Grzegorz Gałęzowski

Test-Driven Development

Extensive Tutorial

Test-Driven Development: Extensive Tutorial

Grzegorz Gałęzowski

This book is available at http://leanpub.com/tdd-ebook

This version was published on 2024-12-16

Tweet This Book!

Please help Grzegorz Gałęzowski by spreading the word about this book on Twitter!

The suggested hashtag for this book is #tddebookxt.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#tddebookxt

Contents

Front Matter . . . 1

Dedications . . . 2

Thanks! . . . 3

About code examples . . . 4
- Notes for C# users . . . 4
- Notes for Java users . . . 4

Part 1: Just the basics . . . 8

Motivation – the first step to learning TDD . . . 9
- What TDD feels like . . . 10
- Let's get it started! . . . 11

The essential tools . . . 12
- Test framework . . . 12
- Mocking framework . . . 17
- Anonymous values generator . . . 23
- Summary . . . 26

It's not (only) a test . . . 27
- When a test becomes something more . . . 27
- Taking it to the software development land . . . 28
- A Specification rather than a test suite . . . 29
- The differences between executable and "traditional" specifications . . . 30

Statement-first programming . . . 31
- What's the point of writing a specification after the fact? . . . 31
- "Test-First" means seeing a failure . . . 33
- "Test-After" often ends up as "Test-Never" . . . 37
- "Test-After" often leads to design rework . . . 38
- Summary . . . 39

Practicing what we have already learned . . . 40
- Let me tell you a story . . . 40
- Act 1: The Car . . . 40
- Act 2: The Customer's Site . . . 41

Act 3: Test-Driven Development . . . 45
Epilogue . . . 60

Sorting out the bits . . . 61

How to start? . . . 62
Start with a good name . . . 62
Start by filling the GIVEN-WHEN-THEN structure with the obvious . . . 66
Start from the end . . . 69
Start by invoking a method if you have one . . . 71
Summary . . . 74

How is TDD about analysis and what does "GIVEN-WHEN-THEN" mean? . . . 75
Is there a commonality between analysis and TDD? . . . 75
Gherkin . . . 76
TODO list... again! . . . 78

What is the scope of a unit-level Statement in TDD? . . . 83
Scope and level . . . 83
On what level do we specify our software? . . . 84
What should be the functional scope of a single Statement? . . . 84
Failing to adhere to the three rules . . . 86
How many assertions do I need? . . . 87
Summary . . . 89

Developing a TDD style and Constrained Non-Determinism . . . 90
A style? . . . 90
Principle: Tests As Specification . . . 90
First technique: Anonymous Input . . . 91
Second technique: Derived Values . . . 92
Third technique: Distinct Generated Values . . . 93
Fourth technique: Constant Specification . . . 95
Summary of the example . . . 97
Constrained non-determinism . . . 97
Summary . . . 98

Specifying functional boundaries and conditions . . . 99
Sometimes, an anonymous value is not enough . . . 99
Exceptions to the rule . . . 100
Rules valid within boundaries . . . 105
Combination of boundaries – ranges . . . 109
Summary . . . 111

Driving the implementation from Specification . . . 113
Type the obvious implementation . . . 113
Fake it ('til you make it) . . . 114
Triangulate . . . 117
Summary . . . 130

Part 2: Object-Oriented World 131

On Object Composability **133**
Another task for Johnny and Benjamin 133
A Quick Retrospective 140

Telling, not asking **141**
Contractors 141
A Quick Retrospective 147

The need for mock objects **148**
Composability... again! 148

Why do we need composability? **149**
Pre-object-oriented approaches 149
Object-oriented programming to the rescue! 151
The power of composition 152
Summary – are you still with me? 157

Web, messages and protocols **158**
So, again, what does it mean to compose objects? 158
Alarms, again! 160
Summary 163

Composing a web of objects **164**
Three important questions 164
A preview of all three answers 164

When are objects composed? **165**

How does a sender obtain a reference to a recipient (i.e. how connections are made)? **166**
Receive as a constructor parameter 166
Receive inside a message (i.e. as a method parameter) 168
Receive in response to a message (i.e. as a method return value) 169
Receive as a registered observer 171

Where are objects composed? **178**
Composition Root 178
Factories 182
Summary 199

Interfaces **200**
Classes vs interfaces 200
Events/callbacks vs interfaces – few words on roles 201
Small interfaces 203

Protocols **208**
Protocols exist 208
Protocol stability 210

Craft messages to reflect the sender's intention . . . 210
Model interactions after the problem domain . . . 211
Message recipients should be told what to do, instead of being asked for information 213
Most of the getters should be removed, return values should be avoided . . . 216
Protocols should be small and abstract . . . 222
Summary . . . 223

Classes . . . 224
Single Responsibility Principle . . . 224
Static recipients . . . 228
Summary . . . 231

Object Composition as a Language . . . 232
More readable composition root . . . 232
Refactoring for readability . . . 234
Composition as a language . . . 243
The significance of a higher-level language . . . 245
Some advice . . . 246
Summary . . . 251

Value Objects . . . 252
What is a value object? . . . 252
Example: money and names . . . 252

Value object anatomy . . . 258
Class signature . . . 259
Hidden data . . . 259
Hidden constructor . . . 259
String conversion methods . . . 265
Equality members . . . 266
The return of investment . . . 267
Summary . . . 269

Aspects of value objects design . . . 270
Immutability . . . 270
Handling of variability . . . 278
Special values . . . 282
Value types and Tell Don't Ask . . . 283
Summary . . . 284

Part 3: TDD in Object-Oriented World . . . 285

Mock Objects as a testing tool . . . 287
A backing example . . . 287
Interfaces . . . 288
Protocols . . . 288
Roles . . . 289

Behaviors . . . 289
Filling in the roles . . . 290
Using a mock channel . . . 292
Mocks as yet another context . . . 294
Summary . . . 294

Test-first using mock objects . . . 295
How to start? – with mock objects . . . 295
Responsibility and Responsibility . . . 295
Channel and DataDispatch one more time . . . 296
The first behavior . . . 296
Second behavior – specifying an error . . . 305
Summary . . . 310

Test-driving at the input boundary . . . 311
Fixing the ticket office . . . 311
Initial objects . . . 312
Bootstrap . . . 315
Writing the first Statement . . . 315
Summary . . . 337

Test-driving at the input boundary – a retrospective . . . 338
Outside-in development . . . 338
Workflow specification . . . 339
Data Transfer Objects and TDD . . . 341
Using a `ReservationInProgress` . . . 345
Interface discovery and the sources of abstractions . . . 349
Do I need all of this to do TDD? . . . 349
What's next? . . . 349

Test-driving object creation . . . 350

Test-driving object creation – a retrospective . . . 356
Limits of creation specification . . . 356
Why specify object creation? . . . 356
What do we specify in the creational Statements? . . . 357
Value object creation . . . 358
Summary . . . 360

Test-driving application logic . . . 361
Summary . . . 371

Test-driving value objects . . . 373
Initial value object . . . 373
Value semantics . . . 374
Case-insensitive comparison . . . 377
Input validation . . . 378
Summary . . . 379

Reaching the web of objects boundaries **380**
What time is it? 380
Timers 383
Threads 386
Others 387

What's inside the object? **388**
What are object's peers? 388
What are object's internals? 388
Examples of internals 390
Summary 397

Design smells visible in the Specification **398**
Design smells' catalog 398
General description 404
Flavors 404

THIS IS ALL I HAVE FOR NOW. WHAT FOLLOWS IS RAW, UNORDERED MATERIAL THAT'S NOT YET READY TO BE CONSUMED AS PART OF THIS TUTORIAL **406**

Mock objects as a design tool **407**
Responsibility-Driven Design 407

Guidance of test smells **408**
Long Statements 408
Lots of stubbing 408
Specifying private members 408

Revisiting topics from chapter 1 . . . . . **409**
Constrained non-determinism in OO world 409
Behavioral boundaries 409
Triangulation 409

Maintainable mock-based Statements **410**
Setup and teardown 410

Refactoring mock code **411**

Part 4: Application architecture 412

On stable/architectural boundaries **413**

Ports and adapters **414**
Physical separation of layers 414

What goes into application? **415**
Application and other layers 415

What goes into ports? **416**

Data transfer objects 416
Ports are not a layer 416

Part 5: TDD on application architecture level 417

Designing automation layer **418**
Adapting screenplay pattern 418
Driver 418
Actors 418
Data builders 418

Further Reading **419**
Motivation – the first step to learning TDD 419
The Essential Tools 419
Value Objects 419

Front Matter

Dedications

Ad Deum qui laetificat iuventutem meam.

To my beloved wife Monika and our lovely son Daniel.

Thanks!

I would like to thank the following people (listed alphabetically by name) for valuable feedback, suggestions, typo fixes and other contributions:

- Brad Appleton
- Borysław Bobulski
- Chris Kucharski
- Daniel Dec
- Daniel Żołopa (cover image)
- Donghyun Lee
- Łukasz Maternia
- Marek Radecki
- Martin Moene
- Michael Whelan
- Polina Kravchenko
- Rafał Bigaj
- Reuven Yagel
- Rémi Goyard
- Robert Pająk
- Wiktor Żołnowski

This book is not original at all. It presents various topics that others invented and I just picked up. Thus, I would also like to thank my mentors and authorities on test-driven development and object-oriented design that I gained most of my knowledge from (listed alphabetically by name):

- Amir Kolsky
- Dan North
- Emily Bache
- Ken Pugh
- Kent Beck
- Mark Seemann
- Martin Fowler
- Nat Pryce
- Philip Schwarz
- Robert C. Martin
- Scott Bain
- Steve Freeman

About code examples

Notes for C# users

The language of choice for code examples is C#, however, I made some exception from the typical C# code conventions.

Dropping "I" from interface names

I am not a big fan of using `ISomething` as an interface name, so I decided to drop the `I` even though most C# developers expect to see it. I hope you can forgive on this one.

Idiomatic C#

Most of the code in this book is not idiomatic C#. I tried to avoid properties, events, and most modern features. My goal is to allow users of other languages (especially Java) to benefit from the book.

Using underscore in field names

Some people like it, some not. I decided to stick to the convention of putting an underscore (_) before a class field name.

Notes for Java users

The language of choice for the code examples is C#. That said, I wanted the book to be as technology-agnostic as possible, to allow especially Java programmers to benefit from it. I tried using a minimum number of C#-specific features and in several places even made remarks targeted at Java users to make it easier for them. Still, there are some things I could not avoid. That's why I wrote up a list containing several of the differences between Java and C# that Java users could benefit from knowing when reading the book.

Naming conventions

Most languages have their default naming conventions. For example, in Java, a class name is written with pascal case (e.g. `UserAccount`), methods and fields are written with camel case, e.g. `payTaxes` and constants/read-only fields are typically written in underscored upper-case (e.g. `CONNECTED_NODES`).

C# uses pascal case for both classes and methods (e.g. `UserAccount`, `PayTaxes`, `ConnectedNodes`). For fields, there are several naming conventions. I chose the one starting with underscore (e.g. `_myDependency`). There are other minor differences, but these are the ones you are going to encounter quite often.

`var` keyword

For example brevity, I chose to use the `var` keyword in the examples. This keyword serves as automatic type inference, e.g.

```
var x = 123; //x inferred as integer
```

Of course, this is no dynamic typing by any means – everything is resolved at compile time.

One more thing – `var` keyword can only be used when the type can be inferred, so occasionally, you will see me declaring types explicitly as in:

```
List<string> list = null; //list cannot be inferred
```

`string` as keyword

C# has a `String` type, similar to Java. It allows, however, to write this type name as keyword, e.g. `string` instead of `String`. This is only syntactic sugar which is used by default by the C# community.

Attributes instead of annotations

In C#, attributes are used for the same purpose as annotations in Java. So, whenever you see:

```
[Whatever]
public void doSomething()
```

think:

```
@Whatever
public void doSomething()
```

`readonly` and `const` instead of `final`

Where Java uses final for constants (together with `static`) and read-only fields, C# uses two keywords: `const` and `readonly`. Without going into details, whenever you see something like:

```
public class User
{
  // a constant with literal:
  private const int DefaultAge = 15;

  // a "constant" object:
  private static readonly TimeSpan DefaultSessionTime
    = TimeSpan.FromDays(2);

  // a read-only instance field:
  private readonly List<int> _marks = new List<int>();
}
```

think:

```
public class User {
  //a constant with literal:
  private static final int DEFAULT_AGE = 15;

  //a "constant" object:
  private static final Duration
    DEFAULT_SESSION_TIME = Duration.ofDays(2);

  // a read-only instance field:
  private final List<Integer> marks = new ArrayList<>();
}
```

A `List<T>`

If you are a Java user, note that in C#, `List<T>` is not an interface, but a concrete class. it is typically used where you would use an `ArrayList`.

Generics

One of the biggest difference between Java and C# is how they treat generics. First of all, C# allows using primitive types in generic declarations, so you can write `List<int>` in C# where in Java you have to write `List<Integer>`.

The other difference is that in C# there is no type erasure as there is in Java. C# code retains all the generic information at runtime. This impacts how most generic APIs are declared and used in C#.

A generic class definition and creation in Java and C# are roughly the same. There is, however, a difference on a method level. A generic method in Java is typically written as:

```
public <T> List<T> createArrayOf(Class<T> type) {
    ...
}
```

and called like this:

```
List<Integer> ints = createArrayOf(Integer.class);
```

whereas in C# the same method would be defined as:

```
public List<T> CreateArrayOf<T>()
{
    ...
}
```

and called as such:

```
List<int> ints = CreateArrayOf<int>();
```

These differences are visible in the design of the library that I use throughout this book for generating test data. While in the C# version, one generates test data by writing:

```
var data = Any.Instance<MyData>();
```

the Java version of the library is used like this:

```
MyData data = Any.instanceOf(MyData.class);
```

Part 1: Just the basics

Status: stable

This chapter will mostly get bugfixes and cosmetic changes.

Without going much into advanced aspects, such as applying TDD to object-oriented systems where multiple objects collaborate (which is a topic of part 2), I introduce the basic TDD philosophy and practices. In terms of design, most of the examples will be about methods of a single object being exercised. The goal is to focus on the core of TDD before going into its specific applications and to slowly introduce some concepts in an easy to grasp manner.

After reading part 1, you will be able to effectively develop classes that have no dependencies on other classes (and on operating system resources) using TDD.

Motivation – the first step to learning TDD

I'm writing this book because I'm an enthusiast of Test-Driven Development (TDD). I believe that TDD is a major improvement over other software development methodologies that I have used to deliver quality software. I also believe that this is true not only for me, but for many other software developers. This raises the question: why don't more people learn and use TDD as their software delivery method of choice? In my professional life, I haven't seen the adoption rate to be high enough to justify the claim that TDD is now mainstream.

I have to respect you for deciding to pick up a book, rather than building your understanding of TDD on the foundation of urban legends and your imagination. I am honored and happy that you chose this one, no matter if this is your first book on TDD or one of many you have opened up in your learning endeavors. As much as I hope you will read this book from cover to cover, I am aware that this doesn't always happen. That makes me want to ask you an important question that may help you decide whether you want to read on: why do you want to learn TDD?

By questioning your motivation, I'm not trying to discourage you from reading this book. Rather, I'd like you to reconsider the goal you want to achieve by reading it. Over time, I have noticed that some of us (myself included) may think we need to learn something (as opposed to wanting to learn something) for various reasons, such as getting a promotion at work, gaining a certificate, adding something to our CV, or just "staying up to date" with recent hypes. Unfortunately, my observation is that Test-Driven Development tends to fall into this category for many people. Such motivation may be difficult to sustain over the long term.

Another source of motivation may be imagining TDD as something it is not. Some of us may only have a vague knowledge of what the real costs and benefits of TDD are. Knowing that TDD is valued and praised by others, we may conclude that it has to be good for us as well. We may have a vague understanding of the reasons, such as "the code will be more tested" for example. As we don't know the real "why" of TDD, we may make up some reasons to practice test-first development, like "to ensure tests are written for everything". Don't get me wrong, these statements might be partially true, but they miss a lot of the essence of TDD. If TDD does not bring the benefits we imagine it might bring, disappointment may creep in. I have heard such disappointed practitioners saying "I don't need TDD, because I need tests that give me confidence on a broader scope" or "Why do I need unit tests[1] when I already have integration tests, smoke tests, sanity tests, exploration tests, etc…?" Many times, I have seen TDD abandoned before it is even understood.

Is learning TDD a high priority for you? Are you determined to try it out and learn it? If you're not, hey, I heard the new series of Game of Thrones is on TV, why don't you check it out instead? Okay, I'm just teasing, but as some say, TDD is "easy to learn, hard to master"[2], so without some

[1]By the way, TDD is not only about unit tests, which we will get to eventually.

[2]I don't know who said it first, I searched the web and found it in few places where none of the writers gave credit to anyone else for it, so I decided just to mention that I'm not the one that coined this phrase.

grit to move on, it will be difficult. Especially since I plan to introduce the content slowly and gradually so that you can get a better explanation of some of the practices and techniques.

What TDD feels like

My brother and I liked to play video games in our childhood – one of the most memorable being Tekken 3 – a Japanese tournament beat'em up for Sony Playstation. Beating the game with all the warriors and unlocking all hidden bonuses, mini-games, etc. took about a day. Some could say the game had nothing to offer since then. Why is it then that we spent more than a year on it?

Tekken3

It is because each fighter in the game had a lot of combos, kicks, and punches that could be mixed in a variety of ways. Some of them were only usable in certain situations, others were something I could throw at my opponent almost anytime without a big risk of being exposed to counterattacks. I could side-step to evade enemy's attacks and, most of all, I could kick another fighter up in the air where they could not block my attacks and I was able to land some nice attacks on them before they fell. These in-the-air techniques were called "juggles". Some magazines published lists of new juggles each month and the hype has stayed in the gaming community for well over a year.

Yes, Tekken was easy to learn – I could put one hour into training the core moves of a character and then be able to "use" this character, but I knew that what would make me a great fighter was

the experience and knowledge on which techniques were risky and which were not, which ones could be used in which situations, which ones, if used one after another, gave the opponent little chance to counterattack, etc. No wonder that soon many tournaments sprang, where players could clash for glory, fame, and rewards. Even today, you can watch some of those old matches on youtube.

TDD is like Tekken. You probably heard the mantra "red-green-refactor" or the general advice "write your test first, then the code", maybe you even did some experiments on your own where you were trying to implement a bubble-sort algorithm or other simple stuff by starting with a test. But that is all like practicing Tekken by trying out each move on its own on a dummy opponent, without the context of real-world issues that make the fight challenging. And while I think such exercises are very useful (in fact, I do a lot of them), I find an immense benefit in understanding the bigger picture of real-world TDD usage as well.

Some people I talk to about TDD sum up what I say to them as, "This is demotivating – there are so many things I have to watch out for, that it makes me never want to start!". Easy, don't panic – remember the first time you tried to ride a bike – you might have been far back then from knowing traffic regulations and following road signs, but that didn't keep you away, did it?

I find TDD very exciting and it makes me excited about writing code as well. Some guys of my age already think they know all about coding, are bored with it and cannot wait until they move to management or requirements or business analysis, but hey! I have a new set of techniques that makes my coding career challenging again! And it is a skill that I can apply to many different technologies and languages, making me a better developer overall! Isn't that something worth aiming for?

Let's get it started!

In this chapter, I tried to provoke you to rethink your attitude and motivation. If you are still determined to learn TDD with me by reading this book, which I hope you are, then let's get to work!

The essential tools

Ever watched Karate Kid, either the old version or the new one? The thing they have in common is that when the "kid" starts learning karate (or kung-fu) from his master, he is given a basic, repetitive task (like taking off a jacket and putting it on again), not knowing yet where it would lead him. Or look at the first Rocky film (yeah, the one starring Sylvester Stallone), where Rocky chases a chicken to train agility.

When I first tried to learn how to play guitar, I found two pieces of advice on the web: the first was to start by mastering a single, difficult song. The second was to play with a single string, learn how to make it sound in different ways and try to play some melodies by ear just with this one string. Do I have to tell you that the second piece of advice worked better?

Honestly, I could dive right into the core techniques of TDD, but I feel this would be like putting you on a ring with a demanding opponent – you would most probably be discouraged before gaining the necessary skills. So, instead of explaining how to win a race, in this chapter we will take a look at what shiny cars we will be driving.

In other words, I will give you a brief tour of the three tools we will use throughout this book.

In this chapter, I will oversimplify some things just to get you up and running without getting into the philosophy of TDD yet (think: physics lessons in primary school). Don't worry about it :-), I will make up for it in the coming chapters!

Test framework

The first tool we'll use is a test framework. A test framework allows us to specify and execute our tests.

Let's assume for the sake of this introduction that we have an application that accepts two numbers from the command line, multiplies them and prints the result on the console. The code is pretty straightforward:

```
1  public static void Main(string[] args)
2  {
3    try
4    {
5      int firstNumber = Int32.Parse(args[0]);
6      int secondNumber = Int32.Parse(args[1]);
7
8      var result =
9        new Multiplication(firstNumber, secondNumber).Perform();
10
11     Console.WriteLine("Result is: " + result);
```

```
  }
  catch(Exception e)
  {
    Console.WriteLine("Multiplication failed because of: " + e);
  }
}
```

Now, let's assume we want to check whether this application produces correct results. The most obvious way would be to invoke it from the command line manually with some exemplary arguments, then check the output to the console and compare it with what we expected to see. Such testing session could look like this:

```
C:\MultiplicationApp\MultiplicationApp.exe 3 7
21
C:\MultiplicationApp\
```

As you can see, our application produces a result of 21 for the multiplication of 3 by 7. This is correct, so we assume the application has passed the test.

Now, what if the application also performed addition, subtraction, division, calculus, etc.? How many times would we have to invoke the application manually to make sure every operation works correctly? Wouldn't that be time-consuming? But wait, we are programmers, right? So we can write programs to do the testing for us! For example, here is a source code of a program that uses the Multiplication class, but in a slightly different way than the original application:

```
public static void Main(string[] args)
{
  var multiplication = new Multiplication(3,7);

  var result = multiplication.Perform();

  if(result != 21)
  {
    throw new Exception("Failed! Expected: 21 but was: " + result);
  }
}
```

It looks simple, isn't it? Now, let's use this code as a basis to build a very primitive test framework, just to show the pieces that such frameworks consist of. As a step in that direction, we can extract the verification of the `result` into a reusable method – after all, we will be adding division in a second, remember? So here goes:

```
public static void Main(string[] args)
{
  var multiplication = new Multiplication(3,7);

  var result = multiplication.Perform();

  AssertTwoIntegersAreEqual(expected: 21, actual: result);
}

//extracted code:
public static void AssertTwoIntegersAreEqual(
  int expected, int actual)
{
  if(actual != expected)
  {
    throw new Exception(
      "Failed! Expected: "
        + expected + " but was: " + actual);
  }
}
```

Note that I started the name of this extracted method with "Assert" – we will get back to the naming soon, for now just assume that this is a good name for a method that verifies that a result matches our expectation. Let's take one last round and extract the test itself so that its code is in a separate method. This method can be given a name that describes what the test is about:

```
public static void Main(string[] args)
{
  Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers();
}

public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Assuming...
  var multiplication = new Multiplication(3,7);

  //when this happens:
  var result = multiplication.Perform();

  //then the result should be...
  AssertTwoIntegersAreEqual(expected: 21, actual: result);
}

public static void AssertTwoIntegersAreEqual(
  int expected, int actual)
```

```
21  {
22    if(actual != expected)
23    {
24      throw new Exception(
25      "Failed! Expected: " + expected + " but was: " + actual);
26    }
27  }
```

And we're done. Now if we need another test, e.g. for division, we can just add a new method call to the `Main()` method and implement it. Inside this new test, we can reuse the `AssertTwoIntegersAreEqual()` method, since the check for division would also be about comparing two integer values.

As you see, we can easily write automated checks like this, using our primitive methods. However, this approach has some disadvantages:

1. Every time we add a new test, we have to update the `Main()` method with a call to the new test. If we forget to add such a call, the test will never be run. At first, it isn't a big deal, but as soon as we have dozens of tests, an omission will become hard to notice.
2. Imagine your system consists of more than one application – you would have some problems trying to gather summary results for all of the applications that your system consists of.
3. Soon you'll need to write a lot of other methods similar to `AssertTwoIntegersAreEqual()` – the one we already have compares two integers for equality, but what if we wanted to check a different condition, e.g. that one integer is greater than another? What if we wanted to check the equality not for integers, but characters, strings, floats, etc.? What if we wanted to check some conditions on collections, e.g. that a collection is sorted or that all items in the collection are unique?
4. Given a test fails, it would be hard to navigate from the command line output to the corresponding line of the source in your IDE. Wouldn't it be easier if you could click on the error message to take you immediately to the code where the failure occurred?

For these and other reasons, advanced automated test frameworks were created such as CppUnit (for C++), JUnit (for Java) or NUnit (C#). Such frameworks are in principle based on the very idea that I sketched above, plus they make up for the deficiencies of our primitive approach. They derive their structure and functionality from Smalltalk's SUnit and are collectively referred to as **xUnit family** of test frameworks.

To be honest, I can't wait to show you how the test we just wrote looks like when a test framework is used. But first, let's recap what we've got in our straightforward approach to writing automated tests and introduce some terminology that will help us understand how automated test frameworks solve our issues:

1. The `Main()` method serves as a **Test List** – a place where it is decided which tests to run.
2. The `Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()` method is a **Test Method**.

3. The `AssertTwoIntegersAreEqual()` method is an **Assertion** – a condition that, when not met, ends a test with failure.

To our joy, those three elements are present as well when we use a test framework. Moreover, they are far more advanced than what we have. To illustrate this, here is (finally!) the same test we wrote above, now using the xUnit.Net[3] test framework:

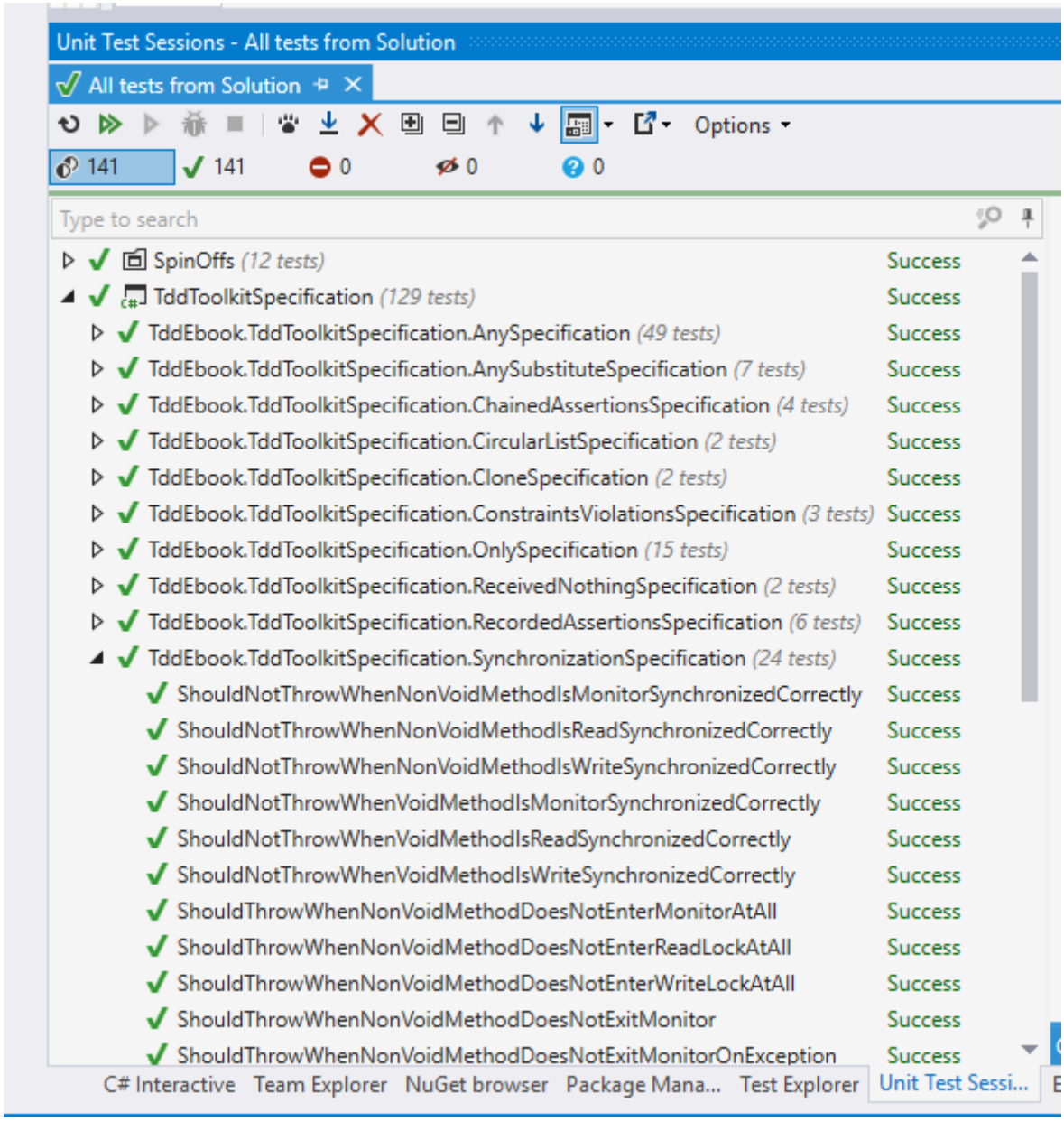
```
[Fact] public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Assuming...
  var multiplication = new Multiplication(3,7);

  //when this happens:
  var result = multiplication.Perform();

  //then the result should be...
  Assert.Equal(21, result);
}
```

Looking at the example, we can see that the test method itself is the only thing that's left – the two methods (the test list and assertion) that we previously had are gone now. Well, to tell you the truth, they are not literally gone – it's just that the test framework offers far better replacements, so we used them instead. Let's reiterate the three elements of the previous version of the test that I promised would be present after the transition to the test framework:

1. The **Test List** is now created automatically by the framework from all methods marked with a `[Fact]` attribute. There's no need to maintain one or more central lists anymore, so the `Main()` method is no more.
2. The **Test Method** is present and looks almost the same as before.
3. The **Assertion** takes the form of a call to the static `Assert.Equal()` method – the xUnit.NET framework is bundled with a wide range of assertion methods, so I used one of them. Of course, no one stops you from writing your custom assertion if the built-in assertion methods don't offer what you are looking for.

Phew, I hope I made the transition quite painless for you. Now the last thing to add – as there is no `Main()` method anymore in the last example, you surely must wonder how we run those tests, right? Ok, the last big secret unveiled – we use an external application for this (we will refer to it using the term **Test Runner**) – we tell it which assemblies to run and then it loads them, runs them, reports the results, etc. A Test Runner can take various forms, e.g. it can be a console application, a GUI application or a plugin for an IDE. Here is an example of a test runner provided by a plugin for Visual Studio IDE called Resharper:

[3] https://xunit.net/



Resharper test runner docked as a window in Visual Studio 2015 IDE

Mocking framework

This introduction is written for those who are not proficient with using mocks. Even though, I accept the fact that the concept may be too difficult for you to grasp. If, while reading this section, you find yourself lost, please skip it. We won't be dealing with mock objects until part 2, where I offer a richer and more accurate description of the concept.

When we want to test a class that depends on other classes, we may think it's a good idea to include those classes in the test as well. This, however, does not allow us to test a single object or

a small cluster of objects in isolation, where we would be able to verify that just a small part of the application works correctly. Thankfully, if we make our classes depend on interfaces rather than other classes, we can easily implement those interfaces with special "fake" classes that can be crafted in a way that makes our testing easier. For example, objects of such classes may contain pre-programmed return values for some methods. They can also record the methods that are invoked on them and allow the test to verify whether the communication between our object under test and its dependencies is correct.

Nowadays, we can rely on tools to generate such a "fake" implementation of a given interface for us and let us use this generated implementation in place of a real object in tests. This happens in a different way, depending on a language. Sometimes, the interface implementations can be generated at runtime (like in Java or C#), sometimes we have to rely more on compile-time generation (e.g. in C++).

Narrowing it down to C# – a mocking framework is just that – a mechanism that allows us to create objects (called "mock objects" or just "mocks"), that adhere to a certain interface, at runtime. It works like this: the type of the interface we want to have implemented is usually passed to a special method which returns a mock object based on that interface (we'll see an example in a few seconds). Aside from the creation of mock objects, such a framework provides an API to configure the mocks on how they behave when certain methods are called on them and allows us to inspect which calls they received. This is a very powerful feature, because we can simulate or verify conditions that would be difficult to achieve or observe using only production code. Mocking frameworks are not as old as test frameworks so they haven't been used in TDD since the very beginning.

I'll give you a quick example of a mocking framework in action now and defer further explanation of their purpose to later chapters, as the full description of mocks and their place in TDD is not so easy to convey.

Let's pretend that we have a class that allows placing orders and then puts these orders into a database (using an implementation of an interface called `OrderDatabase`). Besides, it handles any exception that may occur, by writing it into a log. The class itself does not do any important stuff, but let's try to imagine hard that this is some serious domain logic. Here's the code for this class:

```
1  public class OrderProcessing
2  {
3    OrderDatabase _orderDatabase; //OrderDatabase is an interface
4    Log _log;
5
6    //we get the database object  from outside the class:
7    public OrderProcessing(
8      OrderDatabase database,
9      Log log)
10   {
11     _orderDatabase = database;
12     _log = log;
13   }
14
15   //other code...
```

```
16
17    public void Place(Order order)
18    {
19      try
20      {
21        _orderDatabase.Insert(order);
22      }
23      catch(Exception e)
24      {
25        _log.Write("Could not insert an order. Reason: " + e);
26      }
27    }
28
29    //other code...
30  }
```

Now, imagine we need to test it – how do we do that? I can already see you shake your head and say: "Let's just create a database connection, invoke the `Place()` method and see if the record is added properly into the database". If we did that, the first test would look like this:

```
1   [Fact] public void
2   ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
3   {
4     //GIVEN
5     var orderDatabase = new MySqlOrderDatabase(); //uses real database
6     orderDatabase.Connect();
7     orderDatabase.Clean(); //clean up after potential previous tests
8     var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
9     var order = new Order(
10      name: "Grzesiek",
11      surname: "Galezowski",
12      product: "Agile Acceptance Testing",
13      date: DateTime.Now,
14      quantity: 1);
15
16    //WHEN
17    orderProcessing.Place(order);
18
19    //THEN
20    var allOrders = orderDatabase.SelectAllOrders();
21    Assert.Contains(order, allOrders);
22  }
```

At the beginning of the test, we open a connection to the database and clean all existing orders in it (more on that shortly), then create an order object, insert it into the database and query the database for all orders it contains. At the end, we assert that the order we tried to insert is among all orders in the database.

Why do we clean up the database at the beginning of the test? Remember that a database provides persistent storage. If we don't clean it up before executing the logic of this test, the database may already contain the item we are trying to add, e.g. from previous executions of this test. The database might not allow us to add the same item again and the test would fail. Ouch! It hurts so bad, because we wanted our tests to prove something works, but it looks like it can fail even when the logic is coded correctly. Of what use would be such a test if it couldn't reliably tell us whether the implemented logic is correct or not? So, to make sure that the state of the persistent storage is the same every time we run this test, we clean up the database before each run.

Now that the test is ready, did we get what we wanted from it? I would be hesitant to answer "yes". There are several reasons for that:

1. The test will most probably be slow because accessing the database is relatively slow. It is not uncommon to have more than a thousand tests in a suite and I don't want to wait half an hour for results every time I run them. Do you?
2. Everyone who wants to run this test will have to set up a special environment, e.g. a local database on their machine. What if their setup is slightly different from ours? What if the schema gets outdated – will everyone manage to notice it and update the schema of their local databases accordingly? Should we re-run our database creation script only to ensure we have got the latest schema available to run your tests against?
3. There may be no implementation of the database engine for the operating system running on our development machine if our target is an exotic or mobile platform.
4. Note that the test we wrote is only one out of two. We still have to write another one for the scenario where inserting an order ends with an exception. How do we setup the database in a state where it throws an exception? It is possible, but requires significant effort (e.g. deleting a table and recreating it after the test, for use by other tests that might need it to run correctly), which may lead some to the conclusion that it is not worth writing such tests at all.

Now, let's try to approach this problem differently. Let's assume that the `MySqlOrderDatabase` that queries a real database query is already tested (this is because I don't want to get into a discussion on testing database queries just yet – we'll get to it in later chapters) and that the only thing we need to test is the `OrderProcessing` class (remember, we're trying to imagine really hard that there is some serious domain logic coded here). In this situation we can leave the `MySqlOrderDatabase` out of the test and instead create another, fake implementation of the `OrderDatabase` that acts as if it was a connection to a database but does not write to a real database at all – it only stores the inserted records in a list in memory. The code for such a fake connection could look like this:

```
public class FakeOrderDatabase : OrderDatabase
{
  public Order _receivedArgument;

  public void Insert(Order order)
  {
    _receivedArgument = order;
  }

  public List<Order> SelectAllOrders()
  {
    return new List<Order>() { _receivedOrder };
  }
}
```

Note that the fake order database is an instance of a custom class that implements the same interface as `MySqlOrderDatabase`. Thus, if we try, we can make the tested code use our fake without knowing.

Let's replace the real implementation of the order database by the fake instance in the test:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
{
  //GIVEN
  var orderDatabase = new FakeOrderDatabase();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  var allOrders = orderDatabase.SelectAllOrders();
  Assert.Contains(order, allOrders);
}
```

Note that we do not clean the fake database object as we did with the real database since we create a fresh object each time the test is run and the results are stored in a memory location different for each instance. The test will also be much quicker now because we are not accessing the database anymore. What's more, we can now easily write a test for the error case. How? Just make another fake class, implemented like this:

```
public class ExplodingOrderDatabase : OrderDatabase
{
  public void Insert(Order order)
  {
    throw new Exception();
  }

  public List<Order> SelectAllOrders()
  {
  }
}
```

Ok, so far so good, but now we have two classes of fake objects to maintain (and chances are we will need even more). Any method added to the `OrderDatabase` interface must also be added to each of these fake classes. We can spare some coding by making our mocks a bit more generic so that their behavior can be configured using lambda expressions:

```
public class ConfigurableOrderDatabase : OrderDatabase
{
  public Action<Order> doWhenInsertCalled;
  public Func<List<Order>> doWhenSelectAllOrdersCalled;

  public void Insert(Order order)
  {
    doWhenInsertCalled(order);
  }

  public List<Order> SelectAllOrders()
  {
    return doWhenSelectAllOrdersCalled();
  }
}
```

Now, we don't have to create additional classes for new scenarios, but our syntax becomes awkward. Here's how we configure the fake order database to remember and yield the inserted order:

```
var db = new ConfigurableOrderDatabase();
Order gotOrder = null;
db.doWhenInsertCalled = o => {gotOrder = o;};
db.doWhenSelectAllOrdersCalled = () => new List<Order>() { gotOrder };
```

And if we want it to throw an exception when anything is inserted:

```
var db = new ConfigurableOrderDatabase();
db.doWhenInsertCalled = o => {throw new Exception();};
```

Thankfully, some smart programmers created libraries that provide further automation in such scenarios. One such a library is **NSubstitute**[4]. It provides an API in a form of C# extension methods, which is why it might seem a bit magical at first, especially if you're not familiar with C#. Don't worry, you'll get used to it.

Using NSubstitute, our first test can be rewritten as:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderisPlaced()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

Note that we don't need the `SelectAllOrders()` method on the database connection interface anymore. It was there only to make writing the test easier – no production code used it. We can delete the method and get rid of some more maintenance trouble. Instead of the call to `SelectAllOrders()`, mocks created by NSubstitute record all calls received and allow us to use a special method called `Received()` on them (see the last line of this test), which is actually a camouflaged assertion that checks whether the `Insert()` method was called with the order object as parameter.

This explanation of mock objects is very shallow and its purpose is only to get you up and running. We'll get back to mocks later as we've only scratched the surface here.

Anonymous values generator

Looking at the test data in the previous section we see that many values are specified literally, e.g. in the following code:

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

```
var order = new Order(
  name: "Grzesiek",
  surname: "Galezowski",
  product: "Agile Acceptance Testing",
  date: DateTime.Now,
  quantity: 1);
```

the name, surname, product, date, and quantity are very specific. This might suggest that the exact values are important from the perspective of the behavior we are testing. On the other hand, when we look at the tested code again:

```
public void Place(Order order)
{
  try
  {
    this.orderDatabase.Insert(order);
  }
  catch(Exception e)
  {
    this.log.Write("Could not insert an order. Reason: " + e);
  }
}
```

we can spot that these values are not used anywhere – the tested class does not use or check them in any way. These values are important from the database point of view, but we already took the real database out of the picture. Doesn't it trouble you that we fill the order object with so many values that are irrelevant to the test logic itself and that clutter the structure of the test with needless details? To remove this clutter let's introduce a method with a descriptive name to create the order and hide the details we don't need from the reader of the test:

```
[Fact] public void
ShouldInsertNewOrderToDatabase()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = AnonymousOrder();

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}

public Order AnonymousOrder()
```

```
{
  return new Order(
    name: "Grzesiek",
    surname: "Galezowski",
    product: "Agile Acceptance Testing",
    date: DateTime.Now,
    quantity: 1);
}
```

Now, that's better. Not only did we make the test shorter, we also provided a hint to the reader that the actual values used to create an order don't matter from the perspective of tested order-processing logic. Hence the name `AnonymousOrder()`.

By the way, wouldn't it be nice if we didn't have to provide the anonymous objects ourselves, but could rely on another library to generate these for us? Surprise, surprise, there is one! It's called **Autofixture**[5]. It is an example of a so-called anonymous values generator (although its creator likes to say that it is also an implementation of the Test Data Builder pattern, but let's skip this discussion here).

After changing our test to use AutoFixture, we arrive at the following:

```
private Fixture any = new Fixture();

[Fact] public void
ShouldInsertNewOrderToDatabase()
{
  //GIVEN
  var orderDatabase = Substitute.For<OrderDatabase>();
  var orderProcessing = new OrderProcessing(orderDatabase, new FileLog());
  var order = any.Create<Order>();

  //WHEN
  orderProcessing.Place(order);

  //THEN
  orderDatabase.Received(1).Insert(order);
}
```

In this test, we use an instance of a `Fixture` class (which is a part of AutoFixture) to create anonymous values for us via a method called `Create()`. This allows us to remove the `AnonymousOrder()` method, thus making our test setup shorter.

Nice, huh? AutoFixture has a lot of advanced features, but to keep things simple I like to hide its use behind a static class called `Any`. The simplest implementation of such class would look like this:

[5] https://github.com/AutoFixture/AutoFixture

```
public static class Any
{
  private static any = new Fixture();

  public static T Instance<T>()
  {
    return any.Create<T>();
  }
}
```

In the next chapters, we'll see many different methods from the `Any` type, plus the full explanation of the philosophy behind it. The more you use this class, the more it grows with other methods for creating customized objects.

Summary

This chapter introduced the three tools we'll use in this book that, when mastered, will make your test-driven development flow smoother. If this chapter leaves you with insufficient justification for their use, don't worry – we will dive into the philosophy behind them in the coming chapters. For now, I just want you to get familiar with the tools themselves and their syntax. Go on, download these tools, launch them, try to write something simple with them. You don't need to understand their full purpose yet, just go out and play :-).

It’s not (only) a test

Is the role of a test only to “verify” or “check” whether a piece of software works? Surely, this is a significant part of its runtime value, i.e. the value that we get when we execute the test. However, when we limit our perspective on tests only to this, it could lead us to a conclusion that the only thing that is valuable about having a test is to be able to execute it and view the result. Such acts as designing a test or implementing a test would only have the value of producing something we can run. Reading a test would only have value when debugging. Is this really true?

In this chapter, I argue that the acts of designing, implementing, compiling and reading a test are all very valuable activities. And they let us treat tests as something more than just “automated checks”.

When a test becomes something more

I studied in Łódź, a large city in the center of Poland. As probably all other students in all other countries, we have had lectures, exercises and exams. The exams were pretty difficult. As my computer science group was on the faculty of electronic and electric engineering, we had to grasp a lot of classes that didn’t have anything to do with programming. For instance: electrotechnics, solid-state physics or electronic and electrical metrology.

Knowing that exams were difficult and that it was hard to learn everything during the semester, the lecturers would sometimes give us exemplary exams from previous years. The questions were different from the actual exams that we were to take, but the structure and kinds of questions asked (practice vs. theory etc.) were similar. We would usually get these exemplary questions before we started learning really hard (which was usually at the end of a semester). Guess what happened then? As you might suspect, we did not use the tests we received just to “verify” or “check” our knowledge after we finished learning. Quite the contrary – examining those tests was the very first step of our preparation. Why was that so? What use were the tests when we knew we wouldn’t know most of the answers?

I guess my lecturers would disagree with me, but I find it quite amusing that what we were really doing back then was similar to “lean software development”. Lean is a philosophy where, among other things, there is a rigorous emphasis on eliminating waste. Every feature or product that is produced but is not needed by anyone, is considered a waste. That’s because if something is not needed, there is no reason to assume it will ever be needed. In that case, the entire feature or product adds no value. Even if it ever *will* be needed, it very likely will require rework to fit the customer’s needs at that time. In such a case, the work that went into the parts of the original solution that had to be reworked is a waste – it had a cost, but brought no benefit (I am not talking about such things as customer demos, but finished, polished features or products).

So, to eliminate waste, we usually try to “pull features from demand” instead of “pushing them” into a product, hoping they can become useful one day. In other words, every feature is there to satisfy a concrete need. If not, the effort is considered wasted and the money drowned.

Going back to the exams example, why can the approach of first looking through the exemplary tests be considered "lean"? That's because, when we treat passing an exam as our goal, then everything that does not put us closer to this goal is considered wasteful. Let's suppose the exam concerns theory only – why then practice the exercises? It would probably pay off a lot more to study the theoretical side of the topics. Such knowledge could be obtained from those exemplary tests. So, the tests were a kind of specification of what was needed to pass the exam. It allowed us to pull the value (i.e. our knowledge) from the demand (information obtained from realistic tests) rather that push it from the implementation (i.e. learning everything in a coursebook chapter after chapter).

So the tests became something more. They proved very valuable before the "implementation" (i.e. learning for the exam) because:

1. they helped us focus on what was needed to reach our goal
2. they brought our attention away from what was **not** needed to reach our goal

That was the value of a test before learning. Note that the tests we would usually receive were not exactly what we would encounter at the time of the exam, so we still had to guess. Yet, the role of a **test as a specification of a need** was already visible.

Taking it to the software development land

I chose this lengthy metaphor to show you that a writing a "test" is really another way of specifying a requirement or a need and that it's not counterintuitive to think about it this way – it occurs in our everyday lives. This is also true in software development. Let's take the following "test" and see what kind of needs it specifies:

```
1 var reporting = new ReportingFeature();
2 var anyPowerUser = Any.Of(Users.Admin, Users.Auditor);
3 Assert.True(reporting.CanBePerformedBy(anyPowerUser));
```

(In this example, we used `Any.Of()` method that returns any enumeration value from the specified list. Here, we say "give me a value that is either `Users.Admin` or `Users.Auditor`".)

Let's look at those (only!) three lines of code and imagine that the production code that makes this "test" pass does not exist yet. What can we learn from these three lines about what this production code needs to supply? Count with me:

1. We need a reporting feature.
2. We need to support the notion of users and privileges.
3. We need to support the concept of a power user, who is either an administrator or an auditor.
4. Power users need to be allowed to use the reporting feature (note that it does not specify which other users should or should not be able to use this feature – we would need a separate "test" for that).

Also, we are already after the phase of designing an API (because the test is already using it) that will fulfill the need. Don't you think this is already quite some information about the application functionality from just three lines of code?

A Specification rather than a test suite

I hope you can see now that what we called "a test" can also be seen as a kind of specification. This is also the answer to the question I raised at the beginning of this chapter.

In reality, the role of a test, if written before production code, can be broken down even further:

- designing a scenario – is when we specify our requirements by giving concrete examples of behaviors we expect
- writing the test code – is when we specify an API through which we want to use the code that we are testing
- compiling – is when we get feedback on whether the production code has the classes and methods required by the specification we wrote. If it doesn't, the compilation will fail.
- execution – is where we get feedback on whether the production code exhibits the behaviors that the specification describes
- reading – is where we use the already written specification to obtain knowledge about the production code.

Thus, the name "test" seems like narrowing down what we are doing here too much. My feeling is that maybe a different name would be better – hence the term *specification*.

The discovery of the tests' role as a specification is quite recent and there is no uniform terminology connected to it yet. Some like to call the process of using tests as specifications *Specification By Example* to say that the tests are examples that help specify and clarify the functionality being developed. Some use the term BDD (*Behavior-Driven Development*) to emphasize that writing tests is really about analyzing and describing behaviors. Also, you might encounter different names for some particular elements of this approach, for example, a "test" can be referred to as a "spec", or an "example", or a "behavior description", or a "specification statement" or "a fact about the system" (as you already saw in the chapter on tools, the xUnit.NET framework marks each "test" with a `[Fact]` attribute, suggesting that by writing it, we are stating a single fact about the developed code. By the way, xUnit.NET also allows us to state 'theories' about our code, but let's leave this topic for another time).

Given this variety in terminology, I'd like to make a deal: to be consistent throughout this book, I will establish a naming convention, but leave you with the freedom to follow your own if you so desire. The reason for this naming convention is pedagogical – I am not trying to create a movement to change established terms or to invent a new methodology or anything – I hope that by using this terminology throughout the book, you'll look at some things differently[6]. So, let's agree that for the sake of this book:

Specification Statement (or simply Statement, with a capital 'S')
: will be used instead of the words "test" and "test method"

Specification (or simply Spec, also with a capital 'S')
: will be used instead of the words "test suite" and "test list"

False Statement
: will be used instead of "failing test"

[6] besides, this book is open source, so if you don't like the terminology, you are free to create a fork and change it to your liking!

True Statement
: will be used instead of "passing test"

From time to time I'll refer back to the "traditional" terminology, because it is better established and because you may have already heard some other established terms and wonder how they should be understood in the context of thinking of tests as a specification.

The differences between executable and "traditional" specifications

You may be familiar with requirements specifications or design specifications that are written in plain English or another spoken language. However, our Specifications differ from them in several ways. In particular, the kind of Specification that we create by writing tests:

1. Is not *completely* written up-front like many of such "traditional" specs have been written (which doesn't mean it's written after the code is done – more on this in the next chapters).
2. Is executable – you can run it to see whether the code adheres to the specification or not. This lowers the risk of inaccuracies in the Specification and falling out of sync with the production code.
3. Is written in source code rather than in spoken language – which is both good, as the structure and formality of code leave less room for misunderstanding, and challenging, as great care must be taken to keep such specification readable.

Statement-first programming

What's the point of writing a specification after the fact?

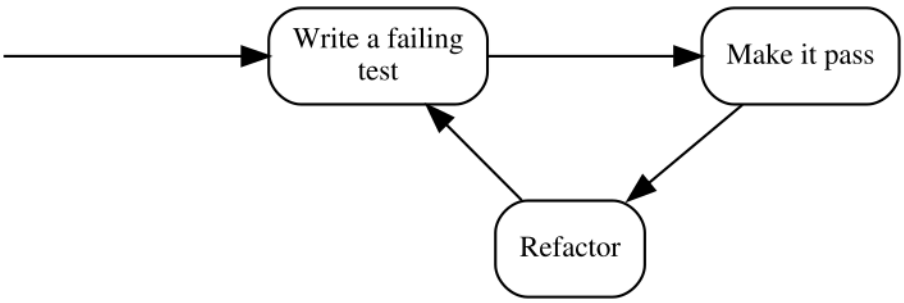
One of the best known thing about TDD is that a failing test for a behavior of a piece of code is written before this behavior is implemented. This concept is often called "test-first development" and seems controversial to many.

In the previous chapter, I said that in TDD a "test" takes an additional role – one of a statement that is part of a specification. If we put it this way, then the whole controversial concept of "writing a test before the code" does not pose a problem at all. Quite the contrary – it only seems natural to specify what we expect from a piece of code to do before we attempt to write it. Does the other way round even make sense? A specification written after completing the implementation is nothing more than an attempt at documenting the existing solution. Sure, such attempts can provide some value when done as a kind of reverse-engineering (i.e. writing the specification for something that was implemented long ago and for which we uncover the previously implicit business rules or policies as we document the existing solution) – it has an excitement of discovery in it, but doing so just after we made all the decisions ourselves doesn't seem to me like a productive way to spend my time, not to mention that I find it dead boring (you can check whether you're like me on this one. Try implementing a simple calculator app and then write specification for it just after it is implemented and manually verified to work). Anyway, I hardly find specifying how something should work after it works creative. Maybe that's the reason why, throughout the years, I have observed the specifications written after a feature is implemented to be much less complete than the ones written before the implementation.

Oh, and did I tell you that without a specification of any kind we don't know whether we are done implementing our changes or not? This is because, to determine if the change is complete, we need to compare the implemented functionality to "something", even if this "something" is only in the customer's head. in TDD, we "compare" it to expectations set by a suite of automated tests.

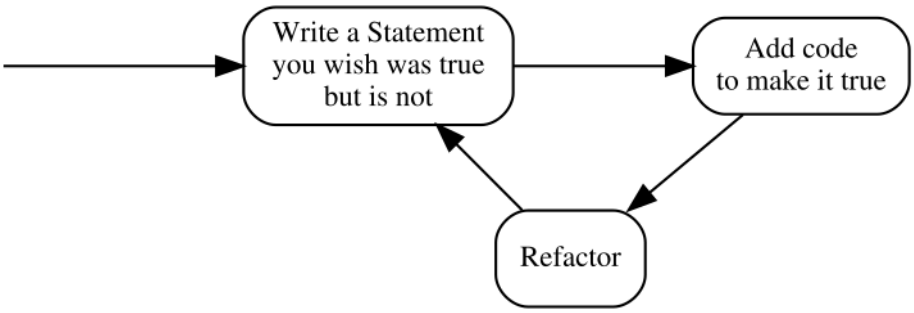
Another thing I mentioned in the previous chapter is that we approach writing a Specification of executable Statements differently from writing a textual design or requirements specification: even though a behavior is implemented after its Specification is ready, we do not write the Specification entirely up-front. The usual sequence is to specify a bit first and then code a bit, repeating it one Statement at a time. When doing TDD, we are traversing repeatedly through a few phases that make up a cycle. We like these cycles to be short, so that we get feedback early and often. This is essential because it allows us to move forward, confident that what we already have works as we intended. It also enables us to make the next cycle more efficient thanks to the knowledge we gained in the previous cycle (if you don't believe me that fast feedback matters, ask yourself a question: "how many times a day do I compile the code I'm working on?").

Reading so much about cycles, it is probably no surprise that the traditional illustration of the TDD process is modeled visually as a circular flow:



Basic TDD cycle

Note that the above form uses the traditional terminology of TDD, so before I explain the steps, here's a similar illustration that uses our terms of Specification and Statements:



Basic TDD cycle with changed terminology

The second version seems more like common sense than the first one – specifying how something should behave before putting that behavior in place is way more intuitive than testing something that does not yet exist.

Anyway, these three steps deserve some explanation. In the coming chapters, I'll give you some examples of how this process works in practice and introduce an expanded version, but in the meantime, it suffices to say that:

Write a Statement you wish were true but is not
: means that the Statement evaluates to false. In the test list, it appears as failing, which most xUnit frameworks mark with red color.

Add code to make it true
: means that we write just enough code to make the Statement true. In the test list, it appears as passing, which most xUnit frameworks mark with green color. Later in the course of the book, you'll see how little can be "just enough".

Refactor
: is a step that I have silently ignored so far and will do so for several more chapters. Don't worry, we'll get back to it eventually. For now, it's important to be aware that the executable Specification can act as a safety net while we are improving the quality of the code without changing its externally visible behavior: by running the Specification often, we quickly discover any mistake we make in the process.

By the way, this process is sometimes referred to as "Red-Green-Refactor", because of the colors that xUnit tools display for failing and passing test. I am just mentioning it here for the record – I will not be using this term further in the book.

"Test-First" means seeing a failure

Explaining the illustration with the TDD process above, I pointed out that we are supposed to write a Statement that we wish was true **but is not**. It means that not only do we have to write a Statement before we provide an implementation that makes it true, we also have to evaluate it (i.e. run it) and watch it fail its assertions before we provide the implementation.

Why is it so important? Isn't it enough to write the Statement first? Why run it and watch it fail? There are several reasons and I will try to outline some of them briefly.

The main reason for writing a Statement and watching it fail is that otherwise, I don't have any proof that the Statement can ever fail.

Every accurate Statement fails when it isn't fulfilled and passes when it is. That's one of the main reasons why we write it – to see this transition from *red* to *green*, which means that what previously was not implemented (and we had a proof for that) is now working (and we have a proof). Observing the transition proves that we made progress.

Another thing to note is that, after being fulfilled, the Statement becomes a part of the executable specification and starts failing as soon as the code stops fulfilling it, for example as a result of a mistake made during code refactoring.

Seeing a Statement proven as false gives us valuable feedback. If we run a Statement only *after* the behavior it describes has been implemented and it is evaluated as true, how do we know whether it accurately describes a need? We never saw it failing, so what proof do we have that it ever will?

The first time I encountered this argument was before I started thinking of tests as an executable specification. "Seriously?" – I thought – "I know what I'm writing. If I make my tests small enough, it is self-evident that I am describing the correct behavior. This is paranoid". However, life quickly verified my claims and I was forced to withdraw my arguments. Let me describe three of the ways I experienced how one can write a Statement that is always true, whether the code is correct or not. There are more ways, however, I think giving you three should be an illustration enough.

Test-first allowed me to avoid the following situations where Statements cheated me into thinking they were fulfilled even when they shouldn't be:

1. Accidental omission of including a Statement in a Specification

It's usually insufficient to just write the code of a Statement – we also have to let the test runner know that a method we wrote is a Statement (not e.g. just a helper method) and it needs to be evaluated, i.e. ran by the runner.

Most xUnit frameworks have some kind of mechanism to mark methods as Statements, whether by using attributes (C#, e.g. `[Fact]`) or annotations (Java, e.g `@Test`), or by using macros (C and C++), or by using a naming convention. We have to use such a mechanism to let the runner know that it should execute such methods.

Let's take xUnit.Net as an example. To turn a method into a Statement in xUnit.Net, we have to mark it with the `[Fact]` attribute like this:

```
public class CalculatorSpecification
{
  [Fact]
  public void ShouldDisplayAdditionResultAsSumOfArguments()
  {
    //...
  }
}
```

There is a chance that we forget to decorate a method with the `[Fact]` attribute – in such a case, this method is never executed by the test runner. However funny it may sound, this is exactly what happened to me several times. Let's take the above Statement as an example and imagine that we are writing this Statement post-factum as a unit test in an environment that has, let's say, more than thirty Statements already written and passing. We have written the code and now we are just creating test after test to ensure the code works. Test – pass, test – pass, test – pass. When I execute tests, I almost always run more than one at a time, since it's easier for me than selecting what to evaluate each time. Besides, I get more confidence this way that I don't make a mistake and break something that is already working. Let's imagine we are doing the same here. Then the workflow is really: Test – all pass, test – all pass, test – all pass…

Over time, I have learned to use code snippets mechanism of my IDE to generate a template body for my Statements. Still, in the early days, I have occasionally written something like this:

```
public class CalculatorSpecification
{
  //... some Statements here

  //oops... forgot to insert the attribute!
  public void ShouldDisplayZeroWhenResetIsPerformed()
  {
    //...
  }
}
```

As you can see, the `[Fact]` attribute is missing, which means this Statement will not be executed. This has happened not only because of not using code generators – sometimes – to create a new Statement – it made sense to copy-paste an existing Statement, change the name and few lines of code[7]. I didn't always remember to include the `[Fact]` attribute in the copied source code. The compiler was not complaining as well.

[7] I know copy-pasting code is considered harmful and we shouldn't be doing that. When writing unit-level Statements, I make some exceptions to that rule. This will be explained in further parts.

The reason I didn't see my mistake was that I was running more than one test at a time – when I got a green bar (i.e. all Statements proven true), I assumed that the Statement I just wrote works as well. It was unattractive for me to search for each new Statement in the list and make sure it's there. The more important reason, however, was that the absence of the `[Fact]` attribute did not disturb my workflow: test – all pass, test – all pass, test – all pass... In other words, my process did not give me any feedback that I made a mistake. So, in such a case, what I end up with is a Statement that not only will never be proven false – **it won't be evaluated at all**.

How does treating tests as Statements and evaluating them before making them true help here? The fundamental difference is that the workflow of TDD is: test – fail – pass, test – fail – pass, test – fail – pass... In other words, we expect each Statement to be proven false at least once. So every time we miss the "fail" stage, we get feedback from our process that something suspicious is happening. This allows us to investigate and fix the problem if necessary.

2. Misplacing Statement setup

Ok, this may sound even funnier, but it happened to me a couple of times as well, so I assume it may happen to you one day, especially if you are in a hurry.

Consider the following toy example: we want to validate a simple data structure that models a frame of data that can arrive via a network. The structure looks like this:

```
public class Frame
{
  public int timeSlot;
}
```

and we need to write a Specification for a `Validation` class that accepts a `Frame` object as an argument and checks whether the time slot (whatever it is) inside it is correct. The correctness is determined by comparing the time slot to a maximum allowed value specified in a constant called `TimeSlot.MaxAllowed` (so it's a constant defined in a `TimeSlot` class). If the frame time slot is higher than the maximum allowed, it is assumed incorrect and the validation should return `false`. Otherwise, `true` should be returned.

Let's take a look at the following Statement which specifies that setting a value higher than allowed to a field of a `frame` should make the validation fail:

```
[Fact]
public void ShouldRecognizeTimeSlotAboveMaximumAllowedAsInvalid()
{
  var frame = new Frame();
  var validation = new Validation();
  var timeSlotAboveMaximumAllowed = TimeSlot.MaxAllowed + 1;
  var result = validation.PerformForTimeSlotIn(frame);
  frame.timeSlot = timeSlotAboveMaximumAllowed;
  Assert.False(result);
}
```

Note how the method `PerformForTimeSlotIn()`, which triggers the specified behavior, is accidentally called *before* value of `timeSlotAboveMaximumAllowed` is set up and thus, this value is not taken into account at the moment when the validation is executed. If, for example, we make a mistake in the implementation of the `Validation` class so that it returns `false` for values below the maximum and not above, such mistake may go unnoticed, because the Statement will always be true.

Again, this is a toy example – I just used it as an illustration of something that can happen when dealing with more complex cases.

3. Using static data inside the production code

Once in a while, we have to jump in and add some new Statements to an existing Specification and some logic to the class it describes. Let's assume that the class and its Specification were written by someone else than us. Imagine the code we are talking about is a wrapper around our product XML configuration file. We decide to write our Statements *after* applying the changes ("well", we may say, "we're all protected by the Specification that is already in place, so we can make our change without the risk of accidentally breaking existing functionality, and then just test our changes and it's all good…").

We start coding… done. Now we start writing this new Statement that describes the functionality we just added. After examining the Specification class, we can see that it has a member field like this:

```
public class XmlConfigurationSpecification
{
  XmlConfiguration config = new XmlConfiguration(xmlFixtureString);

  //...
```

What it does is it sets up an object used by every Statement. So, each Statement uses a `config` object initialized with the same `xmlConfiguration` string value. Another quick examination leads us to discover the following content of the `xmlFixtureString`:

```
<config>
  <section name="General Settings">
    <subsection name="Network Related">
      <parameter name="IP">192.168.3.2</parameter>
      <parameter name="Port">9000</parameter>
      <parameter name="Protocol">AHJ-112</parameter>
    </subsection>
      <subsection name="User Related">
      <parameter name="login">Johnny</parameter>
      <parameter name="Role">Admin</parameter>
      <parameter name="Password Expiry (days)">30</parameter>
    </subsection>
    <!-- and so on and on and on...-->
```

```
14    </section>
15  </config>
```

The string is already pretty large and messy since it contains all information that is required by the existing Statements. Let's assume we need to write tests for a little corner case that does not need all this crap inside this string. So, we decide to start afresh and create a separate object of the `XmlConfiguration` class with your own, minimal string. Our Statement begins like this:

```
1  string customFixture = CreateMyOwnFixtureForThisTestOnly();
2  var configuration = new XmlConfiguration(customFixture);
3  ...
```

And goes on with the scenario. When we execute it, it passes – cool... not. Ok, what's wrong with this? At first sight, everything's OK, until we read the source code of `XmlConfiguration` class carefully. Inside, we can see, how the XML string is stored:

```
1  private static string xmlText; //note the static keyword!
```

It's a static field, which means that its value is retained between instances. What the...? Well, well, here's what happened: the author of this class applied a small optimization. He thought: "In this app, the configuration is only modified by members of the support staff and to do it, they have to shut down the system, so, there is no need to read the XML file every time an XmlConfiguration object is created. I can save some CPU cycles and I/O operations by reading it only once when the first object is created. Later objects will just use the same XML!". Good for him, not so good for us. Why? Because, depending on the order in which the Statements are evaluated, either the original XML string will be used for all Statements or your custom one! Thus the Statements in this Specification may pass or fail for the wrong reason – because they accidentally use the wrong XML.

Starting development from a Statement that we expect to fail may help when such a Statement passes even though the behavior it describes is not implemented yet.

"Test-After" often ends up as "Test-Never"

Consider again the question I already asked in this chapter: did you ever have to write requirements or a design document for something that you already implemented? Was it fun? Was it valuable? Was it creative? As for me, my answer to these questions is *no*. I observed that the same answer applied to formulating my executable Specification. By observing myself and other developers, I concluded that after we've written the code, we have little motivation to specify what we wrote – some of the pieces of code "we can just see are correct", other pieces "we already saw working" when we compiled and deployed our changes and ran a few manual checks... The design is ready... Specification? Maybe next time... Thus, the Specification may never get to be written at all and if it is written, I often find that it covers most of the main flow of the program, but lacks some Statements saying what should happen in case of errors, etc.

Another reason for ending up not writing the Specification might be time pressure, especially in teams that are not yet mature or not have very strong professional ethics. Many times, I have seen people reacting to pressure by dropping everything besides writing the code that directly implements a feature. Among the things that get dropped are design, requirements and tests. And learning as well. I have seen many times teams that, when under pressure, stopped experimenting and learning and reverted to old "safe" behaviors in a mindset of "saving a sinking ship" and "hoping for the best". As in such situations, I've seen pressure raise as the project approached its deadline or milestone, leaving Specification until the end means that it's very likely to get dropped, especially in the case when the changes are (to a degree) tested manually later anyway.

On the other hand, when doing TDD (as we will see in the coming chapters) our Specification grows together with the production code, so there is much less temptation to drop it entirely. Moreover, In TDD, a written Specification Statement is not an addition to the code, but rather *a reason* to write the code. Creating an executable Specification becomes an indispensable part of implementing a feature.

"Test-After" often leads to design rework

I like reading and watching Uncle Bob (Robert C. Martin). One day I was listening to his keynote at Ruby Midwest 2011, called Architecture The Lost Years[8]. At the end, Robert made some digressions, one of them about TDD. He said that writing tests after the code is not TDD and instead called it "a waste of time".

My initial thought was that the comment was maybe a bit too exaggerated and only about missing all the benefits that starting with a false Statement brings me: the ability to see the Statement fail, the ability to do a clean-sheet analysis, etc. However, now I feel that there's much more to it, thanks to something I learned from Amir Kolsky and Scott Bain – to be able to write a maintainable Specification for a piece of code, the code must have a high level of **testability**. We will talk about this quality in part 2 of this book, but for now let's assume the following simplified definition: the higher testability of a piece of code (e.g. a class), the easier it is to write a Statement for its behavior.

Now, where's the waste in writing the Specification after the code is written? To find out, let's compare the Statement-first and code-first approaches. In the Statement-first workflow for new (non-legacy) code, my workflow and approach to testability usually look like this:

1. Write a Statement that is false to start with (during this step, detect and correct testability issues even before the production code is written).
2. Write code to make the Statement true.

And here's what I often see programmers do when they write the code first (extra steps marked with **strong text**):

1. Write some production code without considering how it will be tested (after this step, the testability is often suboptimal as it's usually not being considered at this point).

[8] http://confreaks.tv/videos/rubymidwest2011-keynote-architecture-the-lost-years

2. **Start writing a unit test** (this might not seem like an extra step, since it's also present in the previous approach, but once you reach the step 5, you'll know what I mean).
3. **Notice that unit testing the code we wrote is cumbersome and unsustainable and the tests become looking messy as they try to work around the testability issues.**
4. **Decide to improve testability by restructuring the code, e.g. to be able to isolate objects and use techniques such as mock objects.**
5. Write unit tests (this time it should be easier as the testability of the tested is better).

What is the equivalent of the marked steps in the Statement-first approach? There is none! Doing these things is a waste of time! Sadly, this is a waste I encounter a lot.

Summary

In this chapter, I tried to show you that the choice of *when* we write our Specification often makes a huge difference and that there are numerous benefits of starting with a Statement. When we consider the Specification as what it really is – not only as a suite of tests that check runtime correctness – then the Statement-first approach becomes less awkward and less counter-intuitive.

Practicing what we have already learned

> And now, a taste of things to come!
>
> – Shang Tsung, Mortal Kombat The Movie

The above quote took place just before a fighting scene[9] in which a nameless warrior jumped at Sub-Zero only to be frozen and broken into multiple pieces upon hitting the wall. The scene was not spectacular in terms of fighting technique or length. Also, the nameless guy didn't even try hard – the only thing he did was to jump only to be hit by a freezing ball, which, by the way, he actually could see coming. It looked a lot like the fight was set up only to showcase Sub-Zero's freezing ability. Guess what? In this chapter, we're going to do roughly the same thing – set up a fake, easy scenario just to showcase some of the basic TDD elements!

The previous chapter was filled with a lot of theory and philosophy, don't you think? I hope you didn't fall asleep while reading it. To tell you the truth, we need to grasp much more theory until we can write real-world applications using TDD. To compensate for this somehow, I propose we take a side trip from the trail and try what we already learned in a quick and easy example. As we go through the example, you might wonder how on earth could you possibly write real applications the way we will write our simple program. Don't worry, I will not show you all the tricks yet, so treat it as a "taste of things to come". In other words, the example will be as close to real-world problems as the fight between Sub-Zero and the nameless ninja was to real martial arts fight, but will show you some of the elements of the TDD process.

Let me tell you a story

Meet Johnny and Benjamin, two developers from Buthig Company. Johnny is quite fluent in programming and Test-Driven Development, while Benjamin is an intern under Johnny's mentorship and is eager to learn TDD. They are on their way to their customer, Jane, who requested their presence as she wants them to write a small program for her. Along with them, we will see how they interact with the customer and how Benjamin tries to understand the basics of TDD. Like you, Benjamin is a novice so his questions may reflect yours. However, if you find anything explained in not enough detail, do not worry – in the next chapters, we will be expanding on this material.

Act 1: The Car

Johnny: How do you feel about your first assignment?

[9] https://www.youtube.com/watch?v=b0vhGEGJC8g

Benjamin: I am pretty excited! I hope I can learn some of the TDD stuff you promised to teach me.

Johnny: Not only TDD, but we are also gonna use some of the practices associated with a process called Acceptance Test-Driven Development, albeit in a simplified form.

Benjamin: Acceptance Test-Driven Development? What is that?

Johnny: While TDD is usually referred to as a development technique, Acceptance Test-Driven Development (ATDD) is something more of a collaboration method. Both ATDD and TDD have a bit of analysis in them and work very well together as both use the same underlying principles, just on different levels. We will need only a small subset of what ATDD has to offer, so don't get over-excited.

Benjamin: Sure. Who's our customer?

Johnny: Her name's Jane. She runs a small shop nearby and wants us to write an application for her new mobile. You'll get the chance to meet her in a minute as we're almost there.

Act 2: The Customer's Site

Johnny: Hi, Jane, how are you?

Jane: Thanks, I'm fine, how about you?

Johnny: Me too, thanks. Benjamin, this is Jane, our customer. Jane, this is Benjamin, we'll work together on the task you have for us.

Benjamin: Hi, nice to meet you.

Jane: Hello, nice to meet you too.

Johnny: So, can you tell us a bit about the software you need us to write?

Jane: Sure. Recently, I bought a new smartphone as a replacement for my old one. The thing is, I am really used to the calculator application that ran on my previous phone and I cannot find a counterpart for my current device.

Benjamin: Can't you just use another calculator app? There are probably plenty of them available to download from the web.

Jane: That's right. I checked them all and none has the same behavior as the one I have used for my tax calculations. You see, this app was like a right hand to me and it had some nice shortcuts that made my life easier.

Johnny: So you want us to reproduce the application to run on your new device?

Jane: Exactly.

Johnny: Are you aware that apart from the fancy features that you were using we will have to allocate some effort to implement the basics that all the calculators have?

Jane: Sure, I am OK with that. I got used to my calculator application so much that if I use something else for more than a few months, I will have to pay a psychotherapist instead of you

guys. Apart from that, writing a calculator app seems like an easy task in my mind, so the cost isn't going to be overwhelming, right?

Johnny: I think I get it. Let's get it going then. We will be implementing the functionality incrementally, starting with the most essential features. Which feature of the calculator would you consider the most essential?

Jane: That would be the addition of numbers, I guess.

Johnny: Ok, that will be our target for the first iteration. After the iteration, we will deliver this part of the functionality for you to try out and give us some feedback. However, before we can even deliver the addition feature, we will have to implement displaying digits on the screen as you enter them. Is that correct?

Jane: Yes, I need the display stuff to work as well – it's a prerequisite for other features, so…

Johnny: Ok then, this is a simple functionality, so let me suggest some user stories as I understand what you already said and you will correct me where I am wrong. Here we go:

1. **In order to** know that the calculator is turned on, **As a** taxpayer **I want** to see "0" on the screen as soon as I turn it on.
2. **In order to** see what numbers I am currently operating on, **As a** taxpayer, **I want** the calculator to display the values I enter
3. **In order to** calculate the sum of my different incomes, **As a** taxpayer **I want** the calculator to enable addition of multiple numbers

What do you think?

Jane: The stories pretty much reflect what I want for the first iteration. I don't think I have any corrections to make.

Johnny: Now we'll take each story and collect some examples of how it should work.

Benjamin: Johnny, don't you think it is obvious enough to proceed with implementation straight away?

Johnny: Trust me, Benjamin, if there is one word I fear most in communication, it is "obvious". Miscommunication happens most often around things that people consider obvious, simply because other people do not.

Jane: Ok, I'm in. What do I do?

Johnny: Let's go through the stories one by one and see if we can find some key examples of how the features should work. The first story is…

In order to know that the calculator is turned on, As a taxpayer I want to see "0" on the screen as soon as I turn it on.

Jane: I don't think there's much to talk about. If you display "0", I will be happy. That's all.

Johnny: Let's write this example down using a table:

key sequence	Displayed output	Notes
N/A	0	Initial displayed value

Benjamin: That makes me wonder… what should happen when I press "0" again at this stage?

Johnny: Good catch, that's what these examples are for – they make our thinking concrete. As Ken Pugh says[10]: "Often the complete understanding of a concept does not occur until someone tries to use the concept". Normally, we would put the "pressing zero multiple times" example on a TODO list and leave it for later, because it's a part of a different story. However, it looks like we're done with the current story, so let's move straight ahead. The next story is about displaying entered digits. How about it, Jane?

Jane: Agree.

Johnny: Benjamin?

Benjamin: Yes, go ahead.

In order to see what numbers I am currently operating on, As a taxpayer, I want the calculator to display the values I enter

Johnny: Let's begin with the case raised by Benjamin. What should happen when I input "0" multiple times after I only have "0" on the display?

Jane: A single "0" should be displayed, no matter how many times I press "0".

Johnny: Do you mean this?

key sequence	Displayed output	Notes
0,0,0	0	Zero is a special case – it is displayed only once

Jane: That's right. Other than this, the digits should just show on the screen, like this:

key sequence	Displayed output	Notes
1,2,3	123	Entered digits are displayed

Benjamin: How about this:

key sequence	Displayed output	Notes
1,2,3,4,5,6,7,1,2,3,4,5,6	1234567123456?	Entered digits are displayed?

Jane: Actually, no. My old calculator app has a limit of six digits that I can enter, so it should be:

key sequence	Displayed output	Notes
1,2,3,4,5,6,7,1,2,3,4,5,6	123456	Display limited to six digits

Johnny: Another good catch, Benjamin!

Benjamin: I think I'm beginning to understand why you like working with examples!

Johnny: Good. Is there anything else, Jane?

[10] K. Pugh, Prefactoring, O'Reilly Media, 2005

Jane: No, that's pretty much it. Let's start working on another story.

In order to calculate the sum of my different incomes, As a taxpayer I want the calculator to enable addition of multiple numbers

Johnny: Is the following scenario the only one we have to support?

key sequence	Displayed output	Notes
2,+,3,+,4,=	9	Simple addition of numbers

Jane: This scenario is correct, however, there is also a case when I start with "+" without inputting any number before. This should be treated as adding to zero:

key sequence	Displayed output	Notes
+,1,=	1	Addition shortcut – treated as 0+1

Benjamin: How about when the output is a number longer than six digits limit? Is it OK that we truncate it like this?

key sequence	Displayed output	Notes
9,9,9,9,9,9,+,9,9,9,9,9,9,=	199999	Our display is limited to six digits only

Jane: Sure, I don't mind. I don't add such big numbers anyway.

Johnny: There is still one question we missed. Let's say that I input a number, then press "+" and then another number without asking for result with "=". What should I see?

Jane: Every time you press "+", the calculator should consider entering current number finished and overwrite it as soon as you press any other digit:

key sequence	Displayed output	Notes
2,+,3	3	Digits entered after + operator are treated as digits of a new number, the previous one is stored

Jane: Oh, and asking for the result just after the calculator is turned on should result in "0".

key sequence	Displayed output	Notes
=	0	Result key in itself does nothing

Johnny: Let's sum up our discoveries:

key sequence	Displayed output	Notes
N/A	0	Initial displayed value
1,2,3	123	Entered digits are displayed
0,0,0	0	Zero is a special case – it is displayed only once
1,2,3,4,5,6,7	123456	Our display is limited to six digits only
2,+,3	3	Digits entered after + operator are treated as digits of a new number, the previous one is stored
=	0	Result key in itself does nothing
+,1,=	1	Addition shortcut – treated as 0+1
2,+,3,+,4,=	9	Simple addition of numbers
9,9,9,9,9,9,+,9,9,9,9,9,9,=	199999	Our display is limited to six digits only

Johnny: The limiting of digits displayed looks like a whole new feature, so I suggest we add it to the backlog and do it in another sprint. In this sprint, we will not handle such a situation at all. How about that, Jane?

Jane: Fine with me. It looks like a lot of work. Nice that we discovered it up-front. For me, the limiting capability seemed so obvious that I didn't even think it would be worth mentioning.

Johnny: See? That's why I don't like the word "obvious". Jane, we will get back to you if any more questions arise. For now, I think we know enough to implement these three stories for you.

Jane: good luck!

Act 3: Test-Driven Development

Benjamin: Wow, that was cool. Was that Acceptance Test-Driven Development?

Johnny: In a greatly simplified version, yes. The reason I took you with me was to show you the similarities between working with the customer the way we did and working with the code using TDD process. They are both applying the same set of principles, just on different levels.

Benjamin: I'm dying to see it with my own eyes. Shall we start?

Johnny: Sure. If we followed the ATDD process, we would start writing what we call acceptance-level specification. In our case, however, a unit-level specification will be enough. Let's take the first example:

Statement 1: Calculator should display 0 on creation

key sequence	Displayed output	Notes
N/A	0	Initial displayed value

Johnny: Benjamin, try to write the first Statement.

Benjamin: Oh boy, I don't know how to start.

Johnny: Start by writing the statement in plain English. What should the calculator do?

Benjamin: It should display "0" when I turn the application on.

Johnny: In our case, "turning on" is creating a calculator. Let's write it down as a method name:

```
1  public class CalculatorSpecification
2  {
3
4  [Fact] public void
5  ShouldDisplay0WhenCreated()
6  {
7
8  }
9
10 }
```

Benjamin: Why is the name of the class `CalculatorSpecification` and the name of the method `ShouldDisplay0WhenCreated`?

Johnny: It is a naming convention. There are many others, but this is the one that I like. In this convention, the rule is that when you take the name of the class without the `Specification` part followed by the name of the method, it should form a legit sentence. For instance, if I apply it to what we wrote, it would make a sentence: "Calculator should display `0` when created".

Benjamin: Ah, I see now. So it's a statement of behavior, isn't it?

Johnny: That's right. Now, the second trick I can sell to you is that if you don't know what code to start your Statement with, start with the expected result. In our case, we are expecting that the behavior will end up as displaying "0", right? So let's just write it in the form of an assertion.

Benjamin: You mean something like this?

```
1  public class CalculatorSpecification
2  {
3
4  [Fact] public void
5  ShouldDisplay0WhenCreated()
6  {
7   Assert.Equal("0", displayedResult);
8  }
9
10 }
```

Johnny: Precisely.

Benjamin: But that doesn't even compile. What use is it?

Johnny: The code not compiling is the feedback that you needed to proceed. While before you didn't know where to start, now you have a clear goal – make this code compile. Firstly, where do you get the displayed value from?

Benjamin: From the calculator display, of course!

Johnny: Then write down how you get the value from the display.

Benjamin: Like how?

Johnny: Like this:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 var displayedResult = calculator.Display();

 Assert.Equal("0", displayedResult);
}

}
```

Benjamin: I see. Now the calculator is not created anywhere. I need to create it somewhere now or it will not compile – this is how I know that it's my next step. Is this how it works?

Johnny: Yes, you are catching on quickly.

Benjamin: Ok then, here goes:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 var calculator = new Calculator();

 var displayedResult = calculator.Display();

 Assert.Equal("0", displayedResult);
}

}
```

Johnny: Bravo!

Benjamin: The code doesn't compile yet, because I don't have the `Calculator` class defined at all...

Johnny: Sounds like a good reason to create it.

Benjamin: OK.

```
public class Calculator
{
}
```

Benjamin: Looks like the `Display()` method is missing too. I'll add it.

```
public class Calculator
{
  public string Display()
  {
    return "0";
  }
}
```

Johnny: Hey hey, not so fast!

Benjamin: What?

Johnny: You already provided an implementation of `Display()` that will make our current Statement true. Remember its name? `ShouldDisplay0WhenCreated` – and that's exactly what the code you wrote does. Before we arrive at this point, let's make sure this Statement can ever be evaluated as false. You won't achieve this by providing a correct implementation out of the box. So for now, let's change it to this:

```
public class Calculator
{
  public string Display()
  {
    return "Once upon a time in Africa";
  }
}
```

Johnny: Look, now we can run the Specification and watch that Statement evaluate to false, because it expects "0", but gets "Once upon a time in Africa".

Benjamin: Running... Ok, it is false. By the way, do you always use such silly values to make Statements false?

Johnny: Hahaha, no, I just did it to emphasize the point. Normally, I would write `return "";` or something similarly simple. Now we can evaluate the Statement and see it turn false. Hence, we're sure that we have not yet implemented what is required for the Statement to be true.

Benjamin: I think I get it. For now, the Statement shows that we do not have something we need and gives us a reason to add this "thing". When we do so, this Statement will show that we do have what we need. So what do we do now?

Johnny: Write the simplest thing that makes this Statement true.

Benjamin: like this?

```
public class Calculator
{
  public string Display()
  {
    return "0";
  }
}
```

Johnny: Yes.

Benjamin: But that is not a real implementation. What is the value behind putting in a hardcoded string? The final implementation is not going to be like this for sure!

Johnny: You're right. The final implementation is most probably going to be different. What we did, however, is still valuable because:

1. You're one step closer to implementing the final solution
2. This feeling that this is not the final implementation points you towards writing more Statements. When there are enough Statements to make your implementation complete, it usually means that you have a complete Specification of class behaviors as well.
3. If you treat making every Statement true as an achievement, this practice allows you to evolve your code without losing what you already achieved. If by accident you break any of the behaviors you've already implemented, the Specification is going to tell you because one of the existing Statements that were previously true will turn false. You can then either fix it or undo your changes using version control and start over from the point where all existing Statements were true.

Benjamin: Ok, so it looks like there are some benefits after all. Still, I'll have to get used to this way of working.

Johnny: Don't worry, this approach is an important part of TDD, so you will grasp it in no time. Now, before we go ahead with the next Statement, let's look at what we already achieved. First, we wrote a Statement that turned out false. Then, we wrote just enough code to make the Statement true. Time for a step called Refactoring. In this step, we will take a look at the Statement and the code and remove duplication. Can you see what is duplicated between the Statement and the code?

Benjamin: both of them contain the literal "0". The Statement has it here:

```
Assert.Equal("0", displayedResult);
```

and the implementation here:

```
return "0";
```

Johnny: Good, let's eliminate this duplication by introducing a constant called `InitialValue`. The Statement will now look like this:

```
[Fact] public void
ShouldDisplayInitialValueWhenCreated()
{
 var calculator = new Calculator();

 var displayedResult = calculator.Display();

 Assert.Equal(Calculator.InitialValue, displayedResult);
}
```

and the implementation:

```
public class Calculator
{
  public const string InitialValue = "0";
  public string Display()
  {
    return InitialValue;
  }
}
```

Benjamin: The code looks better and having the "0" constant in one place will make it more maintainable. However, I think the Statement in its current form is weaker than before. I mean, we can change the `InitialValue` to anything and the Statement will still be true since it does not state that this constant needs to have a value of "0".

Johnny: That's right. We need to add it to our TODO list to handle this case. Can you write it down?

Benjamin: Sure. I will write it as "TODO: 0 should be used as an initial value."

Johnny: Ok. We should handle it now, especially since it's part of the story we are currently implementing, but I will leave it for later just to show you the power of TODO list in TDD – whatever is on the list, we can forget and get back to when we have nothing better to do. Our next item from the list is this:

Statement 2: Calculator should display entered digits

key sequence	Displayed output	Notes
1,2,3	123	Entered digits are displayed

Johnny: Benjamin, can you come up with a Statement for this behavior?

Benjamin: I'll try. Here goes:

```
[Fact] public void
ShouldDisplayEnteredDigits()
{
  var calculator = new Calculator();

  calculator.Enter(1);
  calculator.Enter(2);
  calculator.Enter(3);
  var displayedValue = calculator.Display();

  Assert.Equal("123", displayedValue);
}
```

Johnny: I see that you're learning fast. You got the parts about naming and structuring a Statement right. There's one thing we will have to work on here though.

Benjamin: What is it?

Johnny: When we talked to Jane, we used examples with real values. These real values were extremely helpful in pinning down the corner cases and uncovering missing scenarios. They were easier to imagine as well, so they were a perfect suit for conversation. If we were automating these examples on the acceptance level, we would use those real values as well. When we write unit-level Statements, however, we use a different technique to get this kind of specification more abstract. First of all, let me enumerate the weaknesses of the approach you just used:

1. Making a method `Enter()` accept an integer value suggests that one can enter more than one digit at once, e.g. `calculator.Enter(123)`, which is not what we want. We could detect such cases and throw exceptions if the value is outside the 0-9 range, but there are better ways when we know we will only be supporting ten digits (0,1,2,3,4,5,6,7,8,9).
2. The Statement does not clearly show the relationship between input and output. Of course, in this simple case, it's pretty self-evident that the sum is a concatenation of entered digits. In general case, however, we don't want anyone reading our Specification in the future to have to guess such things.
3. The name of the Statement suggests that what you wrote is true for any value, while in reality, it's true only for digits other than "0" since the behavior for "0" is different (no matter how many times we enter "0", the result is just "0"). There are some good ways to communicate it.

Hence, I propose the following:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  string.Format("{0}{1}{2}",
   (int)nonZeroDigit,
   (int)anyDigit1,
   (int)anyDigit2
  ),
  calculator.Display()
 );
}
```

Benjamin: Johnny, I'm lost! Can you explain what's going on here?

Johnny: Sure, what do you want to know?

Benjamin: For instance, what is this `DigitKeys` type doing here?

Johnny: It is supposed to be an enumeration (note that it does not exist yet, we just assume that we have it) to hold all the possible digits a user can enter, which are from the range of 0-9. This is to ensure that the user will not write `calculator.Enter(123)`. Instead of allowing our users to enter any number and then detecting errors, we are giving them a choice from among only the valid values.

Benjamin: Now I get it. So how about the `Any.OtherThan()` and `Any.Of()`? What do they do?

Johnny: They are methods from a small utility library I'm using when writing unit-level Specifications. `Any.OtherThan()` returns any value from enumeration besides the one passed as an argument. Hence, the call `Any.OtherThan(DigitKeys.Zero)` means "any of the values contained in DigitKeys enumeration, but not DigitKeys.Zero".

The `Any.Of()` is simpler – it just returns any value in an enumeration.

Note that by saying:

```
var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
var anyDigit1 = Any.Of<DigitKeys>();
var anyDigit2 = Any.Of<DigitKeys>();
```

I specify explicitly, that the first value entered must be other than "0" and that this constraint does not apply to the second digit, the third one and so on.

By the way, this technique of using generated values instead of literals has its own principles and constraints which you have to know to use it effectively. Let's leave this topic for now and I promise I'll give you a detailed lecture on it later. Agreed?

Benjamin: You better do, because for now, I feel a bit uneasy with generating the values – it seems like the Statement we are writing is getting less deterministic this way. The last question – what about those weird comments you put in the code? `GIVEN`? `WHEN`? `THEN`?

Johnny: Yes, this is a convention that I use, not only in writing, but in thinking as well. I like to think about every behavior in terms of three elements: assumptions (given), trigger (when) and expected result (then). Using the words, we can summarize the Statement we are writing in the following way: "**Given** a calculator, **when** I enter some digits, the first one being non-zero, **then** they should all be displayed in the order they were entered". This is also something that I will tell you more about later.

Benjamin: Sure, for now, I need just enough detail to be able to keep going – we can talk about the principles, pros, and cons later. By the way, the following sequence of casts looks a little bit ugly:

```
string.Format("{0}{1}{2}",
 (int)nonZeroDigit,
 (int)anyDigit1,
 (int)anyDigit2
)
```

Johnny: We will get back to it and make it "smarter" in a second after we make this statement true. For now, we need something obvious. Something we know works. Let's evaluate this Statement. What is the result?

Benjamin: Failed: expected "351", but was "0".

Johnny: Good, now let's write some code to make this Statement true. First, we're going to introduce an enumeration of digits. This enum will contain the digit we use in the Statement (which is `DigitKeys.Zero`) and some bogus values:

```
public enum DigitKeys
{
 Zero = 0,
 TODO1, //TODO - bogus value for now
 TODO2, //TODO - bogus value for now
 TODO3, //TODO - bogus value for now
 TODO4, //TODO - bogus value for now
}
```

Benjamin: What's with all those bogus values? Shouldn't we correctly define values for all the digits we support?

Johnny: Nope, not yet. We still don't have a Statement that would say what digits are supported and which would make us add them, right?

Benjamin: You say you need a Statement for an element to be in an enum?

Johnny: This is a specification we are writing, remember? It should say somewhere which digits we support, shouldn't it?

Benjamin: It's difficult to agree with, I mean, I can see the values in the enum, should I test for something when there's not complexity involved?

Johnny: Again, we're not only testing but also we're specifying. I will try to give you more arguments later. For now, just bear with me and note that when we get to specify the enum elements, adding such a Statement will be almost effortless.

Benjamin: OK.

Johnny: Now for the implementation. Just to remind you – what we have so far looks like this:

```
public class Calculator
{
 public const string InitialValue = "0";
 public string Display()
 {
  return InitialValue;
 }
}
```

This does not support displaying multiple digits yet (as we just proved, because the Statement saying they are supported turned out false). So let's change the code to handle this case:

```
public class Calculator
{
 public const string InitialValue = "0";
 private int _result = 0;

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
 }

 public string Display()
 {
  return _result.ToString();
 }
}
```

Johnny: Now the Statement is true so we can go back to it and make it a little bit prettier. Let's take a second look at it:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  string.Format("{0}{1}{2}",
   (int)nonZeroDigit,
   (int)anyDigit1,
   (int)anyDigit2
  ),
  calculator.Display()
 );
}
```

Johnny: Remember you said that you don't like the part where `string.Format()` is used?

Benjamin: Yeah, it seems a bit unreadable.

Johnny: Let's extract this part into a utility method and make it more general – we will need a way of constructing expected displayed output in many of our future Statements. Here is my go at this helper method:

```
string StringConsistingOf(params DigitKeys[] digits)
{
 var result = string.Empty;

 foreach(var digit in digits)
 {
  result += (int)digit;
 }
 return result;
}
```

Note that this is more general as it supports any number of parameters. And the Statement after this extraction looks like this:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.OtherThan(DigitKeys.Zero);
 var anyDigit1 = Any.Of<DigitKeys>();
 var anyDigit2 = Any.Of<DigitKeys>();

 //WHEN
 calculator.Enter(nonZeroDigit);
 calculator.Enter(anyDigit1);
 calculator.Enter(anyDigit2);

 //THEN
 Assert.Equal(
  StringConsistingOf(nonZeroDigit, anyDigit1, anyDigit2),
  calculator.Display()
 );
}
```

Benjamin: Looks better to me. The Statement is still evaluated as true, which means we got it right, didn't we?

Johnny: Not exactly. With moves such as this one, I like to be extra careful and double-check whether the Statement still describes the behavior accurately. To make sure that's still the case, let's comment out the body of the `Enter()` method and see if this Statement would still turn out false:

```
public void Enter(DigitKeys digit)
{
 //_result *= 10;
 //_result += (int)digit;
}
```

Benjamin: Running... Ok, it is false now. Expected "243", got "0".

Johnny: Good, now we're pretty sure it works OK. Let's uncomment the lines we just commented out and move forward.

Benjamin: But wait, there is one thing that troubles me.

Johnny: I think I know – I was wondering if you'd catch it. Go ahead.

Benjamin: What troubles me is these two lines:

```
public const string InitialValue = "0";
private int _result = 0;
```

Isn't this a duplication? I mean, it's not exactly a code duplication, but in both lines, the value of `0` has the same intent. Shouldn't we remove this duplication somehow?

Johnny: Yes, let's do it. My preference would be to change the `InitialValue` to `int` instead of string and use that. But I can't do it in a single step as I have the two Statements depending on `InitialValue` being a string. if I just changed the type to `int`, I would break those tests as well as the implementation and I always want to be fixing one thing at a time.

Benjamin: So what do we do?

Johnny: Well, my first step would be to go to the Statements that use `InitialValue` and use a `ToString()` method there. For example, in the Statement `ShouldDisplayInitialValueWhenCreated()`, I have an assertion:

```
Assert.Equal(Calculator.InitialValue, displayedResult);
```

which I can change to:

```
Assert.Equal(Calculator.InitialValue.ToString(), displayedResult);
```

Benjamin: But calling `ToString()` on a `string` just returns the same value, what's the point?

Johnny: The point is to make the type of whatever's on the left side of `.ToString()` irrelevant. Then I will be able to change that type without breaking the Statement. The new implementation of `Calculator` class will look like this:

```
public class Calculator
{
 public const int InitialValue = 0;
 private int _result = InitialValue;

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
 }

 public string Display()
 {
  return _result.ToString();
 }
}
```

Benjamin: Oh, I see. And the Statements are still evaluated as true.

Johnny: Yes. Shall we take on another Statement?

Statement 3: Calculator should display only one zero digit if it is the only entered digit even if it is entered multiple times

Johnny: Benjamin, this should be easy for you, so go ahead and try it. It is really a variation of the previous Statement.

Benjamin: Let me try... ok, here it is:

```
[Fact] public void
ShouldDisplayOnlyOneZeroDigitWhenItIsTheOnlyEnteredDigitEvenIfItIsEnteredMultiple\
Times()
{
 //GIVEN
 var calculator = new Calculator();

 //WHEN
 calculator.Enter(DigitKeys.Zero);
 calculator.Enter(DigitKeys.Zero);
 calculator.Enter(DigitKeys.Zero);

 //THEN
 Assert.Equal(
  StringConsistingOf(DigitKeys.Zero),
  calculator.Display()
 );
}
```

Johnny: Good, you're learning fast! Let's evaluate this Statement.

Benjamin: It seems that our current code already fulfills the Statement. Should I try to comment-out some code to make sure this Statement can fail just like you did in the previous Statement?

Johnny: That would be a wise thing to do. When a Statement turns out true without requiring you to change any production code, it's always suspicious. Just like you said, we have to change production code for a second to force this Statement to become false, then undo this modification to make it true again. This isn't as obvious as previously, so let me do it. I will mark all the added lines with `//+` comment so that you can see them easily:

```
1  public class Calculator
2  {
3   public const int InitialValue = 0;
4   private int _result = InitialValue;
5   string _fakeResult = "0"; //+
6
7   public void Enter(DigitKeys digit)
8   {
9    _result *= 10;
10   _result += (int)digit;
11   if(digit == DigitKeys.Zero) //+
12   {  //+
13    _fakeResult += "0";  //+
14   } //+
15  }
16
17  public string Display()
18  {
19   if(_result == 0)  //+
20   {  //+
21    return _fakeResult;  //+
22   }  //+
23   return _result.ToString();
24  }
25 }
```

Benjamin: Wow, looks like a lot of code just to make the Statement false! Is it worth the hassle? We will undo this whole change in a second anyway…

Johnny: Depends on how confident you want to feel. I would say that it's usually worth it – at least you know that you got everything right. It might seem like a lot of work, but it only took me about a minute to add this code and imagine you got it wrong and had to debug it on a production environment. Now *that* would be a waste of time.

Benjamin: Ok, I think I get it. Since we saw this Statement turn false, I will undo this change to make it true again.

Johnny: Sure.

Epilogue

Time to leave Johnny and Benjamin, at least for now. I planned to make this chapter longer, and cover all the other operations, but I fear this would make it boring. You should have a feel of how the TDD cycle looks like, especially since Johnny and Benjamin had a lot of conversations on many other topics in the meantime. I will be revisiting these topics later in the book. For now, if you felt lost or unconvinced on any of the topics mentioned by Johnny, don't worry – I don't expect you to be proficient with any of the techniques shown in this chapter just yet. The time will come for that.

Sorting out the bits

In the last chapter, there has been a lively conversation between Johnny and Benjamin. Even in such a short session, Benjamin, as a TDD novice, had a lot of questions and a lot of things he needed to be sorted out. We will pick up all those questions that were not already answered and try to answer in the coming chapters. Here are the questions:

- How to name a Statement?
- How to start writing a Statement?
- How is TDD about analysis and what does this "GIVEN-WHEN-THEN" mean?
- What exactly is the scope of a Statement? A class, a method, or something else?
- What is the role of TODO list in TDD?
- Why use anonymous generated values instead of literals as the input of a specified behavior?
- Why and how to use the `Any` class?
- What code to extract from a Statement to shared utility methods?
- Why such a strange approach to creating enumerated constants?

A lot of questions, isn't it? Unfortunately, TDD has this high entry barrier, at least for someone used to the traditional way of writing code. Anyway, that is what this tutorial is for – to answer such questions and lower this barrier. Thus, we will try to answer those questions one by one.

How to start?

Whenever I sat down with someone who was about to write code in a Statement-first manner for the first time, the person would stare at the screen, then at me, then would say: "what now?". It's easy to say: "You know how to write code, you know how to write a test for it, just this time start with the latter rather than the first", but for many people, this is something that blocks them completely. If you are one of them, don't worry – you're not alone. I decided to dedicate this chapter solely to techniques for kicking off a Statement when there is no code.

Start with a good name

I already said that a Statement is a description of behavior expressed in code. A thought process leading to creating such an executable Statement might look like the following sequence of questions:

1. What is the scope of the behavior I'm trying to specify? Example answer: I'm trying to specify a behavior of a `Calculator` class.
2. What is the behavior of a `Calculator` class I'm trying to specify? Example answer: it should display all entered digits that are not leading zeroes.
3. How to specify this behavior through code? Example answer: `[Fact] public void ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes() ...` (i.e. a piece of code).

Note that before writing any code, there are at least two questions that can be answered in the human language. Many times answering these questions first before starting to write the code of the Statement makes things easier. Even though, this can still be a challenging process. To apply this advice successfully, some knowledge on how to properly name Statements is required. I know not everybody pays attention to naming their Statements, mainly because the Statements are often considered second-level citizens – as long as they run and "prove the code doesn't contain defects", they are considered sufficient. We will take a look at some examples of bad names and then I'll go into some rules of good naming.

Consequences of bad naming

I have seen many people not caring about how their Statements are named. This is a symptom of treating the Specification as garbage or leftovers – I consider this approach dangerous because I have seen it lead to Specifications that are hard to maintain and that look more like lumps of code put together accidentally in a haste than a kind of "living documentation". Imagine that your Specification consists of Statements named like this:

- `TrySendPacket()`

- `TrySendPacket2()`
- `testSendingManyPackets()`
- `testWrongPacketOrder1()`
- `testWrongPacketOrder2()`

and see how difficult it is to answer the following questions:

1. How do you know what situation each Statement describes?
2. How do you know whether the Statement describes a single situation or several at the same time?
3. How do you know whether the assertions inside those Statements are the right ones assuming each Statement was written by someone else or a long time ago?
4. How do you know whether the Statement should stay or be removed from the Specification when you modify the functionality described by this Statement?
5. If your changes in production code make a Statement turn false, how do you know whether the Statement is no longer correct or the production code is wrong?
6. How do you know whether you will not introduce a duplicate Statement for a scenario when adding to a Specification that was created by another team member?
7. How do you estimate, by looking at the runner tool report, whether the fix for a failing Statement will be easy or not?
8. What do you answer new developers in your team when they ask you "what is this Statement for?"
9. How do you know when your Specification is complete if you can't tell from the Statement names what behaviors you already have covered and what not?

What does a good name contain?

To be of any use, the name of a Statement has to describe its expected behavior. At the minimum, it should describe what happens under what circumstances. Let's take a look at one of the names Steve Freeman and Nat Pryce came up with in their great book Growing Object-Oriented Software Guided By Tests[11]:

```
1 notifiesListenersThatServerIsUnavailableWhenCannotConnectToItsMonitoringPort()
```

Note a few things about the name of the Statement:

1. It describes a behavior of an instance of a specific class. Note that it doesn't contain the name of the method that triggers the behavior, because what is specified is not a single method, but the behavior itself (this will be covered in more detail in the coming chapters). The Statement name simply tells what an instance does ("notifies listeners that server is unavailable") under certain circumstances ("when cannot connect to its monitoring port"). It is important for me because I can derive such a description from thinking about the responsibilities of a class without the need to know any of its method signatures or the code that's inside the class. Hence, this is something I can come up with before implementing – I just need to know why I created this class and build on this knowledge.

[11] http://www.growing-object-oriented-software.com/

2. The name is relatively long. Really, really, **really** don't worry about it. As long as you are describing a single behavior, I'd say it's fine. I've seen people hesitate to give long names to Statements because they tried to apply the same rules to those names as to the names of methods in production code. In production code, a long method name can be a sign that the method has too many responsibilities or that insufficient abstraction level is used to describe the functionality and that the name may needlessly reveal implementation details. My opinion is that these two reasons don't apply as much to Statements. In the case of Statements, the methods are not invoked by anyone besides the automatic test runner, so they will not obfuscate any code that would need to call them with their long names. Besides, the Statements' names need not be as abstract as production code method names – they can reveal more.

   Alternatively, we could put all the information in a comment instead of the Statement name and leave the name short, like this:

```
1  [Fact]
2  //Notifies listeners that server
3  //is unavailable when cannot connect
4  //to its monitoring port
5  public void Statement_002()
6  {
7    //...
8  }
```

   however, there are two downsides to this. First, we now have to add an extra piece of information (`Statement_002`) only to satisfy the compiler, because every method needs to have a name anyway – and there is usually no value a human could derive from a name such as `Statement_002`. The second downside is that when the Statement turns false, the test runner shows the following line: `Statement_002: FAILED` – note that all the information included in the comment is missing from the failure report. I consider it much more valuable to receive a report like:

   `notifiesListenersThatServerIsUnavailableWhenCannotConnectToItsMonitoringPort: FAILED`

   because in such a case, a lot of information about the Statement that fails is available from the test runner report.
3. Using a name that describes a single behavior allows me to find out quickly why the Statement turned false. Let's say a Statement is true when I start refactoring, but at one point it turns false and the report in the runner looks like this: `TrySendingHttpRequest: FAILED` – it only tells me that an attempt was made to send an HTTP request, but, for instance, doesn't tell me whether the object I specified in that Statement is some kind of sender that should try to send this request under some circumstances, or if it is a receiver that should handle such a request properly. To learn what went wrong, I have to go open the source code of the Statement. On the other hand, when I have a Statement named `ShouldRespondWithAnAckWheneverItReceivesAnHttpRequest`, then if it turns false, I know what's broken – the object no longer responds with an ACK to an HTTP request. This may be enough to identify which part of the code is at fault and which of my changes made the Statement false.

My favorite convention

There are many conventions for naming Statements appropriately. My favorite is the one developed by Dan North[12], where each Statement name begins with the word `Should`. So for example, I would name a Statement:

```
ShouldReportAllErrorsSortedAlphabeticallyWhenErrorsOccurDuringSearch()
```

The name of the Specification (i.e. class name) answers the question "who should do it?", i.e. when I have a class named `SortingOperation` and want to say that it "should sort all items in ascending order when performed", I say it like this:

```
1 public class SortingOperationSpecification
2 {
3  [Fact] public void
4  ShouldSortAllItemsInAscendingOrderWhenPerformed()
5  {
6  }
7 }
```

By writing the above, I say that "Sorting operation *(this is derived from the Specification class name)* should sort all items in ascending order when performed *(this is derived from the name of the Statement)*".

The word "should" was introduced by Dan to weaken the statement following it and thus to allow questioning what you are stating and ask yourself the question: "should it really?". If this causes uncertainty, then it is high time to talk to a domain expert and make sure you understand well what you need to accomplish. If you are not a native English speaker, the "should" prefix will probably have a weaker influence on you – this is one of the reasons why I don't insist on you using it. I like it though[13].

When devising a name, it's important to put the main focus on what result or action is expected from an object, not e.g. from one of its methods. If you don't do that, it may quickly become troublesome. As an example, one of my colleagues was specifying a class `UserId` (which consisted of a user name and some other information) and wrote the following name for the Statement about the comparison of two identifiers:

```
EqualOperationShouldFailForTwoInstancesWithTheSameUserName().
```

Note that this name is not written from the perspective of a single object, but rather from the perspective of an operation that is executed on it. We stopped thinking in terms of object responsibilities and started thinking in terms of operation correctness. To reflect an object perspective, this name should be something more like:

```
ShouldNotBeEqualToAnotherIdThatHasDifferentUserName().
```

When I find myself having trouble with naming like this, I suspect one of the following may be the case:

[12] https://dannorth.net/introducing-bdd/

[13] There are also some arguments against using the word "should", e.g. by Kevlin Henney (see https://www.infoq.com/presentations/testing-communication).

1. I am not specifying a behavior of a class, but rather the outcome of a method.
2. I am specifying more than one behavior.
3. The behavior is too complicated and hence I need to change my design (more on this later).
4. I am naming the behavior of an abstraction that is too low-level, putting too many details in the name. I usually only come to this conclusion when all the previous points fail me.

Can't the name really become too long?

A few paragraphs ago, I mentioned you shouldn't worry about the length of Statement names, but I have to admit that the name can become too long occasionally. A rule I try to follow is that the name of a Statement should be easier to read than its content. Thus, if it takes me less time to understand the point of a Statement by reading its body than by reading its name, then I consider the name too long. If this is the case, I try to apply the heuristics described above to find and fix the root cause of the problem.

Start by filling the GIVEN-WHEN-THEN structure with the obvious

This technique can be used as an extension to the previous one (i.e. starting with a good name), by inserting one more question to the question sequence we followed the last time:

1. What is the scope of the behavior I'm trying to specify? Example answer: I'm trying to specify the behavior of a `Calculator` class.
2. What is the behavior of a `Calculator` class I'm trying to specify? Example answer: it should display all entered digits that are not leading zeroes.
3. **What is the context ("GIVEN") of the behavior, the action ("WHEN") that triggers it and the expected reaction ("THEN") of the specified object? Example answer: Given I turn on the calculator, when I enter any digit that's not a 0 followed by any digits, then they should be visible on the display**.
4. How to specify this behavior through code? Example answer: `[Fact] public void ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes() ...` (i.e. a piece of code).

Alternatively, it can be used without the naming step, when it's harder to come up with a name than with a GIVEN-WHEN-THEN structure. In other words, a GIVEN-WHEN-THEN structure can be easily derived from a good name and vice versa.

This technique is about taking the GIVEN, WHEN and THEN parts and translating them into code in an almost literal, brute-force way (without paying attention to missing classes, methods or variables), and then adding all the missing pieces that are required for the code to compile and run.

Example

Let's try it out on a simple problem of comparing two users for equality. We assume that two users should be equal to each other if they have the same name:

```
Given a user with any name
When I compare it to another user with the same name
Then it should appear equal to this other user
```

Let's start with the translation part. Again, remember we're trying to make the translation as literal as possible without paying attention to all the missing pieces for now.

The first line:

```
Given a user with any name
```

can be translated literally to the following piece of code:

```
var user = new User(anyName);
```

Note that we don't have the `User` class yet and we don't bother for now with what `anyName` really is. It's OK.

Then the second line:

```
When I compare it to another user with the same name
```

can be written as:

```
user.Equals(anotherUserWithTheSameName);
```

Great! Again, we don't care what `anotherUserWithTheSameName` is yet. We treat it as a placeholder. Now the last line:

```
Then it should appear equal to this other user
```

and its translation into the code:

```
Assert.True(usersAreEqual);
```

Ok, so now that the literal translation is complete, let's put all the parts together and see what's missing to make this code compile:

```
[Fact] public void
ShouldAppearEqualToAnotherUserWithTheSameName()
{
  //GIVEN
  var user = new User(anyName);

  //WHEN
  user.Equals(anotherUserWithTheSameName);

  //THEN
  Assert.True(usersAreEqual);
}
```

As we expected, this doesn't compile. Notably, our compiler might point us towards the following gaps:

1. Variable `anyName` is not declared.
2. Object `anotherUserWithTheSameName` is not declared.
3. Variable `usersAreEqual` is both not declared and it does not hold the comparison result.
4. If this is our first Statement, we might not even have the `User` class defined at all.

The compiler created a kind of a small TODO list for us, which is nice. Note that while we don't have a compiling piece of code, filling the gaps to make it compile boils down to making a few trivial declarations and assignments:

1. `anyName` can be defined as:

   ```
   var anyName = Any.String();
   ```
2. `anotherUserWithTheSameName` can be defined as:

   ```
   var anotherUserWithTheSameName = new User(anyName);
   ```
3. `usersAreEqual` can be defined as variable which we assign the comparison result to:

   ```
   var usersAreEqual = user.Equals(anotherUserWithTheSameName);
   ```
4. If class `User` does not yet exist, we can add it by simply stating:

   ```
   public class User
   {
     public User(string name) {}
   }
   ```

Putting it all together again, after filling the gaps, gives us:

```
[Fact] public void
ShouldAppearEqualToAnotherUserWithTheSameName()
{
  //GIVEN
  var anyName = Any.String();
  var user = new User(anyName);
  var anotherUserWithTheSameName = new User(anyName);

  //WHEN
  var usersAreEqual = user.Equals(anotherUserWithTheSameName);

  //THEN
  Assert.True(usersAreEqual);
}
```

And that's it – the Statement itself is complete!

Start from the end

This is a technique that I suggest to people that seem to have no idea how to start. I got it from Kent Beck's book Test-Driven Development by Example. It seems funny at first glance, but I found it quite powerful at times. The trick is to write the Statement "backward", i.e. starting with what the result verification (in terms of the *GIVEN-WHEN-THEN* structure, we would say that we start with our *THEN* part).

This works well when we are quite sure of what the outcome in our scenario should be, but not quite so sure of how to get there.

Example

Imagine we are writing a class containing the rules for granting or denying access to a reporting functionality. This reporting functionality is based on roles. We have no idea what the API should look like and how to write our Statement, but we do know one thing: in our domain, the access can be either granted or denied. Let's take the first case we can think of – the "access granted" case – and, as we are moving backward from the end, let's begin with the following assertion:

```
//THEN
Assert.True(accessGranted);
```

Ok, that part was easy, but did we make any progress with that? Of course we did – we now have code that does not compile, with the error caused by the variable `accessGranted`. Now, in contrast to the previous approach where we translated a GIVEN-WHEN-THEN structure into a Statement, our goal is not to make this compile as soon as possible. Instead, we need to answer the question: how do I know whether the access is granted or not? The answer: it is the result of the authorization of the allowed role. Ok, so let's just write it down in code, ignoring everything that stands in our way:

```
//WHEN
var accessGranted
 = access.ToReportingIsGrantedTo(roleAllowedToUseReporting);
```

For now, try to resist the urge to define a class or variable to make the compiler happy, as that may throw you off the track and steal your focus from what is important. The key to doing TDD successfully is to learn to use something that does not exist yet as if it existed and not worry until needed.

Note that we don't know what `roleAllowedToUseReporting` is, neither do we know what `access` object stands for, but that didn't stop us from writing this line. Also, the `ToReportingIsGrantedTo()` method is just taken off the top of our head. It's not defined anywhere, it just made sense to write it like this, because it is the most direct translation of what we had in mind.

Anyway, this new line answers the question about where we take the `accessGranted` value from, but it also makes us ask further questions:

1. Where does the `access` variable come from?
2. Where does the `roleAllowedToUseReporting` variable come from?

As for `access`, we don't have anything specific to say about it other than that it is an object of a class that is not defined yet. What we need to do now is to pretend that we have such a class (but let's not define it yet). How do we call it? The instance name is `access`, so it's quite straightforward to name the class `Access` and instantiate it in the simplest way we can think of:

```
//GIVEN
var access = new Access();
```

Now for the `roleAllowedToUseReporting`. The first question that comes to mind when looking at this is: which roles are allowed to use reporting? Let's assume that in our domain, this is either an Administrator or an Auditor. Thus, we know what is going to be the value of this variable. As for the type, there are various ways we can model a role, but the most obvious one for a type that has few possible values is an enum[14]. So:

```
//GIVEN
var roleAllowedToUseReporting = Any.Of(Roles.Admin, Roles.Auditor);
```

And so, working our way backward, we have arrived at the final solution (in the code below, I already gave the Statement a name – this is the last step):

[14]This approach of picking a single value out of several ones using `Any.From()` does not always work well with enums. Sometimes a parameterized test (a "theory" in XUnit.NET terminology) is better. This topic will be discussed in one of the coming chapters.

```
[Fact] public void
ShouldAllowAccessToReportingWhenAskedForEitherAdministratorOrAuditor()
{
 //GIVEN
 var roleAllowedToUseReporting = Any.Of(Roles.Admin, Roles.Auditor);
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(roleAllowedToUseReporting);

 //THEN
 Assert.True(accessGranted);
}
```

Using what we learned by formulating the Statement, it was easy to give it a name.

Start by invoking a method if you have one

If preconditions for this approach are met, it's the most straightforward one and I use it a lot[15].

Many times, we have to add a new class that implements an already existing interface. The interface imposes what methods the new class must support. If the method signatures are already decided, we can start our Statement with a call to one of the methods and then figure out the rest of the context we need to make it run properly.

Example

Imagine we have an application that, among other things, handles importing an existing database exported from another instance of the application. Given that the database is large and importing it can be a lengthy process, a message box is displayed each time a user performs the import. Assuming the user's name is Johnny, the message box displays the message "Johnny, please sit down and enjoy your coffee for a few minutes as we take time to import your database." The class that implements this looks like:

[15]Look for details in chapter 2.

```
public class FriendlyMessages
{
  public string
  HoldOnASecondWhileWeImportYourDatabase(string userName)
  {
    return string.Format("{0}, "
      + "please sit down and enjoy your coffee "
      + "for a few minutes as we take time "
      + "to import your database",
      userName);
  }
}
```

Now, imagine that we want to ship a trial version of the application with some features disabled, one of which is the database import. One of the things we need to do is display a message saying that this is a trial version and that the import feature is locked. We can do this by extracting an interface from the `FriendlyMessages` class and implement this interface in a new class used when the application is run as the trial version. The extracted interface looks like this:

```
public interface Messages
{
  string HoldOnASecondWhileWeImportYourDatabase(string userName);
}
```

So our new implementation is forced to support the `HoldOnASecondWhileWeImportYourDatabase()` method. Let's call this new class `TrialVersionMessages` (but don't create it yet!) and we can write a Statement for its behavior. Assuming we don't know where to start, we just start with creating an object of the class (we already know the name) and invoking the method we already know we need to implement:

```
[Fact]
public void TODO()
{
 //GIVEN
 var trialMessages = new TrialVersionMessages();

 //WHEN
 trialMessages.HoldOnASecondWhileWeImportYourDatabase();

 //THEN
 Assert.True(false); //so we don't forget this later
}
```

As you can see, we added an assertion that always fails at the end to remind ourselves that the Statement is not finished yet. As we don't have any relevant assertions yet, the Statement will otherwise be considered as true as soon as it compiles and runs and we may not notice that it's

incomplete. As it currently stands, the Statement doesn't compile anyway, because there's no `TrialVersionMessages` class yet. Let's create one with as little implementation as possible:

```
public class TrialVersionMessages : Messages
{
 public string HoldOnASecondWhileWeImportYourDatabase(string userName)
 {
   throw new NotImplementedException();
 }
}
```

Note that there's only as much implementation in this class as required to compile this code. Still, the Statement won't compile yet. This is because the method `HoldOnASecondWhileWeImportYourDatabase()` takes a string argument and we didn't pass any in the Statement. This makes us ask the question of what this argument is and what its role is in the behavior triggered by the `HoldOnASecondWhileWeImportYourDatabase()` method It looks like it's a user name. Thus, we can add it to the Statement like this:

```
[Fact]
public void TODO()
{
 //GIVEN
 var trialMessages = new TrialVersionMessages();
 var userName = Any.String();

 //WHEN
 trialMessages.
  HoldOnASecondWhileWeImportYourDatabase(userName);

 //THEN
 Assert.True(false); //to remember about it
}
```

Now, this compiles but is considered false because of the guard assertion that we put at the end. Our goal is to substitute it with a proper assertion for the expected result. The return value of the call to `HoldOnASecondWhileWeImportYourDatabase` is a string message, so all we need to do is to come up with the message that we expect in case of the trial version:

```
[Fact]
public void TODO()
{
 //GIVEN
 var trialMessages = new TrialVersionMessages();
 var userName = Any.String();
 var expectedMessage =
  string.Format(
    "{0}, better get some pocket money and buy a full version!",
    userName);

 //WHEN
 var message = trialMessages.
  HoldOnASecondWhileWeImportYourDatabase(userName);

 //THEN
 Assert.Equal(expectedMessage, message);
}
```

All what is left is to find a good name for the Statement. This isn't an issue since we already specified the desired behavior in the code, so we can just summarize it as something like `ShouldCreateAPromptForFullVersionPurchaseWhenAskedForImportDatabaseMessage()`.

Summary

When I'm stuck and don't know how to start writing a new failing Statement, the techniques from this chapter help me push things in the right direction. Note that the examples given are simplistic and built on an assumption that there is only one object that takes some kind of input parameter and returns a well-defined result. However, this isn't how most of the object-oriented world is built. In that world, we often have objects that communicate with other objects, send messages, invoke methods on each other and these methods often don't have any return values but are instead declared as `void`. Even though, all of the techniques described in this chapter will still work in such a case and we'll revisit them as soon as we learn how to do TDD in the larger object-oriented world (after the introduction of the concept of mock objects in Part 2). Here, I tried to keep it simple.

How is TDD about analysis and what does "GIVEN-WHEN-THEN" mean?

During the work on the calculator code, Johnny mentioned that TDD is, among other things, about analysis. This chapter further explores this concept. Let's start by answering the following question:

Is there a commonality between analysis and TDD?

From Wikipedia[16]:

> Analysis is the process of breaking a complex topic or substance into smaller parts to gain a better understanding of it.

Thus, for TDD to be about analysis, it would have to fulfill two conditions:

1. It would have to be a process of breaking a complex topic into smaller parts
2. It would have to allow gaining a better understanding of such smaller parts

In the story about Johnny, Benjamin and Jane, I included a part where they analyze requirements using concrete examples. Johnny explained that this is a part of a process called Acceptance Test-Driven Development. This process, followed by the three characters, fulfilled both mentioned conditions for it to be considered analytical. But what about TDD itself?

Although I used parts of the ATDD process in the story to make the analysis part more obvious, similar things happen at pure technical levels. For example, when starting development with a failing application-wide Statement (i.e. one that covers a behavior of an application as a whole. We will talk about the levels of granularity of Statements later. For now, the only thing you need to know is that the so-called "unit tests level" is not the only level of granularity we write Statements on), we may encounter a situation where we need to call a web method and make an assertion on its result. This makes us think: how should this method be named? What are the scenarios it supports? What do I expect to get out of it? How should I, as its user, be notified about errors? Many times, this leads us to either a conversation (if there is another stakeholder that needs to be involved in the decision) or rethinking our assumptions. The same applies on "unit level" - if a class implements a domain rule, there might be some good domain-related questions resulting from trying to write a Statement for it. If a class implements a technical rule, there might be some technical questions to discuss with other developers, etc. This is how we gain a better understanding of the topic we are analyzing, which makes TDD fulfill the second of the two requirements for it to be an analysis method.

[16] https://en.wikipedia.org/wiki/Analysis

But what about the first requirement? What about breaking a complex logic into smaller parts?

If you go back to Johnny and Benjamin's story, you will note that when talking to a customer and when writing code, they used a TODO list. This list was first filled with whatever scenarios they came up with, but later, they would add smaller units of work. When doing TDD, I do the same, essentially decomposing complex topics into smaller items and putting them on the TODO list (this is one of the practices that serve decomposition. The other one is mocking, but let's leave that for now). Thanks to this, I can focus on one thing at a time, crossing off item after item from the list after it's done. If I learn something new or encounter a new issue that needs our attention, I can add it to the TODO list and get back to it later, for now continuing my work on the current item of focus.

An example TODO list from the middle of an implementation task may look like this (don't read through it, I put it here just to give you a glimpse - you're not supposed to understand what the list items are about either):

1. ~~Create an entry point to the module (top-level abstraction)~~
2. ~~Implement main workflow of the module~~
3. ~~Implement `Message` interface~~
4. ~~Implement `MessageFactory` interface~~
5. Implement `ValidationRules` interface
6. ~~Implement behavior required from Wrap method in `LocationMessageFactory` class~~
7. Implement behavior required from ValidateWith method in `LocationMessage` class for Speed field
8. Implement behavior required from ValidateWith method in `LocationMessage` class for Age field
9. Implement behavior required from ValidateWith method in `LocationMessage` class for Sender field

Note that some of the items are already crossed off as done, while others remain pending and waiting to be addressed. All these items are what the article on Wikipedia calls "smaller parts" - a result of breaking down a bigger topic.

For me, the arguments that I gave you are enough to think that TDD is about analysis. The next question is: are there any tools we can use to aid and inform this analysis part of TDD? The answer is yes and you already saw both of them in this book, so now we're going to have a closer look.

Gherkin

Hungry? Too bad, because the Gherkin I am going to tell you about is not edible. It is a notation and a way of thinking about behaviors of the specified piece of code. It can be applied on different levels of granularity – any behavior, whether of a whole system or a single class, may be described using Gherkin.

We already used this notation, we just didn't name it so. Gherkin is the GIVEN-WHEN-THEN structure that you can see everywhere, even as comments in the code samples. This time, we are stamping a name on it and analyzing it further.

In Gherkin, a behavior description consists mostly of three parts:

1. Given – a context
2. When – a cause
3. Then – an effect

In other words, the emphasis is on causality in a given context. There's also a fourth keyword: `And`[17] – we can use it to add more context, more causes or more effects. You'll have a chance to see an example in a few seconds

As I said, there are different levels you can apply it to. Here is an example of such a behavior description from the perspective of its end-user (this is called acceptance-level Statement):

```
Given a bag of tea costs $20
And there is a discount saying "pay half for a second bag"
When I buy two bags
Then I should be charged $30
```

And here is one for unit-level (note again the line starting with "And" that adds to the context):

```
Given a list with 2 items
When I add another item
And check items count
Then the count should be 3
```

While on acceptance level we put such behavior descriptions together with code as a single whole (If this doesn't ring a bell, look at tools such as SpecFlow[18] or Cucumber[19] or FIT[20] to get some examples), on the unit level the description is usually not written down in a literal way, but rather it is translated and written only in form of source code. Still, the structure of GIVEN-WHEN-THEN is useful when thinking about behaviors required from an object or objects, as we saw when we talked about starting from Statement rather than code. I like to put the structure explicitly in my Statements – I find that it helps make them more readable[21]. So most of my unit-level Statements follow this template:

[17] Some claim there are other keywords, like `But` and `Or`. However, we won't need to resort to them so I decided to ignore them in this description.

[18] http://specflow.org/

[19] https://cucumber.io/

[20] http://fit.c2.com/

[21] Seb Rose wrote a blog post where he suggests against the `//GIVEN //WHEN //THEN` comments and states that he only uses empty lines to separate the three sections, see http://claysnow.co.uk/unit-tests-are-your-specification/

```
[Fact]
public void Should__BEHAVIOR__()
{
  //GIVEN
  ...context...

  //WHEN
  ...trigger...

  //THEN
  ...assertions etc....
}
```

Sometimes the WHEN and THEN sections are not so easily separable – then I join them, like in case of the following Statement specifying that an object throws an exception when asked to store null:

```
[Fact]
public void ShouldThrowExceptionWhenAskedToStoreNull()
{
  //GIVEN
  var safeList = new SafeList();

  //WHEN - THEN
  Assert.Throws<Exception>(
    () => safeList.Store(null)
  );
}
```

By thinking in terms of these three parts of behavior, we may arrive at different circumstances (GIVEN) at which the behavior takes place, or additional ones that are needed. The same goes for triggers (WHEN) and effects (THEN). If anything like this comes to our mind, we add it to the TODO list to revisit it later.

TODO list... again!

As I wrote earlier, a TODO list is a repository for our deferred work. This includes anything that comes to our mind when writing or thinking about a Statement but is not a part of the current Statement we are writing. On one hand, we don’t want to forget it, on the other - we don’t want it to haunt us and distract us from our current task, so we write it down as soon as possible and continue with our current task. When we’are finished with it, we take another item from TODO list and start working on it.

Imagine we’re writing a piece of logic that allows users access when they are employees of a zoo, but denies access if they are merely guests of the zoo. Then, after starting writing a Statement we

realize that employees can be guests as well – for example, they might choose to visit the zoo with their families during their vacation. Still, the two previous rules hold, so to avoid being distracted by this third scenario, we can quickly add it as an item to the TODO list (like "TODO: what if someone is an employee, but comes to the zoo as a guest?") and finish the current Statement. When we're finished, you can always come back to the list of deferred items and pick the next item to work on.

There are two important questions related to TODO lists: "what exactly should we add as a TODO list item?" and "How to efficiently manage the TODO list?". We will take care of these two questions now.

What to put on a TODO list?

Everything that we need addressed but is out of the scope of the current Statement. Those items may be related to implementing unimplemented methods, to add whole functionalities (such items are usually broken further into more fine-grained sub-tasks as soon as we start implementing them), they might be reminders to take a better look at something (e.g. "investigate what is this component's policy for logging errors") or questions about the domain that need to get answered. If we tend to get carried away too much in coding and miss our lunch, we can even add a reminder ("TODO: eat lunch!"). I have never encountered a case where I needed to share this TODO list with anyone else, so I tend to treat it as my sketchbook. I recommend the same to you - the list is yours!

How to pick items from a TODO list?

Which item to choose from a TODO list when we have several of them? I have no clear rule, although I tend to take into account the following factors:

1. Risk – if what I learn by implementing or discussing a particular item from the list can have a big impact on the design or behavior of the system, I tend to pick such items first. An example of such item is when I start implementing validation of a request that reaches my application and want to return different error depending on which part of the request is wrong. Then, during the development, I may discover that more than one part of the request can be wrong at the same time and I have to answer a question: which error code should be returned in such a case? Or maybe the return codes should be accumulated for all validations and then returned as a list?
2. Difficulty – depending on my mental condition (how tired I am, how much noise is currently around my desk etc.), I tend to pick items with difficulty that best matches this condition. For example, after finishing an item that requires a lot of thinking and figuring things out, I tend to take on some small and easy items to feel the wind blowing in my sails and to rest a little bit.
3. Completeness – in simplest words, when I finish test-driving an "if" case, I usually pick up the "else" next. For example, after I finish implementing a Statement saying that something should return true for values less than 50, then the next item to pick up is the "greater or equal to 50" case. Usually, when I start test-driving a class, I take items related to this class until I run out of them, then go on to another one.

Of course, a TODO list is just one source of such TODO items. Typically, when searching for items to do, I examine the following sources of items in the following order:

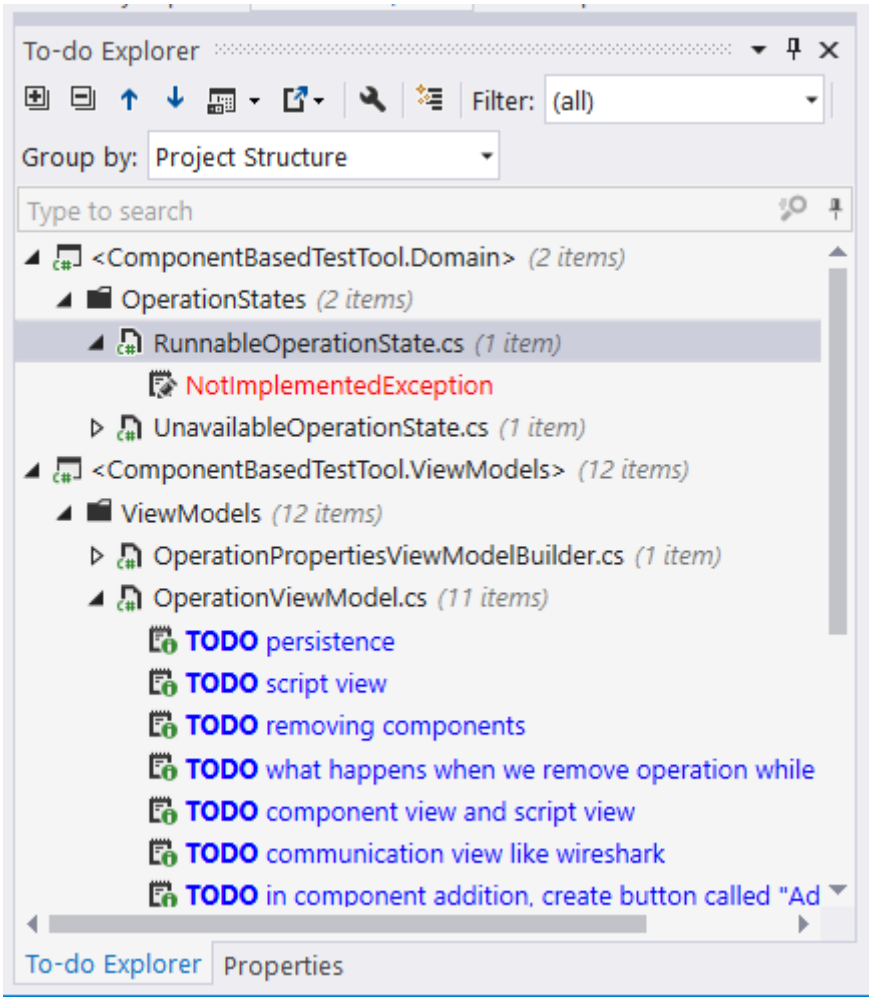
1. compiler failures,
2. False statements,
3. My TODO list.

Where to put a TODO list?

I encountered two ways of maintaining a TODO list. The first one is on a sheet of paper. The drawback is that every time I need to add something to the list, I need to take my hands off the keyboard, grab a pen or a pencil and then get back to coding. Also, the only way a TODO item written on a sheet of paper can tell me which place in my code it is related to, is (obviously) by its text. The good thing about paper is that it is by far one of the best tools for sketching, so when my TODO item is best stored as a diagram or a drawing (which doesn’t happen too often, but sometimes does) , I use pen and paper.

The second alternative is to use a TODO list functionality built-in into an IDE. Most IDEs, such as Visual Studio (and Resharper plugin has an enhanced version), Xamarin Studio, IntelliJ or Eclipse-based IDEs have such functionality. The rules are simple – I insert special comments (e.q. `//TODO do something`) in the code and a special view in my IDE aggregates them for me, allowing me to navigate to each item later. This is my primary way of maintaining a TODO list, because:

1. They don’t force me to take my hands off my keyboard to add an item to the list.
2. I can put a TODO item in a certain place in the code where it makes sense and then navigate back to it later with a click of a mouse. This, apart from other advantages, allows writing shorter notes than if I had to do it on paper. For example, a TODO item saying “TODO: what if it throws an exception?” looks out of place on a sheet of paper, but when added as a comment to my code in the right place, it’s sufficient.
3. Many TODO lists automatically add items for certain things that happen in the code. E.g. in C#, when I’m yet to implement a method that was automatically generated by the IDE, its body usually consists of a line that throws a `NotImplementedException` exception. Guess what – `NotImplementedException` occurrences are added to the TODO list automatically, so I don’t have to manually add items to the TODO list for implementing the methods where they occur.



Resharper TODO Explorer docked as a window in Visual Studio 2015 IDE

The TODO list maintained in the source code has one minor drawback - we have to remember to clear the list when we finish working with it or we may end up pushing the TODO items to the source control repository along with the rest of the source code. Such leftover TODO items may accumulate in the code, effectively reducing the ability to navigate through the items that were only added by a specific developer. There are several strategies for dealing with this:

1. For greenfield projects, I found it relatively easy to set up a static analysis check that runs when the code is built and doesn’t allow the automatic build to pass unless all TODO items are removed. This helps ensure that whenever a change is pushed to a version control system, it’s stripped of the unaddressed TODO items.
2. In some other cases, it’s possible to use a strategy of removing all TODO items from a project before starting working with it. Sometimes it may lead to conflicts between people when TODO items are used for something else than a TDD task list and someone for whatever reason wants them to stay in the code longer. Even though I’m of opinion that such cases of leaving TODO items for longer should be extremely rare at best, however, others may have different opinions.
3. Most modern IDEs offer support markers other than `//TODO` for placing items on a TODO list, for example, `//BUG`. In such a case, I can use the `//BUG` marker to mark just my items

and then I can filter other items out based on that marker. Bug markers are commonly not intended to be left in the code, so it’s much less risky for them to accumulate.
4. As a last resort technique, I can usually define custom markers that are placed on TODO list and, again, use filters to see only the items that were defined by me (plus usually `NotImplementedExceptions`).

TDD process expanded with a TODO list

In one of the previous chapters, I introduced you to the basic TDD process that contained three steps: write false Statement you wish was true, change the production code so that the Statement is true and then refactor the code. TODO list adds new steps to this process leading to the following expanded list:

1. Examine TODO list and pick an item that makes the most sense to implement next.
2. Write false Statement you wish was true.
3. See it reported as false for the right reason.
4. Change the production code to make the Statement true and make sure all already true Statements remain true.
5. Cross off the item from the TODO list.
6. Repeat steps 1-5 until no item is left on the TODO list.

Of course, we can (and should) add new items to the TODO list as we make progress with the existing ones and at the beginning of each cycle the list should be re-evaluated to choose the most important item to implement next, also taking into account the things that were added during the previous cycle.

Potential issues with TODO lists

There are also some issues one may run into when using TODO lists. I already mentioned the biggest of them - that I often saw people add TODO items for means other than to support TDD and they never went back to these items. Some people joke that a TODO comment left in the code means “There was a time when I wanted to do …”. Anyway, such items may pollute our TDD-related TODO list with so much cruft that your items are barely findable.

Another downside is that when you work with multiple workspaces/solutions, your IDE will gather TODO items only from a single solution/workspace, so there may be times when several TODO lists will need to be maintained – one per workspace or solution. Fortunately, this isn’t usually a big deal.

What is the scope of a unit-level Statement in TDD?

In previous chapters, I described how tests form a kind of executable Specification consisting of many Statements. If so, then some fundamental questions regarding these Statements need to be raised, e.g.:

1. What goes into a single Statement?
2. How do I know that I need to write another Statement instead of expanding the existing one?
3. When I see a Statement, how do I know whether it is too big, too small, or just enough?

This can be summarized as one more general question: what should be the scope of a single Statement?

Scope and level

The software we write can be viewed in terms of structure and functionality. Functionality is about the features – things a piece of software does and does not, given certain circumstances. Structure is how this functionality is organized and divided between many subelements, e.g. subsystems, services, components, classes, methods, etc.

A structural element can easily handle several functionalities (either by itself or in cooperation with other elements). For example, many lists implement retrieving added items as well as some kind of searching or sorting. On the other hand, a single feature can easily span several structural elements (e.g. paying for a product in an online store will likely span at least several classes and probably touch a database).

Thus, when deciding what should go into a single Statement, we have to consider both structure and functionality and make the following decisions:

- structure – do we specify what a class should do, or what the whole component should do, or maybe a Statement should be about the whole system? I will refer to such structural decision as “level”.
- functionality – should a single Statement specify everything that structural element does, or maybe only a part of it? If only a part, then which part and how big should that part be? I will refer to such a functional decision as “functional scope”.

Our questions from the beginning of the chapter can be rephrased as:

1. On what level do we specify our software?
2. What should be the functional scope of a single Statement?

On what level do we specify our software?

The answer to the first question is relatively simple – we specify on multiple levels. How many levels there are and which ones we're interested in depends very much on the specific type of application that we write and programming paradigm (e.g. in pure functional programming, we don't have classes).

In this (and next) chapter, I focus mostly on class level (I will refer to it as unit level, since a class is a unit of behavior), i.e. every Statement is written against a public API of a specified class[22].

Does that mean that we can only use a single class in our executable Statement? Let's look at an example of a well-written Statement and try to answer this question:

```
[Fact] public void
ShouldThrowValidationExceptionWithFatalErrorLevelWhenValidatedStringIsEmpty()
{
  //GIVEN
  var validation = new Validation();

  //WHEN
  var exceptionThrown = Assert.Throws<CustomException>(
    () => validation.ApplyTo(string.Empty)
  );

  //THEN
  Assert.True(exceptionThrown.IsFatalError);
}
```

Ok, so let's see… how many real classes take part in this Statement? Three: a string, an exception, and the validation. So even though this is a Statement written against the public API of `Validation` class, the API itself demands using objects of additional classes.

What should be the functional scope of a single Statement?

The short answer to this question is *behavior*. Putting it together with the previous section, we can say that each unit-level Statement specifies a single behavior of a class written against the public API of that class. I like how Liz Keogh[23] says that a unit-level Statement shows one example of how a class is valuable to its users. Also, Amir Kolsky and Scott Bain[24] say that each Statement should "introduce a behavioral distinction not existing before".

What exactly is a behavior? If you read this book from the beginning, you've probably seen a lot of Statements that specify behaviors. Let me show you another one, though.

[22] Some disagree, however, with writing Statements on the class level - see http://blog.cleancoder.com/uncle-bob/2017/03/03/TDD-Harms-Architecture.html or https://vimeo.com/68375232

[23] https://lizkeogh.com/2012/05/30/showcasing-the-language-of-bdd/

[24] http://www.sustainabletdd.com/

Let's consider an example of a class representing a condition for deciding whether some kind of queue is full or not. A single bahavior we can specify is that the condition is met when it is notified three times of something being queued on a queue (so from a bigger-picture point of view, it's an observer of the queue):

```
[Fact] public void
ShouldBeMetWhenNotifiedThreeTimesOfItemQueued()
{
  //GIVEN
  var condition = new FullQueueCondition();
  condition.NotifyItemQueued();
  condition.NotifyItemQueued();
  condition.NotifyItemQueued();

  //WHEN
  var isMet = condition.IsMet();

  //THEN
  Assert.True(isMet);
}
```

The first thing to note is that two methods are called on the `condition` object: `NotifyItemQueued()` (three times) and `IsMet()` (once). I consider this example educative because I have seen people misunderstand the unit level as "specifying a single method". Sure, there is usually a single method triggering the behavior (in this case it's `isMet()`, placed in the `//WHEN` section), but sometimes, more calls are necessary to set up a preconditions for a given behavior (hence the three `Queued()` calls placed in the `//GIVEN` section).

The second thing to note is that the Statement only says what happens when the `condition` object is notified three times – this is the specified behavior. What about the scenario where the `condition` is only notified two times and when asked afterward, should say it isn't met? This is a separate behavior and should be described by a separate Statement. The ideal to which we strive is characterized by three rules by Amir Kolsky and cited by Ken Pugh in his book *Lean-Agile Acceptance Test-Driven Development*:

1. A Statement should turn false for a well-defined reason.
2. No other Statement should turn false for the same reason.
3. A Statement should not turn false for any other reason.

While it's impossible to achieve it in a literal sense (e.g. all Statements specifying the `FullQueueCondition` behaviors must call a constructor, so when I put a `throw new Exception()` inside it, all Statements will turn false), however, we want to keep as close to this goal as possible. This way, each Statement will introduce that "behavioral distinction" I mentioned before, i.e. it will show a new way the class can be valuable to its users.

Most of the time, I specify behaviors using the "GIVEN-WHEN-THEN" thinking framework. A behavior is triggered (`WHEN`) in some kind of context (`GIVEN`) and there are always some kind of results (`THEN`) of that behavior.

Failing to adhere to the three rules

The three rules I mentioned are derived from experience. Let's see what happens if we don't follow one of them.

Our next example is about some kind of buffer size rule. This rule is asked whether the buffer can handle a string of specified length and answers "yes" if this string is at most three-elements long. The writer of a Statement for this class decided to violate the rules we talked about and wrote something like this:

```
[Fact] public void
ShouldReportItCanHandleStringWithLengthOf3ButNotOf4AndNotNullString()
{
  //GIVEN
  var bufferSizeRule = new BufferSizeRule();

  //WHEN
  var resultForLengthOf3
    = bufferSizeRule.CanHandle(Any.StringOfLength(3));
  //THEN
  Assert.True(resultForLengthOf3);

  //WHEN - again?
  var resultForLengthOf4
    = bufferSizeRule.CanHandle(Any.StringOfLength(4))
  //THEN - again?
  Assert.False(resultForLengthOf4);

  //WHEN - again??
  var resultForNull = bufferSizeRule.CanHandle(null);
  //THEN - again??
  Assert.False(resultForNull);
}
```

Note that it specifies three behaviors:

1. Acceptance of a string of allowed size.
2. Refusal of handling a string of size above the allowed limit.
3. A special case of a null string.

As such, the Statement breaks rules: 1 (A Statement should turn false for a well-defined reason) and 3 (A Statement should not turn false for any other reason). In fact, there are three reasons that can make our Statement false.

There are several reasons to avoid writing Statements like this. Some of them are:

1. Most xUnit frameworks stop executing a Statement on first assertion failure. If the first assertion fails in the above Statement, we won't know whether the rest of the behaviors work fine until we fix the first one.
2. Readability tends to be worse as well as the documentation value of our Specification (the names of such Statements tend to be far from helpful).
3. Failure isolation is worse – when a Statement turns false, we'd prefer to know exactly which behavior was broken. Statements such as the one above don't give us this advantage.
4. Throughout a single Statement, we usually work with the same object. When we trigger multiple behaviors on it, we can't be sure how triggering one behavior impacts subsequent behaviors. If we have e.g. four behaviors in a single Statement, we can't be sure how the three earlier ones impact the last one. In the example above, we could get away with this, since the specified object returned its result based only on the input of a specific method (i.e. it did not contain any mutable state). Imagine, however, what could happen if we triggered multiple behaviors on a single list. Would we be sure that it does not contain any leftover element after we added some items, then deleted some, then sorted the list and deleted even more?

How many assertions do I need?

A single assertion by definition checks a single specified condition. If a single Statement is about a single behavior, then what about assertions? Does "single behavior" mean I can only have a single assertion per Statement? That was mostly the case for the Statements you have already seen throughout this book, but not for all.

To tell you the truth, there is a straightforward answer to this question – a rule that says: "have a single assertion per test". What is important to remember is that it applies to "logical assertions", as Robert C. Martin indicated[25].

Before we go further, I'd like to introduce a distinction. A "physical assertion" is a single `AssertXXXXX()` call. A "logical assertion" is one or more physical assertions that together specify one logical condition. To further illustrate this distinction, I'd like to give you two examples of logical assertions.

Logical assertion – example #1

A good example would be an assertion that specifies that all items in a list are unique (i.e. the list contains no duplicates). XUnit.net does not have such an assertion by default, but we can imagine we have written something like that and called it `AssertHasUniqueItems()`. Here's some code that uses this assertion:

[25]Clean Code series, episode 19 (https://cleancoders.com/episode/clean-code-episode-19-p1/show), Robert C. Martin, 2013

```
//some hypothetical code for getting the list:
var list = GetList();

//invoking the assertion:
AssertHasUniqueItems(list);
```

Note that it's a single logical assertion, specifying a well-defined condition. If we peek into the implementation however, we will find the following code:

```
public static void AssertHasUniqueItems<T>(List<T> list)
{
  for(var i = 0 ; i < list.Count ; i++)
  {
    for(var j = 0 ; j < list.Count ; j++)
    {
      if(i != j)
      {
        Assert.NotEqual(list[i], list[j]);
      }
    }
  }
}
```

Which already executes several physical assertions. If we knew the exact number of elements in collection, we could even use three `Assert.NotEqual()` assertions instead of `AssertHasUniqueItems()`:

```
//some hypothetical code for getting the collection:
var list = GetLastThreeElements();

//invoking the assertions:
Assert.NotEqual(list[0], list[1]);
Assert.NotEqual(list[0], list[2]);
Assert.NotEqual(list[1], list[2]);
```

Is it still a single assertion? Physically no, but logically – yes. There is still one logical thing these assertions specify and that is the uniqueness of the items in the list.

Logical assertion – example #2

Another example of a logical assertion is one that specifies exceptions: `Assert.Throws()`. We already encountered one like this in this chapter. Here is the code again:

```
[Fact] public void
ShouldThrowValidationExceptionWithFatalErrorLevelWhenValidatedStringIsEmpty()
{
  //GIVEN
  var validation = new Validation();

  //WHEN
  var exceptionThrown = Assert.Throws<CustomException>(
    () => validation.ApplyTo(string.Empty)
  );

  //THEN
  Assert.True(exceptionThrown.IsFatalError);
}
```

Note that in this case, there are two physical assertions (`Assert.Throws()` and `Assert.True()`), but one intent – to specify the exception that should be thrown. We may as well extract these two physical assertions into a single one with a meaningful name:

```
[Fact] public void
ShouldThrowValidationExceptionWithFatalErrorLevelWhenValidatedStringIsEmpty()
{
  //GIVEN
  var validation = new Validation();

  //WHEN - THEN
  AssertFatalErrorIsThrownWhen(
    () => validation.ApplyTo(string.Empty)
  );
}
```

So every time we have several physical assertions that can be (or are) extracted into a single assertion method with a meaningful name, I consider them a single logical assertion. There is always a gray area in what can be considered a "meaningful name" (but let's agree that `AssertAllConditionsAreMet()` is not a meaningful name, ok?). The rule of thumb is that this name should express our intent better and clearer than the bunch of assertions it hides. If we look again at the example of `AssertHasUniqueItems()` this assertion does a better job of expressing our intent than a set of three `Assert.NotEqual()`.

Summary

In this chapter, we tried to find out how much should go into a single Statement. We examined the notions of level and functional scope to end up with a conclusion that a Statement should cover a single behavior. We backed this statement by three rules by Amir Kolsky and looked at an example of what could happen when we don't follow one of them. Finally, we discussed how the notion of "single Statement per behavior" is related to "single assertion per Statement".

Developing a TDD style and Constrained Non-Determinism

In one of the first chapters, I introduced to you the idea of an anonymous values generator. I showed you the `Any` class which I use for generating such values. Throughout the chapters that followed, I have used it quite extensively in many of the Statements I wrote.

The time has come to explain a little bit more about the underlying principles of using anonymous values in Statements. Along the way, we'll also examine developing a style of TDD.

A style?

Yep. Why am I wasting your time writing about style instead of giving you the hardcore technical details? Here's my answer: before I started writing this tutorial, I read four or five books solely on TDD and maybe two others that contained chapters on TDD. All of this added up to about two or three thousands of paper pages, plus numerous posts on many blogs. And you know what I noticed? No two authors use the same set of techniques for test-driving their code! I mean, sometimes, when you look at the techniques they suggest, two authorities contradict each other. As each authority has their followers, it isn't uncommon to observe and take part in discussions about whether this or that technique is better than a competing one or which technique is "a smell"[26] and leads to trouble in the long run.

I've done a lot of this, too. I also tried to understand how come people praise techniques I (thought I) KNEW were wrong and led to disaster. Over time, I came to understand that this is not a "technique A vs. technique B" debate. There are certain sets of techniques that work together and symbiotically enhance each other. Choosing one technique leaves us with issues we have to resolve by adopting other techniques. This is how a style is developed.

Developing a style starts with a set of problems to solve and an underlying set of principles we consider important. These principles lead us to adopt our first technique, which makes us adopt another one and, ultimately, a coherent style emerges. Using Constrained Non-Determinism as an example, I will try to show you how part of a style gets derived from a technique that is derived from a principle.

Principle: Tests As Specification

As I already stressed, I strongly believe that tests should constitute an executable specification. Thus, they should not only pass input values to an object and assert on the output, they should also convey to their reader the rules according to which objects and functions work. The following toy example shows a Statement where it isn't explicitly explained what the relationship between input and output is:

[26] One of such articles can be found at https://martinfowler.com/articles/mocksArentStubs.html

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostname()
{
  //GIVEN
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo("MY_HOSTNAME");

  //THEN
  Assert.Equal("backup_MY_HOSTNAME.zip", name);
}
```

Although in this case the relationship can be guessed quite easily, it still isn't explicitly stated, so in more complex scenarios it might not be as trivial to spot. Also, seeing code like that makes me ask questions like:

- Is the `"backup_"` prefix always applied? What if I pass the prefix itself instead of `"MY_HOSTNAME"`? Will the name be `"backup_backup_.zip"`, or just `"backup_.zip"`?
- Is this object responsible for any validation of passed string? If I pass `"MY HOST NAME"` (with spaces) will this throw an exception or just apply the formatting pattern as usual?
- Last but not least, what about letter casing? Why is `"MY_HOSTNAME"` written as an upper-case string? If I pass `"my_HOSTNAME"`, will it be rejected or accepted? Or maybe it will be automatically converted to upper case?

This makes me adopt the first technique to provide my Statements with better support for the principle I follow.

First technique: Anonymous Input

I can wrap the actual value `"MY_HOSTNAME"` with a method and give it a name that better documents the constraints imposed on it by the specified functionality. In this case, the `BackupFileNamePattern()` method should accept whatever string I feed it (the object is not responsible for input validation), so I will name the wrapping method `AnyString()`:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostname()
{
  //GIVEN
  var hostname = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname);
```

```
10
11    //THEN
12    Assert.Equal("backup_MY_HOSTNAME.zip", name);
13  }
14
15  public string AnyString()
16  {
17    return "MY_HOSTNAME";
18  }
```

By using **anonymous input**, I provided better documentation of the input value. Here, I wrote `AnyString()`, but of course, there can be a situation where I use more constrained data, e.g. I would invent a method called `AnyAlphaNumericString()` if I needed a string that doesn't contain any characters other than letters and digits.

Anonymous input and equivalence classes

Note that this technique is useful only when we specify a scenario that should occur for all members of some kind of equivalence class. An example of an equivalence class is "a string starting with a number" or "a positive integer" or "any legal URI". When a behavior should occur only for a single specific input value, there is no room for making it anonymous. Taking authorization as an example, when a certain behavior occurs only when the input value is `Users.Admin`, we have no useful equivalence class and we should just use the literal value of `Users.Admin`. On the other hand, for a scenario that occurs for all values other than `Users.Admin`, it makes sense to use a method like `AnyUserOtherThan(Users.Admin)` or even `AnyNonAdminUser()`, because this is a useful equivalence class.

Now that the Statement itself is freed from the knowledge of the concrete value of `hostname` variable, the concrete value of `"backup_MY_HOSTNAME.zip"` in the assertion looks kind of weird. That's because, there is still no clear indication of the kind of relationship between input and output and whether there is any at all (as it currently is, the Statement suggests that the result of the `ApplyTo()` is the same for any `hostname` value). This leads us to another technique.

Second technique: Derived Values

To better document the relationship between input and output, we have to simply derive the expected value we assert on from the input value. Here is the same Statement with the assertion changed:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostname()
{
  //GIVEN
  var hostname = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname);

  //THEN
  Assert.Equal($"backup_{hostname}.zip", name);
}
public string AnyString()
{
  return "MY_HOSTNAME";
}
```

This looks more like a part of a specification, because we are documenting the format of the backup file name and show which part of the format is variable and which part is fixed. This is something you would probably find documented in a paper specification for the application you are writing – it would probably contain a sentence saying: "The format of a backup file should be **backup_H.zip**, where **H** is the current local hostname". What we used here was a **derived value**.

Derived values are about defining expected output in terms of the input that was passed to provide a clear indication on what kind of "transformation" the production code is required to perform on its input.

Third technique: Distinct Generated Values

Let's assume that sometime after our initial version is shipped, we are asked to change the backup feature so that it stores backed up data separately for each user that invokes this functionality. As the customer does not want to have any name conflicts between files created by different users, we are asked to add the name of the user doing a backup to the backup file name. Thus, the new format is **backup_H_U.zip**, where H is still the hostname and U is the user name. Our Statement for the pattern must change as well to include this information. Of course, we are trying to use the anonymous input again as a proven technique and we end up with:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserName()
{
  //GIVEN
  var hostname = AnyString();
  var userName = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal($"backup_{hostname}_{userName}.zip", name);
}

public string AnyString()
{
  return "MY_HOSTNAME";
}
```

Now, we can see that there is something wrong with this Statement. `AnyString()` is used twice and each time it returns the same value, which means that evaluating the Statement does not give us any guarantee, that both arguments of the `ApplyTo()` method are used and that they are used in the correct places. For example, the Statement will be considered true when user name value is used in place of a hostname by the `ApplyTo()` method. This means that if we still want to use the anonymous input effectively without running into false positives[27], we have to make the two values distinct, e.g. like this:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserName()
{
  //GIVEN
  var hostname = AnyString1();
  var userName = AnyString2(); //different value
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal($"backup_{hostname}_{userName}.zip", name);
}

public string AnyString1()
{
```

[27] A “false positive” is a test that should be failing but is passing.

```
18    return "MY_HOSTNAME";
19  }
20
21  public string AnyString2()
22  {
23    return "MY_USER_NAME";
24  }
```

We solved the problem (for now) by introducing another helper method. However, as you can see, this is not a very scalable solution. Thus, let's try to reduce the number of helper methods for string generation to one and make it return a different value each time:

```
1   [Fact] public void
2   ShouldCreateBackupFileNameContainingPassedHostnameAndUserName()
3   {
4     //GIVEN
5     var hostname = AnyString();
6     var userName = AnyString();
7     var fileNamePattern = new BackupFileNamePattern();
8
9     //WHEN
10    var name = fileNamePattern.ApplyTo(hostname, userName);
11
12    //THEN
13    Assert.Equal($"backup_{hostname}_{userName}.zip", name);
14  }
15
16  public string AnyString()
17  {
18    return Guid.NewGuid().ToString();
19  }
```

This time, the `AnyString()` method returns a GUID instead of a human-readable text. Generating a new GUID each time gives us a fairly strong guarantee that each value would be distinct. The string not being human-readable (contrary to something like `"MY_HOSTNAME"`) may leave you worried that maybe we are losing something, but hey, didn't we say **Any**String()?

Distinct generated values means that each time we generate an anonymous value, it's different (if possible) than the previous one and each such value is generated automatically using some kind of heuristics.

Fourth technique: Constant Specification

Let's consider another modification that we are requested to make – this time, the backup file name needs to contain the version number of our application as well. Remembering that

we want to use the derived values technique, we will not hardcode the version number into our Statement. Instead, we will use a constant that's already defined somewhere else in the application's production code (this way we also avoid duplication of this version number across the application). Let's imagine this version number is stored as a constant called `Number` in `Version` class. The Statement updated for the new requirements looks like this:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostnameAndUserNameAndVersion()
{
  //GIVEN
  var hostname = AnyString();
  var userName = AnyString();
  var fileNamePattern = new BackupFileNamePattern();

  //WHEN
  var name = fileNamePattern.ApplyTo(hostname, userName);

  //THEN
  Assert.Equal(
    $"backup_{hostname}_{userName}_{Version.Number}.zip", name);
}

public string AnyString()
{
  return Guid.NewGuid().ToString();
}
```

Again, rather than a literal value of something like `5.0`, I used the `Version.Number` constant which holds the value. This allowed me to use a derived value in the assertion but left me a little worried about whether the `Version.Number` itself is correct – after all, I used the production code constant for the creation of expected value. If I accidentally modified this constant in my code to an invalid value, the Statement would still be considered true, even though the behavior itself would be wrong.

To keep everyone happy, I usually solve this dilemma by writing a single Statement just for the constant to specify what the value should be:

```
public class VersionSpecification
{
 [Fact] public void
 ShouldContainNumberEqualTo1_0()
 {
   Assert.Equal("1.0", Version.Number);
 }
}
```

By doing so, I made sure that there is a Statement that will turn false whenever I accidentally change the value of `Version.Number`. This way, I don't have to worry about it in the rest of the

Specification. As long as this Statement holds, the rest can use the constant from the production code without worries.

Summary of the example

By showing you this example, I tried to demonstrate how a style can evolve from the principles we believe in and the constraints we encounter when applying those principles. I did so for two reasons:

1. To introduce to you a set of techniques (although it would be more accurate to use the word "patterns") I use and recommend. Giving an example was the best way of describing them fluently and logically that I could think of.
2. To help you better communicate with people that are using different styles. Instead of just throwing "you are doing it wrong" at them, consider understanding their principles and how their techniques of choice support those principles.

Now, let's take a quick summary of all the techniques introduced in the backup file name example:

Derived Values
: I define my expected output in terms of the input to document the relationship between input and output.

Anonymous Input
: When I want to document the fact that a particular value is not relevant for the current Statement, I use a special method that produces the value for me. I name this method after the equivalence class that I need it to belong to (e.g. `Any.AlphaNumericString()`) and this way, I make my Statement agnostic of what particular value is used.

Distinct Generated Values
: When using anonymous input, I generate a distinct value each time (in case of types that have very few values, like boolean, try at least not to generate the same value twice in a row) to make the Statement more reliable.

Constant Specification
: I write a separate Statement for a constant to specify what its value should be. This way, I can use the constant instead of its literal value in all the other Statements to create a Derived Value without the risk that changing the value of the constant would not be detected by my executable Specification.

Constrained non-determinism

When we combine anonymous input with distinct generated values, we get something that is called **Constrained Non-Determinism**. This is a term coined by Mark Seemann[28] and means three things:

[28] http://blog.ploeh.dk/2009/03/05/ConstrainedNon-Determinism/

1. Values are anonymous i.e. we don't know the actual value we are using.
2. The values are generated in as distinct as possible sequence (which means that, whenever possible, no two values generated one after another hold the same value)
3. The non-determinism in generation of the values is constrained, which means that the algorithms for generating values are carefully picked to provide values that belong to a specific equivalence class and that are not "evil" (e.g. when generating "any integer", we'd rather not allow generating '0' as it is usually a special-case-value that often deserves a separate Statement).

There are multiple ways to implement constrained non-determinism. Mark Seemann himself has invented the AutoFixture library for C# that is freely available to download[29]. Here is the shortest possible snippet to generate an anonymous integer using AutoFixture:

```
1 Fixture fixture = new Fixture();
2 var anonymousInteger = fixture.Create<int>();
```

I, on the other hand, follow Amir Kolsky and Scott Bain, who recommend using `Any` class as seen in the previous chapters of this book. `Any` takes a slightly different approach than AutoFixture (although it uses AutoFixture internally). My implementation of `Any` class is available to download as well[30].

Summary

I hope that this chapter gave you some understanding of how different TDD styles came into existence and why I use some of the techniques I do (and how these techniques are not just a series of random choices). In the next chapters, I will try to introduce some more techniques to help you grow a bag of neat tricks – a coherent style[31].

[29] https://github.com/AutoFixture/AutoFixture
[30] https://github.com/grzesiek-galezowski/tdd-toolkit
[31] For the biggest collection of such techniques, or more precisely, patterns, see XUnit Test Patterns by Gerard Meszaros.

Specifying functional boundaries and conditions

A Disclaimer

Before I begin, I have to disclaim that this chapter draws from a series of posts by Scott Bain and Amir Kolsky from the blog Sustainable Test-Driven Development and their upcoming book by the same title. I like how they adapt the idea of boundary testing[32] so much that I learned to follow their guidelines. This chapter is going to be a rephrase of these guidelines. I encourage you to read the original blog posts on this subject on http://www.sustainabletdd.com/ (and buy the upcoming book by Scott, Amir and Max Guernsey).

Sometimes, an anonymous value is not enough

In the last chapter, I described how anonymous values are useful when we specify a behavior that should be the same no matter what arguments we pass to the constructor or invoked methods. An example would be pushing an integer onto a stack and popping it back to see whether it's the same item we pushed – the behavior is consistent for whatever number we push and pop:

```
1  [Fact] public void
2  ShouldPopLastPushedItem()
3  {
4    //GIVEN
5    var lastPushedItem = Any.Integer();
6    var stack = new Stack<int>();
7    stack.Push(Any.Integer());
8    stack.Push(Any.Integer());
9    stack.Push(lastPushedItem);
10
11   //WHEN
12   var poppedItem = stack.Pop();
13
14   //THEN
15   Assert.Equal(lastPushedItem, poppedItem);
16 }
```

[32] https://en.wikipedia.org/wiki/Boundary_testing

In this case, the values of the first two integer numbers pushed on the stack do not matter – the described relationship between input and output is independent of the actual values we use. As we saw in the last chapter, this is the typical case where we would apply Constrained Non-Determinism.

Sometimes, however, specified objects exhibit different behaviors based on what is passed to their constructors or methods or what they get by calling other objects. For example:

- in our application, we may have a licensing policy where a feature is allowed to be used only when the license is valid, and denied after it has expired. In such a case, the behavior before the expiry date is different than after – the expiry date is the functional behavior boundary.
- Some shops are open from 10 AM to 6 PM, so if we had a query in our application whether the shop is currently open, we would expect it to be answered differently based on what the current time is. Again, the open and closed dates are functional behavior boundaries.
- An algorithm calculating the absolute value of an integer number returns the same number for inputs greater than or equal to `0` but negated input for negative numbers. Thus, `0` marks the functional boundary in this case.

In such cases, we need to carefully choose our input values to gain a sufficient confidence level while avoiding overspecifying the behaviors with too many Statements (which usually introduces penalties in both Specification run time and maintenance). Scott and Amir build on the proven practices from the testing community[33] and give us some advice on how to pick the values. I'll describe these guidelines (slightly modified in several places) in three parts:

1. specifying exceptions to the rules – where behavior is the same for every input values except one or more explicitly specified values,
2. specifying boundaries
3. specifying ranges – where there are more boundaries than one.

Exceptions to the rule

There are times when a Statement is true for every value except one (or several) explicitly specified. My approach varies a bit depending on the set of possible values and the number of exceptions. I'm going to give you three examples to help you understand these variations better.

Example 1: a single exception from a large set of values

In some countries, some digits are avoided e.g. in floor numbers in some hospitals and hotels due to some local superstitions or just sounding similar to another word that has a very negative meaning. One example of this is /tetraphobia/[34], which leads to skipping the digit `4`, as in some languages, it sounds similar to the word "death". In other words, any number containing `4` is

[33] see e.g. chapter 4.3 of ISQTB Foundation Level Syllabus at http://www.istqb.org/downloads/send/2-foundation-level-documents/3-foundation-level-syllabus-2011.html4

[34] https://en.wikipedia.org/wiki/Tetraphobia

avoided and when you enter the building, you might not find a fourth floor (or fourteenth). Let's imagine we have several such rules for our hotels in different parts of the world and we want the software to tell us if a certain digit is allowed by local superstitions. One of these rules is to be implemented by a class called `Tetraphobia`:

```
public class Tetraphobia : LocalSuperstition
{
  public bool Allows(char number)
  {
    throw new NotImplementedException("not implemented yet");
  }
}
```

It implements the `LocalSuperstition` interface which has an `Allows()` method, so for the sake of compile-correctness, we had to create the class and the method. Now that we have it, we want to test-drive the implementation. What Statements do we write?

Obviously we need a Statement that says what happens when we pass a disallowed digit:

```
[Fact] public void
ShouldReject4()
{
  //GIVEN
  var tetraphobia = new Tetraphobia();

  //WHEN
  var isFourAccepted = tetraphobia.Allows('4');

  //THEN
  Assert.False(isFourAccepted);
}
```

Note that we use the specific value for which the exceptional behavior takes place. Still, it may be a very good idea to extract `4` into a constant. In one of the previous chapters, I described a technique called **Constant Specification**, where we write an explicit Statement about the value of the named constant and use the named constant itself everywhere else instead of its literal value. So why did I not use this technique this time? The only reason is that this might have looked a little bit silly with such an extremely trivial example. In reality, I should have used the named constant. Let's do this exercise now and see what happens.

```
[Fact] public void
ShouldRejectSuperstitiousValue()
{
  //GIVEN
  var tetraphobia = new Tetraphobia();

  //WHEN
  var isSuperstitiousValueAccepted =
    tetraphobia.Allows(Tetraphobia.SuperstitiousValue);

  //THEN
  Assert.False(isSuperstitiousValueAccepted);
}
```

When we do that, we have to document the named constant with the following Statement:

```
[Fact] public void
ShouldReturn4AsSuperstitiousValue()
{
  Assert.Equal('4', Tetraphobia.SuperstitiousValue);
}
```

Time for a Statement that describes the behavior for all non-exceptional values. This time, we are going to use a method of the `Any` class named `Any.OtherThan()`, that generates any value other than the one specified (and produces nice, readable code as a side effect):

```
[Fact] public void
ShouldAcceptNonSuperstitiousValue()
{
  //GIVEN
  var tetraphobia = new Tetraphobia();

  //WHEN
  var isNonSuperstitiousValueAccepted =
    tetraphobia.Allows(Any.OtherThan(Tetraphobia.SuperstitiousValue);

  //THEN
  Assert.True(isNonSuperstitiousValueAccepted);
}
```

and that's it – I don't usually write more Statements in such cases. There are so many possible input values that it would not be rational to specify all of them. Drawing from Kent Beck's famous comment from Stack Overflow[35], I think that our job is not to write as many Statements as we can, but as little as possible to truly document the system and give us a desired level of confidence.

[35]https://stackoverflow.com/questions/153234/how-deep-are-your-unit-tests/153565#153565

Example 2: a single exception from a small set of values

The situation is different, however, when the exceptional value is chosen from a small set – this is often the case where the input value type is an enumeration. Let's go back to an example from one of our previous chapters, where we specified that there is some kind of reporting feature and it can be accessed by either an administrator role or by an auditor role. Let's modify this example for now and say that only administrators are allowed to access the reporting feature:

```
[Fact] public void
ShouldGrantAdministratorsAccessToReporting()
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(Roles.Admin);

 //THEN
 Assert.True(accessGranted);
}
```

The approach to this part is no different than what I did in the first example – I wrote a Statement for the single exceptional value. Time to think about the other Statement – the one that specifies what should happen for the rest of the roles. I'd like to describe two ways this task can be tackled.

The first way is to do it like in the previous example – pick a value different than the exceptional one. This time we will use `Any.OtherThan()` method, which is suited for such a case:

```
[Fact] public void
ShouldDenyAnyRoleOtherThanAdministratorAccessToReporting()
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(Any.OtherThan(Roles.Admin));

 //THEN
 Assert.False(accessGranted);
}
```

This approach has two advantages:

1. Only one Statement is executed for the "access denied" case, so there is no significant run time penalty.

2. In case we expand our enum in the future, we don't have to modify this Statement – the added enum member will get a chance to be picked up.

However, there is also one disadvantage – we can't be sure the newly added enum member is used in this Statement. In the previous example, we didn't care that much about the values that were used, because:

- `char` range was quite large so specifying the behaviors for all the values could prove troublesome and inefficient given our desired confidence level,
- `char` is a fixed set of values – we can't expand `char` as we expand enums, so there is no need to worry about the future.

So what if there are only two more roles except `Roles.Admin`, e.g. `Auditor` and `CasualUser`? In such cases, I sometimes write a Statement that's executed against all the non-exceptional values, using xUnit.net's `[Theory]` attribute that allows me to execute the same Statement code with different sets of arguments. An example here would be:

```
1  [Theory]
2  [InlineData(Roles.Auditor)]
3  [InlineData(Roles.CasualUser)]
4  public void
5  ShouldDenyAnyRoleOtherThanAdministratorAccessToReporting(Roles role)
6  {
7   //GIVEN
8   var access = new Access();
9
10  //WHEN
11  var accessGranted
12      = access.ToReportingIsGrantedTo(role);
13
14  //THEN
15  Assert.False(accessGranted);
16 }
```

The Statement above is executed for both `Roles.Auditor` and `Roles.CasualUser`. The downside of this approach is that each time we expand an enumeration, we need to go back here and update the Statement. As I tend to forget such things, I try to keep at most one Statement in the system depending on the enum – if I find more than one place where I vary my behavior based on values of a particular enumeration, I change the design to replace enum with polymorphism. Statements in TDD can be used as a tool to detect design issues and I'll provide a longer discussion on this in a later chapter.

Example 3: More than one exception

The previous two examples assume there is only one exception to the rule. However, this concept can be extended to more values, as long as it is a finished, discrete set. If multiple exceptional

values produce the same behavior, I usually try to cover them all by using the mentioned `[Theory]` feature of xUnit.net. I'll demonstrate it by taking the previous example of granting access and assuming that this time, both administrators and auditors are allowed to use the feature. A Statement for behavior would look like this:

```
[Theory]
[InlineData(Roles.Admin)]
[InlineData(Roles.Auditor)]
public void
ShouldAllowAccessToReportingBy(Roles role)
{
 //GIVEN
 var access = new Access();

 //WHEN
 var accessGranted
     = access.ToReportingIsGrantedTo(role);

 //THEN
 Assert.False(accessGranted);
}
```

In the last example, I used this approach for the non-exceptional values, saying that this is what I sometimes do. However, when specifying multiple exceptions to the rule, this is my default approach – the nature of the exceptional values is that they are strictly specified, so I want them all to be included in my Specification.

This time, I'm not showing you the Statement for non-exceptional values as it follows the approach I outlined in the previous example.

Rules valid within boundaries

Sometimes, a behavior varies around a boundary. A simple example would be a rule on how to determine whether someone is an adult or not. One is usually considered an adult after reaching a certain age, let's say, of 18. Pretending we operate at the granule of years (not taking months into account), the rule is:

1. When one's age in years is less than 18, they are considered not an adult.
2. When one's age in years is at least 18, they are considered an adult.

As you can see, there is a boundary between the two behaviors. The "right edge" of this boundary is 18. Why do I say "right edge"? That is because the boundary always has two edges, which, by the way, also means it has a length. If we assume we are talking about the mentioned adult age rule and that our numerical domain is that of integer numbers, we can as well use `17` instead of `18` as edge value and say that:

1. When one's age in years is at most 17, they are considered not an adult.
2. When one's age in years is more than 17, they are considered an adult.

So a boundary is not a single number – it always has a length – the length between last value of the previous behavior and the first value of the next behavior. In the case of our example, the length between `17` (left edge – last non-adult age) and `18` (right edge – first adult value) is `1`.

Now, imagine that we are not talking about integer values anymore, but we treat the time as what it is – a continuum. Then the right edge value would still be 18 years. But what about the left edge? It would not be possible for it to stay 17 years, as the rule would apply to e.g. 17 years and 1 day as well. So what is the correct right edge value and the correct length of the boundary? Would the left edge need to be 17 years and 11 months? Or 17 years, 11 months, 365/366 days (we have the leap year issue here)? Or maybe 17 years, 11 months, 365/366 days, 23 hours, 59 minutes, etc.? This is harder to answer and depends heavily on the context – it must be answered for each particular case based on the domain and the business needs – this way we know what kind of precision is expected of us.

In our Specification, we have to document the boundary length somehow. This brings an interesting question: how to describe the boundary length with Statements? To illustrate this, I want to show you two Statements describing the mentioned adult age calculation expressed using the granule of years (so we leave months, days, etc. out).

The first Statement is for values smaller than 18 and we want to specify that for the left edge value (i.e. `17`), but calculated relative to the right edge value (i.e. instead of writing `17`, we write `18-1`):

```
[Fact] public void
ShouldNotBeSuccessfulForAgeLessThan18()
{
  //GIVEN
  var detection = new AdultAgeDetection();
  var notAnAdult = 18 - 1; //more on this later

  //WHEN
  var isSuccessful = detection.PerformFor(notAnAdult);

  //THEN
  Assert.False(isSuccessful);
}
```

And the next Statement for values greater than or equal to 18 and we want to use the right edge value:

```
[Fact] public void
ShouldBeSuccessfulForAgeGreaterThanOrEqualTo18()
{
  //GIVEN
  var detection = new AdultAgeDetection();
  var adult = 18;

  //WHEN
  var isSuccessful = detection.PerformFor(adult);

  //THEN
  Assert.True(isSuccessful);
}
```

There are two things to note about these examples. The first one is that I didn't use any kind of `Any` methods. I use `Any` in cases where I don't have a boundary or when I consider no value from an equivalence class better than others in any particular way. When I specify boundaries, however, instead of using methods like `Any.IntegerGreaterOrEqualTo(18)`, I use the edge values as I find that they more strictly define the boundary and drive the right implementation. Also, explicitly specifying the behaviors for the edge values allows me to document the boundary length.

The second thing to note is the usage of literal constant `18` in the example above. In one of the previous chapters, I described a technique called **Constant Specification** which is about writing an explicit Statement about the value of the named constant and use the named constant everywhere else instead of its literal value. So why didn't I use this technique this time?

The only reason is that this might have looked a little bit silly with such an extremely trivial example as detecting adult age. In reality, I should have used the named constant in both Statements and it would show the boundary length even more clearly. Let's perform this exercise now and see what happens.

First, let's document the named constant with the following Statement:

```
[Fact] public void
ShouldIncludeMinimumAdultAgeEqualTo18()
{
  Assert.Equal(18, Age.MinimumAdult);
}
```

Now we've got everything we need to rewrite the two Statements we wrote earlier. The first would look like this:

```
[Fact] public void
ShouldNotBeSuccessfulForLessThanMinimumAdultAge()
{
  //GIVEN
  var detection = new AdultAgeDetection();
  var notAnAdultYet = Age.MinimumAdult - 1;

  //WHEN
  var isSuccessful = detection.PerformFor(notAnAdultYet);

  //THEN
  Assert.False(isSuccessful);
}
```

And the next Statement for values greater than or equal to 18 would look like this:

```
[Fact] public void
ShouldBeSuccessfulForAgeGreaterThanOrEqualToMinimumAdultAge()
{
  //GIVEN
  var detection = new AdultAgeDetection();
  var adultAge = Age.MinimumAdult;

  //WHEN
  var isSuccessful = detection.PerformFor(adultAge);

  //THEN
  Assert.True(isSuccessful);
}
```

As you can see, the first Statement contains the following expression:

```
Age.MinimumAdult - 1
```

where `1` is the exact length of the boundary. As I mentioned earlier, the example is so trivial that it may look silly and funny, however, in real-life scenarios, this is a technique I apply anytime, anywhere.

Boundaries may look like they apply only to numeric input, but they occur at many other places. There are boundaries associated with date/time (e.g. the adult age calculation would be this kind of case if we didn't stop at counting years but instead considered time as a continuum – the decision would need to be made whether we need precision in seconds or maybe in ticks), or strings (e.g. validation of user name where it must be at least 2 characters, or password that must contain at least 2 special characters). They also apply to regular expressions. For example, for a simple regex `\d+`, we would surely specify for at least three values: an empty string, a single digit, and a single non-digit.

Combination of boundaries – ranges

The previous examples focused on a single boundary. So, what about a situation when there are more, i.e. a behavior is valid within a range?

Example – driving license

Let's consider the following example: we live in a country where a citizen can get a driving license only after their 18th birthday, but before 65th (the government decided that people after 65 may have worse sight and that it's safer not to give them new driving licenses). Let's assume that we are trying to develop a class that answers the question of whether we can apply for a driving license and the values returned by this query is as follows:

1. Age < 18 – returns enum value `QueryResults.TooYoung`
2. 18 <= age <= 65 – returns enum value `QueryResults.AllowedToApply`
3. Age > 65 – returns enum value `QueryResults.TooOld`

Now, remember I wrote that I specify the behaviors with boundaries by using the edge values? This approach, when applied to the situation I just described, would give me the following Statements:

1. Age = 17, should yield result `QueryResults.TooYoung`
2. Age = 18, should yield result `QueryResults.AllowedToApply`
3. Age = 65, should yield result `QueryResults.AllowedToApply`
4. Age = 66, should yield result `QueryResults.TooOld`

thus, I would describe the behavior where the query should return `AllowedToApply` value twice. This is not a big issue if it helps me document the boundaries.

The first Statement says what should happen up to the age of 17:

```
[Fact]
public void ShouldRespondThatAgeLessThan18IsTooYoung()
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(18-1);

  //THEN
  Assert.Equal(QueryResults.TooYoung, result);
}
```

The second Statement tells us that the range of 18 – 65 is where a citizen is allowed to apply for a driving license. I write it as a theory (again using the `[InlineData()]` attribute of xUnit.net) because this range has two boundaries around which the behavior changes:

```
[Theory]
[InlineData(18, QueryResults.AllowedToApply)]
[InlineData(65, QueryResults.AllowedToApply)]
public void ShouldRespondThatDrivingLicenseCanBeAppliedForInRangeOf18To65(
  int age, QueryResults expectedResult
)
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(age);

  //THEN
  Assert.Equal(expectedResult, result);
}
```

The last Statement specifies what should be the response when someone is older than 65:

```
[Fact]
public void ShouldRespondThatAgeMoreThan65IsTooOld()
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(65+1);

  //THEN
  Assert.Equal(QueryResults.TooOld, result);
}
```

Note that I used 18-1 and 65+1 instead of 17 and 66 to show that 18 and 65 are the boundary values and that the lengths of the boundaries are, in both cases, 1. Of course, I should've used constants in places of 18 and 65 (maybe something like `MinimumApplicantAge` and `MaximumApplicantAge`) – I'll leave that as an exercise to the reader.

Example – setting an alarm

In the previous example, we were quite lucky because the specified logic was purely functional (i.e. it returned different results based on different inputs). Thanks to this, when writing out the theory for the age range of 18-65, we could parameterize input values together with expected results. This is not always the case. For example, let's imagine that we have a `Clock` class that allows us to schedule an alarm. The class allows us to set the hour safely between 0 and 24, otherwise, it throws an exception.

This time, I have to write two parameterized Statements – one where a value is returned (for valid cases) and one where the exception is thrown (for invalid cases). The first would look like this:

```
[Theory]
[InlineData(Hours.Min)]
[InlineData(Hours.Max)]
public void
ShouldBeAbleToSetHourBetweenMinAndMax(int inputHour)
{
  //GIVEN
  var clock = new Clock();
  clock.SetAlarmHour(inputHour);

  //WHEN
  var setHour = clock.GetAlarmHour();

  //THEN
  Assert.Equal(inputHour, setHour);
}
```

and the second:

```
[Theory]
[InlineData(Hours.Min-1)]
[InlineData(Hours.Max+1)]
public void
ShouldThrowOutOfRangeExceptionWhenTryingToSetAlarmHourOutsideValidRange(
  int inputHour)
{
  //GIVEN
  var clock = new Clock();

  //WHEN - THEN
  Assert.Throws<OutOfRangeException>(
    ()=> clock.SetAlarmHour(inputHour)
  );
}
```

Other than that, I used the same approach as the last time.

Summary

In this chapter, I described specifying functional boundaries with a minimum amount of code and Statements, so that the Specification is more maintainable and runs faster. There is one more

kind of situation left: when we have compound conditions (e.g. a password must be at least 10 characters and contain at least 2 special characters) – we'll get back to those when we introduce mock objects.

Driving the implementation from Specification

As one of the last topics of the core TDD techniques that don't require us to delve into the object-oriented design world, I'd like to show you three techniques for turning a false Statement true. The names of the techniques come from a book by Kent Beck, Test-Driven Development: By Example[36] and are:

1. Type the obvious implementation
2. Fake it ('til you make it)
3. Triangulate

Don't worry if these names don't tell you anything, the techniques are not that difficult to grasp and I will try to give an example of each of them.

Type the obvious implementation

The first technique simply says: when you know the correct and final implementation to turn a Statement true, then just type it. If the implementation is obvious, this approach makes a lot of sense - after all, the number of Statements required to specify (and test) a functionality should reflect our desired level of confidence. If this level is very high, we can just type the correct code in response to a single Statement. Let's see it in action on a trivial example of adding two numbers:

```
1  [Fact] public void
2  ShouldAddTwoNumbersTogether()
3  {
4    //GIVEN
5    var addition = new Addition();
6
7    //WHEN
8    var sum = addition.Of(3,5);
9
10   //THEN
11   Assert.Equal(8, sum);
12 }
```

[36] https://isbnsearch.org/isbn/9780321146533

You may remember that in one of the previous chapters I wrote that we usually should write the simplest production code that would make the Statement true. The mentioned approach would encourage us to just return `8` from the `Of()` method because it would be sufficient to make the Statement true. Instead of doing that, however, we may decide that the logic is so obvious, that we can just type it in its final form:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

and that's it. Note that I didn't use Constrained Non-Determinism in the Statement, because using it kind of enforces using the "type the obvious implementation" approach. This is also one of the reasons that many Statements I wrote so far in the previous chapters were implemented by typing the correct implementation. Just to illustrate it, let's take a look at how the above Statement would look if I used Constrained Non-Determinism:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var a = Any.Integer();
  var b = Any.Integer();
  var addition = new Addition();

  //WHEN
  var sum = addition.Of(a,b);

  //THEN
  Assert.Equal(a + b, sum);
}
```

The most obvious implementation that would make this Statement true is the correct implementation – I can't get away with returning a constant value as I could when I didn't use Constrained Non-Determinism. This is because this time I just don't know what the expected result is as it is strictly dependent on the input values which I don't know as well.

Fake it ('til you make it)

The second technique made me smile when I first learned about it. I don't recall myself ever using it in real production code, yet I find it so interesting that I want to show it to you anyway. It is so simple you will not regret these few minutes even if just for broadening your horizons.

Let's assume we already have a false Statement written and are about to make it true by writing production code. At this moment, we apply *fake it ('till you make it)* in two steps:

1. We start with a "fake it" step. Here, we turn a false Statement true by using the most obvious implementation possible, even if it's not the correct implementation (hence the name of the step - we "fake" the real implementation to "cheat" the Statement). Usually, returning a literal constant is enough at the beginning.
2. Then we proceed with the "make it" step - we rely on our sense of duplication between the Statement and (fake) implementation to gradually transform both into their more general forms that eliminate this duplication. Usually, we achieve this by changing constants into variables, variables into parameters, etc.

An example would be handy just about now, so let's apply *fake it...* to the same addition example as in the *type the obvious implementation* section. The Statement looks the same as before:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var addition = new Addition();

  //WHEN
  var sum = addition.Of(3, 5);

  //THEN
  Assert.Equal(8, sum);
}
```

For the implementation, however, we are going to use the most obvious code that will turn the Statement true. As mentioned, this most obvious implementation is almost always returning a constant:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 8; //we faked the real implementation
  }
}
```

The Statement turns true (green) now, even though the implementation is obviously wrong. Now is the time to remove duplication between the Statement and the production code.

First, let's note that the number `8` is duplicated between Statement and implementation – the implementation returns it and the Statement asserts on it. To reduce this duplication, let's break the `8` in the implementation into an addition:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 3 + 5;
  }
}
```

Note the smart trick I did. I changed the duplication between implementation and *expected result* of the Statement to duplication between implementation and *input values* of the Statement. I changed the production code to use

```
return 3 + 5;
```

exactly because the Statement used these two values like this:

```
var sum = addition.Of(3, 5);
```

This kind of duplication is different from the previous one in that it can be removed using parameters (this applies not only to input parameters of a method but to anything we have access to before triggering specified behavior – constructor parameters, fields, etc. in contrast to result which we normally don't know until we invoke the behavior). The duplication of number 3 can be eliminated by changing the production code to use the value passed from the Statement. So this:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 3 + 5;
  }
}
```

Is transformed into this:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + 5;
  }
}
```

This way we eliminated the duplication of number 3 - we used a method parameter to transfer the value of 3 from Statement to the `Of()` implementation, so we have it in a single place now. After this transformation, we only have the number 5 left duplicated, so let's transform it the same way we transformed 3:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

And that's it - we arrived at the correct implementation. I used a trivial example, since I don't want to spend too much time on this, but you can find more advanced ones in Kent Beck's book if you like.

Triangulate

Triangulation is considered the most conservative technique of the described trio, because following it involves the tiniest possible steps to arrive at the right solution. The term *Triangulation* seems mysterious at first - at least it was to me, especially that it didn't bring anything related to software engineering to my mind. The name was taken from radar triangulation[37] where outputs from at least two radars must be used to determine the position of a unit. Also, in radar triangulation, the position is measured indirectly, by combining the following data: range (not position!) between two radars, measurement done by each radar and the positions of the radars (which we know, because we are the ones who put the radars there). From this data, we can derive a triangle, so we can use trigonometry to calculate the position of the third point of the triangle, which is the desired position of the unit (two remaining points are the positions of radars). Such measurement is indirect in nature, because we don't measure the position directly, but calculate it from other helper measurements.

These two characteristics: indirect measurement and using at least two sources of information are at the core of TDD triangulation. Here's how it can be translated from radars to code:

1. **Indirect measurement**: in code, it means we derive the internal implementation and design of a module from several known examples of its desired externally visible behavior by looking at what varies in these examples and changing the production code so that this variability is handled generically. For example, variability might lead us from changing a constant to a variable, because several different examples use different input values.
2. **Using at least two sources of information**: in code, it means we start with the simplest possible implementation of behavior and make it more general **only** when we have two or more different examples of this behavior (i.e. Statements that describe the desired functionality for several different inputs). Then new examples can be added and generalization can be done again. This process is repeated until we reach the desired implementation. Robert C. Martin developed a maxim on this, saying that "As the tests get more specific, the code gets more generic"[38].

[37] http://encyclopedia2.thefreedictionary.com/radar+triangulation
[38] http://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

Usually, when TDD is showcased on simple examples, triangulation is the primary technique used, so many novices mistakenly believe TDD is all about triangulation.

I consider it an important technique because:

1. Many TDD practitioners use it and demonstrate it, so I assume you will see it sooner or later and most likely have questions regarding it.
2. It allows us to arrive at the right implementation by taking very tiny steps (tiniest than any you have seen so far in this book) and I find it very useful when I'm uncertain about how the correct implementation and design should look like.

Example 1 - adding numbers

Before I show you a more advanced example of triangulation, I would like to get back to our toy example of adding two integer numbers. This will allow us to see how triangulation differs from the other two techniques mentioned earlier.

For writing the examples, we will use the xUnit.net's feature of parameterized Statements, i.e. theories - this will allow us to give many examples of the desired functionality without duplicating the code.

The first example looks like this:

```
1  [Theory]
2  [InlineData(0,0,0)]
3  public void ShouldAddTwoNumbersTogether(
4    int addend1,
5    int addend2,
6    int expectedSum)
7  {
8    //GIVEN
9    var addition = new Addition();
10
11   //WHEN
12   var sum = addition.Of(addend1, addend2);
13
14   //THEN
15   Assert.Equal(expectedSum, sum);
16 }
```

Note that we parameterized not only the input values but also the expected result (`expectedSum`). The first example specifies that `0 + 0 = 0`.

The implementation, similarly to *fake it ('till you make it)* is, for now, to just return a constant:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return 0;
  }
}
```

Now, contrary to *fake it...* technique, we don't try to remove duplication between the Statement and the code. Instead, we add another example of the same rule. What do I mean by "the same rule"? Well, we need to consider our axes of variability. In the addition operation, two things can vary - either the first addend, or the second - thus, we have two axes of variability. For our second example, we need to keep one of them unchanged while changing the other. Let's say that we decide to keep the second input value the same as in the previous example (which is 0) and change the first value to 1. So this single example:

```
[Theory]
[InlineData(0,0,0)]
```

Becomes a set of two examples:

```
[Theory]
[InlineData(0,0,0)]
[InlineData(1,0,1)] //NEW!
```

Again, note that the second input value stays the same in both examples and the first one varies. The expected result needs to be different as well.

As for the implementation, we still try to make the Statement true by using as dumb implementation as possible:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 1) return 1;
    return 0;
  }
}
```

We already have two examples, so if we see a repeating pattern, we may try to generalize it. Let's assume, however, that we don't have an idea on how to generalize the implementation yet, so let's add a third example:

```
[Theory]
[InlineData(0,0,0)]
[InlineData(1,0,1)]
[InlineData(2,0,2)]
```

And the implementation is expanded to:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return 2;
    if(a == 1) return 1;
    return 0;
  }
}
```

Now, looking at this code, we can notice a pattern - for every input values so far, we return the value of the first one: for 1 we return 1, for 2 we return 2, for 0 we return 0. Thus, we can generalize this implementation. Let's generalize only the part related to the handling number 2 to see whether the direction is right:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return a; //changed from 2 to a
    if(a == 1) return 1;
    return 0;
  }
}
```

The examples should still be true at this point, so we haven't broken the existing code. Time to change the second `if` statement:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return a;
    if(a == 1) return a; //changed from 1 to a
    return 0;
  }
}
```

We still have the green bar, so the next step would be to generalize the `return 0` part to `return a`:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(a == 2) return a;
    if(a == 1) return a;
    return a; //changed from 0 to a
  }
}
```

The examples should still be true. By the way, triangulation doesn't force us to take as tiny steps as in this case, however, I wanted to show you that it makes it possible. The ability to take smaller steps when needed is something I value a lot when I use TDD. Anyway, we can notice that each of the conditions ends with the same result, so we don't need the conditions at all. We can remove them and leave only:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a;
  }
}
```

Thus, we have generalized the first axis of variability, which is the first addend. Time to vary the second one, by leaving the first addend unchanged. To the following existing examples:

```
[Theory]
[InlineData(0,0,0)] //0+0=0
[InlineData(1,0,1)] //1+0=1
[InlineData(2,0,2)] //2+0=2
```

We add the following one:

```
[InlineData(2,1,3)] //2+1=3
```

Note that we already used the value of 2 for the first addend in one of the previous examples, so this time we decide to freeze it and vary the second addend, which has so far always been 0. The implementation would be something like this:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(b == 1)
    {
      return a + 1;
    }
    else
    {
      return a;
    }
  }
}
```

We already have two examples for the variation of the second addend, so we could generalize. Let's say, however, we don't see the pattern yet. We add another example for a different value of second addend:

```
[Theory]
[InlineData(0,0,0)] //0+0=0
[InlineData(1,0,1)] //1+0=1
[InlineData(2,0,2)] //2+0=2
[InlineData(2,1,3)] //2+1=3
[InlineData(2,2,4)] //2+2=4
```

So, we added 2+2=4. Again, the implementation should be as naive as possible:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(b == 1)
    {
      return a + 1;
    }
    else if(b == 2)
    {
      return a + 2;
    }
    else
    {
      return a;
    }
  }
}
```

Now we can see the pattern more clearly. Whatever value of `b` we pass to the `Of()` method, it gets added to `a`. Let's try to generalize, this time using a little bigger step:

```
public class Addition
{
  public int Of(int a, int b)
  {
    if(b == 1)
    {
      return a + b; //changed from 1 to b
    }
    else if(b == 2)
    {
      return a + b; //changed from 2 to b
    }
    else
    {
      return a + b; //added "+ b"
    }
  }
}
```

Again, this step was bigger, because we modified three places in a single change. Remember triangulation allows us to choose the size of the step, so this time I chose a bigger one because I felt more confident. Anyway, we can see that the result for each branch is exactly the same: `a + b`, so we can remove the conditions altogether and get:

```
public class Addition
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

and there we go - we have successfully triangulated the addition function. Now, I understand that it must have felt extremely over-the-top for you to derive an obvious addition this way. Remember I did this exercise only to show you the mechanics, not to provide a solid case for triangulation usefulness.

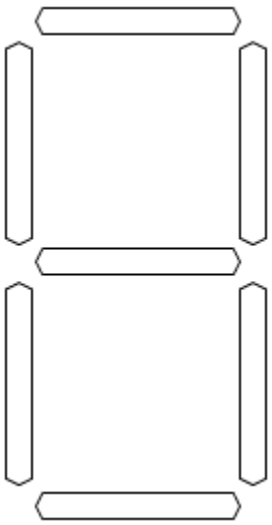
Example 2 - LED display

I don't blame you if the first example did little to convince you that triangulation can be useful. After all, that was calculating a sum of two integers! The next example is going to be something less obvious. I would like to warn you, however, that I will take my time to describe the problem

and will show you only part of the solution, so if you have enough of triangulation already, just skip this example and get back to it later.

Now that we're through with the disclaimer, here goes the description.

Imagine we need to write a class that produces a 7-segment LED display ASCII art. In real life, such displays are used to display digits:



A 7-segment LED display mockup

An example of an ASCII art that is expected from our class looks like this:

```
1 .-.
2 |.|
3 .-.
4 |.|
5 .-.
```

Note that there are three kinds of symbols:

- . means either space (there is no segment there) or a segment that is not lit.
- - means a lit horizontal segment
- | means a lit vertical segment

The functionality we need to implement should allow one to not only display numbers but to light any combination of segments at will. So, we can decide to not light any segment, thus getting the following output:

```
...
...
...
...
...
```

Or to light only the upper segment, which leads to the following output:

```
.-.
...
...
...
...
```

How do we tell our class to light this or that segment? We pass it a string of segment names. The segments are named A, B, C, D, E, F, G and the mapping of each name to a specific segment can be visualized as:

```
.A.
F.B
.G.
E.C
.D.
```

So to achieve the described earlier output where only the upper segment is lit, we need to pass the input consisting of `"A"`. If we want to light all segments, we pass `"ABCDEFG"`. If we want to keep all segments turned off, we pass `""` (or a C# equivalent: `string.Empty`).

The last thing I need to say before we begin is that for the sake of this exercise, we focus only on the valid input (e.g. we assume we won't get inputs such as "AAAA", or "abc" or "ZXVN"). Of course, in real projects, invalid input cases should be specified as well.

Time for the first Statement. For starters, I'm going to specify the case of empty input that results in all segments turned off:

```
[Theory]
[InlineData("", new [] {
  "...",
  "...",
  "...",
  "...",
  "...",
})]
public void ShouldConvertInputToAsciiArtLedDisplay(
  string input, string[] expectedOutput
)
{
```

```
13    //GIVEN
14    var asciiArts = new LedAsciiArts();
15
16    //WHEN
17    var asciiArtString = asciiArts.ConvertToLedArt(input);
18
19    //THEN
20    Assert.Equal(expectedOutput, asciiArtString);
21  }
```

Again, as I described in the previous example, on the production code side, we do the easiest thing just to make this example true. In our case, this would be:

```
1   public string[] ConvertToLedArt(string input)
2   {
3     return new [] {
4       "...",
5       "...",
6       "...",
7       "...",
8       "...",
9     };
10  }
```

The example is now implemented. Of course, this is not the final implementation of the whole conversion logic. This is why we need to choose the next example to specify. This choice will determine which axis of change we will pursue first. I decided to specify the uppermost segment (i.e. the `A` segment) - we already have an example that says when this segment is turned off, now we need one that will say what should happen when I turn it on. I will reuse the same Statement body and just add another `InlineData` attribute to execute the Statement for the new set of input and expected output:

```
1   [InlineData("A", new [] {
2     ".-.", // note the '-' character
3     "...",
4     "...",
5     "...",
6     "...",
7   })]
```

This time, I'm passing `"A"` as the input and expect to receive almost the same output as before, only that this time the first line reads ". - . " instead of ". . . ".

I implement this example using, again, the most naive and easiest to write code. The result is:

```
public string[] ConvertToLedArt(string input)
{
  if(input == "A"))
  {
    return new [] {
      ".-.",
      "...",
      "...",
      "...",
      "...",
    };
  }
  else
  {
    return new [] {
      "...",
      "...",
      "...",
      "...",
      "...",
    };
  }
}
```

The implementation is pretty dumb, but now that we have two examples, we can spot a pattern. Note that, depending on the input string, two possible results can be returned. All of the rows are the same except the first row, which, so far, is the only one that depends on the value of `input`. Thus, we can generalize the production code by extracting the duplication into something like this:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      (input == "A") ? ".-." : "...",
      "...",
      "...",
      "...",
      "...",
    };
}
```

Note that I changed the code so that only the first row depends on the `input`. This isn't over, however. When looking at the condition for the first row:

```
(input == "A") ? ".-." : "..."
```

we can further note that it's only the middle character that changes depending on what we pass. Both the left-most and the right-most character of the first row are always `.`. Thus, let's generalize even further, to end up with something like this:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      "." + ((input == "A") ? "-" : ".") + ".",
      "...",
      "...",
      "...",
      "...",
    };
}
```

Now, if we look closer at the expression:

```
((input == "A") ? "-" : ".")
```

We may note that its responsibility is to determine whether the value of the current segment based on the `input`. We can use this knowledge to extract it into a method with an intent-revealing name. The method body is:

```
public string DetermineSegmentValue(
  string input,
  string turnOnToken,
  string turnOnValue)
{
  return ((input == turnOnToken) ? turnOnValue : ".");
}
```

After this extraction, our `ConvertToLedArt` method becomes:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      "." + DetermineSegmentValue(input, "A", "-") + ".",
      "...",
      "...",
      "...",
      "...",
    };
}
```

And we're done triangulating the `A` segment.

Additional conclusions from the LED display example

The fact that I'm done triangulating along one axis of variability does not mean I can't do triangulation along other axes. For example, when we look again at the code of the `DetermineSegmentValue()` method:

```
public string DetermineSegmentValue(
  string input,
  string turnOnToken,
  string turnOnValue)
{
  return ((input == turnOnToken) ? turnOnValue : ".");
}
```

We can clearly see that the method is detecting a token by doing a direct string comparison: `input == turnOnToken`. This will fail e.g. if I pass `"AB"`, so we probably need to triangulate along this axis to arrive at the correct implementation. I won't show the steps here, but the final result of this triangulation would be something like:

```
public string DetermineSegmentValue(
  string input,
  string turnOnToken,
  string turnOnValue)
{
  return ((input.Contains(turnOnToken) ? turnOnValue : ".");
}
```

And after we do it, the `DetermineSegmentValue` method will be something we will be able to use to implement lighting other segments - no need to discover it again using triangulation for every segment. So, assuming this method is in its final form, when I write an example for the B segment, I will make it true by using the `DetermineSegmentValue()` method right from the start instead of putting an `if` first and then generalizing. The implementation will the look like this:

```
public string[] ConvertToLedArt(string input)
{
    return new [] {
      "." + DetermineSegmentValue(input, "A", "-") + ".",
      ".." + DetermineSegmentValue(input, "B", "|"),
      "...",
      "...",
      "...",
    };
}
```

So note that this time, I used the *type the obvious implementation* approach - this is because, due to previous triangulation, this step *became* obvious.

The two lessons from this are:

1. When I stop triangulating along one axis, I may still need to triangulate along the others.
2. Triangulation allows me to take smaller steps when I *need* to and when I don't, I use another approach. There are many things I don't triangulate.

I hope that, by showing you this example, I made a more compelling case for triangulation. I'd like to stop here, leaving the rest of this exercise for the reader.

Summary

In this lengthy chapter, I tried to demonstrate three techniques for going from a false Statement to a true one:

1. Type the obvious implementation
2. Fake it ('til you make it)
3. Triangulate

I hope this was an easy-to-digest introduction and if you want to know more, be sure to check Kent Beck's book, where he uses these techniques extensively on several small exercises.

Part 2: Object-Oriented World

Status: pretty stable

This chapter will probably get one big review in the far future. While I am pleased with the content, I will be looking for a better structure and wording, making little changes here and there. I may also add several sections explaining things to existing chapters on things I find were not sufficiently explained. Still, if you read it as it is now, you're not going to miss anything significant.

Most of the examples in the previous part were about a single object that did not have dependencies on other objects (with an exception of some values – strings, integers, enums, etc.). This is not how most OO systems are built. In this part, we are finally going to look at scenarios where multiple objects work together as a system.

This brings about some issues that need to be discussed. One of them is the approach to object-oriented design and how it influences the tools we use to test-drive our code. You probably heard something about a tool called mock objects (at least from one of the introductory chapters of this book) or, in a broader sense, test doubles. If you open your web browser and type "mock objects break encapsulation", you will find a lot of different opinions – some saying that mocks are great, others blaming them for all the evil in the world, and a lot of opinions that fall in between. The discussions are still heated, even though mocks were introduced more than ten years ago. My goal in this chapter is to outline the context and forces that lead to the adoption of mocks and how to use them for your benefit, not failure.

Steve Freeman, one of the godfathers of using mock objects with TDD, wrote[39]: "mocks arise naturally from doing responsibility-driven OO. All these mock/not-mock arguments are completely missing the point. If you're not writing that kind of code, people, please don't give me a hard time". I am going to introduce mocks in a way that will not give Steve a hard time, I hope.

To do this, I need to cover some topics of object-oriented design. In fact, I decided to dedicate the entire part 2 solely for that purpose. Thus, this chapter will be about object-oriented techniques, practices, and qualities you need to know to use TDD effectively in the object-oriented world. The key quality that we'll focus on is objects' composability.

[39] https://groups.google.com/forum/#!msg/growing-object-oriented-software/rwxCURI_3kM/2UcNAlF_Jh4J

Teaching one thing at a time

During this part of the book, you will see me do a lot of code and design examples without writing any test. This may make you wonder whether you are still reading a TDD book.

I want to make it very clear that by omitting tests in these chapters I am not advocating writing code or refactoring without tests. The only reason I am doing this is that teaching and learning several things at the same time may make everything harder, both for the teacher and for the student. So, while explaining the necessary object-oriented design topics, I want you to focus only on them.

Don't worry. After I've laid the groundwork for mock objects, I'll re-introduce TDD in part 3 and write lots of tests. Please trust me and be patient.

After reading part 2, you will understand an opinionated approach to object-oriented design that is based on the idea of the object-oriented system being a web of nodes (objects) that pass messages to each other. This will give us a good starting point for introducing mock objects and mock-based TDD in part 3.

On Object Composability

In this chapter, I will try to outline briefly why object composability is a goal worth achieving and how it can be achieved. I am going to start with an example of an unmaintainable code and will gradually fix its flaws in the next chapters. For now, we are going to fix just one of the flaws, so the code we will end up will not be perfect by any means, still, it will be better by one quality.

In the coming chapters, we will learn more valuable lessons resulting from changing this little piece of code.

Another task for Johnny and Benjamin

Remember Johnny and Benjamin? Looks like they managed their previous task and are up to something else. Let's listen to their conversation as they are working on another project…

Benjamin: So, what's this assignment about?

Johnny: Actually, it's nothing exciting – we'll have to add two features to a legacy application that's not prepared for the changes.

Benjamin: What is the code for?

Johnny: It is a C# class that implements company policies. As the company has just started using this automated system and it was started recently, there is only one policy implemented: yearly incentive plan. Many corporations have what they call incentive plans. These plans are used to promote good behaviors and exceeding expectations by employees of a company.

Benjamin: You mean, the project has just started and is already in a bad shape?

Johnny: Yep. The guys writing it wanted to "keep it simple", whatever that means, and now it looks pretty bad.

Benjamin: I see…

Johnny: By the way, do you like riddles?

Benjamin: Always!

Johnny: So here's one: how do you call a development phase when you ensure high code quality?

Benjamin: … … No clue… So what is it called?

Johnny: It's called "now".

Benjamin: Oh!

Johnny: Getting back to the topic, here's the company incentive plan.

Every employee has a pay grade. An employee can be promoted to a higher pay grade, but the mechanics of how that works is something we will not need to deal with.

Normally, every year, everyone gets a raise of 10%. But to encourage behaviors that give an employee a higher pay grade, such an employee cannot get raises indefinitely on a given pay grade. Each grade has its associated maximum pay. If this amount of money is reached, an employee does not get a raise anymore until they reach a higher pay grade.

Additionally, every employee on their 5th anniversary of working for the company gets a special, one-time bonus which is twice their current payment.

Benjamin: Looks like the source code repository just finished synchronizing. Let's take a bite at the code!

Johnny: Sure, here you go:

```
public class CompanyPolicies : IDisposable
{
  readonly SqlRepository _repository
    = new SqlRepository();

  public void ApplyYearlyIncentivePlan()
  {
    var employees = _repository.CurrentEmployees();

    foreach(var employee in employees)
    {
      var payGrade = employee.GetPayGrade();
      //evaluate raise
      if(employee.GetSalary() < payGrade.Maximum)
      {
        var newSalary
          = employee.GetSalary()
          + employee.GetSalary()
          * 0.1;
        employee.SetSalary(newSalary);
      }

      //evaluate one-time bonus
      if(employee.GetYearsOfService() == 5)
      {
        var oneTimeBonus = employee.GetSalary() * 2;
        employee.SetBonusForYear(2014, oneTimeBonus);
      }

      employee.Save();
    }
  }

  public void Dispose()
  {
```

```
36      _repository.Dispose();
37    }
38  }
```

Benjamin: Wow, there is a lot of literal constants all around and functional decomposition is barely done!

Johnny: Yeah. We won't be fixing all of that today. Still, we will follow the boy scout rule and "leave the campground cleaner than we found it".

Benjamin: What's our assignment?

Johnny: First of all, we need to provide our users with a choice between an SQL database and a NoSQL one. To achieve our goal, we need to be somehow able to make the `CompanyPolicies` class database type-agnostic. For now, as you can see, the implementation is coupled to the specific `SqlRepository`, because it creates a specific instance itself:

```
1  public class CompanyPolicies : IDisposable
2  {
3    readonly SqlRepository _repository
4      = new SqlRepository();
```

Now, we need to evaluate the options we have to pick the best one. What options do you see, Benjamin?

Benjamin: Well, we could certainly extract an interface from `SqlRepository` and introduce an if statement to the constructor like this:

```
1  public class CompanyPolicies : IDisposable
2  {
3    readonly Repository _repository;
4
5    public CompanyPolicies()
6    {
7      if(...)
8      {
9        _repository = new SqlRepository();
10     }
11     else
12     {
13       _repository = new NoSqlRepository();
14     }
15   }
```

Johnny: True, but this option has few deficiencies. First of all, remember we're trying to follow the boy scout rule and by using this option we introduce more complexity to the `CommonPolicies` class. Also, let's say tomorrow someone writes another class for, say, reporting and this class will also need to access the repository – they will need to make the same decision on repositories in

their code as we do in ours. This effectively means duplicating code. Thus, I'd rather evaluate further options and check if we can come up with something better. What's our next option?

Benjamin: Another option would be to change the `SqlRepository` itself to be just a wrapper around the actual database access, like this:

```
public class SqlRepository : IDisposable
{
  readonly Repository _repository;

  public SqlRepository()
  {
    if(...)
    {
      _repository = new RealSqlRepository();
    }
    else
    {
      _repository = new RealNoSqlRepository();
    }
  }

  IList<Employee> CurrentEmployees()
  {
    return _repository.CurrentEmployees();
  }
```

Johnny: Sure, this is an approach that could work and would be worth considering for very serious legacy code, as it does not force us to change the `CompanyPolicies` class at all. However, there are some issues with it. First of all, the `SqlRepository` name would be misleading. Second, look at the `CurrentEmployees()` method – all it does is delegating a call to the implementation chosen in the constructor. With every new method required of the repository, we'll need to add new delegating methods. In reality, it isn't such a big deal, but maybe we can do better than that?

Benjamin: Let me think, let me think... I evaluated the option where `CompanyPolicies` class chooses between repositories. I also evaluated the option where we hack the `SqlRepository` to makes this choice. The last option I can think of is leaving this choice to another, "3rd party" code, that would choose the repository to use and pass it to the `CompanyPolicies` via its constructor, like this:

```
public class CompanyPolicies : IDisposable
{
  private readonly Repository _repository;

  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }
```

This way, the `CompanyPolicies` won't know what exactly is passed to it via its constructor and we can pass whatever we like – either an SQL repository or a NoSQL one!

Johnny: Great! This is the option we're looking for! For now, just believe me that this approach will lead us to many good things – you'll see why later.

Benjamin: OK, so let me just pull the `SqlRepository` instance outside the `CompanyPolicies` class and make it an implementation of `Repository` interface, then create a constructor and pass the real instance through it...

Johnny: Sure, I'll go get some coffee.

... 10 minutes later

Benjamin: Haha! Look at this! I am SUPREME!

```
public class CompanyPolicies : IDisposable
{
  //_repository is now an interface
  readonly Repository _repository;

  // repository is passed from outside.
  // We don't know what exact implementation it is.
  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }

  public void ApplyYearlyIncentivePlan()
  {
    //... body of the method. Unchanged.
  }

  public void Dispose()
  {
    _repository.Dispose();
  }
}
```

Johnny: Hey, hey, hold your horses! There is one thing wrong with this code.

Benjamin: Huh? I thought this is what we were aiming at.

Johnny: Yes, except the `Dispose()` method. Look closely at the `CompanyPolicies` class. it is changed so that it is not responsible for creating a repository for itself, but it still disposes of it. This is could cause problems because `CompanyPolicies` instance does not have any right to assume it is the only object that is using the repository. If so, then it cannot determine the moment when the repository becomes unnecessary and can be safely disposed of.

Benjamin: Ok, I get the theory, but why is this bad in practice? Can you give me an example?

Johnny: Sure, let me sketch a quick example. As soon as you have two instances of `CompanyPolicies` class, both sharing the same instance of `Repository`, you're cooked. This is because one instance of `CompanyPolicies` may dispose of the repository while the other one may still want to use it.

Benjamin: So who is going to dispose of the repository?

Johnny: The same part of the code that creates it, for example, the `Main` method. Let me show you an example of how this may look like:

```
public static void Main(string[] args)
{
  using(var repo = new SqlRepository())
  {
    var policies = new CompanyPolicies(repo);

    //use above created policies
    //for anything you like
  }
}
```

This way the repository is created at the start of the program and disposed of at the end. Thanks to this, the `CompanyPolicies` has no disposable fields and it does not have to be disposable itself – we can just delete the `Dispose()` method:

```
//not implementing IDisposable anymore:
public class CompanyPolicies
{
  //_repository is now an interface
  readonly Repository _repository;

  //New constructor
  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }

  public void ApplyYearlyIncentivePlan()
  {
    //... body of the method. No changes
```

```
16    }
17
18    //no Dispose() method anymore
19  }
```

Benjamin: Cool. So, what now? Seems we have the `CompanyPolicies` class depending on a repository abstraction instead of an actual implementation, like SQL repository. I guess we will be able to make another class implementing the interface for NoSQL data access and just pass it through the constructor instead of the original one.

Johnny: Yes. For example, look at `CompanyPolicies` component. We can compose it with a repository like this:

```
var policies
  = new CompanyPolicies(new SqlRepository());
```

or like this:

```
var policies
  = new CompanyPolicies(new NoSqlRepository());
```

without changing the code of `CompanyPolicies`. This means that `CompanyPolicies` does not need to know what `Repository` exactly it is composed with, as long as this `Repository` follows the required interface and meets expectations of `CompanyPolicies` (e.g. does not throw exceptions when it is not supposed to do so). An implementation of `Repository` may be itself a very complex and composed of another set of classes, for example, something like this:

```
new SqlRepository(
  new ConnectionString("..."),
  new AccessPrivileges(
    new Role("Admin"),
    new Role("Auditor")
  ),
  new InMemoryCache()
);
```

but the `CompanyPolicies` neither knows or cares about this, as long as it can use our new `Repository` implementation just like other repositories.

Benjamin: I see... So, getting back to our task, shall we proceed with making a NoSQL implementation of the `Repository` interface?

Johnny: First, show me the interface that you extracted while I was looking for the coffee.

Benjamin: Here:

```
public interface Repository
{
  IList<Employee> CurrentEmployees();
}
```

Johnny: Ok, so what we need is to create just another implementation and pass it through the constructor depending on what data source is chosen and we're finished with this part of the task.

Benjamin: You mean there's more?

Johnny: Yeah, but that's something for tomorrow. I'm exhausted today.

A Quick Retrospective

In this chapter, Benjamin learned to appreciate composability of an object, i.e. the ability to replace its dependencies, providing different behaviors, without the need to change the code of the object class itself. Thus, an object, given replaced dependencies, starts using the new behaviors without noticing that any change occurred at all.

The code mentioned has some serious flaws. For now, Johnny and Benjamin did not encounter a desperate need to address them.

Also, after we part again with Johnny and Benjamin, we are going to reiterate the ideas they stumble upon in a more disciplined manner.

Telling, not asking

In this chapter, we'll get back to Johnny and Benjamin as they introduce another change in the code they are working on. In the process, they discover the impact that return values and getters have on the composability of objects.

Contractors

Johnny: G'morning. Ready for another task?

Benjamin: Of course! What's next?

Johnny: Remember the code we worked on yesterday? It contains a policy for regular employees of the company. But the company wants to start hiring contractors as well and needs to include a policy for them in the application.

Benjamin: So this is what we will be doing today?

Johnny: That's right. The policy is going to be different for contractors. While, just as regular employees, they will be receiving raises and bonuses, the rules will be different. I made a small table to allow comparing what we have for regular employees and what we want to add for contractors:

Employee Type	Raise	Bonus
Regular Employee	+10% of current salary if not reached a maximum on a given pay grade	+200% of current salary one time after five years
Contractor	+5% of average salary calculated for last 3 years of service (or all previous years of service if they have worked for less than 3 years	+10% of current salary when a contractor receives score more than 100 for the previous year

So while the workflow is going to be the same for both a regular employee and a contractor:

1. Load from repository
2. Evaluate raise
3. Evaluate bonus
4. Save

the implementation of some of the steps will be different for each type of employee.

Benjamin: Correct me if I am wrong, but these "load" and "save" steps do not look like they belong with the remaining two – they describe something technical, while the other steps describe something strictly related to how the company operates…

Johnny: Good catch, however, this is something we'll deal with later. Remember the boy scout rule – just don't make it worse. Still, we're going to fix some of the design flaws today.

Benjamin: Aww... I'd just fix all of it right away.

Johnny: Ha ha, patience, Luke. For now, let's look at the code we have now before we plan further steps.

Benjamin: Let me just open my IDE... OK, here it is:

```
1  public class CompanyPolicies
2  {
3    readonly Repository _repository;
4
5    public CompanyPolicies(Repository repository)
6    {
7      _repository = repository;
8    }
9
10   public void ApplyYearlyIncentivePlan()
11   {
12     var employees = _repository.CurrentEmployees();
13
14     foreach(var employee in employees)
15     {
16       var payGrade = employee.GetPayGrade();
17
18       //evaluate raise
19       if(employee.GetSalary() < payGrade.Maximum)
20       {
21         var newSalary
22           = employee.GetSalary()
23           + employee.GetSalary()
24           * 0.1;
25         employee.SetSalary(newSalary);
26       }
27
28       //evaluate one-time bonus
29       if(employee.GetYearsOfService() == 5)
30       {
31         var oneTimeBonus = employee.GetSalary() * 2;
32         employee.SetBonusForYear(2014, oneTimeBonus);
33       }
34
35       employee.Save();
36     }
37   }
38 }
```

Benjamin: Look, Johnny, the class, in fact, contains all the four steps you mentioned, but they are not named explicitly – instead, their internal implementation for regular employees is just inserted in here. How are we supposed to add the variation of the employee type?

Johnny: Time to consider our options. We have a few of them. Well?

Benjamin: For now, I can see two. The first one would be to create another class similar to `CompanyPolicies`, called something like `CompanyPoliciesForContractors` and implement the new logic there. This would let us leave the original class as is, but we would have to change the places that use `CompanyPolicies` to use both classes and choose which one to use somehow. Also, we would have to add a separate method to the repository for retrieving the contractors.

Johnny: Also, we would miss our chance to communicate through the code that the sequence of steps is intentionally similar in both cases. Others who read this code in the future will see that the implementation for regular employees follows the steps: load, evaluate raise, evaluate bonus, save. When they look at the implementation for contractors, they will see the same order of steps, but they will be unable to tell whether the similarity is intentional, or a pure accident.

Benjamin: So our second option is to put an `if` statement into the differing steps inside the `CompanyPolicies` class, to distinguish between regular employees and contractors. The `Employee` class would have an `isContractor()` method and depending on what it would return, we would either execute the logic for regular employees or contractors. Assuming that the current structure of the code looks like this:

```
foreach(var employee in employees)
{
  //evaluate raise
  ...

  //evaluate one-time bonus
  ...

  //save employee
}
```

the new structure would look like this:

```
foreach(var employee in employees)
{
  if(employee.IsContractor())
  {
    //evaluate raise for contractor
    ...
  }
  else
  {
    //evaluate raise for regular
    ...
```

```
  }

  if(employee.IsContractor())
  {
    //evaluate one-time bonus for contractor
    ...
  }
  else
  {
    //evaluate one-time bonus for regular
    ...
  }

  //save employee
  ...
}
```

this way we would show that the steps are the same, but the implementation is different. Also, this would mostly require us to add code and not move the existing code around.

Johnny: The downside is that we would make the class even uglier than it was when we started. So despite initial easiness, we'll be doing a huge disservice to future maintainers. We have at least one another option. What would that be?

Benjamin: Let's see... we could move all the details concerning the implementation of the steps from `CompanyPolicies` class into the `Employee` class itself, leaving only the names and the order of steps in `CompanyPolicies`:

```
foreach(var employee in employees)
{
  employee.EvaluateRaise();
  employee.EvaluateOneTimeBonus();
  employee.Save();
}
```

Then, we could change the `Employee` into an interface, so that it could be either a `RegularEmployee` or `ContractorEmployee` – both classes would have different implementations of the steps, but the `CompanyPolicies` would not notice, since it would not be coupled to the implementation of the steps anymore – just the names and the order.

Johnny: This solution would have one downside – we would need to significantly change the current code, but you know what? I'm willing to do it, especially that I was told today that the logic is covered by some tests which we can run to see if a regression was introduced.

Benjamin: Cool, what do we start with?

Johnny: The first thing that is between us and our goal are these getters on the `Employee` class:

```
GetSalary();
GetGrade();
GetYearsOfService();
```

They just expose too much information specific to the regular employees. It would be impossible to use different implementations when these are around. These setters don't help much:

```
SetSalary(newSalary);
SetBonusForYear(year, amount);
```

While these are not as bad, we'd better give ourselves more flexibility. Thus, let's hide all of it behind more abstract methods that only reveal our intention.

First, take a look at this code:

```
//evaluate raise
if(employee.GetSalary() < payGrade.Maximum)
{
  var newSalary
    = employee.GetSalary()
    + employee.GetSalary()
    * 0.1;
  employee.SetSalary(newSalary);
}
```

Each time you see a block of code separated from the rest with blank lines and starting with a comment, you see something screaming "I want to be a separate method that contains this code and has a name after the comment!". Let's grant this wish and make it a separate method on the `Employee` class.

Benjamin: Ok, wait a minute... here:

```
employee.EvaluateRaise();
```

Johnny: Great! Now, we've got another example of this species here:

```
//evaluate one-time bonus
if(employee.GetYearsOfService() == 5)
{
  var oneTimeBonus = employee.GetSalary() * 2;
  employee.SetBonusForYear(2014, oneTimeBonus);
}
```

Benjamin: This one should be even easier... Ok, take a look:

```
employee.EvaluateOneTimeBonus();
```

Johnny: Almost good. I'd only leave out the information that the bonus is one-time from the name.

Benjamin: Why? Don't we want to include what happens in the method name?

Johnny: Actually, no. What we want to include is our intention. The bonus being one-time is something specific to the regular employees and we want to abstract away the details about this or that kind of employee, so that we can plug in different implementations without making the method name lie. The names should reflect that we want to evaluate a bonus, whatever that means for a particular type of employee. Thus, let's make it:

```
employee.EvaluateBonus();
```

Benjamin: Ok, I get it. No problem.

Johnny: Now let's take a look at the full code of the `EvaluateIncentivePlan` method to see whether it is still coupled to details specific to regular employees. Here's the code:

```
public void ApplyYearlyIncentivePlan()
{
  var employees = _repository.CurrentEmployees();

  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
    employee.Save();
  }
}
```

Benjamin: It seems that there is no coupling to the details about regular employees anymore. Thus, we can safely make the repository return a combination of regulars and contractors without this code noticing anything. Now I think I understand what you were trying to achieve. If we make interactions between objects happen on a more abstract level, then we can put in different implementations with less effort.

Johnny: True. Can you see another thing related to the lack of return values on all of the `Employee`'s methods in the current implementation?

Benjamin: Not really. Does it matter?

Johnny: Well, if `Employee` methods had return values and this code depended on them, all subclasses of `Employee` would be forced to supply return values as well and these return values would need to match the expectations of the code that calls these methods, whatever these expectations were. This would make introducing other kinds of employees harder. But now that there are no return values, we can, for example:

- introduce a `TemporaryEmployee` that has no raises, by leaving its `EvaluateRaise()` method empty, and the code that uses employees will not notice.
- introduce a `ProbationEmployee` that has no bonus policy, by leaving its `EvaluateBonus()` method empty, and the code that uses employees will not notice.
- introduce an `InMemoryEmployee` that has an empty `Save()` method, and the code that uses employees will not notice.

As you see, by asking the objects less, and telling it more, we get greater flexibility to create alternative implementations and the composability, which we talked about yesterday, increases!

Benjamin: I see... So telling objects what to do instead of asking them for their data makes the interactions between objects more abstract, and so, more stable, increasing composability of interacting objects. This is a valuable lesson – it is the first time I hear this and it seems a pretty powerful concept.

A Quick Retrospective

In this chapter, Benjamin learned that the composability of an object (not to mention clarity) is reinforced when interactions between it and its peers are: abstract, logical and stable. Also, he discovered, with Johnny's help, that it is further strengthened by following a design style where objects are told what to do instead of asked to give away information to somebody who then decides on their behalf. This is because if an API of an abstraction is built around answering specific questions, the clients of the abstraction tend to ask it a lot of questions and are coupled to both those questions and some aspects of the answers (i.e. what is in the return values). This makes creating another implementation of abstraction harder, because each new implementation of the abstraction needs to not only provide answers to all those questions, but the answers are constrained to what the client expects. When abstraction is merely told what its client wants it to achieve, the clients are decoupled from most of the details of how this happens. This makes introducing new implementations of abstraction easier – it often even lets us define implementations with all methods empty without the client noticing at all.

These are all important conclusions that will lead us towards TDD with mock objects.

Time to leave Johnny and Benjamin for now. In the next chapter, I'm going to reiterate their discoveries and put them in a broader context.

The need for mock objects

We already experienced mock objects in the chapter about tools, although at that point, I gave you an oversimplified and deceiving explanation of what a mock object is, promising that I will make up for it later. Now is the time.

Mock objects were made with a specific goal in mind. I hope that when you understand the real goal, you will probably understand the means to the goal far better.

In this chapter, we will explore the qualities of object-oriented design which make mock objects a viable tool.

Composability... again!

In the two previous chapters, we followed Johnny and Benjamin in discovering the benefits of and prerequisites for composability of objects. Composability is the number one quality of the design we're after. After reading Johhny and Benjamin's story, you might have some questions regarding composability. Hopefully, they are among the ones answered in the next few chapters. Ready?

Why do we need composability?

It might seem stupid to ask this question here – if you have managed to stay with me this long, then you're probably motivated enough not to need a justification? Well, anyway, it's still worth discussing it a little. Hopefully, you'll learn as much reading this back-to-basics chapter as I did writing it.

Pre-object-oriented approaches

Back in the days of procedural programming[40], when we wanted to execute different code based on some factor, it was usually achieved using an 'if' statement. For example, if our application was in need to be able to use different kinds of alarms, like a loud alarm (that plays a loud sound) and a silent alarm (that does not play any sound, but instead silently contacts the police) interchangeably, then usually, we could achieve this using a conditional like in the following function:

```
1  void triggerAlarm(Alarm* alarm)
2  {
3    if(alarm->kind == LOUD_ALARM)
4    {
5      playLoudSound(alarm);
6    }
7    else if(alarm->kind == SILENT_ALARM)
8    {
9      notifyPolice(alarm);
10   }
11 }
```

The code above makes a decision based on the alarm kind which is embedded in the alarm structure:

```
1  struct Alarm
2  {
3    int kind;
4    //other data
5  };
```

If the alarm kind is the loud one, it executes behavior associated with a loud alarm. If this is a silent alarm, the behavior for silent alarms is executed. This seems to work. Unfortunately, if we

[40] I am simplifying the discussion on purpose, leaving out e.g. functional languages and assuming that "pre-object-oriented" means procedural or structural. While this is not true in general, this is how the reality looked like for many of us. If you are good at functional programming, you already understand the benefits of composability.

wanted to make a second decision based on the alarm kind (e.g. we needed to disable the alarm), we would need to query the alarm kind again. This would mean duplicating the conditional code, just with a different set of actions to perform, depending on what kind of alarm we were dealing with:

```
void disableAlarm(Alarm* alarm)
{
  if(alarm->kind == LOUD_ALARM)
  {
    stopLoudSound(alarm);
  }
  else if(alarm->kind == SILENT_ALARM)
  {
    stopNotifyingPolice(alarm);
  }
}
```

Do I have to say why this duplication is bad? Do I hear a "no"? My apologies then, but I'll tell you anyway. The duplication means that every time a new kind of alarm is introduced, a developer has to remember to update both places that contain 'if-else' – the compiler will not force this. As you are probably aware, in the context of teams, where one developer picks up work that another left and where, from time to time, people leave to find another job, expecting someone to "remember" to update all the places where the logic is duplicated is asking for trouble.

So, we see that the duplication is bad, but can we do something about it? To answer this question, let's take a look at the reason the duplication was introduced. And the reason is: We have two things we want to be able to do with our alarms: triggering and disabling. In other words, we have a set of questions we want to be able to ask an alarm. Each kind of alarm has a different way of answering these questions – resulting in having a set of "answers" specific to each alarm kind:

Alarm Kind	Triggering	Disabling
Loud Alarm	`playLoudSound()`	`stopLoudSound()`
Silent Alarm	`notifyPolice()`	`stopNotifyingPolice()`

So, at least conceptually, as soon as we know the alarm kind, we already know which set of behaviors (represented as a row in the above table) it needs. We could just decide the alarm kind once and associate the right set of behaviors with the data structure. Then, we would not have to query the alarm kind in several places as we did, but instead, we could say: "execute triggering behavior from the set of behaviors associated with this alarm, whatever it is".

Unfortunately, procedural programming does not allow binding behavior with data easily. The whole paradigm of procedural programming is about separating behavior and data! Well, honestly, they had some answers to those concerns, but these answers were mostly awkward (for those of you that still remember C language: I'm talking about macros and function pointers). So, as data and behavior are separated, we need to query the data each time we want to pick a behavior based on it. That's why we have duplication.

Object-oriented programming to the rescue!

On the other hand, object-oriented programming has for a long time made available two mechanisms that enable what we didn't have in procedural languages:

1. Classes – that allow binding behavior together with data.
2. Polymorphism – allows executing behavior without knowing the exact class that holds them, but knowing only a set of behaviors that it supports. This knowledge is obtained by having an abstract type (interface or an abstract class) define this set of behaviors, with no real implementation. Then we can make other classes that provide their own implementation of the behaviors that are declared to be supported by the abstract type. Finally, we can use the instances of those classes where an instance of the abstract type is expected. In the case of statically-typed languages, this requires implementing an interface or inheriting from an abstract class.

So, in case of our alarms, we could make an interface with the following signature:

```
public interface Alarm
{
  void Trigger();
  void Disable();
}
```

and then make two classes: `LoudAlarm` and `SilentAlarm`, both implementing the `Alarm` interface. Example for `LoudAlarm`:

```
public class LoudAlarm : Alarm
{
  public void Trigger()
  {
    //play very loud sound
  }

  public void Disable()
  {
    //stop playing the sound
  }
}
```

Now, we can make parts of code use the alarm, but by knowing the interface only instead of the concrete classes. This makes the parts of the code that use alarm this way not having to check which alarm they are dealing with. Thus, what previously looked like this:

```
if(alarm->kind == LOUD_ALARM)
{
  playLoudSound(alarm);
}
else if(alarm->kind == SILENT_ALARM)
{
  notifyPolice(alarm);
}
```

becomes just:

```
alarm.Trigger();
```

where `alarm` is either `LoudAlarm` or `SilentAlarm` but seen polymorphically as `Alarm`, so there's no need for 'if-else' anymore.

But hey, isn't this cheating? Even provided I can execute the trigger behavior on an alarm without knowing the actual class of the alarm, I still have to decide which class it is in the place where I create the actual instance:

```
// we must know the exact type here:
alarm = new LoudAlarm();
```

so it looks like I am not eliminating the 'else-if' after all, just moving it somewhere else! This may be true (we will talk more about it in future chapters), but the good news is that I eliminated at least the duplication by making our dream of "picking the right set of behaviors to use with certain data once" come true.

Thanks to this, I create the alarm once, and then I can take it and pass it to ten, a hundred or a thousand different places where I will not have to determine the alarm kind anymore to use it correctly.

This allows writing a lot of classes that do not know about the real class of the alarm they are dealing with, yet they can use the alarm just fine only by knowing a common abstract type – `Alarm`. If we can do that, we arrive at a situation where we can add more alarms implementing `Alarm` and watch existing objects that are already using `Alarm` work with these new alarms without any change in their source code! There is one condition, however – the **creation of the alarm instances must be moved out of the classes that use them**. That's because, as we already observed, to create an alarm using a `new` operator, we have to know the exact type of the alarm we are creating. So whoever creates an instance of `LoudAlarm` or `SilentAlarm`, loses its uniformity, since it is not able to depend solely on the `Alarm` interface.

The power of composition

Moving creation of alarm instances away from the classes that use those alarms brings up an interesting problem – if an object does not create the objects it uses, then who does it? A solution

is to make some special places in the code that are only responsible for composing a system from context-independent objects[41]. We saw this already as Johnny was explaining composability to Benjamin. He used the following example:

```
new SqlRepository(
  new ConnectionString("..."),
  new AccessPrivileges(
    new Role("Admin"),
    new Role("Auditor")
  ),
  new InMemoryCache()
);
```

We can do the same with our alarms. Let's say that we have a secure area that has three buildings with different alarm policies:

- Office building – the alarm should silently notify guards during the day (to keep office staff from panicking) and loud during the night, when guards are on patrol.
- Storage building – as it is quite far and the workers are few, we want to trigger loud and silent alarms at the same time.
- Guards building – as the guards are there, no need to notify them. However, a silent alarm should call the police for help instead, and a loud alarm is desired as well.

Note that besides just triggering a loud or silent alarm, we are required to support a combination ("loud and silent alarms at the same time") and a conditional ("silent during the day and loud during the night"). we could just hardcode some `for`s and `if-else`s in our code, but instead, let's factor out these two operations (combination and choice) into separate classes implementing the alarm interface.

Let's call the class implementing the choice between two alarms `DayNightSwitchedAlarm`. Here is the source code:

```
public class DayNightSwitchedAlarm : Alarm
{
  private readonly Alarm _dayAlarm;
  private readonly Alarm _nightAlarm;

  public DayNightSwitchedAlarm(
    Alarm dayAlarm,
    Alarm nightAlarm)
  {
    _dayAlarm = dayAlarm;
    _nightAlarm = nightAlarm;
  }

```

[41] More on context-independence and what these "special places" are, in the next chapters.

```
14    public void Trigger()
15    {
16      if(/* is day */)
17      {
18        _dayAlarm.Trigger();
19      }
20      else
21      {
22        _nightAlarm.Trigger();
23      }
24    }
25
26    public void Disable()
27    {
28      _dayAlarm.Disable();
29      _nightAlarm.Disable();
30    }
31  }
```

Studying the above code, it is apparent that this is not an alarm *per se*, e.g. it does not raise any sound or notification, but rather, it contains some rules on how to use other alarms. This is the same concept as power splitters in real life, which act as electric devices but do not do anything other than redirecting the electricity to other devices.

Next, let's use the same approach and model the combination of two alarms as a class called `HybridAlarm`. Here is the source code:

```
1   public class HybridAlarm : Alarm
2   {
3     private readonly Alarm _alarm1;
4     private readonly Alarm _alarm2;
5
6     public HybridAlarm(
7       Alarm alarm1,
8       Alarm alarm2)
9     {
10      _alarm1 = alarm1;
11      _alarm2 = alarm2;
12    }
13
14    public void Trigger()
15    {
16      _alarm1.Trigger();
17      _alarm2.Trigger();
18    }
19
20    public void Disable()
```

```
21    {
22      _alarm1.Disable();
23      _alarm2.Disable();
24    }
25  }
```

Using these two classes along with already existing alarms, we can implement the requirements by composing instances of those classes like this:

```
1   new SecureArea(
2     new OfficeBuilding(
3       new DayNightSwitchedAlarm(
4         new SilentAlarm("222-333-444"),
5         new LoudAlarm()
6       )
7     ),
8     new StorageBuilding(
9       new HybridAlarm(
10        new SilentAlarm("222-333-444"),
11        new LoudAlarm()
12      )
13    ),
14    new GuardsBuilding(
15      new HybridAlarm(
16        new SilentAlarm("919"), //call police
17        new LoudAlarm()
18      )
19    )
20  );
```

Note that the fact that we implemented combination and choice of alarms as separate objects implementing the `Alarm` interface allows us to define new, interesting alarm behaviors using the parts we already have, but composing them together differently. For example, we might have, as in the above example:

```
1   new DayNightSwitchAlarm(
2     new SilentAlarm("222-333-444"),
3     new LoudAlarm());
```

which would mean triggering silent alarm during a day and loud one during the night. However, instead of this combination, we might use:

```
new DayNightSwitchAlarm(
  new SilentAlarm("222-333-444"),
  new HybridAlarm(
    new SilentAlarm("919"),
    new LoudAlarm()
  )
)
```

Which would mean that we use a silent alarm to notify the guards during the day, but a combination of silent (notifying police) and loud during the night. Of course, we are not limited to combining a silent alarm with a loud one only. We can as well combine two silent ones:

```
new HybridAlarm(
  new SilentAlarm("919"),
  new SilentAlarm("222-333-444")
)
```

Additionally, if we suddenly decided that we do not want alarm at all during the day, we could use a special class called `NoAlarm` that would implement `Alarm` interface, but have both `Trigger` and `Disable` methods do nothing. The composition code would look like this:

```
new DayNightSwitchAlarm(
  new NoAlarm(), // no alarm during the day
  new HybridAlarm(
    new SilentAlarm("919"),
    new LoudAlarm()
  )
)
```

And, last but not least, we could completely remove all alarms from the guards building using the following `NoAlarm` class (which is also an `Alarm`):

```
public class NoAlarm : Alarm
{
  public void Trigger()
  {
  }

  public void Disable()
  {
  }
}
```

and passing it as the alarm to guards building:

```
new GuardsBuilding(
  new NoAlarm()
)
```

Noticed something funny about the last few examples? If not, here goes an explanation: in the last few examples, we have twisted the behaviors of our application in wacky ways, but all of this took place in the composition code! We did not have to modify any other existing classes! True, we had to write a new class called `NoAlarm`, but did not need to modify any code except the composition code to make objects of this new class work with objects of existing classes!

This ability to change the behavior of our application just by changing the way objects are composed together is extremely powerful (although you will always be able to achieve it only to certain extent), especially in evolutionary, incremental design, where we want to evolve some pieces of code with as little as possible other pieces of code having to realize that the evolution takes place. This ability can be achieved only if our system consists of composable objects, thus the need for composability – an answer to a question raised at the beginning of this chapter.

Summary – are you still with me?

We started with what seemed to be a repetition from a basic object-oriented programming course, using a basic example. It was necessary though to make a fluent transition to the benefits of composability we eventually introduced at the end. I hope you did not get overwhelmed and can understand now why I am putting so much stress on composability.

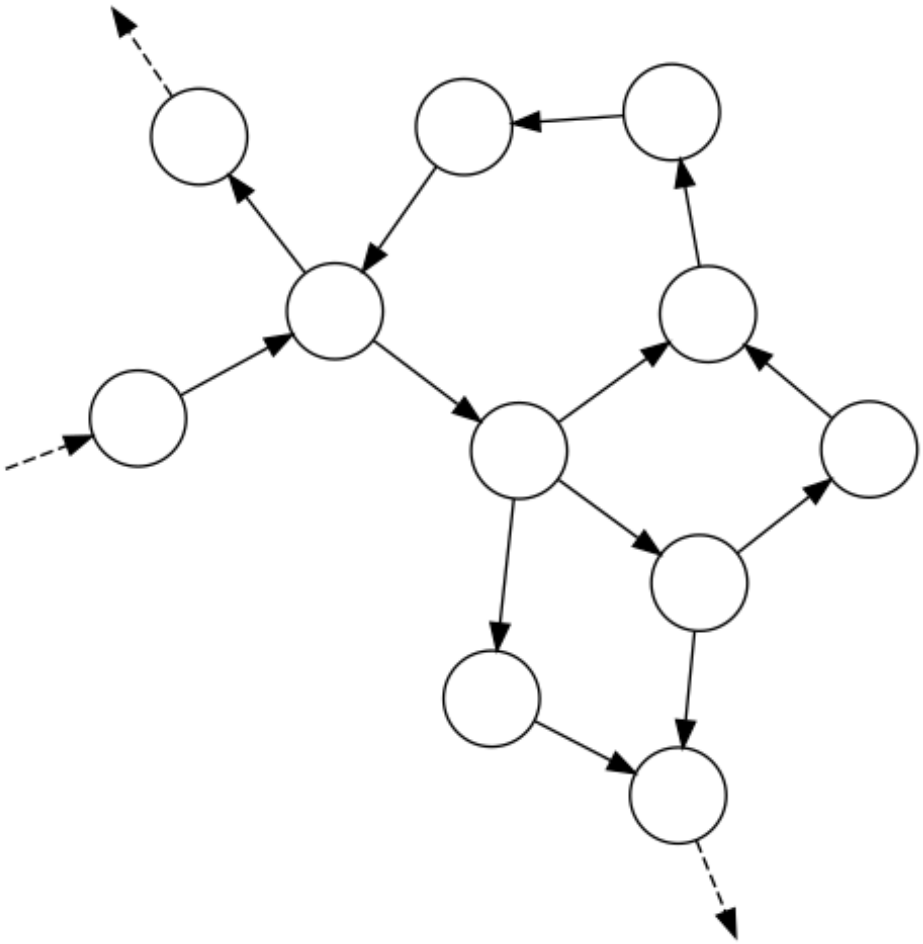
In the next chapter, we will take a closer look at composing objects itself.

Web, messages and protocols

In the previous chapter, we talked a little bit about why composability is valuable, now let's flesh out a little bit of terminology to get more precise understanding.

So, again, what does it mean to compose objects?

Roughly, it means that an object has obtained a reference to another object and can invoke methods on it. By being composed together, two objects form a small system that can be expanded with more objects as needed. Thus, a bigger object-oriented system forms something similar to a web:



Web of objects – the circles are the objects and the arrows are methods invocations from one object on another

If we take the web metaphor a little bit further, we can note some similarities to e.g. a TCP/IP network:

1. An object can send **messages** to other objects (i.e. call methods on them – arrows on the above diagram) via **interfaces**. Each message has a **sender** and at least one **recipient**.
2. To send a message to a recipient, a sender has to acquire an **address** of the recipient, which, in the object-oriented world, we call a reference (and in languages such as C++, references are just that – addresses in memory).
3. Communication between the sender and recipients has to follow a certain **protocol**. For example, the sender usually cannot invoke a method passing nulls as all arguments, or should expect an exception if it does so. Don't worry if you don't see the analogy now – I'll follow up with more explanation of this topic later).

Alarms, again!

Let's try to apply this terminology to an example. Imagine that we have an anti-fire alarm system in an office. When triggered, this alarm system makes all lifts go to the bottom floor, opens them and then disables each of them. Among others, the office contains automatic lifts, that contain their own remote control systems and mechanical lifts, that are controlled from the outside by a special custom-made mechanism.

Let's try to model this behavior in code. As you might have guessed, we will have some objects like alarm, automatic lift, and mechanical lift. The alarm will control the lifts when triggered.

Firstly, we don't want the alarm to have to distinguish between an automatic and a mechanical lift – this would only add complexity to the alarm system, especially that there are plans to add a third kind of lift – a more modern one – in the future. So, if we made the alarm aware of the different kinds of lifts, we would have to modify it each time a new kind of lift is introduced. Thus, we need a special **interface** (let's call it `Lift`) to communicate with both `AutoLift` and `MechanicalLift` (and `ModernLift` in the future). Through this interface, an alarm will be able to send messages to both types of lifts without having to know the difference between them.

```
1  public interface Lift
2  {
3    ...
4  }
5
6  public class AutoLift : Lift
7  {
8    ...
9  }
10
11 public class MechanicalLift : Lift
12 {
13   ...
14 }
```

Next, to be able to communicate with specific lifts through the `Lift` interface, an alarm object has to acquire "**addresses**" of the lift objects (i.e. references to them). We can pass these references e.g. through a constructor:

```
public class Alarm
{
  private readonly IEnumerable<Lift> _lifts;

  //obtain "addresses" here
  public Alarm(IEnumerable<Lift> lifts)
  {
    //store the "addresses" for later use
    _lifts = lifts;
  }
}
```

Then, the alarm can send three kinds of **messages**: `GoToBottomFloor()`, `OpenDoor()`, and `DisablePower()` to any of the lifts through the `Lift` interface:

```
public interface Lift
{
  void GoToBottomFloor();
  void OpenDoor();
  void DisablePower();
}
```

and, as a matter of fact, it sends all these messages when triggered. The `Trigger()` method on the alarm looks like this:

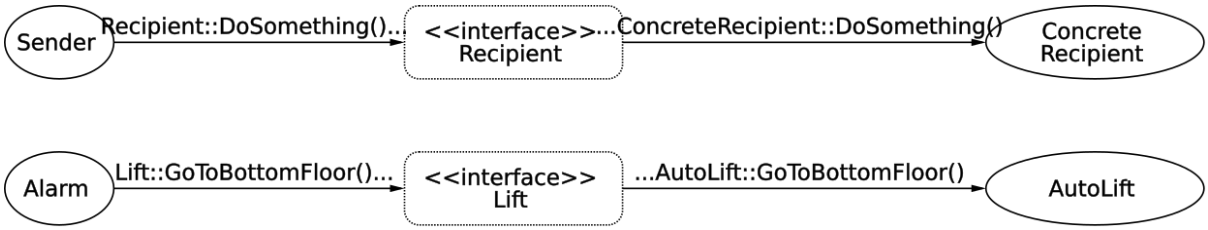
```
public void Trigger()
{
  foreach(var lift in _lifts)
  {
    lift.GoToBottomFloor();
    lift.OpenDoor();
    lift.DisablePower();
  }
}
```

By the way, note that the order in which the messages are sent **does** matter. For example, if we disabled the power first, asking the powerless lift to go anywhere would be impossible. This is a first sign of a **protocol** existing between the `Alarm` and a `Lift`.

In this communication, `Alarm` is a **sender** – it knows what it sends (messages that control lifts), it knows why (because the alarm is triggered), but does not know what exactly are the recipients going to do when they receive the message – it only knows what it **wants** them to do, but does not know **how** they are going to achieve it. The rest is left to objects that implement `Lift` (namely, `AutoLift` and `MechanicalLift`). They are the **recipients** – they don't know who they received the message from (unless they are told in the content of the message somehow – but even then they can be cheated), but they know how to react, based on who they are (`AutoLift`

has its specific way of reacting and `MechanicalLift` has its own as well). They also know what kind of message they received (a lift does a different thing when asked to go to the bottom floor than when it is asked to open its door) and what's the message content (i.e. method arguments – in this simplistic example there are none).

To illustrate that this separation between a sender and a recipient exists, I'll say that we could even write an implementation of a `Lift` interface that would just ignore the messages it got from the `Alarm` (or fake that it did what it was asked for) and the `Alarm` will not even notice. We sometimes say that this is because deciding on a specific reaction is not the `Alarm`'s responsibility.



Sender, interface, and recipient

New requirements

It was decided that whenever any malfunction happens in the lift when it is executing the alarm emergency procedure, the lift object should report this by throwing an exception called `LiftUnoperationalException`. This affects both `Alarm` and implementations of the `Lift` interface:

1. The `Lift` implementations need to know that when a malfunction happens, they should report it by throwing the exception.
2. The `Alarm` must be ready to handle the exception thrown from lifts and act accordingly (e.g. still try to secure other lifts).

This is another example of a **protocol** existing between `Alarm` and `Lift` that must be adhered to by both sides. Here is an exemplary code of `Alarm` keeping to its part of the protocol, i.e. handling the malfunction reports in its `Trigger()` method:

```
1  public void Trigger()
2  {
3    foreach(var lift in _lifts)
4    {
5      try
6      {
7        lift.GoToBottomFloor();
8        lift.OpenDoor();
9        lift.DisablePower();
10     }
11     catch(LiftUnoperationalException e)
12     {
```

```
13          report.ThatCannotSecure(lift);
14        }
15      }
16  }
```

Summary

Each of the objects in the web can receive messages and most of them send messages to other objects. Throughout the next chapters, I will refer to an object sending a message as **sender** and an object receiving a message as **recipient**.

For now, it may look unjustified to introduce this metaphor of webs, protocols, interfaces, etc. Still, I have two reasons for doing so:

- This is the way I interpret Alan Kay's[42] mental model of what object-oriented programming is about.
- I find it useful for some things want to explain in the next chapters: how to make connections between objects, how to design an object boundary and how to achieve strong composability.

By the way, the example from this chapter is a bit naive. For one, in real production code, the lifts would have to be notified in parallel, not sequentially. Also, I would probably use some kind of observer pattern to separate the instructions give to each lift from raising an event (I will demonstrate an example of using observers in this fashion in the next chapter). These two choices, in turn, would probably make me rethink error handling - there is a chance I wouldn't be able to get away with just catching exceptions. Anyway, I hope the naive form helped explain the idea of protocols and messages without raising the bar in other topics.

[42] http://c2.com/cgi/wiki?AlanKayOnMessaging

Composing a web of objects

Three important questions

Now that we know that such thing as a web of objects exists, that there are connections, protocols and such, time to mention the one thing I left out: how does a web of objects come into existence?

This is, of course, a fundamental question, because if we are unable to build a web, we don't have a web. Also, this is a question that is a little more tricky than it looks like at first glance. To answer it, we have to find the answer to three other questions:

1. When are objects composed (i.e. when are the connections made)?
2. How does an object obtain a reference to another one on the web (i.e. how are the connections made)?
3. Where are objects composed (i.e. where are connections made)?

At first sight, finding the difference between these questions may be tedious, but the good news is that they are the topic of this chapter, so I hope we'll have that cleared shortly.

A preview of all three answers

Before we take a deep dive, let's try to answer these three questions for a naive example code of a console application:

```
1  public static void Main(string[] args)
2  {
3    var sender = new Sender(new Recipient());
4
5    sender.Work();
6  }
```

This is a piece of code that creates two objects and connects them, then it tells the `sender` object to work on something. For this code, the answers to the three questions I raised are:

1. When are objects composed? Answer: during application startup (because `Main()` method is called at console application startup).
2. How does an object (`Sender`) obtain a reference to another one (`Recipient`)? Answer: the reference is obtained by receiving a `Recipient` as a constructor parameter.
3. Where are objects composed? Answer: at the application entry point (`Main()` method)

Depending on circumstances, we may have different sets of answers. Also, to avoid rethinking this topic each time we create an application, I like to have a set of default answers to these questions. I'd like to demonstrate these answers by tackling each of the three questions in-depth, one by one, in the coming chapters.

When are objects composed?

The quick answer to this question is: as early as possible. Now, that wasn't too helpful, was it? So here goes a clarification.

Many of the objects we use in our applications can be created and connected up-front when the application starts and can stay alive until the application finishes executing. Let's call this part the **static part** of the web.

Apart from that, there's something I'll call **dynamic part** – the objects that are created, connected and destroyed many times during the application lifecycle. There are at least two reasons this dynamic part exists:

1. Some objects represent requests or user actions that arrive during the application runtime, are processed and then discarded. These objects cannot be created up-front, but only as early as the events they represent occur. Also, these objects do not live until the application is terminated but are discarded as soon as the processing of a request is finished. Other objects represent e.g. items in cache that live for some time and then expire, so, again, we don't have enough information to compose these objects up-front and they often don't live as long as the application itself. Such objects come and go, making temporary connections.
2. There are objects that have life spans as long as the application has, but the nature of their connections are temporary. Consider an example where we want to encrypt our data storage for export, but depending on circumstances, we sometimes want to export it using one algorithm and sometimes using another. If so, we may sometimes invoke the encryption method like this:

```
1    database.encryptUsing(encryption1);
```

and sometimes like this:

```
1    database.encryptUsing(encryption2);
```

In the first case, `database` and `encryption1` are only connected temporarily, for the time it takes to perform the encryption. Still, nothing prevents these objects from being created during the application startup. The same applies to the connection of `database` and `encryption2` - this connection is temporary as well.

Given these definitions, it is perfectly possible for an object to be part of both static and dynamic part – some of its connections may be made up-front, while others may be created later, e.g. when its reference is passed inside a message sent to another object (i.e. when it is passed as method parameter).

How does a sender obtain a reference to a recipient (i.e. how connections are made)?

There are several ways a sender can obtain a reference to a recipient, each of them being useful in certain circumstances. These ways are:

1. Receive as a constructor parameter
2. Receive inside a message (i.e. as a method parameter)
3. Receive in response to a message (i.e. as a method return value)
4. Receive as a registered observer

Let's take a closer look at what each of them is about and which one to choose in what circumstances.

Receive as a constructor parameter

Two objects can be composed by passing one into a constructor of another:

```
sender = new Sender(recipient);
```

A sender that receives the recipient then saves a reference to it in a private field for later, like this:

```
private Recipient _recipient;

public Sender(Recipient recipient)
{
  _recipient = recipient;
}
```

Starting from this point, the `Sender` may send messages to `Recipient` at will:

```
public void DoSomething()
{
  //... other code

  _recipient.DoSomethingElse();

  //... other code
}
```

Advantage: "what you hide, you can change"

Composing using constructors has one significant advantage. Let's look again at how `Sender` is created:

```
sender = new Sender(recipient);
```

and at how it's used:

```
sender.DoSomething();
```

Note that only the code that creates a `Sender` needs to be aware of it having access to a `Recipient`. When it comes to invoking a method, this private reference is invisible from outside. Now, remember when I described the principle of separating object use from its construction? If we follow this principle here, we end up with the code that creates a `Sender` being in a totally different place than the code that uses it. Thus, every code that uses a `Sender` will not be aware of it sending messages to a `Recipient` at all. There is a maxim that says: "what you hide, you can change"[43] – in this particular case, if we decide that the `Sender` does not need a `Recipient` to do its job, all we have to change is the composition code to remove the `Recipient`:

```
//no need to pass a reference to Recipient anymore
new Sender();
```

and the code that uses `Sender` doesn't need to change at all – it still looks the same as before, since it never knew `Recipient`:

```
sender.DoSomething();
```

Communication of intent: required recipient

Another advantage of the constructor approach is that it allows stating explicitly what the required recipients are for a particular sender. For example, a `Sender` accepts a `Recipient` in its constructor:

[43] I got this saying from Amir Kolsky and Scott Bain

```
public Sender(Recipient recipient)
{
 //...
}
```

The signature of the constructor makes it explicit that a reference to `Recipient` is required for a `Sender` to work correctly – the compiler will not allow creating a `Sender` without passing *something* as a `Recipient`[44].

Where to apply

Passing into a constructor is a great solution in cases we want to compose a sender with a recipient permanently (i.e. for the lifetime of a `Sender`). To be able to do this, a `Recipient` must, of course, exist before a `Sender` does. Another less obvious requirement for this composition is that a `Recipient` must be usable at least as long as a `Sender` is usable. A simple example of violating this requirement is this code:

```
sender = new Sender(recipient);

recipient.Dispose(); //but sender is unaware of it
                     //and may still use recipient later:
sender.DoSomething();
```

In this case, when we tell `sender` to `DoSomething()`, it uses a recipient that is already disposed of, which may lead to some nasty bugs.

Receive inside a message (i.e. as a method parameter)

Another common way of composing objects together is passing one object as a parameter of another object's method call:

```
sender.DoSomethingWithHelpOf(recipient);
```

In such a case, the objects are most often composed temporarily, just for the time of execution of this single method:

[44]Sure, we could pass a `null` but then we are the ones asking for trouble.

```
public void DoSomethingWithHelpOf(Recipient recipient)
{
  //... perform some logic

  recipient.HelpMe();

  //... perform some logic
}
```

Where to apply

Contrary to the constructor approach, where a `Sender` could hide from its user the fact that it needs a `Recipient`, in this case, the user of `Sender` is explicitly responsible for supplying a `Recipient`. In other words, there needs to be some kind of coupling between the code using `Sender` and a `Recipient`. It may look like this coupling is a disadvantage, but I know of some scenarios where it's **required** for code using `Sender` to be able to provide its own `Recipient` – it allows us to use the same sender with different recipients at different times (most often from different parts of the code):

```
//in one place
sender.DoSomethingWithHelpOf(recipient);

//in another place:
sender.DoSomethingWithHelpOf(anotherRecipient);

//in yet another place:
sender.DoSomethingWithHelpOf(yetAnotherRecipient);
```

If this ability is not required, I strongly prefer the constructor approach as it removes the (then) unnecessary coupling between code using `Sender` and a `Recipient`, giving me more flexibility.

Receive in response to a message (i.e. as a method return value)

This method of composing objects relies on an intermediary object – often an implementation of a factory pattern[45] – to supply recipients on request. To simplify things, I will use factories in examples presented in this section, although what I tell you is true for some other creational patterns[46] as well (also, later in this chapter, I'll cover some aspects of the factory pattern in depth).

To be able to ask a factory for recipients, the sender needs to obtain a reference to it first. Typically, a factory is composed with a sender through its constructor (an approach I already described). For example:

[45] http://www.netobjectives.com/PatternRepository/index.php?title=TheAbstractFactoryPattern
[46] https://en.wikipedia.org/wiki/Creational_pattern

```
var sender = new Sender(recipientFactory);
```

The factory can then be used by the `Sender` at will to get a hold of new recipients:

```
public class Sender
{
  //...

  public void DoSomething()
  {
    //ask the factory for a recipient:
    var recipient = _recipientFactory.CreateRecipient();

    //use the recipient:
    recipient.DoSomethingElse();
  }
}
```

Where to apply

I find this kind of composition useful when a new recipient is needed each time `DoSomething()` is called. In this sense, it may look much like in case of the previously discussed approach of receiving a recipient inside a message. There is one difference, however. Contrary to passing a recipient inside a message, where the code using the `Sender` passed a `Recipient` "from outside" of the `Sender`, in this approach, we rely on a separate object that is used by a `Sender` "from the inside".

To be more clear, let's compare the two approaches. Passing recipient inside a message looks like this:

```
//Sender gets a Recipient from the "outside":
public void DoSomething(Recipient recipient)
{
  recipient.DoSomethingElse();
}
```

and obtaining it from a factory:

```
//a factory is used "inside" Sender
//to obtain a recipient
public void DoSomething()
{
  var recipient = _factory.CreateRecipient();
  recipient.DoSomethingElse();
}
```

So in the first example, the decision on which `Recipient` is used is made by whoever calls `DoSomething()`. In the factory example, whoever calls `DoSomething()` does not know at all about the `Recipient` and cannot directly influence which `Recipient` is used. The factory makes this decision.

Factories with parameters

So far, all of the factories we considered had creation methods with empty parameter lists, but this is not a requirement of any sort - I just wanted to make the examples simple, so I left out everything that didn't help make my point. As the factory remains the decision-maker on which `Recipient` is used, it can rely on some external parameters passed to the creation method to help it make the decision.

Not only factories

Throughout this section, we have used a factory as our role model, but the approach of obtaining a recipient in response to a message is wider than that. Other types of objects that fall into this category include, among others: repositories[47], caches[48], builders[49], collections[50]. While they are all important concepts (which you can look up on the web if you like), they are not required to progress through this chapter so I won't go through them now.

Receive as a registered observer

This means passing a recipient to an **already created** sender (contrary to passing as constructor parameter where the recipient was passed **during** creation) as a parameter of a method that stores the reference for later use. Usually, I meet two kinds of registrations:

1. a "setter" method, where someone registers an observer by calling something like `sender.SetRecipient(rec`
   Honestly, even though it's a setter, I don't like naming it according to the convention "setWhatever()" – after Kent Beck[51] I find this convention too much implementation-focused instead of purpose-focused. Thus, I pick different names based on what domain concept is modeled by the registration method or what is its purpose. Anyway, this approach allows only one observer and setting another overwrites the previous one.

[47] https://martinfowler.com/eaaCatalog/repository.html
[48] https://en.wikipedia.org/wiki/Cache_%28computing%29
[49] http://www.blackwasp.co.uk/Builder.aspx
[50] If you never used collections before and you are not a copy-editor, then you are probably reading the wrong book :-)
[51] Kent Beck, Implementation Patterns

2. an "addition" method - where someone registers an observer by calling something like `sender.addRecipient(recipient)` - in this approach, a collection of observers needs to be maintained somewhere and the recipient registered as an observer is merely added to the collection.

Note that there is one similarity to the "passing inside a message" approach – in both, a recipient is passed inside a message. The difference is that this time, contrary to "pass inside a message" approach, the passed recipient is not used immediately (and then forgotten), but rather it's remembered (registered) for later use.

I hope I can clear up the confusion with a quick example.

Example

Suppose we have a temperature sensor that can report its current and historically mean value to whoever subscribes to it. If no one subscribes, the sensor still does its job, because it still has to collect the data for calculating a history-based mean value in case anyone subscribes later.

We may model this behavior by using an observer pattern and allow observers to register in the sensor implementation. If no observer is registered, the values are not reported (in other words, a registered observer is not required for the object to function, but if there is one, it can take advantage of the reports). For this purpose, let's make our sensor depend on an interface called `TemperatureObserver` that could be implemented by various concrete observer classes. The interface declaration looks like this:

```
public interface TemperatureObserver
{
  void NotifyOn(
    Temperature currentValue,
    Temperature meanValue);
}
```

Now we're ready to look at the implementation of the temperature sensor itself and how it uses this `TemperatureObserver` interface. Let's say that the class representing the sensor is called `TemperatureSensor`. Part of its definition could look like this:

```
public class TemperatureSensor
{
  private TemperatureObserver _observer
    = new NullObserver(); //ignores reported values

  private Temperature _meanValue
    = Temperature.Celsius(0);

  // + maybe more fields related to storing historical data

```

```
11    public void Run()
12    {
13      while(/* needs to run */)
14      {
15        var currentValue = /* get current value somehow */;
16        _meanValue = /* update mean value somehow */;
17
18        _observer.NotifyOn(currentValue, _meanValue);
19
20        WaitUntilTheNextMeasurementTime();
21      }
22    }
23  }
```

As you can see, by default, the sensor reports its values to nowhere (`NullObserver`), which is a safe default value (using a `null` for a default value instead would cause exceptions or force us to put a null check inside the `Run()` method). We have already seen such "null objects"[52] a few times before (e.g. in the previous chapter, when we introduced the `NoAlarm` class) – `NullObserver` is just another incarnation of this pattern.

Registering observers

Still, we want to be able to supply our own observer one day, when we start caring about the measured and calculated values (the fact that we "started caring" may be indicated to our application e.g. by a network packet or an event from the user interface). This means we need to have a method inside the `TemperatureSensor` class to overwrite this default "do-nothing" observer with a custom one **after** the `TemperatureSensor` instance is created. As I said, I don't like the "SetXYZ()" convention, so I will name the registration method `FromNowOnReportTo()` and make the observer an argument. Here are the relevant parts of the `TemperatureSensor` class:

```
1   public class TemperatureSensor
2   {
3     private TemperatureObserver _observer
4       = new NullObserver(); //ignores reported values
5
6     //... ... ...
7
8     public void FromNowOnReportTo(TemperatureObserver observer)
9     {
10      _observer = observer;
11    }
12
13    //... ... ...
14  }
```

[52] This pattern has a name and the name is... Null Object (surprise!). You can read more on this pattern at http://www.cs.oberlin.edu/~jwalker/nullObjPattern/ and http://www.cs.oberlin.edu/~jwalker/refs/woolf.ps (a little older document)

This allows us to overwrite the current observer with a new one should we ever need to do it. Note that, as I mentioned, this is the place where the registration approach differs from the "pass inside a message" approach, where we also received a recipient in a message, but for immediate use. Here, we don't use the recipient (i.e. the observer) when we get it, but instead, we save it for later use.

Communication of intent: optional dependency

Allowing registering recipients after a sender is created is a way of saying: "the recipient is optional – if you provide one, fine, if not, I will do my work without it". Please, don't use this kind of mechanism for **required** recipients – these should all be passed through a constructor, making it harder to create invalid objects that are only partially ready to work.

Let's examine an example of a class that:

- accepts a recipient in its constructor,
- allows registering a recipient as an observer,
- accepts a recipient for a single method invocation

This example is annotated with comments that sum up what these three approaches say:

```
1  public class Sender
2  {
3    //"I will not work without a Recipient1"
4    public Sender(Recipient1 recipient1) {...}
5
6    //"I will do fine without Recipient2 but you
7    //can overwrite the default here if you are
8    //interested in being notified about something
9    //or want to customize my default behavior"
10   public void Register(Recipient2 recipient2) {...}
11
12   //"I need a recipient3 only here and you get to choose
13   //what object to give me each time you invoke
14   //this method on me"
15   public void DoSomethingWith(Recipient3 recipient3) {...}
16 }
```

More than one observer

Now, the observer API we just skimmed over gives us the possibility to have a single observer at any given time. When we register a new observer, the reference to the old one is overwritten. This is not really useful in our context, is it? With real sensors, we often want them to report their measurements to multiple places (e.g. we want the measurements printed on the screen,

saved to a database, and used as part of more complex calculations). This can be achieved in two ways.

The first way would be to just hold a collection of observers in our sensor, and add to this collection whenever a new observer is registered:

```
private IList<TemperatureObserver> _observers
  = new List<TemperatureObserver>();

public void FromNowOnReportTo(TemperatureObserver observer)
{
  _observers.Add(observer);
}
```

In such case, reporting would mean iterating over the list of observers:

```
...
foreach(var observer in _observers)
{
  observer.NotifyOn(currentValue, meanValue);
}
...
```

The approach shown above places the policy for notifying observers inside the sensor. Many times this could be sufficient. Still, the sensor is coupled to the answers to at least the following questions:

- In what order do we notify the observers? In the example above, we notify them in order of registration.
- How do we handle errors (e.g. one of the observers throws an exception) - do we stop notifying further observers, or log an error and continue, or maybe do something else? In the example above, we stop on the first observer that throws an exception and rethrow the exception. Maybe it's not the best approach for our case?
- Is our notification model synchronous or asynchronous? In the example above, we are using a synchronous `for` loop.

We can gain a bit more flexibility by extracting the notification logic into a separate observer that would receive a notification and pass it to other observers. We can call it "a broadcasting observer". The implementation of such an observer could look like this:

```
public class BroadcastingObserver
  : TemperatureObserver,
    TemperatureObservable //I'll explain it in a second
{
  private IList<TemperatureObserver> _observers
    = new List<TemperatureObserver>();

  public void FromNowOnReportTo(TemperatureObserver observer)
  {
    _observers.Add(observer);
  }

  public void NotifyOn(
    Temperature currentValue,
    Temperature meanValue)
  {
    foreach(var observer in _observers)
    {
      observer.NotifyOn(currentValue, meanValue);
    }
  }
}
```

This `BroadcastingObserver` could be instantiated and registered like this:

```
//instantiation:
var broadcastingObserver
  = new BroadcastingObserver();

...
//somewhere else in the code...:
sensor.FromNowOnReportTo(broadcastingObserver);

...
//somewhere else in the code...:
broadcastingObserver.FromNowOnReportTo(
      new DisplayingObserver())
...
//somewhere else in the code...:
broadcastingObserver.FromNowOnReportTo(
      new StoringObserver());
...
//somewhere else in the code...:
broadcastingObserver.FromNowOnReportTo(
      new CalculatingObserver());
```

With this design, the other observers register with the broadcasting observer. However, they don't

really need to know who they are registering with - to hide it, I introduced a special interface called `TemperatureObservable`, which has the `FromNowOnReportTo()` method:

```
public interface TemperatureObservable
{
  public void FromNowOnReportTo(TemperatureObserver observer);
}
```

This way, the code that registers an observer does not need to know what the concrete observable object is.

The additional benefit of modeling broadcasting as an observer is that it would allow us to change the broadcasting policy without touching either the sensor code or the other observers. For example, we might replace our `for` loop-based observer with something like `ParallelBroadcastingObserver` that would notify each of its observers asynchronously instead of sequentially. The only thing we would need to change is the observer object that's registered with a sensor. So instead of:

```
//instantiation:
var broadcastingObserver
  = new BroadcastingObserver();

...
//somewhere else in the code...:
sensor.FromNowOnReportTo(broadcastingObserver);
```

We would have

```
//instantiation:
var broadcastingObserver
  = new ParallelBroadcastingObserver();

...
//somewhere else in the code...:
sensor.FromNowOnReportTo(broadcastingObserver);
```

and the rest of the code would remain unchanged. This is because the sensor implements:

- `TemperatureObserver` interface, which the sensor depends on,
- `TemperatureObservable` interface which the code that registers the observers depends on.

Anyway, as I said, use registering instances very wisely and only if you specifically need it. Also, if you do use it, evaluate how allowing changing observers at runtime is affecting your multithreading scenarios. This is because a collection of observers might potentially be modified by two threads at the same time.

Where are objects composed?

Ok, we went through some ways of passing a recipient to a sender. We did it from the "internal" perspective of a sender that is given a recipient. What we left out, for the most part, is the "external" perspective, i.e. who should pass the recipient into the sender?

For almost all of the approaches described in the previous chapter, there is no limitation – you pass the recipient from where you need to pass it.

There is one approach, however, that is more limited, and this approach is **passing as a constructor parameter**.

Why is that? Because, we are trying to be true to the principle of "separating objects creation from use" and this, in turn, is a result of us striving for composability.

Anyway, if an object cannot both use and create another object, we have to make special objects just for creating other objects (there are some design patterns for how to design such objects, but the most popular and useful is a **factory**) or defer the creation up to the application entry point (there is also a pattern for this, called **composition root**).

So, we have two cases to consider. I'll start with the second one – composition root.

Composition Root

let's assume, just for fun, that we are creating a mobile game where a player has to defend a castle. This game has two levels. Each level has a castle to defend. When we manage to defend the castle long enough, the level is considered completed and we move to the next one. So, we can break down the domain logic into three classes: a `Game` that has two `Levels` and each of them that contains a `Castle`. Let's also assume that the first two classes violate the principle of separating use from construction, i.e. that a `Game` creates its own levels and each `Level` creates its own castle.

A `Game` class is created in the `Main()` method of the application:

```
1  public static void Main(string[] args)
2  {
3    var game = new Game();
4
5    game.Play();
6  }
```

The `Game` creates its own `Level` objects of specific classes implementing the `Level` interface and stores them in an array:

```
public class Game
{
  private Level[] _levels = new[] {
    new Level1(), new Level2()
  };

  //some methods here that use the levels
}
```

And the `Level` implementations create their own castles and assign them to fields of interface type `Castle`:

```
public class Level1
{
  private Castle _castle = new SmallCastle();

  //some methods here that use the castle
}

public class Level2
{
  private Castle _castle = new BigCastle();

  //some methods here that use the castle
}
```

Now, I said (and I hope you see it in the code above) that the `Game`, `Level1` and `Level2` classes violate the principle of separating use from construction. We don't like this, do we? Let's try to make them more compliant with the principle.

Achieving separation of use from construction

First, let's refactor the `Level1` and `Level2` according to the principle by moving the instantiation of their castles outside. As the existence of a castle is required for a level to make sense at all – we will say this in code by using the approach of passing a castle through a `Level`'s constructor:

```
public class Level1
{
  private Castle _castle;

  //now castle is received as
  //constructor parameter
  public Level1(Castle castle)
  {
    _castle = castle;
```

```
  }

  //some methods here that use the castle
}

public class Level2
{
  private Castle _castle;

  //now castle is received as
  //constructor parameter
  public Level2(Castle castle)
  {
    _castle = castle;
  }

  //some methods here that use the castle
}
```

This was easy, wasn't it? However, it leaves us with an issue to resolve: if the instantiations of castles are not in `Level1` and `Level2` anymore, then they have to be passed by whoever creates the levels. In our case, this falls on the shoulders of `Game` class:

```
public class Game
{
  private Level[] _levels = new[] {
    //now castles are created here as well:
    new Level1(new SmallCastle()),
    new Level2(new BigCastle())
  };

  //some methods here that use the levels
}
```

But remember – this class suffers from the same violation of not separating objects use from construction as the levels did. Thus, to make this class compliant to the principle as well, we have to treat it the same as we did the level classes – move the creation of levels outside:

```
public class Game
{
  private Level[] _levels;

  //now levels are received as
  //constructor parameter
  public Game(Level[] levels)
  {
    _levels = levels;
  }

  //some methods here that use the levels
}
```

There, we did it, but again, the levels now must be supplied by whoever creates the `Game`. Where do we put them? In our case, the only choice left is the `Main()` method of our application, so this is exactly where we are going to create all the objects that we pass to a `Game`:

```
public static void Main(string[] args)
{
  var game =
    new Game(
      new Level[] {
        new Level1(new SmallCastle()),
        new Level2(new BigCastle())
  });

  game.Play();
}
```

By the way, the `Level1` and `Level2` are differed only by the castle types and this difference is no more as we refactored it out, so we can make them a single class and call it e.g. `TimeSurvivalLevel` (because such level is considered completed when we manage to defend our castle for a specific period). After this move, now we have:

```
public static void Main(string[] args)
{
  var game =
    new Game(
      new Level[] {
        new TimeSurvivalLevel(new SmallCastle()),
        new TimeSurvivalLevel(new BigCastle())
  });

  game.Play();
}
```

Looking at the code above, we might come to another funny conclusion – this violates the principle of separating use from construction as well! First, we create and connect the web of objects and then send the `Play()` message to the `game` object. Can we fix this as well?

I would say "no", for two reasons:

1. There is no further place we can defer the creation. Sure, we could move the creation of the `Game` object and its dependencies into a separate object responsible only for the creation (e.g. a factory), but it would still leave us with the question: where do we create the factory? Of course, we could use a static method to call it in our `Main()` like this: `var app = ApplicationRoot.Create()`, but that would be delegating the composition, not pulling it up.
2. The whole point of the principle we are trying to apply is decoupling, i.e. giving ourselves the ability to change one thing without having to change another. When we think of it, there is no point of decoupling the entry point of the application from the application itself, since this is the most application-specific and non-reusable part of the application we can imagine.

What I consider important is that we reached a place where the web of objects is created using the constructor approach and we have no place left to defer the creation of the web (in other words, it is as close as possible to application entry point). Such a place is called **a composition root**[53].

We say that composition root is "as close as possible" to the application entry point because there may be different frameworks in control of your application and you will not always have the `Main()` method at your service[54].

Apart from the constructor invocations, the composition root may also contain, e.g., registrations of observers (see registration approach to passing recipients) if such observers are already known at this point. It is also responsible for disposing of all objects it created that require explicit disposal after the application finishes running. This is because it creates them and thus it is the only place in the code that can safely determine when they are not needed.

The composition root above looks quite small, but you can imagine it growing a lot in bigger applications. There are techniques of refactoring the composition root to make it more readable and cleaner – we will explore such techniques in a dedicated chapter.

Factories

Earlier, I described how it isn't always possible to pass everything through the constructor. One of the approaches we discussed that we can use in such cases is **a factory**.

When we previously talked about factories, we focused on it being just a source of objects. This time we will have a much closer look at what factories are and what are their benefits.

But first, let's look at an example of a factory emerging in code that was not using it, as a mere consequence of trying to follow the principle of separating objects use from construction.

[53] http://blog.ploeh.dk/2011/07/28/CompositionRoot/
[54] For details, check Dependency Injection in .NET by Mark Seemann.

Emerging factory – example

Consider the following code that receives a frame that came from the network (as raw data), then packs it into an object, validates and applies to the system:

```
public class MessageInbound
{
  //...initialization code here...

  public void Handle(Frame frame)
  {
    // determine the type of message
    // and wrap it with an object
    ChangeMessage change = null;
    if(frame.Type == FrameTypes.Update)
    {
      change = new UpdateRequest(frame);
    }
    else if(frame.Type == FrameTypes.Insert)
    {
      change = new InsertRequest(frame);
    }
    else
    {
      throw
        new InvalidRequestException(frame.Type);
    }

    change.ValidateUsing(_validationRules);
    _system.Apply(change);
  }
}
```

Note that this code violates the principle of separating use from construction. The `change` is first created, depending on the frame type, and then used (validated and applied) in the same method. On the other hand, if we wanted to separate the construction of `change` from its use, we have to note that it is impossible to pass an instance of the `ChangeMessage` through the `MessageInbound` constructor, because this would require us to create the `ChangeMessage` before we create the `MessageInbound`. Achieving this is impossible because we can create messages only as soon as we know the frame data which the `MessageInbound` receives.

Thus, our choice is to make a special object that we would move the creation of new messages into. It would produce new instances when requested, hence the name **factory**. The factory itself can be passed through a constructor since it does not require a frame to exist – it only needs one when it is asked to create a message.

Knowing this, we can refactor the above code to the following:

```
public class MessageInbound
{
  private readonly
    MessageFactory _messageFactory;
  private readonly
    ValidationRules _validationRules;
  private readonly
    ProcessingSystem _system;

  public MessageInbound(
    //this is the factory:
    MessageFactory messageFactory,
    ValidationRules validationRules,
    ProcessingSystem system)
  {
    _messageFactory = messageFactory;
    _validationRules = validationRules;
    _system = system;
  }

  public void Handle(Frame frame)
  {
    var change = _messageFactory.CreateFrom(frame);
    change.ValidateUsing(_validationRules);
    _system.Apply(change);
  }
}
```

This way we have separated message construction from its use.

By the way, note that we extracted not only a single constructor, but the whole object creation logic. It's in the factory now:

```
public class InboundMessageFactory
 : MessageFactory
{
  ChangeMessage CreateFrom(Frame frame)
  {
    if(frame.Type == FrameTypes.Update)
    {
      return new UpdateRequest(frame);
    }
    else if(frame.Type == FrameTypes.Insert)
    {
      return new InsertRequest(frame);
    }
    else
```

```
15      {
16        throw
17          new InvalidRequestException(frame.Type);
18      }
19    }
20  }
```

And that's it. We have a factory now and the way we got to this point was by trying to adhere to the principle of separating use from construction.

Now that we are through with the example, we're ready for some more general explanation on factories.

Reasons to use factories

As demonstrated in the example, factories are objects responsible for creating other objects. They are used to achieve the separation of objects construction from their use. They are useful for creating objects that live shorter than the objects that use them. Such a shorter lifespan results from not all of the context necessary to create an object being known up-front (i.e. until a user enters credentials, we would not be able to create an object representing their account). We pass the part of the context we know up-front (a so-called **global context**) in the factory via its constructor and supply the rest that becomes available later (the so-called **local context**) in a form of factory method parameters when it becomes available:

```
1  var factory = new Factory(globalContextKnownUpFront);
2
3  //... some time later:
4  factory.CreateInstance(localContext);
```

Another case for using a factory is when we need to create a new object each time some kind of request is made (a message is received from the network or someone clicks a button):

```
1  var factory = new Factory(globalContext);
2
3  //...
4
5  //we need a fresh instance
6  factory.CreateInstance();
7
8  //...
9
10 //we need another fresh instance
11 factory.CreateInstance();
```

In the above example, two independent instances are created, even though both are created identically (there is no local context that would differentiate them).

Both these reasons were present in our example from the last chapter:

1. We were unable to create a `ChangeMessage` before knowing the actual `Frame`.
2. For each `Frame` received, we needed to create a new `ChangeMessage` instance.

Simplest factory

The simplest possible example of a factory object is something along the following lines:

```
public class MyMessageFactory
{
  public MyMessage CreateMyMessage()
  {
    return new MyMessage();
  }
}
```

Even in this primitive shape the factory already has some value (e.g. we can make `MyMessage` an abstract type and return instances of its subclasses from the factory, and the only place impacted by the change is the factory itself[55]). More often, however, when talking about simple factories, I think about something like this:

```
//Let's assume MessageFactory
//and Message are interfaces
public class XmlMessageFactory : MessageFactory
{
  public Message CreateSessionInitialization()
  {
    return new XmlSessionInitialization();
  }
}
```

Note the two things that the factory in the second example has that the one in the first example did not:

- it implements an interface (a level of indirection is introduced)
- its `CreateSessionInitialization()` method declares a return type to be an interface (another level of indirection is introduced)

Thus, we introduced two additional levels of indirection. For you to be able to use factories effectively, I need you to understand why and how these levels of indirection are useful, especially when I talk with people, they often do not understand the benefits of using factories, "because we already have the `new` operator to create objects". The point is, by hiding (encapsulating) certain information, we achieve more flexibility:

[55] A. Shalloway et al., Essential Skills For The Agile Developer.

Factories allow creating objects polymorphically (encapsulation of type)

Each time we invoke a `new` operator, we have to put the name of a concrete type next to it:

```
new List<int>(); //OK!
new IList<int>(); //won't compile...
```

This means that whenever we change our mind and instead of using `List<int>()` we want to use an object of another class (e.g. `SortedList<int>()`), we have to either change the code to delete the old type name and put new type name or provide some kind of conditional (`if-else`). Both options have drawbacks:

- changing the name of the type requires a code change in the class that calls the constructor each time we change our mind, effectively tying us to a single implementation,
- conditionals require us to know all the possible subclasses up-front and our class lacks extensibility that we often require.

Factories allow dealing with these deficiencies. Because we get objects from a factory by invoking a method, not by saying explicitly which class we want to get instantiated, we can take advantage of polymorphism, i.e. our factory may have a method like this:

```
IList<int> CreateContainerForData() {...}
```

which returns any instance of a real class that implements `IList<int>` (say, `List<int>`):

```
public IList<int> /* return type is interface */
CreateContainerForData()
{
  return new List<int>(); /* instance of concrete class */
}
```

Of course, it makes little sense for the return type of the factory to be a library class or interface like in the above example (rather, we use factories to create instances of our own classes), but you get the idea, right?

Anyway, it's typical for a declared return type of a factory to be an interface or, at worst, an abstract class. This means that whoever uses the factory, it knows only that it receives an object of a class that is implementing an interface or is derived from an abstract class. But it doesn't know exactly what *concrete* type it is. Thus, a factory may return objects of different types at different times, depending on some rules only it knows.

Time to look at a more realistic example of how to apply this. Let's say we have a factory of messages like this:

```
public class Version1ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    if(rawData.IsSessionInit())
    {
      return new SessionInit(rawData);
    }
    else if(rawData.IsSessionEnd())
    {
      return new SessionEnd(rawData);
    }
    else if(rawData.IsSessionPayload())
    {
      return new SessionPayload(rawData);
    }
    else
    {
      throw new UnknownMessageException(rawData);
    }
  }
}
```

The factory can create many different types of messages depending on what is inside the raw data, but from the perspective of the user of the factory, this is irrelevant. All that it knows is that it gets a `Message`, thus, it (and the rest of the code operating on messages in the whole application for that matter) can be written as general-purpose logic, containing no "special cases" dependent on the type of message:

```
var message = _messageFactory.NewInstanceFrom(rawData);
message.ValidateUsing(_primitiveValidations);
message.ApplyTo(_sessions);
```

Note that this code doesn't need to change in case we want to add a new type of message that's compatible with the existing flow of processing messages[56]. The only place we need to modify in such a case is the factory. For example, imagine we decided to add a session refresh message. The modified factory would look like this:

[56] although it does need to change when the rule "first validate, then apply to sessions" changes

```
public class Version1ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    if(rawData.IsSessionInit())
    {
      return new SessionInit(rawData);
    }
    else if(rawData.IsSessionEnd())
    {
      return new SessionEnd(rawData);
    }
    else if(rawData.IsSessionPayload())
    {
      return new SessionPayload(rawData);
    }
    else if(rawData.IsSessionRefresh())
    {
      //new message type!
      return new SessionRefresh(rawData);
    }
    else
    {
      throw new UnknownMessageException(rawData);
    }
  }
}
```

and the rest of the code could remain untouched.

Using the factory to hide the real type of message returned makes maintaining the code easier, because there are fewer places in the code impacted by adding new types of messages to the system or removing existing ones (in our example – in case when we do not need to initiate a session anymore) [57] – the factory hides that and the rest of the application is coded against the general scenario.

The above example demonstrated how a factory can hide that many classes can play the same role (i.e. different messages could play the role of a `Message`), but we can as well use factories to hide that the same class plays many roles. An object of the same class can be returned from different factory method, each time as a different interface and clients cannot access the methods it implements from other interfaces.

Factories are themselves polymorphic (encapsulation of rule)

Another benefit of factories over inline constructor calls is that if a factory is received an object that can be passed as an interface, which allows us to use another factory that implements the

[57] Note that this is an application of a Gang of Four guideline: “encapsulate what varies”.

same interface in its place via polymorphism. This allows replacing the rule used to create objects with another one, by replacing one factory implementation with another.

Let's get back to the example from the previous section, where we had a `Version1ProtocolMessageFactory` that could create different kinds of messages based on some flags being set on raw data (e.g. `IsSessionInit()`, `IsSessionEnd()` etc.). Imagine we decided we don't like this version anymore. The reason is that having so many separate boolean flags is too cumbersome, as with such a design, we risk receiving a message where two or more flags are set to true (e.g. someone might send a message that indicates that it's both a session initialization and a session end). Supporting such cases (e.g. by validating and rejecting such messages) requires additional effort in the code. We want to make it better before more customers start using the protocol. Thus, a new version of the protocol is conceived – a version 2. This version, instead of using several flags, uses an enum (called `MessageTypes`) to specify the message type:

```
public enum MessageTypes
{
  SessionInit,
  SessionEnd,
  SessionPayload,
  SessionRefresh
}
```

thus, instead of querying different flags, version 2 allows querying a single value that defines the message type.

Unfortunately, to sustain backward compatibility with some clients, both versions of the protocol need to be supported, each version hosted on a separate endpoint. The idea is that when all clients migrate to the new version, the old one will be retired.

Before introducing version 2, the composition root had code that looked like this:

```
var controller = new MessagingApi(new Version1ProtocolMessageFactory());
//...
controller.HostApi(); //start listening to messages
```

where `MessagingApi` has a constructor accepting the `MessageFactory` interface:

```
public MessagingApi(MessageFactory messageFactory)
{
  _messageFactory = messageFactory;
}
```

and some general message handling code:

```
var message = _messageFactory.NewInstanceFrom(rawData);
message.ValidateUsing(_primitiveValidations);
message.ApplyTo(_sessions);
```

This logic needs to remain the same in both versions of the protocol. How do we achieve this without duplicating this code for each version?

The solution is to create another message factory, i.e. another class implementing the `MessageFactory` interface. Let's call it `Version2ProtocolMessageFactory` and implement it like this:

```
//note that now it is a version 2 protocol factory
public class Version2ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    switch(rawData.GetMessageType())
    {
      case MessageTypes.SessionInit:
        return new SessionInit(rawData);
      case MessageTypes.SessionEnd:
        return new SessionEnd(rawData);
      case MessageTypes.SessionPayload:
        return new SessionPayload(rawData);
      case MessageTypes.SessionRefresh:
        return new SessionRefresh(rawData);
      default:
        throw new UnknownMessageException(rawData);
    }
  }
}
```

Note that this factory can return objects of the same classes as version 1 factory, but it makes the decision using the value obtained from `GetMessageType()` method instead of relying on the flags.

Having this factory enables us to create a `MessagingApi` instance working with either the version 1 protocol:

```
new MessagingApi(new Version1ProtocolMessageFactory());
```

or the version 2 protocol:

```
new MessagingApi(new Version2ProtocolMessageFactory());
```

and, since for the time being we need to support both versions, our composition root will have this code somewhere[58]:

[58]The two versions of the API would probably be hosted on different URLs or different ports. In a real-world scenario, these different values would probably need to be passed as constructor parameters as well.

```
var v1Controller = new MessagingApi(new Version1ProtocolMessageFactory());
var v2Controller = new MessagingApi(new Version2ProtocolMessageFactory());
//...
v1Controller.HostApi(); //start listening to messages
v2Controller.HostApi(); //start listening to messages
```

Note that the `MessagingApi` class itself did not need to change. As it depends on the `MessageFactory` interface, all we had to do was supplying a different factory object that made its decision differently.

This example shows something I like calling "encapsulation of rule". The logic inside the factory is a rule on how, when and which objects to create. Thus, if we make our factory implement an interface and have other objects depend on this interface only, we will be able to switch the rules of object creation by providing another factory without having to modify these objects (as in our case where we did not need to modify the `MessagingApi` class).

Factories can hide some of the created object dependencies (encapsulation of global context)

Let's consider another toy example. We have an application that, again, can process messages. One of the things that are done with those messages is saving them in a database and another is validation. The processing of the message is, like in previous examples, handled by a `MessageProcessing` class, which, this time, does not use any factory, but creates the messages based on the frame data itself. Let's look at this class:

```
public class MessageProcessing
{
  private DataDestination _database;
  private ValidationRules _validation;

  public MessageProcessing(
    DataDestination database,
    ValidationRules validation)
  {
    _database = database;
    _validation = validation;
  }

  public void ApplyTo(MessageData data)
  {
    //note this creation:
    var message =
      new Message(data, _database, _validation);

    message.Validate();
    message.Persist();
```

```
22
23      //... other actions
24    }
25  }
```

There is one noticeable thing about the `MessageProcessing` class. It depends on both `DataDestination` and `ValidationRules` interfaces but does not use them. The only thing it needs those interfaces for is to supply them as parameters to the constructor of a `Message`. As the number of `Message` constructor parameters grows, the `MessageProcessing` will have to change to take more parameters as well. Thus, the `MessageProcessing` class gets polluted by something that it does not directly need.

We can remove these dependencies from `MessageProcessing` by introducing a factory that would take care of creating the messages in its stead. This way, we only need to pass `DataDestination` and `ValidationRules` to the factory, because `MessageProcessing` never needed them for any reason other than creating messages. This factory may look like this:

```
1   public class MessageFactory
2   {
3     private DataDestination _database;
4     private ValidationRules _validation;
5
6     public MessageFactory(
7       DataDestination database,
8       ValidationRules validation)
9     {
10      _database = database;
11      _validation = validation;
12    }
13
14    //clients only need to pass data here:
15    public Message CreateFrom(MessageData data)
16    {
17      return
18        new Message(data, _database, _validation);
19    }
20  }
```

Now, note that the creation of messages was moved to the factory, along with the dependencies needed for this. The `MessageProcessing` does not need to take these dependencies anymore, and can stay more true to its real purpose:

```
public class MessageProcessing
{
  private MessageFactory _factory;

  //now we depend on the factory only:
  public MessageProcessing(
    MessageFactory factory)
  {
    _factory = factory;
  }

  public void ApplyTo(MessageData data)
  {
    //no need to pass database and validation
    //since they already are inside the factory:
    var message = _factory.CreateFrom(data);

    message.Validate();
    message.Persist();

    //... other actions
  }
}
```

So, instead of `DataDestination` and `ValidationRules` interfaces, the `MessageProcessing` depends only on the factory. This may not sound like a very attractive tradeoff (taking away two dependencies and introducing one), but note that whenever the `MessageFactory` needs another dependency that is like the existing two, the factory is all that will need to change. The `MessageProcessing` will remain untouched and still coupled only to the factory.

The last thing that I want to mention is that not all dependencies can be hidden inside a factory. Note that the factory still needs to receive the `MessageData` from whoever is asking for a `Message`, because the `MessageData` is not available when the factory is created. You may remember that I call such dependencies a **local context** (because it is specific to a single use of a factory and passed from where the factory creation method is called). On the other hand, what a factory accepts through its constructor can be called a **global context** (because it is the same throughout the factory lifetime). Using this terminology, the local context cannot be hidden from users of the factory, but the global context can. Thanks to this, the classes using the factory do not need to know about the global context and can stay cleaner, coupled to less things and more focused.

Factories can help increase readability and reveal intention (encapsulation of terminology)

Let's assume we are writing an action-RPG game that consists of many game levels (not to be mistaken with experience levels). Players can start a new game or continue a saved game. When they choose to start a new game, they are immediately taken to the first level with an empty

inventory and no skills. Otherwise, when they choose to continue an old game, they have to select a file with a saved state (then the game level, skills, and inventory are loaded from the file). Thus, we have two separate workflows in our game that end up with two different methods being invoked: `OnNewGame()` for a new game mode and `OnContinue()` for resuming a saved game:

```
public void OnNewGame()
{
  //...
}

public void OnContinue(PathToFile savedGameFilePath)
{
  //...
}
```

In each of these methods, we have to somehow assemble a `Game` class instance. The constructor of `Game` allows composing it with a starting level, character's inventory and a set of skills the character can use:

```
public class FantasyGame : Game
{
  public FantasyGame(
      Level startingLevel,
      Inventory inventory,
      Skills skills)
  {
  }
}
```

There is no special class for "new game" or for "resumed game" in our code. A new game is just a game starting from the first level with an empty inventory and no skills:

```
var newGame = new FantasyGame(
  new FirstLevel(),
  new BackpackInventory(),
  new KnightSkills());
```

In other words, the "new game" concept is expressed by a composition of objects rather than by a single class, called e.g. `NewGame`.

Likewise, when we want to create a game object representing a resumed game, we do it like this:

```
try
{
  saveFile.Open();

  var loadedGame = new FantasyGame(
    saveFile.LoadLevel(),
    saveFile.LoadInventory(),
    saveFile.LoadSkills());
}
finally
{
  saveFile.Close();
}
```

Again, the concept of "resumed game" is represented by a composition rather than a single class, just like in the case of "new game". On the other hand, the concepts of "new game" and "resumed game" are part of the domain, so we must make them explicit somehow or we lose readability.

One of the ways to do this is to use a factory[59]. We can create such a factory and put inside two methods: one for creating a new game, another for creating a resumed game. The code of the factory could look like this:

```
public class FantasyGameFactory : GameFactory
{
  public Game NewGame()
  {
    return new FantasyGame(
      new FirstLevel(),
      new BackpackInventory(),
      new KnightSkills());
  }

  public Game GameSavedIn(PathToFile savedGameFilePath)
  {
    var saveFile = new SaveFile(savedGameFilePath);
    try
    {
      saveFile.Open();

      var loadedGame = new FantasyGame(
        saveFile.LoadLevel(),
        saveFile.LoadInventory(),
        saveFile.LoadSkills());

      return loadedGame;
```

[59]There are simpler ways, yet none is as flexible as using factories.

```
24      }
25      finally
26      {
27        saveFile.Close();
28      }
29    }
30  }
```

Now we can use the factory in the place where we are notified of the user choice. Remember? This was the place:

```
1  public void OnNewGame()
2  {
3    //...
4  }
5
6  public void OnContinue(PathToFile savedGameFilePath)
7  {
8    //...
9  }
```

When we fill the method bodies with the factory usage, the code ends up like this:

```
1  public void OnNewGame()
2  {
3    var game = _gameFactory.NewGame();
4    game.Start();
5  }
6
7  public void OnContinue(PathToFile savedGameFilePath)
8  {
9    var game = _gameFactory.GameSavedIn(savedGameFilePath);
10   game.Start();
11 }
```

Note that using a factory helps in making the code more readable and intention-revealing. Instead of using a nameless set of connected objects, the two methods shown above ask using terminology from the domain (explicitly requesting either `NewGame()` or `GameSavedIn(path)`). Thus, the domain concepts of "new game" and "resumed game" become explicit. This justifies the first part of the name I gave this section (i.e. "Factories can help increase readability and reveal intention").

There is, however, the second part of the section name: "encapsulating terminology" which I need to explain. Here's an explanation: note that the factory is responsible for knowing what exactly the terms "new game" and "resumed game" mean. As the meaning of the terms is encapsulated in the factory, we can change this meaning throughout the application merely by changing the

code inside the factory. For example, we can say that new game starts with inventory that is not empty, but contains a basic sword and a shield, by changing the `NewGame()` method of the factory to this:

```
  public Game NewGame()
  {
    return new FantasyGame(
      new FirstLevel(),
      new BackpackInventory(
        new BasicSword(),
        new BasicShield()),
      new KnightSkills());
  }
```

Putting it all together, factories allow giving names to some specific object compositions to increase readability and they allow hiding the meaning of some of the domain terms for easier change in the future because we can modify the meaning of the encapsulated term by changing the code inside the factory methods.

Factories help eliminate redundancy

Redundancy in code means that at least two things need to change for the same reason in the same way[60]. Usually, it is understood as code duplication, but I consider "conceptual duplication" a better term. For example, the following two methods are not redundant, even though the code seems duplicated (by the way, the following is not an example of good code, just a simple illustration):

```
public int MetersToCentimeters(int value)
{
  return value*100;
}

public int DollarsToCents(int value)
{
  return value*100;
}
```

As I said, I don't consider this to be redundancy, because the two methods represent different concepts that would change for different reasons. Even if I was to extract "common logic" from the two methods, the only sensible name I could come up with would be something like `MultiplyBy100()` which, in my opinion, wouldn't add any value at all.

Note that so far, we considered four things factories encapsulate about creating objects:

[60] A. Shalloway et al., Essential Skills For The Agile Developer.

1. Type
2. Rule
3. Global context
4. Terminology

Thus, if factories didn't exist, all these concepts would leak to surrounding classes (we saw an example when we were talking about encapsulation of global context). Now, as soon as there is more than one class that needs to create instances, these things leak to all of these classes, creating redundancy. In such a case, any change to how instances are created probably means a change to all classes needing to create those instances.

Thankfully, by having a factory – an object that takes care of creating other objects and nothing else – we can reuse the ruleset, the global context and the type-related decisions across many classes without any unnecessary overhead. All we need to do is reference the factory and ask it for an object.

There are more benefits to factories, but I hope I managed to explain why I consider them a pretty darn beneficial concept for a reasonably low cost.

Summary

In the last chapter several chapters, I tried to show you a variety of ways of composing objects together. Don't worry if you feel overwhelmed, for the most part, just remember to follow the principle of separating use from construction and you should be fine.

The rules outlined here apply to most of the objects in our application. Wait, did I say most of? Not all? So there are exceptions? Yes, there are and we'll talk about them shortly when I introduce value objects, but first, we need to further examine the influence composability has on our object-oriented design approach.

Interfaces

Some objects are harder to compose with other objects, others are easier. Of course, we are striving for higher composability. Numerous factors influence this. I already discussed some of them indirectly, so time to sum things up and fill in the gaps. This chapter will deal with the role interfaces play in achieving high composability and the next one will deal with the concept of protocols.

Classes vs interfaces

As we said, a sender is composed with a recipient by obtaining a reference to it. Also, we said that we want our senders to be able to send messages to many different recipients. This is, of course, done using polymorphism.

So, one of the questions we have to ask ourselves in our quest for high composability is: on what should a sender depend on to be able to work with as many recipients as possible? Should it depend on classes or interfaces? In other words, when we plug in an object as a message recipient like this:

```
1  public Sender(Recipient recipient)
2  {
3    this._recipient = recipient;
4  }
```

Should the `Recipient` be a class or an interface?

If we assume that `Recipient` is a class, we can get the composability we want by deriving another class from it and implementing abstract methods or overriding virtual ones. However, depending on a class as a base type for a recipient has the following disadvantages:

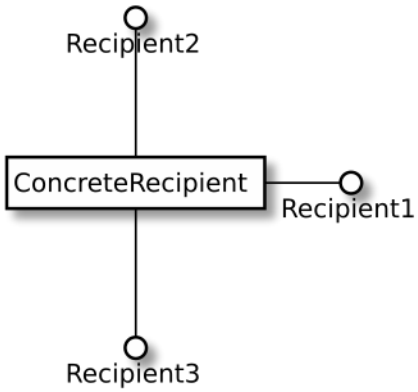
1. The recipient class may have some real dependencies. For example, if our `Recipient` depends on Windows Communication Foundation (WCF) stack, then all classes depending directly on `Recipient` will indirectly depend on WCF, including our `Sender`. The more damaging version of this problem is where such a `Recipient` class does something like opening a network connection in a constructor – the subclasses are unable to prevent it, no matter if they like it or not because a subclass has to call a superclass' constructor.
2. `Recipient`'s constructor must be invoked by any class deriving from it, which may be smaller or bigger trouble, depending on what kind of parameters the constructor accepts and what it does.
3. In languages that support single inheritance only, deriving from `Recipient` class uses up the only inheritance slot, constraining our design.

4. We must make sure to mark all the methods of `Recipient` class as `virtual` to enable overriding them by subclasses. otherwise, we won't have full composability. Subclasses will not be able to redefine all of the `Recipient` behaviors, so they will be very constrained in what they can do.

As you see, there are some difficulties using classes as "slots for composability", even if composition is technically possible this way. Interfaces are far better, just because they do not have the above disadvantages.

It is decided then that if a sender wants to be composable with different recipients, it has to accept a reference to a recipient in a form of interface reference. We can say that, by being lightweight and behaviorless, **interfaces can be treated as "slots" or "sockets" for plugging in different objects**.

As a matter of fact, on UML diagrams, one way to depict a class implementing an interface is by drawing it with a plug. Thus, it seems that the "interface as slot for pluggability" concept is not so unusual.



ConcreteRecipient class implementing three interfaces in UML. The interfaces are shown as "plugs" exposed by the class meaning it can be plugged into anything that uses any of the three interfaces

As you may have already guessed from the previous chapters, we are taking the idea of pluggability and composability to the extreme, making it one of the top priorities.

Events/callbacks vs interfaces – few words on roles

Did I just say that composability is "one of the top priorities" in our design approach? Wow, that's quite a statement, isn't it? Unfortunately for me, it also lets you raise the following argument: "Hey, interfaces are not the most extreme way of achieving composability! What about e.g. C# events feature? Or callbacks that are supported by some other languages? Wouldn't it make the classes even more context-independent and composable, if we connected them through events or callbacks, not interfaces?"

Yes, it would, but it would also strip us from another very important aspect of our design approach that I did not mention explicitly until now. This aspect is: roles. When we use interfaces, we can say that each interface stands for a role for a real object to play. When these roles are explicit, they help design and describe the communication between objects.

Let's look at an example of how not defining explicit roles can remove some clarity from the design. This is a sample method that sends some messages to two recipients held as interfaces:

```
//role players:
private readonly Role1 recipient1;
private readonly Role2 recipient2;

public void SendSomethingToRecipients()
{
  recipient1.DoX();
  recipient1.DoY();
  recipient2.DoZ();
}
```

and we compare it with similar effect achieved using callback invocation:

```
//callbacks:
private readonly Action DoX;
private readonly Action DoY;
private readonly Action DoZ;

public void SendSomethingToRecipients()
{
  DoX();
  DoY();
  DoZ();
}
```

We can see that in the second case we are losing the notion of which message belongs to which recipient – each callback is standalone from the sender's point of view. This is unfortunate because, in our design approach, we want to highlight the roles each recipient plays in the communication, to make it readable and logical. Also, ironically, decoupling using events or callbacks can make composability harder. This is because roles tell us which sets of behaviors belong together and thus, need to change together. If each behavior is triggered using a separate event or callback, an overhead is placed on us to remember which behaviors should be changed together, and which ones can change independently.

This does not mean that events or callbacks are bad. It's just that they are not fit for replacing interfaces – in reality, their purpose is a little bit different. We use events or callbacks not to tell somebody to do something, but to indicate what happened (that's why we call them events, after all...). This fits well with the observer pattern we already talked about in the previous chapter. So, instead of using observer objects, we may consider using events or callbacks instead (as in everything, there are some tradeoffs for each of the solutions). In other words, events and callbacks have their use in composition, but they are fit for a case so specific, that they cannot be treated as a default choice. The advantage of interfaces is that they bind together messages that represent coherent abstractions and convey roles in the communication. This improves readability and clarity.

Small interfaces

Ok, so we said that the interfaces are "the way to go" for reaching the strong composability we're striving for. Does merely using interfaces guarantee us that the composability will be strong? The answer is "no" – while using interfaces as "slots" is a necessary step in the right direction, it alone does not produce the best composability.

One of the other things we need to consider is the size of interfaces. Let's state one thing that is obvious regarding this:

All other things equal, smaller interfaces (i.e. with fewer methods) are easier to implement than bigger interfaces.

The obvious conclusion from this is that if we want to have strong composability, our "slots", i.e. interfaces, have to be as small as possible (but not smaller – see the previous section on interfaces vs events/callbacks). Of course, we cannot achieve this by blindly removing methods from interfaces, because this would break classes that use these methods e.g. when someone is using an interface implementation like this:

```
1  public void Process(Recipient recipient)
2  {
3    recipient.DoSomething();
4    recipient.DoSomethingElse();
5  }
```

It is impossible to remove either of the methods from the `Recipient` interface because it would cause a compile error saying that we are trying to use a method that does not exist.

So, what do we do then? We try to separate groups of methods used by different senders and move them to separate interfaces so that each sender has access only to the methods it needs. After all, a class can implement more than one interface, like this:

```
1  public class ImplementingObject
2  : InterfaceForSender1,
3    InterfaceForSender2,
4    InterfaceForSender3
5  { ... }
```

This notion of creating a separate interface per sender instead of a single big interface for all senders is known as the Interface Segregation Principle[61].

A simple example: separation of reading from writing

Let's assume we have a class in our application that represents enterprise organizational structure. This application exposes two APIs. The first one serves for notifications about changes of organizational structure by an administrator (so that our class can update its data). The second one is for client-side operations on the organizational data, like listing all employees. The interface for the organizational structure class may contain methods used by both these APIs:

[61] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

```
public interface
OrganizationStructure
{
  //////////////////////
  //used by administrator:
  //////////////////////

  void Make(Change change);
  //...other administrative methods

  //////////////////////
  //used by clients:
  //////////////////////

  void ListAllEmployees(
    EmployeeDestination destination);
  //...other client-side methods
}
```

However, the administrative API handling is done by a different code than the client-side API handling. Thus, the administrative part has no use of the knowledge about listing employees and vice-versa – the client-side one has no interest in making administrative changes. We can use this knowledge to split our interface into two:

```
public interface
OrganizationalStructureAdminCommands
{
  void Make(Change change);
  //... other administrative methods
}

public interface
OrganizationalStructureClientCommands
{
  void ListAllEmployees(
    EmployeeDestination destination);
  //... other client-side methods
}
```

Note that this does not constrain the implementation of these interfaces – a real class can still implement both of them if this is desired:

```
public class InMemoryOrganizationalStructure
: OrganizationalStructureAdminCommands,
  OrganizationalStructureClientCommands
{
  //...
}
```

In this approach, we create more interfaces (which some of you may not like), but that shouldn't bother us much because, in return, each interface is easier to implement (because the number of methods to implement is smaller than in case of one big interface). This means that composability is enhanced, which is what we want the most.

It pays off. For example, one day, we may get a requirement that all writes to the organizational structure (i.e. the admin-related operations) have to be traced. In such case, all we have to do is to create a proxy class implementing `OrganizationalStructureAdminCommands` interface, which wraps the original class' methods with a notification to an observer (that can be either the trace that is required or anything else we like):

```
public class NotifyingAdminCommands : OrganizationalStructureAdminCommands
{
  public NotifyingCommands(
    OrganizationalStructureAdminCommands wrapped,
    ChangeObserver observer)
  {
    _wrapped = wrapped;
    _observer = observer;
  }

  void Make(Change change)
  {
    _wrapped.Make(change);
    _observer.NotifyAbout(change);
  }

  //...other administrative methods
}
```

Note that when defining the above class, we only had to implement one interface:`OrganizationalStructureAdmi` and could ignore the existence of `OrganizationalStructureClientCommands`. This is because of the interface split we did before. If we had not separated interfaces for admin and client access, our `NotifyingAdminCommands` class would have to implement the `ListAllEmployees` method (and others) and make it delegate to the original wrapped instance. This is not difficult, but it's unnecessary effort. Splitting the interface into two smaller ones spared us this trouble.

Interfaces should model roles

In the above example, we split the one bigger interface into two smaller, in reality exposing that the `InMemoryOrganizationalStructure` class objects can play two roles.

Considering roles is another powerful way of separating interfaces. For example, in the organizational structure we mentioned above, we may have objects of class `Employee`, but that does not mean this class has to implement an interface called `IEmployee` or `EmployeeIfc` of anything like that. Honestly speaking, this is a situation that we may start of with, when we don't have better ideas yet, but would like to get away from as soon as we can through refactoring. What we would like to do as soon as we can is to recognize valid roles. In our example, from the point of view of the structure, the employee might play a `Node` role. If it has a parent (e.g. an organization unit) it belongs to, from its perspective it might play a `ChildUnit` role. Likewise, if it has any children in the structure (e.g. employees he manages), he can be considered their `Parent` or `DirectSupervisor`. All of these roles should be modeled using interfaces which `Employee` class implements:

```
public class Employee : Node, ChildUnit, DirectSupervisor
{
 //...
```

and each of those interfaces should be given only the methods that are needed from the point of view of objects interacting with a role modeled with this interface.

Interfaces should depend on abstractions, not implementation details

It is tempting to think that every interface is an abstraction by definition. I believe otherwise – while interfaces abstract away the concrete type of the class that implements it, they may still contain some other things not abstracted that are implementation details. Let's look at the following interface:

```
public interface Basket
{
  void WriteTo(SqlConnection sqlConnection);
  bool IsAllowedToEditBy(SecurityPrincipal user);
}
```

See the arguments of those methods? `SqlConnection` is a library object for interfacing directly with an SQL Server database, so it is a very concrete dependency. `SecurityPrincipal` is one of the core classes of .NET's authorization library that works with a user database on a local system or in Active Directory. So again, a very concrete dependency. With dependencies like that, it will be very hard to write other implementations of this interface, because we will be forced to drag around concrete dependencies and mostly will not be able to work around that if we want something different. Thus, we may say that these concrete types I mentioned are implementation details exposed in the interface. Thus, this interface is a failed abstraction. It is essential to abstract these implementation details away, e.g. like this:

```
public interface Basket
{
  void WriteTo(ProductOutput output);
  bool IsAllowedToEditBy(BasketOwner user);
}
```

This is better. For example, as `ProductOutput` is a higher-level abstraction (most probably an interface, as we discussed earlier) no implementation of the `WriteTo` method must be tied to any particular storage kind. This means that we have more freedom to develop different implementations of this method. Also, each implementation of the `WriteTo` method is more useful as it can be reused with different kinds of `ProductOutputs`.

Another example might be a data interface, i.e. an interface with getters and setters only. Looking at this example:

```
public interface Employee
{
  HumanName Name { get; set; }
  HumanAge  Age { get; set; }
  Address Address { get; set; }
  Money Pay { get; set; }
  EmploymentStatus EmploymentStatus { get; set; }
}
```

in how many different ways can we implement such an interface? Not many – the only question we can answer differently in different implementations of `Employee` is: "what is the data storage?". Everything besides this question is exposed, making this a very poor abstraction. This is similar to what Johnny and Benjamin were battling in the payroll system when they wanted to introduce another kind of employee – a contractor employee. Thus, most probably, a better abstraction would be something like this:

```
public interface Employee
{
  void Sign(Document document);
  void Send(PayrollReport payrollReport);
  void Fire();
  void GiveRaiseBy(Percentage percentage);
}
```

So the general rule is: make interfaces real abstractions by abstracting away the implementation details from them. Only then are you free to create different implementations of the interface that are not constrained by dependencies they do not want or need.

Protocols

You already know that objects are connected (composed) together and communicate through interfaces, just as in IP network. There is one more similarity, that's as important. It's *protocols*. In this section, we will look at protocols between objects and their place on our design approach.

Protocols exist

I do not want to introduce any scientific definition, so let's just establish an understanding that protocols are sets of rules about how objects communicate with each other.

Really? Are there any rules? Is it not enough the objects can be composed together through interfaces, as I explained in previous sections? Well, no, it's not enough and let me give you a quick example.

Let's imagine a class `Sender` that, in one of its methods, asks `Recipient` (let's assume `Recipient` is an interface) to extract status code from some kind of response object and makes a decision based on that code whether or not to notify an observer about an error:

```
1  if(recipient.ExtractStatusCodeFrom(response) == -1)
2  {
3    observer.NotifyErrorOccured();
4  }
```

This design is a bit simplistic but nevermind. Its role is to make a certain point. Whoever the `recipient` is, it is expected to report an error by returning a value of `-1`. Otherwise, the `Sender` (which is explicitly checking for this value) will not be able to react to the error situation appropriately. Similarly, if there is no error, the recipient must not report this by returning `-1`, because if it does, the `Sender` will mistakenly recognize this as an error. So, for example, this implementation of `Recipient`, although implementing the interface required by `Sender`, is wrong, because it does not behave as `Sender` expects it to:

```
1  public class WrongRecipient : Recipient
2  {
3    public int ExtractStatusFrom(Response response)
4    {
5      if( /* success */ )
6      {
7        return -1; // but -1 is for errors!
8      }
9      else
10     {
```

```
11        return 1; // -1 should be used!
12      }
13    }
14  }
```

So as you see, we cannot just write anything in a class implementing an interface, because of a protocol that imposes certain constraints on both a sender and a recipient.

This protocol may not only determine the return values necessary for two objects to interact properly, but it can also determine types of exceptions thrown or the order of method calls. For example, anybody using some kind of connection object would imagine the following way of using the connection: first, open it, then do something with it and close it when finished, e.g.

```
1  connection.Open();
2  connection.Send(data);
3  connection.Close();
```

Assuming the above `connection` is an implementation of `Connection` interface, if we were to implement it like this:

```
1  public class WrongConnection : Connection
2  {
3    public void Open()
4    {
5      // imagine implementation
6      // for *closing* the connection is here!!
7    }
8
9    public void Close()
10   {
11     // imagine implementation for
12     // *opening* the connection is here!!
13   }
14 }
```

then it would compile just fine but fail badly when executed. This is because the behavior would be against the protocol set between `Connection` abstraction and its user. All implementations of `Connection` must follow this protocol.

So, again, there are rules that restrict the way two objects can communicate. Both a sender and a recipient of a message must adhere to the rules, or they will not be able to work together.

The good news is that most of the time, *we* are the ones who design these protocols, along with the interfaces, so we can design them to be either easier or harder to follow by different implementations of an interface. Of course, we are wholeheartedly for the “easier” part.

Protocol stability

Remember the last story about Johnny and Benjamin when they had to make a design change to add another kind of employees (contractors) to the application? To do that, they had to change existing interfaces and add new ones. This was a lot of work. We don't want to do this much work every time we make a change, especially when we introduce a new variation of a concept that is already present in our design (e.g. Johnny and Benjamin already had the concept of "employee" and they were adding a new variation of it, called "contractor").

To achieve this, we need the protocols to be more stable, i.e. less prone to change. By drawing some conclusions from experiences of Johnny and Benjamin, we can say that they had problems with protocols stability because the protocols were:

1. complicated rather than simple
2. concrete rather than abstract
3. large rather than small

Based on the analysis of the factors that make the stability of the protocols bad, we can come up with some conditions under which these protocols could be more stable:

1. protocols should be simple
2. protocols should be abstract
3. protocols should be logical
4. protocols should be small

And some heuristics help us get closer to these qualities:

Craft messages to reflect the sender's intention

The protocols are simpler if they are designed from the perspective of the object that sends the message, not the one that receives it. In other words, method signatures should reflect the intention of senders rather than the capabilities of recipients.

As an example, let's look at a code for logging in that uses an instance of an `AccessGuard` class:

```
1 accessGuard.SetLogin(login);
2 accessGuard.SetPassword(password);
3 accessGuard.Login();
```

In this little snippet, the sender must send three messages to the `accessGuard` object: `SetLogin()`, `SetPassword()` and `Login()`, even though there is no real need to divide the logic into three steps – they are all executed in the same place anyway. The maker of the `AccessGuard` class might have thought that this division makes the class more "general purpose", but it seems this is a "premature optimization" that only makes it harder for the sender to work with the `accessGuard` object. Thus, the protocol that is simpler from the perspective of a sender would be:

```
accessGuard.LoginWith(login, password);
```

Naming by intention

Another lesson learned from the above example is: setters (like `SetLogin` and `SetPassword` in our example) rarely reflect senders' intentions – more often they are artificial "things" introduced to directly manage object state. This may also have been the reason why someone introduced three messages instead of one – maybe the `AccessGuard` class was implemented to hold two fields (login and password) inside, so the programmer might have thought someone would want to manipulate them separately from the login step... Anyway, setters should be either avoided or changed to something that reflects the intention better. For example, when dealing with the observer pattern, we don't want to say: `SetObserver(screen)`, but rather something like `FromNowOnReportCurrentWeatherTo(screen)`.

The issue of naming can be summarized with the following statement: a name of an interface should be assigned after the *role* that its implementations play and methods should be named after the *responsibilities* we want the role to have. I love the example that Scott Bain gives in his Emergent Design book[62]: if I asked you to give me your driving license number, you might've reacted differently based on whether the driving license is in your pocket, or your wallet, or your bag, or in your house (in which case you would need to call someone to read it for you). The point is: I, as a sender of this "give me your driving license number" message, do not care how you get it. I say `RetrieveDrivingLicenseNumber()`, not `OpenYourWalletAndReadTheNumber()`.

This is important because if the name represents the sender's intention, the method will not have to be renamed when new classes are created that fulfill this intention in a different way.

Model interactions after the problem domain

Sometimes at work, I am asked to conduct a design workshop. The example I often give to my colleagues is to design a system for order reservations (customers place orders and shop deliverers can reserve who gets to deliver which order). The thing that struck me the first few times I did this workshop was that even though the application was all about orders and their reservation, nearly none of the attendees introduced any kind of `Order` interface or class with `Reserve()` method on it. Most of the attendees assume that `Order` is a data structure and handle reservation by adding it to a "collection of reserved items" which can be imagined as the following code fragment:

```
// order is just a data structure,
// added to a collection
reservedOrders.Add(order)
```

While this achieves the goal in technical terms (i.e. the application works), the code does not reflect the domain.

[62]Scott Bain, Emergent Design

If roles, responsibilities, and collaborations between objects reflect the domain, then any change that is natural in the domain is natural in the code. If this is not the case, then changes that seem small from the perspective of the problem domain end up touching many classes and methods in highly unusual ways. In other words, the interactions between objects become less stable (which is exactly what we want to avoid).

On the other hand, let's assume that we have modeled the design after the domain and have introduced a proper `Order` role. Then, the logic for reserving an order may look like this:

```
order.ReserveBy(deliverer);
```

Note that this line is as stable as the domain itself. It needs to change e.g. when orders are not reserved anymore, or someone other than deliverers starts reserving the orders. Thus, I'd say the stability of this tiny interaction is darn high.

Even in cases when the understanding of the domain evolves and changes rapidly, the stability of the domain, although not as high as usual, is still one of the highest the world around us has to offer.

Another example

Let's assume that we have a code for handling alarms. When an alarm is triggered, all gates are closed, sirens are turned on and a message is sent to special forces with the highest priority to arrive and terminate the intruder. Any error in this procedure leads to shutting down power in the building. If this workflow is coded like this:

```
try
{
  gates.CloseAll();
  sirens.TurnOn();
  specialForces.NotifyWith(Priority.High);
}
catch(SecurityFailure failure)
{
  powerSystem.TurnOffBecauseOf(failure);
}
```

Then the risk of this code changing for other reasons than the change of how domain works (e.g. we do not close the gates anymore but activate laser guns instead) is small. Thus, interactions that use abstractions and methods that directly express domain rules are more stable.

So, to sum up – if a design reflects the domain, it is easier to predict how a change of domain rules will affect the design. This contributes to the maintainability and stability of the interactions and the design as a whole.

Message recipients should be told what to do, instead of being asked for information

Let's say we are paying an annual income tax yearly and are too busy (i.e. have too many responsibilities) to do this ourselves. Thus, we hire a tax expert to calculate and pay the taxes for us. He is an expert on paying taxes, knows how to calculate everything, where to submit it, etc. but there is one thing he does not know – the context. In other words, he does not know which bank we are using or what we have earned this year that we need to pay the tax for. This is something we need to give him.

Here's the deal between us and the tax expert summarized as a table:

Who?	Needs	Can provide
Us	The tax paid	context (bank, income documents)
Tax Expert	context (bank, income documents)	The service of paying the tax

It is us who hire the expert and us who initiate the deal, so we need to provide the context, as seen in the above table. If we were to model this deal as an interaction between two objects, it could e.g. look like this:

```
1 taxExpert.PayAnnualIncomeTax(
2   ourIncomeDocuments,
3   ourBank);
```

One day, our friend, Joan, tells us she needs a tax expert as well. We are happy with the one we hired, so we recommend him to Joan. She has her own income documents, but they are functionally similar to ours, just with different numbers here and there and maybe some different formatting. Also, Joan uses a different bank, but interacting with any bank these days is almost identical. Thus, our tax expert knows how to handle her request. If we model this as interaction between objects, it may look like this:

```
1 taxExpert.PayAnnualIncomeTax(
2   joansIncomeDocuments,
3   joansBank);
```

Thus, when interacting with Joan, the tax expert can still use his abilities to calculate and pay taxes the same way as in our case. This is because his skills are independent of the context.

Another day, we decide we are not happy anymore with our tax expert, so we decide to make a deal with a new one. Thankfully, we do not need to know how tax experts do their work – we just tell them to do it, so we can interact with the new one just as with the previous one:

```
//this is the new tax expert,
//but no change to the way we talk to him:

taxExpert.PayAnnualIncomeTax(
  ourIncomeDocuments,
  ourBank);
```

This small example should not be taken literally. Social interactions are far more complicated and complex than what objects usually do. But I hope I managed to illustrate with it an important aspect of the communication style that is preferred in object-oriented design: the *Tell Don't Ask* heuristic.

Tell Don't Ask means that each object, as an expert in its job, handles it well while delegating other responsibilities to other objects that are experts in their respective jobs and provide them with all the context they need to achieve the tasks it wants them to do as parameters of the messages it sends to them.

This can be illustrated with a generic code pattern:

```
recipient.DoSomethingForMe(allTheContextYouNeedToKnow);
```

This way, a double benefit is gained:

1. Our recipient (e.g. `taxExpert` from the example) can be used by other senders (e.g. pay tax for Joan) without needing to change. All it needs is a different context passed inside a constructor and messages.
2. We, as senders, can easily use different recipients (e.g. different tax experts that do the task they are assigned differently) without learning how to interact with each new one.

If you look at it, as much as bank and documents are a context for the tax expert, the tax expert is a context for us. Thus, we may say that *a design that follows the Tell Don't Ask principle creates classes that are context-independent.*

This has a very profound influence on the stability of the protocols. As much as objects are context-independent, they (and their interactions) do not need to change when the context changes.

Again, quoting Scott Bain, "what you hide, you can change". Thus, telling an object what to do requires less knowledge than asking for data and information. Again using the driver license metaphor: I may ask another person for a driving license number to make sure they have the license and that it is valid (by checking the number somewhere). I may also ask another person to provide me with directions to the place I want the first person to drive. But isn't it easier to just tell "buy me some bread and butter"? Then, whoever I ask, has the freedom to either drive or walk (if they know a good store nearby) or ask yet another person to do it instead. I don't care as long as tomorrow morning, I find the bread and butter in my fridge.

All of these benefits are, by the way, exactly what Johnny and Benjamin were aiming at when refactoring the payroll system. They went from this code, where they *asked* `employee` a lot of questions:

```
var newSalary
  = employee.GetSalary()
  + employee.GetSalary()
  * 0.1;
employee.SetSalary(newSalary);
```

to this design that *told* `employee` do do its job:

```
employee.EvaluateRaise();
```

This way, they were able to make this code interact with both `RegularEmployee` and `ContractorEmployee` the same way.

This guideline should be treated very, very seriously and applied in almost an extreme way. There are, of course, few places where it does not apply and we'll get back to them later.

Oh, I almost forgot one thing! The context that we are passing is not necessarily data. It is even more frequent to pass around behavior than to pass data. For example, in our interaction with the tax expert:

```
taxExpert.PayAnnualIncomeTax(
  ourIncomeDocuments,
  ourBank);
```

The `Bank` class is probably not a piece of data. Rather, I would imagine the `Bank` to implement an interface that looks like this:

```
public interface Bank
{
  void TransferMoney(
    Amount amount,
    AccountId sourceAccount,
    AccountId destinationAccount);
}
```

So as you can see, this `Bank` exposes behavior, not data, and it follows the Tell Don't Ask style as well (it does something well and takes all the context it needs from outside).

Where Tell Don't Ask does not apply

As I mentioned earlier, there are places where Tell Don't Ask does not apply. Here are some examples off the top of my head:

1. Factories – these are objects that produce other objects for us, so they are inherently "pull-based" – they are always asked to deliver objects.

2. Collections – they are merely containers for objects, so all we want from them is adding objects and retrieving objects (by index, by a predicate, using a key, etc.). Note, however, that when we write a class that wraps a collection inside, we want this class to expose interface shaped in a Tell Don't Ask manner.
3. Data sources, like databases – again, these are storage for data, so it is more probable that we will need to ask for this data to get it.
4. Some APIs accessed via network – while it is good to use as much Tell Don't Ask as we can, web APIs have one limitation – it is hard or impossible to pass behaviors as polymorphic objects through them. Usually, we can only pass data.
5. So-called "fluent APIs", also called "internal domain-specific languages"[63]

Even in cases where we obtain other objects from a method call, we want to be able to apply Tell Don't Ask to these other objects. For example, we want to avoid the following chain of calls:

```
1 Radio radio = radioRepository().GetRadio(12);
2 var userName = radio.GetUsers().First().GetName();
3 primaryUsersList.Add(userName);
```

This way we make the communication tied to the following assumptions:

1. Radio has many users
2. Radio must have at least one user
3. Each user must have a name
4. The name is not null

On the other hand, consider this implementation:

```
1 Radio radio = radioRepository().GetRadio(12);
2 radio.AddPrimaryUserNameTo(primaryUsersList);
```

It does not have any of the weaknesses of the previous example. Thus, it is more stable in the face of change.

Most of the getters should be removed, return values should be avoided

The above-stated guideline of "Tell Don't Ask" has a practical implication of getting rid of (almost) all the getters. We did say that each object should stick to its work and tell other objects to do their work, passing context to them, didn't we? If so, then why should we "get" anything from other objects?

For me, the idea of "no getters" was very extreme at first, but in a short time, I learned that this is in fact how I am supposed to write object-oriented code. I started learning to program using

[63]This topic is outside the scope of the book, but you can take a look at: M. Fowler, Domain-Specific Languages, Addison-Wesley 2010

procedural languages such as C, where a program was divided into procedures or functions and data structures. Then I moved on to object-oriented languages that had far better mechanisms for abstraction, but my style of coding didn't change much. I would still have procedures and functions, just divided into objects. I would still have data structures, but now more abstract, e.g. objects with setters, getters, and some query methods.

But what alternatives do we have? Well, I already introduced Tell Don't Ask, so you should know the answer. Even though you should, I want to show you another example, this time specifically about getters and setters.

Let's say that we have a piece of software that handles user sessions. A session is represented in code using a `Session` class. We want to be able to do three things with our sessions: display them on the GUI, send them through the network and persist them. In our application, we want each of these responsibilities handled by a separate class, because we think it is good if they are not tied together.

So, we need three classes dealing with data owned by the session. This means that each of these classes should somehow obtain access to the data. Otherwise, how can this data be e.g. persisted? It seems we have no choice and we have to expose it using getters.

Of course, we might re-think our choice of creating separate classes for sending, persistence, etc. and consider a choice where we put all this logic inside a `Session` class. If we did that, however, we would make a core domain concept (a session) dependent on a nasty set of third-party libraries (like a particular GUI library), which would mean that e.g. every time some GUI displaying concept changes, we will be forced to tinker in core domain code, which is pretty risky. Also, if we did that, the `Session` would be hard to reuse, because every place we would want to reuse this class, we would need to take all these heavy libraries it depends on with us. Plus, we would not be able to e.g. use `Session` with different GUI or persistence libraries. So, again, it seems like our (not so good, as we will see) only choice is to introduce getters for the information pieces stored inside a session, like this:

```
1  public interface Session
2  {
3    string GetOwner();
4    string GetTarget();
5    DateTime GetExpiryTime();
6  }
```

So yeah, in a way, we have decoupled `Session` from these third-party libraries and we may even say that we have achieved context-independence as far as `Session` itself is concerned – we can now pull all its data e.g. in a GUI code and display it as a table. The `Session` does not know anything about it. Let's see:

```
// Display sessions as a table on GUI
foreach(var session in sessions)
{
  var tableRow = TableRow.Create();
  tableRow.SetCellContentFor("owner", session.GetOwner());
  tableRow.SetCellContentFor("target", session.GetTarget());
  tableRow.SetCellContentFor("expiryTime", session.GetExpiryTime());
  table.Add(tableRow);
}
```

It seems we solved the problem by separating the data from the context it is used in and pulling data to a place that has the context, i.e. knows what to do with this data. Are we happy? We may be unless we look at how the other parts look like – remember that in addition to displaying sessions, we also want to send them and persist them. The sending logic looks like this:

```
//part of sending logic
foreach(var session in sessions)
{
  var message = SessionMessage.Blank();
  message.Owner = session.GetOwner();
  message.Target = session.GetTarget();
  message.ExpiryTime = session.GetExpiryTime();
  connection.Send(message);
}
```

and the persistence logic like this:

```
//part of storing logic
foreach(var session in sessions)
{
  var record = Record.Blank();
  dataRecord.Owner = session.GetOwner();
  dataRecord.Target = session.GetTarget();
  dataRecord.ExpiryTime = session.GetExpiryTime();
  database.Save(record);
}
```

See anything disturbing here? If no, then imagine what happens when we add another piece of information to the `Session`, say, a priority. We now have three places to update and we have to remember to update all of them every time. This is called "redundancy" or "asking for trouble". Also, the composability of these three classes is pretty bad, because they will have to change a lot only because data in a session changes.

The reason for this is that we made the `Session` class effectively a data structure. It does not implement any domain-related behaviors, it just exposes data. There are two implications of this:

1. This forces all users of this class to define session-related behaviors on behalf of the `Session`, meaning these behaviors are scattered all over the place[64]. If one is to make a change to the session, they must find all related behaviors and correct them.
2. As a set of object behaviors is generally more stable than its internal data (e.g. a session might have more than one target one day, but we will always be starting and stopping sessions), this leads to brittle interfaces and protocols – certainly the opposite of what we are striving for.

Bummer, this solution is pretty bad, but we seem to be out of options. Should we just accept that there will be problems with this implementation and move on? Thankfully, we don't have to. So far, we have found the following options to be troublesome:

1. The `Session` class containing the display, store and send logic, i.e. all the context needed – too much coupling to heavy dependencies.
2. The `Session` class to expose its data via getters, so that we may pull it where we have enough context to know how to use it – communication is too brittle and redundancy creeps in (by the way, this design will also be bad for multithreading, but that's something for another time).

Thankfully, we have a third alternative, which is better than the two we already mentioned. We can just **pass** the context **into** the `Session` class. "Isn't this just another way to do what we outlined in point 1? If we pass the context in, isn't `Session` still coupled to this context?", you may ask. The answer is: not necessarily because we can make the `Session` class depend on interfaces only instead of the real thing to make it context-independent enough.

Let's see how this plays out in practice. First, let's remove those getters from the `Session` and introduce a new method called `DumpInto()` that will take a `Destination` interface implementation as a parameter:

```
1  public interface Session
2  {
3    void DumpInto(Destination destination);
4  }
```

The implementation of `Session`, e.g. a `RealSession` can pass all fields into this destination like so:

[64]This is sometimes called Feature Envy. It means that a class is more interested in other class' data than its own.

```
public class RealSession : Session
{
  //...

  public void DumpInto(Destination destination)
  {
    destination.AcceptOwner(this.owner);
    destination.AcceptTarget(this.target);
    destination.AcceptExpiryTime(this.expiryTime);
    destination.Done();
  }

  //...
}
```

And the looping through sessions now looks like this:

```
foreach(var session : sessions)
{
  session.DumpInto(destination);
}
```

In this design, `RealSession` itself decides which parameters to pass and in what order (if that matters) – no one is asking for its data. This `DumpInto()` method is fairly general, so we can use it to implement all three mentioned behaviors (displaying, persistence, sending), by creating an implementation for each type of destination, e.g. for GUI, it might look like this:

```
public class GuiDestination : Destination
{
  private TableRow _row;
  private Table _table;

  public GuiDestination(Table table, TableRow row)
  {
    _table = table;
    _row = row;
  }

  public void AcceptOwner(string owner)
  {
    _row.SetCellContentFor("owner", owner);
  }

  public void AcceptTarget(string target)
  {
    _row.SetCellContentFor("target", target);
```

```
20    }
21
22    public void AcceptExpiryTime(DateTime expiryTime)
23    {
24      _row.SetCellContentFor("expiryTime", expiryTime);
25    }
26
27    public void Done()
28    {
29      _table.Add(_row);
30    }
31  }
```

The protocol is now more stable as far as the consumers of session data are concerned. Previously, when we had the getters in the `Session` class:

```
1  public class Session
2  {
3    string GetOwner();
4    string GetTarget();
5    DateTime GetExpiryTime();
6  }
```

the getters **had to** return **something**. So what if we had sessions that could expire and decided we want to ignore them when they do (i.e. do not display, store, send or do anything else with them)? In case of the "getter approach" seen in the snippet above, we would have to add another getter, e.g. called `IsExpired()` to the session class and remember to update each consumer the same way – to check the expiry before consuming the data... you see where this is going, don't you? On the other hand, with the current design of the `Session` interface, we can e.g. introduce a feature where the expired sessions are not processed at all in a single place:

```
1  public class TimedSession : Session
2  {
3    //...
4
5    public void DumpInto(Destination destination)
6    {
7      if(!IsExpired())
8      {
9        destination.AcceptOwner(this.owner);
10       destination.AcceptTarget(this.target);
11       destination.AcceptExpiryTime(this.expiryTime);
12       destination.Done();
13     }
14   }
15
```

```
16    //...
17  }
```

and there is no need to change any other code to get this working[65].

Another advantage of designing/making `Session` to not return anything from its methods is that we have more flexibility in applying patterns such as proxy and decorator to the `Session` implementations. For example, we can use proxy pattern to implement hidden sessions that are not displayed/stored/sent at all, but at the same time behave like another session in all the other cases. Such a proxy forwards all messages it receives to the original, wrapped `Session` object, but discards the `DumpInto()` calls:

```
1   public class HiddenSession : Session
2   {
3     private Session _innerSession;
4
5     public HiddenSession(Session innerSession)
6     {
7       _innerSession = innerSession;
8     }
9
10    public void DoSomething()
11    {
12      // forward the message to wrapped instance:
13      _innerSession.DoSomething();
14    }
15
16    //...
17
18    public void DumpInto(Destination destination)
19    {
20      // discard the message - do nothing
21    }
22
23    //...
24  }
```

The clients of this code will not notice this change at all. When we are not forced to return anything, we are more free to do as we like. Again, "tell, don't ask".

Protocols should be small and abstract

I already said that interfaces should be small and abstract, so am I not just repeating myself here? The answer is: there is a difference between the size of protocols and the size of interfaces. As an extreme example, let's take the following interface:

[65] We can even further refactor this into a state machine using a Gang of Four *State* pattern. There would be two states in such a state machine: started and expired.

```
public interface Interpreter
{
  public void Execute(string command);
}
```

Is the interface small? Of course! Is it abstract? Well, kind of, yes. Tell Don't Ask? Sure! But let's see how it's used by one of its collaborators:

```
public void RunScript()
{
  _interpreter.Execute("cd dir1");
  _interpreter.Execute("copy *.cs ../../dir2/src");
  _interpreter.Execute("copy *.xml ../../dir2/config");
  _interpreter.Execute("cd ../../dir2/");
  _interpreter.Execute("compile *.cs");
  _interpreter.Execute("cd dir3");
  _interpreter.Execute("copy *.cs ../../dir4/src");
  _interpreter.Execute("copy *.xml ../../dir4/config");
  _interpreter.Execute("cd ../../dir4/");
  _interpreter.Execute("compile *.cs");
  _interpreter.Execute("cd dir5");
  _interpreter.Execute("copy *.cs ../../dir6/src");
  _interpreter.Execute("copy *.xml ../../dir6/config");
  _interpreter.Execute("cd ../../dir6/");
  _interpreter.Execute("compile *.cs");
}
```

The point is: the protocol is neither abstract nor small. Thus, making implementations of an interface that is used as such can be pretty painful.

Summary

In this lengthy chapter, I tried to show you the often underrated value of designing communication protocols between objects. They are not a "nice thing to have", but rather a fundamental part of the design approach that makes mock objects useful, as you will see when finally we get to them. But first, I need you to swallow a few more object-oriented design ideas. I promise it will pay off.

Classes

We already covered interfaces and protocols. In our quest for composability, We need to look at classes as well. Classes:

- implement interfaces (i.e. play roles)
- communicate through interfaces to other services
- follow protocols in this communication

So in a way, what is "inside" a class is a byproduct of how objects of this class acts on the "outside". Still, it does not mean there is nothing to say about classes themselves that contributes to better composability.

Single Responsibility Principle

I already said that we want our system to be a web of composable objects. An object is a granule of composability – we cannot e.g. unplug a half of an object and plug in another half. Thus, a valid question to ask is: how big should an object be to make the composability comfortable – to let us unplug as much logic as we want, leaving the rest untouched and ready to work with the new recipients we plug in?

The answer comes with a *Single Responsibility Principle* (in short: SRP) for classes[66], which says[67]:

> A code of a class should have only one reason to change.

There has been a lot written about the principle on the web, so I am not going to be wiser than your favorite web search engine (my recent search yielded over 74 thousand results). Still, I believe it is useful to explain this principle in terms of composability.

Usually, the hard part about this principle is how to understand "a reason to change". Robert C. Martin explains[68] that this is about a single source of entropy that generates changes to the class. Which leads us to another trouble of defining a "source of entropy". So I think it's better to just give you an example.

Separating responsibilities

Remember the code Johnny and Benjamin used to apply incentive plans to employees? In case you don't, here it is (it's just a single method, not a whole class, but it should be enough for our needs):

[66] This principle can be applied to methods as well, but we are not going to cover this part, because it is not directly tied to the notion of composability and this is not a design book ;-).

[67] http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[68] https://stackoverflow.fogbugz.com/default.asp?W29030

```
public void ApplyYearlyIncentivePlan()
{
  var employees = _repository.CurrentEmployees();

  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
    employee.Save();
  }
}
```

So... how many reasons to change does this piece of code have? If we weren't talking about "reason to change" but simply a "change", the answer would be "many". For example, someone may decide that we are not giving raises anymore and the `employee.EvaluateRaise()` line would be gone. Likewise, a decision could be made that we are not giving bonuses, then the `employee.EvaluateBonus()` line would have to be removed. So, there are undoubtedly many ways this method could change. But would it be for different reasons? Actually, no. The reason in both cases would be (probably) that the CEO approved a new incentive plan. So, there is one "source of entropy" for these two changes, although there are many ways the code can change. Hence, the two changes are for the same reason.

Now the more interesting part of the discussion: what about saving the employees – is the reason for changing how we save employees the same as for the bonuses and pays? For example, we may decide that we are not saving each employee separately, because it would cause a huge performance load on our data store, but instead, we will save them together in a single batch after we finish processing the last one. This causes the code to change, e.g. like this:

```
public void ApplyYearlyIncentivePlan()
{
  var employees = _repository.CurrentEmployees();

  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
  }

  //now all employees saved once
  _repository.SaveAll(employees);
}
```

So, as you might've already guessed, the reason for this change is different than one for changing the incentive plan, thus, it is a separate responsibility and the logic for reading and storing employees should be separated from this class. The method after the separation and extraction into a new class would look something like this:

```
public void ApplyYearlyIncentivePlanTo(IEnumerable<Employee> employees)
{
  foreach(var employee in employees)
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
  }
}
```

In the example above, we moved reading and writing employees out, so that it is handled by different code – thus, the responsibilities are separated. Do we now have a code that adheres to the Single Responsibility Principle? We may, but consider this situation: the evaluation of the raises and bonuses begins getting slow and, instead of doing this for all employees in a sequential `for` loop, we would rather parallelize it to process every employee at the same time in a separate thread. After applying this change, the code could look like this (This uses C#-specific API for parallel looping, but I hope you get the idea):

```
public void ApplyYearlyIncentivePlanTo(IEnumerable<Employee> employees)
{
  Parallel.ForEach(employees, employee =>
  {
    employee.EvaluateRaise();
    employee.EvaluateBonus();
  });
}
```

Is this a new reason to change? Of course it is! Decisions on parallelizing processing come from different source than incentive plan modifications. So, we may say we encountered another responsibility and separate it. The code that remains in the `ApplyYearlyIncentivePlanTo()` method looks like this now:

```
public void ApplyYearlyIncentivePlanTo(Employee employee)
{
  employee.EvaluateRaise();
  employee.EvaluateBonus();
}
```

The looping, which is a separate responsibility, is now handled by a different class.

How far do we go?

The above example begs some questions:

1. Can we reach a point where we have separated all responsibilities?
2. If we can, how can we be sure we have reached it?

The answer to the first question is: probably no. While some reasons to change are common sense, and others can be drawn from our experience as developers or knowledge about the domain of the problem, there are always some that are unexpected and until they surface, we cannot foresee them. Thus, the answer for the second question is: "there is no way". Which does not mean we should not try to separate the different reasons we see – quite the contrary. We just don't get overzealous trying to predict every possible change.

I like the comparison of responsibilities to our usage of time in real life. The brewing time of black tea is usually around three to five minutes. This is what is usually printed on the package we buy: "3 — 5 minutes". Nobody gives the time in seconds, because such granularity is not needed. If seconds made a noticeable difference in the process of brewing tea, we would probably be given time in seconds. But they don't. When we estimate tasks in software engineering, we also use different time granularity depending on the need[69] and the granularity becomes finer as we reach a point where the smaller differences matter more.

Likewise, a simplest software program that prints "hello world" on the screen may fit into a single "main" method and we will probably not see it as several responsibilities. But as soon as we get a requirement to write "hello world" in a native language of the currently running operating system, obtaining the text becomes a separate responsibility from putting it on the screen. It all depends on what granularity we need at the moment (which, as mentioned, may be spotted from code or, in some cases, known up-front from our experience as developers or domain knowledge).

The mutual relationship between the Single Responsibility Principle and composability

The reason I am writing all this is that responsibilities[70] are the real granules of composability. The composability of objects that I have talked about a lot already is a means to achieve composability of responsibilities. So, this is what our real goal is. If we have two collaborating objects, each having a single responsibility, we can easily replace the way our application achieves one of these responsibilities without touching the other. Thus, objects conforming to SRP are the most comfortably composable and the right size.[71].

A good example from another playground where single responsibility goes hand in hand with composability is UNIX. UNIX is famous for its collection of single-purpose command-line tools, like `ls`, `grep`, `ps`, `sed` etc. The single-purposeness of these utilities along with the ability of the UNIX command line to pass an output stream of one command to the input stream of another by using the `|` (pipe) operator. For example, we may combine three commands: `ls` (lists contents of a directory), `sort` (sorts passed input) and `more` (allows comfortably viewing on the screen input that takes more than one screen) into a pipeline:

```
1 ls | sort | more
```

[69]Provided we are not using a measure such as story points.

[70]Note that I'm writing about responsibility in terms of the single responsibility principle. In Responsibility-Driven Design, responsibility means something different. See Rebecca Wirfs-Brock's clarification.

[71]Note that I am talking about responsibilities the way SRP talks about them, not the way they are understood by e.g. Responsibility-Driven Design. Thus, I am talking about the responsibilities of a class, not the responsibilities of its API.

Which displays sorted content of the current directory for a more comfortable view. This philosophy of composing a set of single-purpose tools into a more complex and more useful whole is what we are after, only that in object-oriented software development, we're using objects instead of executables. We will talk more about it in the next chapter.

Static recipients

While static fields in a class body may sometimes seem like a good idea of "sharing" recipient references between its instances and a smart way to make the code more "memory-efficient", they hurt composability more often than not. Let's take a look at a simple example to get a feeling of how static fields constraint our design.

SMTP Server

Imagine we need to implement an e-mail server that receives and sends SMTP messages[72]. We have an `OutboundSmtpMessage` class which symbolizes SMTP messages we send to other parties. To send the message, we need to encode it. For now, we always use an encoding called *Quoted-Printable*, which is declared in a separate class called `QuotedPrintableEncoding` and the class `OutboundSmtpMessage` declares a private field of this type:

```
1  public class OutboundSmtpMessage
2  {
3    //... other code
4
5    private Encoding _encoding = new QuotedPrintableEncoding();
6
7    //... other code
8  }
```

Note that each message owns its encoding object, so when we have, say, 1000000 messages in memory, we also have the same amount of encoding objects.

Premature optimization

One day we notice that it is a waste for each message to define its encoding object since an encoding is a pure algorithm and each use of this encoding does not affect further uses in any way – so we can as well have a single instance and use it in all messages – it will not cause any conflicts. Also, it may save us some CPU cycles, since creating an encoding each time we create a new message has its cost in high throughput scenarios.

But how we make the encoding shared between all instances? Out first thought – static fields! A static field seems fit for the job since it gives us exactly what we want – a single object shared across many instances of its declaring class. Driven by our (supposedly) excellent idea, we modify our `OutboundSmtpMessage` message class to hold `QuotedPrintableEncoding` instance as a static field:

[72]SMTP stands for Simple Mail Transfer Protocol and is a standard protocol for sending and receiving e-mail. You can read more on Wikipedia.

```
public class OutboundSmtpMessage
{
  //... other code

  private static Encoding _encoding = new QuotedPrintableEncoding();

  //... other code
}
```

There, we fixed it! But didn't our mommies tell us not to optimize prematurely? Oh well...

Welcome, change!

One day it turns out that in our messages, we need to support not only Quoted-Printable encoding but also another one, called *Base64*. With our current design, we cannot do that, because, as a result of using a static field, a single encoding is shared between all messages. Thus, if we change the encoding for a message that requires Base64 encoding, it will also change the encoding for the messages that require Quoted-Printable. This way, we constraint the composability with this premature optimization – we cannot compose each message with the encoding we want. All of the messages use either one encoding or another. A logical conclusion is that no instance of such class is context-independent – it cannot obtain its own context, but rather, context is forced on it.

So what about optimizations?

Are we doomed to return to the previous solution to have one encoding per message? What if this really becomes a performance or memory problem? Is our observation that we don't need to create the same encoding many times useless?

Not at all. We can still use this observation and get a lot (albeit not all) of the benefits of a static field. How do we do it? How do we achieve the sharing of encodings without the constraints of a static field? Well, we already answered this question few chapters ago – give each message an encoding through its constructor. This way, we can pass the same encoding to many, many `OutboundSmtpMessage` instances, but if we want, we can always create a message that has another encoding passed. Using this idea, we will try to achieve the sharing of encodings by creating a single instance of each encoding in the composition root and have it passed it to a message through its constructor.

Let's examine this solution. First, we need to create one of each encoding in the composition root, like this:

```
// We are in a composition root!

//...some initialization

var base64Encoding = new Base64Encoding();
var quotedPrintableEncoding = new QuotedPrintableEncoding();

//...some more initialization
```

Ok, encodings are created, but we still have to pass them to the messages. In our case, we need to create a new `OutboundSmtpMessage` object at the time we need to send a new message, i.e. on-demand, so we need a factory to produce the message objects. This factory can (and should) be created in the composition root. When we create the factory, we can pass both encodings to its constructor as a global context (remember that factories encapsulate global context?):

```
// We are in a composition root!

//...some initialization

var messageFactory
  = new StmpMessageFactory(base64Encoding, quotedPrintableEncoding);

//...some more initialization
```

The factory itself can be used for the on-demand message creation that we talked about. As the factory receives both encodings via its constructor, it can store them as private fields and pass whichever one is appropriate to a message object it creates:

```
public class SmtpMessageFactory : MessageFactory
{
  private Encoding _quotedPrintable;
  private Encoding _base64;

  public SmtpMessageFactory(
    Encoding quotedPrintable,
    Encoding base64)
  {
    _quotedPrintable = quotedPrintable;
    _base64 = base64;
  }

  public Message CreateFrom(string content, MessageLanguage language)
  {
    if(language.IsLatinBased)
    {
      //each message gets the same instance of encoding:
```

```
19        return new StmpMessage(content, _quotedPrintable);
20      }
21      else
22      {
23        //each message gets the same instance of encoding:
24        return new StmpMessage(content, _base64);
25      }
26    }
27  }
```

The performance and memory saving is not exactly as big as when using a static field (e.g. each `OutboundSmtpMessage` instance must store a separate reference to the received encoding), but it is still a huge improvement over creating a separate encoding object per message.

Where statics work?

What I wrote does not mean that statics do not have their uses. They do, but these uses are very specific. I will show you one of such uses in the next chapters after I introduce value objects.

Summary

In this chapter, I tried to give you some advice on designing classes that do not come so naturally from the concept of composability and interactions as those described in previous chapters. Still, as I hope I was able to show, they enhance composability and are valuable to us.

Object Composition as a Language

While most of the earlier chapters talked a lot about viewing object composition as a web, this one will take a different view – one of a language. These two views are remarkably similar and complement each other in guiding design.

It might surprise you that I am comparing object composition to a language, but, as I hope you'll see, there are many similarities. Don't worry, we'll get there step by step, the first step being taking a second look at the composition root.

More readable composition root

When describing object composition and composition root in particular, I promised to get back to the topic of making the composition code cleaner and more readable.

Before I do this, however, we need to get one important question answered…

Why bother?

By now you have to be sick and tired of how I stress the importance of composability. I do so, however, because I believe it is one of the most important aspects of well-designed classes. Also, I said that to reach high composability of a class, it has to be context-independent. To explain how to reach this independence, I introduced the principle of separating object use from construction, pushing the construction part away into specialized places in code. I also said that a lot can be contributed to this quality by making the interfaces and protocols abstract and having them expose as little implementation details as possible.

All of this has its cost, however. Striving for high context-independence takes away from us the ability to look at a single class and determine its context just by reading its code. Such class knows very little about the context it operates in. For example, a few chapters back we dealt with dumping sessions and I showed you that such dump method may be implemented like this:

```
1  public class RealSession : Session
2  {
3    //...
4
5    public void DumpInto(Destination destination)
6    {
7      destination.AcceptOwner(this.owner);
8      destination.AcceptTarget(this.target);
9      destination.AcceptExpiryTime(this.expiryTime);
10     destination.Done();
11   }
```

```
12
13    //...
14  }
```

Here, the session knows that whatever the destination is, `Destination` it accepts owner, target and expiry time and needs to be told when all information is passed to it. Still, reading this code, we cannot tell where the destination leads to, since `Destination` is an interface that abstracts away the details. It is a role that can be played by a file, a network connection, a console screen or a GUI widget. Context-independence enables composability.

On the other hand, as much as context-independent classes and interfaces are important, the behavior of the application as a whole is important as well. Didn't I say that the goal of composability is to be able to change the behavior of application more easily? But how can we consciously decide about changing the application's behavior when we do not understand it? And no further than the last paragraph we concluded that merely reading a class after class is not enough. We need to have a view of how these classes work together as a system. So, where is the overall context that defines the behavior of the application?

The context is in the composition code – the code that connects objects, passing real collaborators to each of them and showing how the connected parts make a whole.

Example

I assume you barely remember the alarms example I gave you in one of the first chapters of this part of the book to explain changing behavior by changing object composition. Anyway, just to remind you, we ended with a code that looked like this:

```
1  new SecureArea(
2    new OfficeBuilding(
3      new DayNightSwitchedAlarm(
4        new SilentAlarm("222-333-444"),
5        new LoudAlarm()
6      )
7    ),
8    new StorageBuilding(
9      new HybridAlarm(
10       new SilentAlarm("222-333-444"),
11       new LoudAlarm()
12     )
13   ),
14   new GuardsBuilding(
15     new HybridAlarm(
16       new SilentAlarm("919"), //call police
17       new LoudAlarm()
18     )
19   )
20 );
```

So we had three buildings all armed with alarms. The nice property of this code was that we could read the alarm setups from it, e.g. the following part of the composition:

```
new OfficeBuilding(
  new DayNightSwitchedAlarm(
    new SilentAlarm("222-333-444"),
    new LoudAlarm()
  )
),
```

meant that we were arming an office building with an alarm that calls number `222-333-444` when triggered during the day, but plays loud sirens when activated during the night. We could read this straight from the composition code, provided we knew what each object added to the overall composite behavior. So, again, the composition of parts describes the behavior of the whole. There is, however, one more thing to note about this piece of code: it describes the behavior without explicitly stating its control flow (`if`, `else`, `for`, etc.). Such description is often called *declarative* – by composing objects, we write *what* we want to achieve without writing *how* to achieve it – the control flow itself is hidden inside the objects.

Let's sum up these two conclusions with the following statement:

The composition code is a declarative description of the overall behavior of our application.

Wow, this is quite a statement, isn't it? But, as we already noticed, it is true. There is, however, one problem with treating the composition code as an overall application description: readability. Even though the composition *is* the description of the system, it doesn't read naturally. We want to see the description of behavior, but most of what we see is: `new`, `new`, `new`, `new`, `new`... There is a lot of syntactic noise involved, especially in real systems, where composition code is much longer than this tiny example. Can't we do something about it?

Refactoring for readability

The declarativeness of composition code goes hand in hand with an approach of defining so-called *fluent interfaces*. A fluent interface is an API made with readability and flow-like reading in mind. It is usually declarative and targeted towards specific domain, thus another name: *internal domain-specific languages*, in short: DSL.

There are some simple patterns for creating such domain-specific languages. One of them that can be applied to our situation is called *nested function*[73], which, in our context, means wrapping a call to `new` with a more descriptive method. Don't worry if that confuses you, we'll see how it plays out in practice in a second. We will do this step by step, so there will be a lot of repeated code, but hopefully, you will be able to closely watch the process of improving the readability of composition code.

Ok, Let's see the code again before making any changes to it:

[73] M. Fowler, Domain-Specific Languages, Addison-Wesley 2010.

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
      new SilentAlarm("222-333-444"),
      new LoudAlarm()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      new SilentAlarm("919"), //call police
      new LoudAlarm()
    )
  )
);
```

Note that we have several places where we create `SilentAlarm`. Let's move the creation of these objects into a separate method:

```
public Alarm Calls(string number)
{
  return new SilentAlarm(number);
}
```

This step may look silly, (after all, we are introducing a method wrapping a single line of code), but there is a lot of sense to it. First of all, it lets us reduce the syntax noise – when we need to create a silent alarm, we do not have to say `new` anymore. Another benefit is that we can describe the role a `SilentAlarm` instance plays in our composition (I will explain later why we are doing it using passive voice).

After replacing each invocation of `SilentAlarm` constructor with a call to this method, we get:

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      Calls("222-333-444"),
      new LoudAlarm()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
```

```
      Calls("222-333-444"),
      new LoudAlarm()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      Calls("919"), //police number
      new LoudAlarm()
    )
  )
);
```

Next, let's do the same with `LoudAlarm`, wrapping its creation with a method:

```
public Alarm MakesLoudNoise()
{
  return new LoudAlarm();
}
```

and the composition code after applying this method looks like this:

```
new SecureArea(
  new OfficeBuilding(
    new DayNightSwitchedAlarm(
      Calls("222-333-444"),
      MakesLoudNoise()
    )
  ),
  new StorageBuilding(
    new HybridAlarm(
      Calls("222-333-444"),
      MakesLoudNoise()
    )
  ),
  new GuardsBuilding(
    new HybridAlarm(
      Calls("919"), //police number
      MakesLoudNoise()
    )
  )
);
```

Note that we have removed some more `new`s in favor of something more readable. This is exactly what I meant by "reducing syntax noise".

Now let's focus a bit on this part:

```
new GuardsBuilding(
  new HybridAlarm(
    Calls("919"), //police number
    MakesLoudNoise()
  )
)
```

and try to apply the same trick of introducing a factory method to `HybridAlarm` creation. You know, we are always told that class names should be nouns and that's why `HybridAlarm` is named like this. But it does not act well as a description of what the system does. Its real function is to trigger both alarms when it is triggered. Thus, we need to come up with a better name. Should we name the method `TriggersBothAlarms()`? Naah, it's too much noise – we already know it's alarms that we are triggering, so we can leave the "alarms" part out. What about "triggers"? It says what the hybrid alarm does, which might seem good, but when we look at the composition, `Calls()` and `MakesLoudNoise()` already say what is being done. The `HybridAlarm` only says that both of those things happen simultaneously. We could leave `Trigger` if we changed the names of the other methods in the composition to look like this:

```
new GuardsBuilding(
  TriggersBoth(
    Calling("919"), //police number
    LoudNoise()
  )
)
```

But that would make the names `Calling()` and `LoudNoise()` out of place everywhere it is not being nested as `TriggersBoth()` arguments. For example, if we wanted to make another building that would only use a loud alarm, the composition would look like this:

```
new OtherBuilding(LoudNoise());
```

or if we wanted to use silent one:

```
new OtherBuilding(Calling("919"));
```

Instead, let's try to name the method wrapping construction of `HybridAlarm` just `Both()` – it is simple and communicates well the role hybrid alarms play – after all, they are just a kind of combining operators, not real alarms. This way, our composition code is now:

```
new GuardsBuilding(
  Both(
    Calls("919"), //police number
    MakesLoudNoise()
  )
)
```

and, by the way, the `Both()` method is defined as:

```
public Alarm Both(Alarm alarm1, Alarm alarm2)
{
  return new HybridAlarm(alarm1, alarm2);
}
```

Remember that `HybridAlarm` was also used in the `StorageBuilding` instance composition:

```
new StorageBuilding(
  new HybridAlarm(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

which now becomes:

```
new StorageBuilding(
  Both(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

Now the most difficult part – finding a way to make the following piece of code readable:

```
new OfficeBuilding(
  new DayNightSwitchedAlarm(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

The difficulty here is that `DayNightSwitchedAlarm` accepts two alarms that are used alternatively. We need to invent a term that:

1. Says it's an alternative.

2. Says what kind of alternative it is (i.e. that one happens at day, and the other during the night).
3. Says which alarm is attached to which condition (silent alarm is used during the day and loud alarm is used at night).

If we introduce a single name, e.g. `FirstDuringDayAndSecondAtNight()`, it will feel awkward and we will lose the flow. Just look:

```
new OfficeBuilding(
  FirstDuringDayAndSecondAtNight(
    Calls("222-333-444"),
    MakesLoudNoise()
  )
),
```

It just doesn't feel well... We need to find another approach to this situation. There are two approaches we may consider:

Approach 1: use named parameters

Named parameters are a feature of languages like Python or C#. In short, when we have a method like this:

```
public void DoSomething(int first, int second)
{
  //...
}
```

we can call it with the names of its arguments stated explicitly, like this:

```
DoSomething(first: 12, second: 33);
```

We can use this technique to refactor the creation of `DayNightSwitchedAlarm` into the following method:

```
public Alarm DependingOnTimeOfDay(
    Alarm duringDay, Alarm atNight)
{
  return new DayNightSwitchedAlarm(duringDay, atNight);
}
```

This lets us write the composition code like this:

```
new OfficeBuilding(
  DependingOnTimeOfDay(
    duringDay: Calls("222-333-444"),
    atNight: MakesLoudNoise()
  )
),
```

which is quite readable. Using named parameters has this small added benefit that it lets us pass the arguments in a different order they were declared, thanks to their names stated explicitly. This makes both following invocations valid:

```
//this is valid:
DependingOnTimeOfDay(
  duringDay: Calls("222-333-444"),
  atNight: MakesLoudNoise()
)

//arguments in different order,
//but this is valid as well:
DependingOnTimeOfDay(
  atNight: MakesLoudNoise(),
  duringDay: Calls("222-333-444")
)
```

Now, on to the second approach.

Approach 2: use method chaining

This approach is better translatable to different languages and can be used e.g. in Java and C++. This time, before I show you the implementation, let's look at the final result we want to achieve:

```
new OfficeBuilding(
  DependingOnTimeOfDay
    .DuringDay(Calls("222-333-444"))
    .AtNight(MakesLoudNoise())
  )
),
```

So as you see, this is very similar in reading, the main difference being that it's more work. It might not be obvious from the start how this kind of parameter passing works:

```
DependingOnTimeOfDay
  .DuringDay(...)
  .AtNight(...)
```

so, let's decipher it. First, `DependingOnTimeOfDay`. This is just a class:

```
public class DependingOnTimeOfDay
{
}
```

which has a static method called `DuringDay()`:

```
//note: this method is static
public static
DependingOnTimeOfDay DuringDay(Alarm alarm)
{
  return new DependingOnTimeOfDay(alarm);
}

//The constructor is private:
private DependingOnTimeOfDay(Alarm dayAlarm)
{
  _dayAlarm = dayAlarm;
}
```

Now, this method seems strange, doesn't it? It is a static method that returns an instance of its enclosing class (not an actual alarm!). Also, the private constructor stores the passed alarm inside for later... why?

The mystery resolves itself when we look at another method defined in the `DependingOnTimeOfDay` class:

```
//note: this method is NOT static
public Alarm AtNight(Alarm nightAlarm)
{
  return new DayNightSwitchedAlarm(_dayAlarm, nightAlarm);
}
```

This method is not static and it returns the alarm that we were trying to create. To do so, it uses the first alarm passed through the constructor and the second one passed as its parameter. So if we were to take this construct:

```
DependingOnTimeOfDay //class
  .DuringDay(dayAlarm) //static method
  .AtNight(nightAlarm) //non-static method
```

and assign a result of each operation to a separate variable, it would look like this:

```
DependingOnTimeOfDay firstPart = DependingOnTimeOfDay.DuringDay(dayAlarm);
Alarm alarm = firstPart.AtNight(nightAlarm);
```

Now, we can just chain these calls and get the result we wanted to:

```
new OfficeBuilding(
  DependingOnTimeOfDay
    .DuringDay(Calls("222-333-444"))
    .AtNight(MakesLoudNoise())
  )
),
```

The advantage of this solution is that it does not require your programming language of choice to support named parameters. The downside is that the order of the calls is strictly defined. The `DuringDay` returns an object on which `AtNight` is invoked, so it must come first.

Discussion continued

For now, I will assume we have chosen approach 1 because it is simpler.

Our composition code looks like this so far:

```
new SecureArea(
  new OfficeBuilding(
    DependingOnTimeOfDay(
      duringDay: Calls("222-333-444"),
      atNight: MakesLoudNoise()
    )
  ),
  new StorageBuilding(
    Both(
      Calls("222-333-444"),
      MakesLoudNoise()
    )
  ),
  new GuardsBuilding(
    Both(
      Calls("919"), //police number
      MakesLoudNoise()
    )
  )
);
```

There are a few more finishing touches we need to make. First of all, let's try and extract these dial numbers like `222-333-444` into constants. When we do so, then, for example, this code:

```
Both(
  Calls("919"), //police number
  MakesLoudNoise()
)
```

becomes

```
Both(
  Calls(Police),
  MakesLoudNoise()
)
```

And the last thing is to hide the creation of the following classes: `SecureArea`, `OfficeBuilding`, `StorageBuilding`, `GuardsBuilding` and we have this:

```
SecureAreaContaining(
  OfficeBuildingWithAlarmThat(
    DependingOnTimeOfDay(
      duringDay: Calls(Guards),
      atNight: MakesLoudNoise()
    )
  ),
  StorageBuildingWithAlarmThat(
    Both(
      Calls(Guards),
      MakesLoudNoise()
    )
  ),
  GuardsBuildingWithAlarmThat(
    Both(
      Calls(Police),
      MakesLoudNoise()
    )
  )
);
```

And here it is – the real, declarative description of our application! The composition reads better than when we started, doesn't it?

Composition as a language

Written this way, object composition has another important property – it is extensible and can be extended using the same terms that are already used (of course we can add new ones as well).

For example, using the methods we invented to make the composition more readable, we may write something like this:

```
Both(
  Calls(Police),
  MakesLoudNoise()
)
```

but, using the same terms, we may as well write this:

```
Both(
  Both(
    Calls(Police),
    Calls(Security)),
  Both(
    Calls(Boss),
    MakesLoudNoise()))
)
```

to obtain different behavior. Note that we have invented something that has these properties:

1. It defines some kind of *vocabulary* – in our case, the following "words" are form part of the vocabulary: `Both`, `Calls`, `MakesLoudNoise`, `DependingOnTimeOfDay`, `atNight`, `duringDay`, `SecureAreaContaining`, `GuardsBuildingWithAlarmThat`, `OfficeBuildingWithAlarmThat`.
2. It allows combining the words from the vocabulary. These combinations have meaning, which is based solely on the meaning of used words and the way they are combined. For example: `Both(Calls(Police), Calls(Guards))` has the meaning of "calls both police and guards when triggered" – thus, it allows us to combine words into *sentences*.
3. Although we are quite liberal in defining behaviors for alarms, there are some rules as to what can be composed with what (for example, we cannot compose guards building with an office, but each of them can only be composed with alarms). Thus, we can say that the *sentences* we write have to obey certain rules that look a lot like *a grammar*.
4. The vocabulary is *constrained to the domain* of alarms. On the other hand, it *is more powerful and expressive* as a description of this domain than a combination of `if` statements, `for` loops, variable assignments and other elements of a general-purpose language. It is tuned towards describing rules of a domain on a *higher level of abstraction*.
5. The sentences written define the behavior of the application – so by writing sentences like this, we still write software! Thus, what we do by combining *words* into *sentences* constrained by a *grammar* is still *programming*!

All of these points suggest that we have created a *Domain-Specific Language*[74], which, by the way, is a *higher-level language*, meaning we describe our software on a higher level of abstraction.

[74] M. Fowler, Domain-Specific Languages, Addison-Wesley 2010.

The significance of a higher-level language

So… why do we need a higher-level language to describe the behavior of our application? After all, expressions, statements, loops and conditions (and objects and polymorphism) are our daily bread and butter. Why invent something that moves us away from this kind of programming into something "domain-specific"?

My main answer is: to deal with complexity more effectively.

What's complexity? For our purpose, we can approximately define it as the number of different decisions our application needs to make. As we add new features and fix errors or implement missed requirements, the complexity of our software grows. What can we do when it grows larger than we can manage? We have the following choices:

1. Remove some decisions – i.e. remove features from our application. This is very cool when we *can* do this, but there are times when this might be unacceptable from the business perspective.
2. Optimize away redundant decisions – this is about making sure that each decision is made once in the code base – I already showed you some examples how polymorphism can help with that.
3. Use 3rd party component or a library to handle some of the decisions for us – while this is quite easy for "infrastructure" code and utilities, it is very, very hard (impossible?) to find a library that will describe our "domain rules" for us. So if these rules are where the real complexity lies (and often they are), we are still left alone with our problem.
4. Hide the decisions by programming on a higher level of abstraction – this is what we did in this chapter so far. The advantage is that it allows us to reduce the complexity of our domain, by creating bigger building blocks from which a behavior description can be created.

So, as you see, only the last of the above points really helps in reducing domain complexity. This is where the idea of domain-specific languages falls in. If we carefully craft our object composition into a set of domain-specific languages (one is often too little in all but simplest cases), one day we may find that we are adding new features by writing new sentences in these languages in a declarative way rather than adding new imperative code. Thus, if we have a good language and a firm understanding of its vocabulary and grammar, we can program on a higher level of abstraction which is more expressive and less complex.

This is very hard to achieve – it requires, among others:

1. A huge discipline across a development team.
2. A sense of direction of how to structure the composition and where to lead the language designs as they evolve.
3. Merciless refactoring.
4. Some minimal knowledge of language design and experience in doing so.
5. Knowledge of some techniques (like the ones we used in our example) that make constructs written in general-purpose language look like another language.

Of course, not all parts of the composition make good material to being structured like a language. Despite these difficulties, I think it's well worth the effort. Programming on a higher level of abstraction with declarative code rather than imperative is where I place my hope for writing maintainable and understandable systems.

Some advice

So, eager to try this approach? Let me give you a few pieces of advice first:

Evolve the language as you evolve code

At the beginning of this chapter, we achieved our higher-level language by refactoring already existing object composition. This does not at all mean that in real projects we need to wait for a lot of composition code to appear and then try to wrap all of it. I indeed did just that in the alarm example, but this was just an example and its purpose was mainly didactical.

In reality, the language is better off evolving along with the composition it describes. One reason for this is because there is a lot of feedback about the composability of the design gained by trying to put a language on top of it. As I said in the chapter on single responsibility, if objects are not comfortably composable, something is probably wrong with the distribution of responsibilities between them (for comparison of wrongly placed responsibilities, imagine a general-purpose language that would not have a separate `if` and `for` constructs but only a combination of them called `forif` :-)). Don't miss out on this feedback!

The second reason is that even if you can safely refactor all the code because you have an executable Specification protecting you from making mistakes, it's just too many decisions to handle at once (plus it takes a lot of time and your colleagues keep adding new code, don't they?). Good language grows and matures organically rather than being created in a big bang effort. Some decisions take time and a lot of thought to be made.

Composition is not a single DSL, but a series of mini DSLs[75]

I already briefly noted this. While it may be tempting to invent a single DSL to describe the whole application, in practice it is hardly possible, because our applications have different subdomains that often use different sets of terms. Rather, it pays off to hunt for such subdomains and create smaller languages for them. The alarm example shown above would probably be just a small part of the real composition. Not all parts would lend themselves to shape this way, at least not instantly. What starts as a single class might become a subdomain with its own vocabulary at some point. We need to pay attention. Hence, we still want to apply some of the DSL techniques even to those parts of the composition that are not easily turned into DSLs and hunt for an occasion when we can do so.

As Nat Pryce puts it[76], it's all about:

[75] A reader noted that the ideas in this section are remarkably similar to the notion of Bounded Contexts in a book: E. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, Prentice Hall 2003.

[76] http://www.natpryce.com/articles/000783.html

> (…) clearly expressing the dependencies between objects in the code that composes them, so that the system structure can easily be refactored, and aggressively refactoring that compositional code to remove duplication and express intent, and thereby raising the abstraction level at which we can program (…). The end goal is to need less and less code to write more and more functionality as the system grows.

For example, a mini-DSL for setting up handling of an application configuration updates might look like this:

```
return ConfigurationUpdates(
  Of(log),
  Of(localSettings),
  OfResource(Departments()),
  OfResource(Projects()));
```

Reading this code should not be difficult, especially when we know what each term in the sentence means. This code returns an object handling configuration updates of four things: application log, local settings, and two resources (in this subdomain, resources mean things that can be added, deleted and modified). These two resources are: departments and projects (e.g. we can add a new project or delete an existing one).

Note that the constructs of this language make sense only in a context of creating configuration update handlers. Thus, they should be restricted to this part of composition. Other parts that have nothing to do with configuration updates, should not need to know these constructs.

Do not use an extensive amount of DSL tricks

In creating internal DSLs, one can use a lot of neat tricks, some of them being very "hacky" and twisting the general-purpose language in many ways to achieve "flluent" syntax. But remember that the composition code is to be maintained by your team. Unless each and every member of your team is an expert on creating such DSLs, do not show off with too many, too sophisticated tricks. Stick with a few of the proven ones that are simple to use and work, like the ones I have used in the alarm example.

Martin Fowler[77] describes a lot of tricks for creating such DSLs and at the same time warns against using too many of them in the same language.

Factory method nesting is your best friend

One of the DSL techniques, the one I have used the most, is factory method nesting. Basically, it means wrapping a constructor (or constructors – no one said each factory method must wrap exactly one `new`) invocation with a method that has a name more fitting for a context it is used in (and which hides the obscurity of the `new` keyword). This technique is what makes this:

[77] M. Fowler, Domain-Specific Languages, Addison-Wesley 2010.

```
new HybridAlarm(
  new SilentAlarm("222-333-444"),
  new LoudAlarm()
)
```

look like this:

```
Both(
  Calls("222-333-444"),
  MakesLoudNoise()
)
```

As you probably remember, in this case, each method wraps a constructor, e.g. `Calls()` is defined as:

```
public Alarm Calls(string number)
{
  return new SilentAlarm(number);
}
```

This technique is great for describing any kind of tree and graph-like structures as each method provides a natural scope for its arguments:

```
Method1( //beginning of scope
  NestedMethod1(),
  NestedMethod2()
);       //end of scope
```

Thus, it is a natural fit for object composition, which *is* a graph-like structure.

This approach looks great on paper but it's not like everything just fits in all the time. There are two issues with factory methods that we need to address.

Where to put these methods?

In the usual case, we want to be able to invoke these methods without any qualifier before them, i.e. we want to call `MakesLoudNoise()` instead of `alarmsFactory.MakesLoudNoise()` or `this.MakesLoudNoise()` or anything.

If so, where do we put such methods?

There are two options[78]:

1. Put the methods in the class that performs the composition.
2. Put the methods in a superclass.

[78]In some languages, there is a third way: Java lets us use static imports which are part of C# as well starting with version 6.0. C++ has always supported bare functions, so it's not a topic there.

Apart from that, we can choose between:

1. Making the factory methods static.
2. Making the factory methods non-static.

First, let's consider the dilemma of putting in composing class vs having a superclass to inherit from. This choice is mainly determined by reuse needs. The methods that we use in one composition only and do not want to reuse are mostly better off as private methods in the composing class. On the other hand, the methods that we want to reuse (e.g. in other applications or services belonging to the same system), are better put in a superclass that we can inherit from. Also, a combination of the two approaches is possible, where the superclass contains a more general method while the composing class wraps it with another method that adjusts the creation to the current context. By the way, remember that in many languages, we can inherit from a single class only – thus, putting methods for each language in a separate superclass forces us to distribute the composition code across several classes, each inheriting its own set of methods and returning an object or several objects. This is not bad at all – quite the contrary, this is something we'd like to have because it enables us to evolve a language and sentences written in this language in an isolated context.

The second choice between static and non-static is one of having access to instance fields – instance methods have this access, while static methods do not. Thus, if the following is an instance method of a class called `AlarmComposition`:

```
public class AlarmComposition
{
  //...

  public Alarm Calls(string number)
  {
    return new SilentAlarm(number);
  }

  //...
}
```

and I need to pass an additional dependency to `SilentAlarm` that I do not want to show in the main composition code, I am free to change the `Calls` method to:

```
public Alarm Calls(string number)
{
  return new SilentAlarm(
    number,
    _hiddenDependency) //field
}
```

and this new dependency may be passed to the `AlarmComposition` via its constructor:

```
public AlarmComposition(
  HiddenDependency hiddenDependency)
{
  _hiddenDependency = hiddenDependency;
}
```

This way, I can hide it from the main composition code. This is the freedom I do not have with static methods.

Use implicit collections instead of explicit ones

Most object-oriented languages support passing variable argument lists (e.g. in C# this is achieved with the `params` keyword, while Java has `...` operator). This is valuable in composition, because we often want to be able to pass an arbitrary number of objects to some places. Again, coming back to this composition:

```
return ConfigurationUpdates(
  Of(log),
  Of(localSettings),
  OfResource(Departments()),
  OfResource(Projects()));
```

the `ConfigurationUpdates()` method is using variable argument list:

```
public ConfigurationUpdates ConfigurationUpdates(
  params ConfigurationUpdate[] updates)
{
  return new MyAppConfigurationUpdates(updates);
}
```

Note that we could, of course, pass the array of `ConfigurationUpdate` instances using the explicit definition: `new ConfigurationUpdate[] {...}`, but that would greatly hinder readability and flow of this composition. See for yourself:

```
return ConfigurationUpdates(
  new [] { //explicit definition brings noise
    Of(log),
    Of(localSettings),
    OfResource(Departments()),
    OfResource(Projects())
  }
);
```

Not so pretty, huh? This is why we like the ability to pass variable argument lists as it enhances readability.

A single method can create more than one object

No one said each factory method must create one and only one object. For example, take a look again at this method creating configuration updates:

```
public ConfigurationUpdates ConfigurationUpdates(
  params ConfigurationUpdate[] updates)
{
  return new MyAppConfigurationUpdates(updates);
}
```

Now, let's assume we need to trace each invocation on the instance of `ConfigurationUpdates` class and we want to achieve this by wrapping the `MyAppConfigurationUpdates` instance with a tracing proxy (a wrapping object that passes the calls along to a real object, but writes some trace messages before and after it does). For this purpose, we can reuse the method we already have, just adding the additional object creation there:

```
public ConfigurationUpdates ConfigurationUpdates(
  params ConfigurationUpdate[] updates)
{
  //now two objects created instead of one:
  return new TracedConfigurationUpdates(
    new MyAppConfigurationUpdates(updates)
  );
}
```

Note that the `TracedConfigurationUpdates` is not important from the point of view of the composition – it is pure infrastructure code, not a new domain rule. Because of that, it may be a good idea to hide it inside the factory method.

Summary

In this chapter, I tried to convey to you a vision of object composition as a language, with its own vocabulary, its own grammar, keywords and arguments. We can compose the words from the vocabulary in different sentences to create new behaviors on a higher level of abstraction.

This area of object-oriented design is something I am still experimenting with, trying to catch up with what authorities on this topic share. Thus, I am not as fluent in it as in other topics covered in this book. Expect this chapter to grow (maybe into several chapters) or to be clarified in the future. For now, if you feel you need more information, please take a look at the video by Steve Freeman and Nat Pryce called "Building on SOLID foundations"[79].

[79] https://vimeo.com/105785565

Value Objects

I spent several chapters talking about composing objects in a web where real implementation was hidden and only interfaces were exposed. These objects exchanged messages and modeled roles in our domain.

However, this is just one part of the object-oriented design approach that I'm trying to explain. Another part of the object-oriented world, complementary to what we have been talking about are value objects. They have their own set of design constraints and ideas, so most of the concepts from the previous chapters do not apply to them, or apply differently.

What is a value object?

In short, values are usually seen as immutable quantities, measurements[80] or other objects that are compared by their content, not their identity. There are some examples of values in the libraries of our programming languages. For example, `String` class in Java or C# is a value object because it is immutable and every two strings are considered equal when they contain the same data. Other examples are the primitive types that are built-in into most programming languages, like numbers or characters.

Most of the values that are shipped with general-purpose libraries are quite primitive or general. There are many times, however, when we want to model a domain abstraction as a value object. Some examples include date and time (which nowadays is usually a part of the standard library, because it is usable in so many domains), money, temperature, but also things such as file paths or resource identifiers.

As you may have already spotted when reading this book, I'm really bad at explaining things without examples, so here is one:

Example: money and names

Imagine we are developing a web store for a customer. There are different kinds of products sold and the customer wants to have the ability to add new products.

Each product has at least two important attributes: name and price (there are others like quantity, but let's leave them alone for now).

Now, imagine how you would model these two things - would the name be modeled as a mere `string` and price be a `double` or a `decimal` type?

Let's say that we have indeed decided to use a `decimal` to hold a price, and a `string` to hold a name. Note that both are generic library types, not connected to any domain. Is it a good choice to use "library types" for domain abstractions? We shall soon find out…

[80] S. Freeman, N. Pryce, Growing Object-Oriented Software Guided by Tests, Addison-Wesley Professional, 2009

Time passes...

One day, it turns out that these values must be shared across several subdomains of the system. For example:

1. The website needs to display them
2. They are used in income calculations
3. They are taken into account when defining and checking discount rules (e.g. "buy three, pay for two")
4. They must be supplied when printing invoices

etc.

The code grows larger and larger and, as the concepts of product name and price are among the main concepts of the application, they tend to land in many places.

Change request

Now, imagine that one of the following changes must make its way into the system:

1. The product name must be compared as case insensitive since the names of the products are always printed in uppercase on the invoice. Thus, creating two products that differ only in a letter case (eg. "laptop" and "LAPTOP") would confuse the customers as both these products look the same on the invoice. Also, the only way one would create two products that differ by letter case only is by mistake and we want to avoid that.
2. The product name is not enough to differentiate a product. For example, PC manufacturers have the same models of notebooks in different configurations (e.g. different amount of RAM or different processor models inside). So each product will receive an additional identifier that will have to be taken into account during comparisons.
3. To support customers from different countries, new currencies must be supported.

In the current situation, these changes are painful to make. Why? It's because we used primitive types to represent the things that would need to change, which means we're coupled in multiple places to a particular implementation of the product name (`string`) and a particular implementation of money (e.g. `decimal`). This wouldn't be so bad, if not for the fact that we're coupled to implementation we cannot change!

Are we sentenced to live with issues like that in our code and cannot do anything about it? Let's find out by exploring the options we have.

From now on, let's put the money concept aside and focus only on the product name, as both name and price are similar cases with similar solutions, so it's sufficient for us to consider just one of them.

What options do we have to address product name changes?

To support new requirements, we have to find all places where we use the product name (by the way, an IDE will not help us much in this search, because we would be searching for all the occurrences of type `string`) and make the same change. Every time we need to do something like this (i.e. we have to make the same change in multiple places and there is a non-zero possibility we'll miss at least one of those places), it means that we have introduced redundancy. Remember? We talked about redundancy when discussing factories and mentioned that redundancy is about conceptual duplication that forces us to make the same change (not literally, but conceptually) in several places.

Al Shalloway coined a humorous "law" regarding redundancy, called *The Shalloway's Law*, which says:

> Whenever the same change needs to be applied in N places and N > 1, Shalloway will find at most N-1 such places.

An example application of this law would be:

> Whenever the same change needs to be applied in 4 places, Shalloway will find at most 3 such places.

While making fun of himself, Al described something that I see common of myself and some other programmers - that conceptual duplication makes us vulnerable and when dealing with it, we have no advanced tools to help us - just our memory and patience.

Thankfully, there are multiple ways to approach this redundancy. Some of them are better and some are worse[81].

Option one - just modify the implementation in all places

This option is about leaving the redundancy where it is and just making the change in all places, hoping that this is the last time we change anything related to the product name.

So let's say we want to add comparison with the letter case ignored. Using this option would lead us to find all the places where we do something like this:

```
1 if(productName == productName2)
2 {
3 ..
```

or

[81]All engineering decisions are trade-offs anyway, so I should say "some of them make better trade-offs in our context, and some make worse".

```
if(String.Equals(productName, productName2))
{
..
```

And change them to a comparisong that ignores case, e.g.:

```
if(String.Equals(productName, productName2,
   StringComparison.OrdinalIgnoreCase))
{
..
```

This deals with the problem, at least for now, but in the long run, it can cause some trouble:

1. It will be very hard[82] to find all these places and chances are you'll miss at least one. This is an easy way for a bug to creep in.
2. Even if this time you'll be able to find and correct all the places, every time the domain logic for product name comparisons changes (e.g. we'll have to use `InvariantIgnoreCase` option instead of `OrdinalIgnoreCase` for some reasons, or handle the case I mentioned earlier where comparison includes an identifier of a product), you'll have to do it over. And Shalloway's Law applies the same every time. In other words, you're not making things better.
3. Everyone who adds new logic that needs to compare product names in the future will have to remember that character case is ignored in such comparisons. Thus, they will need to keep in mind that they should use `OrdinalIgnoreCase` option whenever they add new comparisons somewhere in the code. If you want to know my opinion, accidental violation of this convention in a team that has either a fair size or more than minimal staff turnover rate is just a matter of time.
4. Also, there are other changes that will be tied to the concept of product name equality in a different way (for example, hash sets and hash tables use hash codes to help find the right objects) and you'll need to find those places and make changes there as well.

So, as you can see, this approach does not make things any better. It is this approach that led us to the trouble we are trying to get away in the first place.

Option two - use a helper class

We can address the issues #1 and #2 of the above list (i.e. the necessity to change multiple places when the comparison logic of product names changes) by moving this comparison into a static helper method of a helper class, (let's simply call it `ProductNameComparison`) and make this method a single place that knows how to compare product names. This would make each of the places in the code when comparison needs to be made look like this:

[82] http://www.netobjectives.com/blogs/shalloway%E2%80%99s-law-and-shalloway%E2%80%99s-principle

```
if(ProductNameComparison.AreEqual(productName, productName2))
{
..
```

Note that the details of what it means to compare two product names are now hidden inside the newly created static `AreEqual()` method. This method has become the only place that knows these details and each time the comparison needs to be changed, we have to modify this method alone. The rest of the code just calls this method without knowing what it does, so it won't need to change. This frees us from having to search and modify this code each time the comparison logic changes.

However, while it protects us from the change of comparison logic indeed, it's still not enough. Why? Because the concept of a product name is still not encapsulated - a product name is still a `string` and it allows us to do everything with it that we can do with any other `string`, even when it does not make sense for product names. This is because, in the domain of the problem, product names are not sequences of characters (which `stringss` are), but an abstraction with a special set of rules applicable to it. By failing to model this abstraction appropriately, we can run into a situation where another developer who starts adding some new code may not even notice that product names need to be compared differently than other strings and just use the default comparison of a `string` type.

Other deficiencies of the previous approach apply as well (as mentioned, except for issues #1 and #2).

Option three - encapsulate the domain concept and create a "value object"

I think it's more than clear now that a product name is a not "just a string", but a domain concept and as such, it deserves its own class. Let us introduce such a class then, and call it `ProductName`. Instances of this class will have `Equals()` method overridden[83] with the logic specific to product names. Given this, the comparison snippet is now:

```
// productName and productName2
// are both instances of ProductName
if(productName.Equals(productName2))
{
..
```

How is it different from the previous approach where we had a helper class, called `ProductNameComparison`? Previously the data of a product name was publicly visible (as a string) and we used the helper class only to store a function operating on this data (and anybody could create their own functions somewhere else without noticing the ones we already added). This time, the data of the product name is hidden[84] from the outside world. The only available way to operate on this data is through the `ProductName`'s public interface (which exposes only those methods that we

[83] and, for C#, overriding equality operators (== and !=) is probably a good idea as well, not to mention `GetHashCode()` (See https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/equality-operators)

[84] In reality, this is only partially true. For example, we will have to override `ToString()` somewhere anyway to ensure interoperability with 3rd party libraries that don't know about our `ProductName` type, but will accept string arguments. Also, one can always use reflection to get private data. I hope you get the point though :-).

think make sense for product names and no more). In other words, whereas before we were dealing with a general-purpose type we couldn't change, now we have a domain-specific type that's completely under our control. This means we can freely change the meaning of two names being equal and this change will not ripple throughout the code.

In the following chapters, I will further explore this example of product names to show you some properties of value objects.

Value object anatomy

In the previous chapter, we saw a value object - `ProductName` in action. In this chapter, we'll study its anatomy - line by line, field by field, method after method. After doing this, you'll hopefully have a better feel of some of the more general properties of value objects.

Let's begin our examination by taking a look at the definition of the type `ProductName` from the previous chapter (the code I will show you is not legal C# - I omitted method bodies, putting a `;` after each method declaration. I did this because it would be a lot of code to grasp otherwise and I don't necessarily want to delve into the code of each method). Each section of the `ProductName` class definition is marked with a comment. These comments mark the topics we'll be discussing throughout this chapter.

So here is the promised definition of `ProductName`:

```
1  //This is the class we created and used
2  //in the previous chapter
3
4  // class signature
5  public sealed class ProductName
6    : IEquatable<ProductName>
7  {
8    // Hidden data:
9    private string _value;
10
11   // Constructor - hidden as well:
12   private ProductName(string value);
13
14   // Static method for creating new instances:
15   public static ProductName For(string value);
16
17   // Overridden version of ToString()
18   // from Object class
19   public override string ToString();
20
21   // Non-standard version of ToString().
22   // I will explain its purpose later
23   public string ToString(Format format);
24
25   // For value types, we need to implement all the equality
26   // methods and operators, plus GetHashCode():
27   public override bool Equals(Object other);
28   public bool Equals(ProductName other);
29   public override int GetHashCode();
```

```
30    public static bool operator ==(ProductName a, ProductName b);
31    public static bool operator !=(ProductName a, ProductName b);
32  }
```

Using the comments, I divided the class into sections and will describe them in order.

Class signature

There are two things to note about the class signature. The first one is that the class is `sealed` (in Java that would be `final`), i.e. I disallow inheriting from it. This is because I want the objects of this class to be immutable. At first sight, sealing the class has nothing to do with immutability. I will explain it in the next chapter when I discuss the aspects of value object design.

The second thing to note is that the class implements an `IEquatable` interface that adds more strongly typed versions of the `Equals(T object)` method. This is not strictly required as in C#, every object has a default `Equals(Object o)` method, but is typically considered good practice since it allows e.g. more efficient use of value objects with C# collections such as `Dictionary`[85].

Hidden data

The actual data is private:

```
1  private string _value;
```

Only the methods we publish can be used to operate on the state. This is useful for three things:

1. To restrict allowed operations to what we think makes sense to do with a product name. Everything else (i.e. what we think does not make sense to do) is not allowed.
2. To achieve immutability of `ProductName` instances (more on why we want the type to be immutable later), which means that when we create an instance, we cannot modify it. If the `_value` field was public, everyone could modify the state of `ProductName` instance by writing something like: `csharp productName.data = "something different";`
3. To protect against creating a product name with an invalid state. When creating a product name, we have to pass a string containing a name through a static `For()` method that can perform the validation (more on this later). If there are no other ways we can set the name, we can rest assured that the validation will happen every time someone wants to create a `ProductName`.

Hidden constructor

Note that the constructor is made private as well:

[85] https://stackoverflow.com/questions/2734914/whats-the-difference-between-iequatable-and-just-overriding-object-equals

```
private ProductName(string value)
{
  _value = value;
}
```

and you probably wonder why. I'd like to decompose the question further into two others:

1. How should we create new instances then?
2. Why private and not public?

Let's answer them one by one.

How should we create new instances?

The `ProductName` class contains a special static factory method, called `For()`. It invokes the constructor and handles all input parameter validations[86]. An example implementation could be:

```
public static ProductName For(string value)
{
  if(string.IsNullOrWhiteSpace(value))
  {
    //validation failed
    throw new ArgumentException(
      "Product names must be human readable!");
  }
  else
  {
    //here we call the constructor
    return new ProductName(value);
  }
}
```

There are several reasons for not exposing a constructor directly but use a static factory method instead. Below, I briefly describe some of them.

Explaining intention

Just like factories, static factory methods help explaining intention, because, unlike constructors, they can have names, while constructors have the constraint of being named after their class[87]. One can argue that the following:

[86]By the way, the code contains a call to `IsNullOrEmpty()`. There are several valid arguments against using this method, e.g. by Mark Seemann (http://blog.ploeh.dk/2014/11/18/the-isnullorwhitespace-trap/), but in this case, I put it in to make the code shorter as the validation logic itself is not that important at the moment.

[87]This is true for languages like Java, C# or C++. There are other languages (like Ruby), with different rules regarding object construction. Still, the original argument - that the naming of methods responsible for object creation is constrained - holds.

```
ProductName.For("super laptop X112")
```

is not that more readable than:

```
new ProductName("super laptop X112");
```

but note that in our example, we have a single, simple factory method. The benefit would be more visible when we would need to support an additional way of creating a product name. Let's assume that in above example of "super laptop X112", the "super laptop" is a model and "X112" is a specific configuration (since the same laptop models are often sold in several different configurations, with more or less RAM, different operating systems, etc.) and we find it comfortable to pass these two pieces of information as separate arguments in some places (e.g. because we may obtain them from different sources) and let the `ProductName` combine them. If we used a constructor for that, we would write:

```
// assume model is "super laptop"
// and configuration is "X112"
new ProductName(model, configuration)
```

On the other hand, we can craft a factory method and say:

```
ProductName.CombinedOf(model, configuration)
```

which reads more fluently. Or, if we like to be super explicit:

```
ProductName.FromModelAndConfig(model, configuration)
```

which is not my favorite way of writing code, because I don't like repeating the same information in method name and argument names. I wanted to show you that we can do this if we want though.

I met a lot of developers that find using factory methods somehow unfamiliar, but the good news is that factory methods for value objects are getting more and more mainstream. Just to give you two examples, `TimeSpan` type in C# uses them (e.g. we can write `TimeSpan.FromSeconds(12)` and `Period` type in Java (e.g. `Period.ofNanos(2222)`).

Ensuring consistent initialization of objects

In the case where we have different ways of initializing an object that all share a common part (i.e. whichever way we choose, part of the initialization must always be done the same), having several constructors that delegate to one common seems like a good idea. For example, we can have two constructors, one delegating to the other, that holds a common initialization logic:

```
// common initialization logic
public ProductName(string value)
{
  _value = value;
}

//another constructor that uses the common initialization
public ProductName(string model, string configuration)
 : this(model + " " + configuration) //delegation to "common" constructor
{
}
```

Thanks to this, the field `_value` is initialized in a single place and we have no duplication.

The issue with this approach is this binding between constructors is not enforced - we can use it if we want, otherwise, we can skip it. For example, we can as well use a separate set of fields in each constructor:

```
public ProductName(string value)
{
  _value = value;
}

public ProductName(string model, string configuration)
 //oops, no delegation to the other constructor
{
}
```

which leaves room for mistakes - we might forget to initialize all the fields all the time and allow creating objects with an invalid state.

I argue that using several static factory methods while leaving just a single constructor is safer in that it enforces every object creation to pass through this single constructor. This constructor can then ensure all fields of the object are properly initialized. There is no way in such case that we can bypass this constructor in any of the static factory methods, e.g.:

```
public ProductName CombinedOf(string model, string configuration)
{
  // no way to bypass the constructor here,
  // and to avoid initializing the _value field
  return new ProductName(model + " " + configuration);
}
```

What I wrote above might seem an unnecessary complication as the example of product names is trivial and we are unlikely to make a mistake like the one I described above, however:

1. There are more complex cases when we can indeed forget to initialize some fields in multiple constructors.

2. It is always better to be protected by the compiler than not when the price for the protection is considerably low. At the very least, when something happens, we'll have one place less to search for bugs.

Better place for input validation

Let's look again at the `For()` factory method:

```
public static ProductName For(string value)
{
  if(string.IsNullOrWhiteSpace(value))
  {
    //validation failed
    throw new ArgumentException(
      "Product names must be human readable!");
  }
  else
  {
    //here we call the constructor
    return new ProductName(value);
  }
}
```

and note that it contains some input validation, while the constructor did not. Is it a wise decision to move the validation to such a method and leave constructor only for assigning fields? The answer to this question depends on the answer to another one: are there cases where we do not want to validate constructor arguments? If no, then the validation should go to the constructor, as its purpose is to ensure an object is properly initialized.

Apparently, there are cases when we want to keep validations out of the constructor. Consider the following case: we want to create bundles of two product names as one. For this purpose, we introduce a new method on `ProductName`, called `BundleWith()`, which takes another product name:

```
public ProductName BundleWith(ProductName other)
{
  return new ProductName(
    "Bundle: " + _value + other._value);
}
```

Note that the `BundleWith()` method doesn't contain any validations but instead just calls the constructor. It is safe to do so in this case because we know that:

1. The string will be neither null nor empty since we are appending the values of both product names to the constant value of `"Bundle: "`. The result of such an append operation will never give us an empty string or a `null`.

2. The `_value` fields of both `this` and the `other` product name components must be valid because if they were not, the two product names that contain those values would fail to be created in the first place.

This was a case where we didn't need the validation because we were sure the input was valid. There may be another case - when it is more convenient for a static factory method to provide validation on its own. Such validation may be more detailed and helpful as it is in a factory method made for a specific case and knows more about what this case is. For example, let's look at the method we already saw for combining the model and configuration into a product name. If we look at it again (it does not contain any validations yet):

```
public ProductName CombinedOf(string model, string configuration)
{
  return ProductName.For(model + " " + configuration);
}
```

We may argue that this method would benefit from a specialized set of validations because probably both model and configuration need to be validated separately (by the way, it sometimes may be a good idea to create value objects for model and configuration as well - it depends on where we get them and how we use them). We could then go as far as to throw a different exception for each case, e.g.:

```
public ProductName CombinedOf(string model, string configuration)
{
  if(!IsValidModel(model))
  {
    throw new InvalidModelException(model);
  }

  if(!IsValidConfiguration(configuration))
  {
    throw new InvalidConfigurationException(configuration);
  }

  return ProductName.For(model + " " + configuration);
}
```

What if we need the default validation in some cases? We can still put them in a common factory method and invoke it from other factory methods. This looks a bit like going back to the problem with multiple constructors, but I'd argue that this issue is not as serious - in my mind, the problem of validations is easier to spot than mistakenly missing a field assignment as in the case of constructors. You may have different preferences though.

Remember we asked two questions and I have answered just one of them. Thankfully, the other one - why the constructor is private, not public - is much easier to answer now.

Why private and not public?

My reasons for it are: validation and separating use from construction.

Validation

Looking at the constructor of `ProductName` - we already discussed that it does not validate its input. This is OK when the constructor is used internally inside `ProductName` (as I just demonstrated in the previous section), because it can only be called by the code we, as creators of `ProductName` class, can trust. On the other hand, there probably is a lot of code that will create instances of `ProductName`. Some of this code is not even written yet, most of it we don't know, so we cannot trust it. For such code, we want to use only the "safe" methods that validate input and raise errors, not the constructor.

Separating use from construction[88]

I already mentioned that most of the time, we do not want to use polymorphism for values, as they do not play any roles that other objects can fill. Even though, I consider it wise to reserve some degree of flexibility to be able to change our decision more easily in the future, especially when the cost of the flexibility is very low.

Static factory methods provide more flexibility when compared to constructors. For example, when we have a static factory method like this:

```
1  public static ProductName For(string value)
2  {
3    //validations skipped for brevity
4    return new ProductName(value);
5  }
```

and all our code depends on it for creating product names rather than on the constructor, we are free to make the `ProductName` class abstract at some point and have the `For()` method return an instance of a subclass of `ProductName`. This change would impact just this static method, as the constructor is hidden and accessible only from inside the `ProductName` class. Again, this is something I don't recommend doing by default, unless there is a very strong reason. But if there is, the capability to do so is here.

String conversion methods

The overridden version of `ToString()` usually returns the internally held value or its string representation. It can be used to interact with a third party APIs or other code that does not know about our `ProductName` type. For example, if we want to save the product name inside the

[88] A. Shalloway et al., Essential Skills For The Agile Developer.

database, the database API has no idea about `ProductName`, but rather accepts library types such as strings, numbers, etc. In such a case, we can use `ToString()` to make passing the product name possible:

```
// let's assume that we have a variable
// productName of type ProductName.

var dataRecord = new DataRecord();
dataRecord["Product Name"] = productName.ToString();

//...

database.Save(dataRecord);
```

Things get more complicated when a value object has multiple fields or when it wraps another type like `DateTime` or an `int` - we may have to implement other accessor methods to obtain this data. `ToString()` can then be used for diagnostic purposes to allow printing user-friendly data dump.

Apart from the overridden `ToString()`, our `ProductName` type has an overload with signature `ToString(Format format)`. This version of `ToString()` is not inherited from any other class, so it's a method we made to fit our goals. The `ToString()` name is used only out of convenience, as the name is good enough to describe what the method does and it feels familiar. Its purpose is to be able to format the product name differently for different outputs, e.g. reports and on-screen printing. True, we could introduce a special method for each of the cases (e.g. `ToStringForScreen()` and `ToStringForReport()`), but that could make the `ProductName` know too much about how it is used - we would have to extend the type with new methods every time we wanted to print it differently. Instead, the `ToString()` accepts a `Format` (which is an interface, by the way) which gives us a bit more flexibility.

When we need to print the product name on screen, we can say:

```
var name = productName.ToString(new ScreenFormat());
```

and for reports, we can say:

```
var name = productName.ToString(new ReportingFormat());
```

Nothing forces us to call this method `ToString()` - we can use another name if we want to.

Equality members

For values such as `ProductName`, we need to implement all equality operations plus `GetHashCode()`. The purpose of equality operations to give product names value semantics and allow the following expressions:

```
ProductName.For("a").Equals(ProductName.For("a"));
ProductName.For("a") == ProductName.For("a");
```

to return `true`, since the state of the compared objects is the same despite them being separate instances in terms of references. In Java, of course, we can only override `equals()` method - we are unable to override equality operators as their behavior is fixed to comparing references (except primitive types), but Java programmers are so used to this, that it's rarely a problem.

One thing to note about the implementation of `ProductName` is that it implements `IEquatable<ProductName>` interface. In C#, overriding this interface when we want to have value semantics is considered a good practice. The `IEquatable<T>` interface is what forces us to create a strongly typed `Equals()` method:

```
public bool Equals(ProductName other);
```

while the one inherited from `object` accepts an `object` as a parameter. The use and existence of `IEquatable<T>` interface are mostly C#-specific, so I won't go into the details here, but you can always look it up in the documentation[89].

When we override `Equals()`, the `GetHashCode()` method needs to be overridden as well. The rule is that all objects that are considered equal should return the same hash code and all objects considered not equal should return different hash codes. The reason is that hash codes are used to identify objects in hash tables or hash sets - these data structures won't work properly with values if `GetHashCode()` is not properly implemented. That would be too bad because values are often used as keys in various hash-based dictionaries.

The return of investment

There are some more aspects of values that are not visible on the `ProductName` example, but before I explain them in the next chapter, I'd like to consider one more thing.

Looking into the `ProductName` anatomy, it may seem like it's a lot of code just to wrap a single string. Is it worth it? Where is the return of investment?

To answer that, I'd like to get back to our original problem with product names and remind you that I introduced a value object to limit the impact of some changes that could occur to the codebase where product names are used. As it's been a long time, here are the changes that we wanted to impact our code as little as possible:

1. Changing the comparison of product names to case-insensitive
2. Changing the comparison to take into account not only a product name but also a configuration in which a product is sold.

Let's find out whether introducing a value object would pay off in these cases.

[89] https://msdn.microsoft.com/en-us/library/ms131187.aspx

First change - case-insensitivity

This one is easy to perform - we just have to modify the equality operators, `Equals()` and `GetHashCode()` operations, so that they treat names with the same content in different letter case equal. I won't go over the code now as it's not too interesting, I hope you imagine how that implementation would look like. We would need to change all comparisons between strings to use an option to ignore case, e.g. `OrdinalIgnoreCase`. This would need to happen only inside the `ProductName` class as it's the only one that knows how what it means for two product names to be equal. This means that the encapsulation we've introduced without `ProductName` class has paid off.

Second change - additional identifier

This change is more complex, but having a value object in place makes it much easier anyway over the raw string approach. To make this change, we need to modify the creation of `ProductName` class to take an additional parameter, called `config`:

```
1  private ProductName(string value, string config)
2  {
3    _value = value;
4    _config = config;
5  }
```

Note that this is an example we mentioned earlier. There is one difference, however. While earlier we assumed that we don't need to hold value and configuration separately inside a `ProductName` instance and concatenated them into a single string when creating an object, this time we assume that we will need this separation between name and configuration later.

After modifying the constructor, the next thing is to add additional validations to the factory method:

```
1  public static ProductName CombinedOf(string value, string config)
2  {
3    if(string.IsNullOrWhiteSpace(value))
4    {
5      throw new ArgumentException(
6            "Product names must be human readable!");
7    }
8    else if(string.IsNullOrWhiteSpace(config))
9    {
10     throw new ArgumentException(
11           "Configs must be human readable!");
12   }
13   else
14   {
15     return new ProductName(value, config);
```

```
16     }
17 }
```

Note that this modification requires changes all over the code base (because additional argument is needed to create an object), however, this is not the kind of change that we're afraid of too much. That's because changing the signature of the method will trigger compiler errors. Each of these errors will need to be fixed before the compilation can pass (we can say that the compiler creates a nice TODO list for us and makes sure we address each and every item on that list). This means that we don't fall into the risk of forgetting to make one of the places where we need to make a change. This greatly reduces the risk of violating the Shalloway's Law.

The last part of this change is to modify equality operators, `Equals()` and `GetHashCode()`, to compare instances not only by name, but also by configuration. And again, I will leave the code of those methods as an exercise to the reader. I'll just briefly note that this modification won't require any changes outside the `ProductName` class.

Summary

So far, we have talked about value objects using a specific example of product names. I hope you now have a feel of how such objects can be useful. The next chapter will complement the description of value objects by explaining some of their general properties.

Aspects of value objects design

In the last chapter, we examined the anatomy of a value object. Still, there are several more aspects of value objects design that I still need to mention to give you a full picture.

Immutability

I mentioned before that value objects are usually immutable. Some say immutability is the core part of something being a value (e.g. Kent Beck goes as far as to say that 1 is always 1 and will never become 2), while others don't consider it as a hard constraint. One way or another, designing value objects as immutable has served me exceptionally well to the point that I don't even consider writing value object classes that are mutable. Allow me to describe three of the reasons I consider immutability a key constraint for value objects.

Accidental change of hash code

Many times, values are used as keys in hash maps (.e.g .NET's `Dictionary<K,V>` is essentially a hash map). Let's imagine we have a dictionary indexed with instances of a type called `KeyObject`:

```
1  Dictionary<KeyObject, AnObject> _objects;
```

When we use a `KeyObject` to insert a value into a dictionary:

```
1  KeyObject key = ...
2  _objects[key] = anObject;
```

then its hash code is calculated and stored separately from the original key.

When we read from the dictionary using the same key:

```
1  AnObject anObject = _objects[key];
```

then its hash code is calculated again and only when the hash codes match are the key objects compared for equality.

Thus, to successfully retrieve an object from a dictionary with a key, this key object must meet the following conditions regarding the key we previously used to put the object in:

1. The `GetHashCode()` method of the key used to retrieve the object must return the same hash code as that of the key used to insert the object did during the insertion,

2. The `Equals()` method must indicate that both the key used to insert the object and the key used to retrieve it are equal.

The bottom line is: if any of the two conditions is not met, we cannot expect to get the item we inserted.

I mentioned in the previous chapter that the hash code of a value object is calculated based on its state. A conclusion from this is that each time we change the state of a value object, its hash code changes as well. So, let's assume our `KeyObject` allows changing its state, e.g. by using a method `SetName()`. Thus, we can do the following:

```
KeyObject key = KeyObject.With("name");
_objects[key] = new AnObject();

// we mutate the state:
key.SetName("name2");

//do we get the inserted object or not?
var objectIInsertedTwoLinesAgo = _objects[key];
```

This will throw a `KeyNotFoundException` (this is the dictionary's behavior when it is indexed with a key it does not contain), as the hash code when retrieving the item is different than it was when inserting it. By changing the state of the `key` with the statement: `key.SetName("name2");`, I also changed its calculated hash code, so when I asked for the previously inserted object with `_objects[val]`, I tried to access an entirely different place in the dictionary than the one where my object is stored.

As I find it a quite common situation that value objects end up as keys inside dictionaries, I'd rather leave them immutable to avoid nasty surprises.

Accidental modification by foreign code

I bet many who code or coded in Java know its `Date` class. `Date` behaves like a value (it has overloaded equality and hash code generation), but is mutable (with methods like `setMonth()`, `setTime()`, `setHours()` etc.).

Typically, value objects tend to be passed a lot throughout an application and used in calculations. Many Java programmers at least once exposed a `Date` value using a getter:

```
public class ObjectWithDate {

  private final Date date = new Date();

  //...

  public Date getDate() {
    //oops...
    return this.date;
  }
}
```

The `getDate()` method allows users of the `ObjectWithDate` class to access the date. But remember, a date object is mutable and a getter returns a reference! Everyone who calls the getter gains access to the internally stored instance of `Date` and can modify it like this:

```
ObjectWithDate o = new ObjectWithDate();

o.getDate().setTime(date.getTime() + 10000); //oops!

return date;
```

Of course, almost no one would probably do it in the same line like in the snippet above, but usually, this date would be accessed, assigned to a variable and then passed through several methods, one of which would do something like this:

```
public void doSomething(Date date) {
  date.setTime(date.getTime() + 10000); //oops!
  this.nextUpdateTime = date;
}
```

This led to unpredicted situations as the date objects were accidentally modified far, far away from the place they were retrieved[90].

As most of the time it wasn't the intention, the problem of date mutability forced us to manually create a copy each time their code returned a date:

```
public Date getDate() {
  return (Date)this.date.clone();
}
```

which many of us tended to forget. This cloning approach, by the way, may have introduced a performance penalty because the objects were cloned every time, even when the code that was calling the `getDate()` had no intention of modifying the date[91].

Even when we follow the suggestion of avoiding getters, the same applies when our class passes the date somewhere. Look at the body of a method, called `dumpInto()`:

[90] This is sometimes called "aliasing bug": https://martinfowler.com/bliki/AliasingBug.html
[91] Unless Java optimizes it somehow, e.g. by using the copy-on-write approach.

```
public void dumpInto(Destination destination) {
  destination.write(this.date); //passing reference to mutable object
}
```

Here, the `destination` is allowed to modify the date it receives anyway it likes, which, again, is usually against developers' intentions.

I saw many, many issues in production code caused by the mutability of Java `Date` type alone. That's one of the reasons the new time library in Java 8 (`java.time`) contains immutable types for time and date. When a type is immutable, you can safely return its instance or pass it somewhere without having to worry that someone will overwrite your local state against your will.

Thread safety

When mutable values are shared between threads, there is a risk that they are changed by several threads at the same time or modified by one thread while being read by another. This can cause data corruption. As I mentioned, value objects tend to be created many times in many places and passed inside methods or returned as results a lot - this seems to be their nature. Thus, this risk of data corruption or inconsistency raises.

Imagine our code took hold of a value object of type `Credentials`, containing username and password. Also, let's assume `Credentials` objects are mutable. If so, one thread may accidentally modify the object while it is used by another thread, leading to data inconsistency. So, provided we need to pass login and password separately to a third-party security mechanism, we may run into the following:

```
public void LogIn(Credentials credentials)
{
  thirdPartySecuritySystem.LogIn(
    credentials.GetLogin(),
    //imagine password is modified before the next line
    //from a different thread
    credentials.GetPassword())
}
```

On the other hand, when an object is immutable, there are no multithreading concerns. If a piece of data is read-only, it can be safely read by as many threads as we like. After all, no one can modify the state of an object, so there is no possibility of inconsistency[92].

If not mutability, then what?

For the reasons I described, I consider immutability a crucial aspect of value object design and in this book, when I talk about value objects, I assume they are immutable.

Still, there is one question that remains unanswered: what about a situation when I need to:

[92]This is one of the reasons why functional languages, where data is immutable by default, gain a lot of attention in domains where doing many things in parallel is necessary.

- replace all occurrences of letter ‘r’ in a string to letter ‘l’?
- move a date forward by five days?
- add a file name to an absolute directory path to form an absolute file path (e.g. “C:” + “myFile.txt” = “C:\myFile.txt”)?

If I am not allowed to modify an existing value, how can I achieve such goals?

The answer is simple - value objects have operations that, instead of modifying the existing object, return a new one, with the state we are expecting. The old value remains unmodified. This is the way e.g. strings behave in Java and C#.

Just to address the three examples I mentioned

- when I have an existing `string` and want to replace every occurence of letter `r` with letter `l`:

```
string oldString = "rrrr";
string newString = oldString.Replace('r', 'l');
//oldString is still "rrrr", newString is "llll"
```

- When I want to move a date forward by five days:

```
DateTime oldDate = DateTime.Now;
DateTime newString = oldDate + TimeSpan.FromDays(5);
//oldDate is unchanged, newDate is later by 5 days
```

- When I want to add a file name to a directory path to form an absolute file path[93]:

```
AbsoluteDirectoryPath oldPath
  = AbsoluteDirectoryPath.Value(@"C:\Directory");
AbsoluteFilePath newPath = oldPath + FileName.Value("file.txt");
//oldPath is "C:\Directory", newPath is "C:\Directory\file.txt"
```

So, again, anytime we want to have a value based on a previous value, instead of modifying the previous object, we create a new object with the desired state.

Immutability gotchas

Watch out for the constructor!

Going back to the Java’s `Date` example - you may think that it’s fairly easy to get used to cases such as that one and avoid them by just being more careful, but I find it difficult due to many gotchas associated with immutability in languages such as C# or Java. For example, another variant of the `Date` case from Java could be something like this: Let’s imagine we have a `Money` type, which is defined as:

[93] this example uses a library called Atma Filesystem: https://www.nuget.org/packages/AtmaFilesystem/

```
public sealed class Money
{
  private readonly int _amount;
  private readonly Currencies _currency;
  private readonly List<ExchangeRate> _exchangeRates;

  public Money(
    int amount,
    Currencies currency,
    List<ExchangeRate> exchangeRates)
  {
    _amount = amount;
    _currency = currency;
    _exchangeRates = exchangeRates;
  }

  //... other methods
}
```

Note that this class has a field of type `List<>`, which is itself mutable. But let's also imagine that we have carefully reviewed all of the methods of this class so that this mutable data is not exposed. Does it mean we are safe?

The answer is: as long as our constructor stays as it is, no. Note that the constructor takes a mutable list and just assigns it to a private field. Thus, someone may do something like this:

```
List<ExchangeRate> rates = GetExchangeRates();

Money dollars = new Money(100, Currencies.USD, rates);

//modify the list that was passed to dollars object
rates.Add(GetAnotherExchangeRate());
```

In the example above, the `dollars` object was changed by modifying the list that was passed inside. To get the immutability, one would have to either use an immutable collection library or change the following line:

```
_exchangeRates = exchangeRates;
```

to:

```
_exchangeRates = new List<ExchangeRate>(exchangeRates);
```

Inheritable dependencies can surprise you!

Another gotcha has to do with objects of types that can be subclassed (i.e. are not `sealed`). Let's take a look at the example of a class called `DateWithZone`, representing a date with a time zone. Let's say that this class has a dependency on another class called `ZoneId` and is defined as such:

```
public sealed class DateWithZone : IEquatable<DateWithZone>
{
  private readonly ZoneId _zoneId;

  public DateWithZone(ZoneId zoneId)
  {
    _zoneId = zoneId;
  }

  //... some equality methods and operators...

  public override int GetHashCode()
  {
    return (_zoneId != null ? _zoneId.GetHashCode() : 0);
  }

}
```

Note that for simplicity, I made the `DateWithZone` type consist *only* of zone id, which of course in reality does not make any sense. I am doing this only because I want this example to be stripped to the bone. This is also why, for the sake of this example, `ZoneId` type is defined simply as:

```
public class ZoneId
{

}
```

There are two things to note about this class. First, it has an empty body, so no fields and methods defined. The second thing is that this type is not `sealed` (OK, the third thing is that this type does not have value semantics, since its equality operations are inherited as reference-based from the `Object` class, but, again for the sake of simplification, let's ignore that).

I just said that the `ZoneId` does not have any fields and methods, didn't I? Well, I lied. A class in C# inherits from `Object`, which means it implicitly inherits some fields and methods. One of these methods is `GetHashCode()`, which means that the following code compiles:

```
var zoneId = new ZoneId();
Console.WriteLine(zoneId.GetHashCode());
```

The last piece of information that we need to see the bigger picture is that methods like `Equals()` and `GetHashCode()` can be overridden. This, combined with the fact that our `ZoneId` is not `sealed`, means that somebody can do something like this:

```
public class EvilZoneId : ZoneId
{
  private int _i = 0;

  public override GetHashCode()
  {
    _i++;
    return i;
  }
}
```

When calling `GetHashCode()` on an instance of this class multiple times, it's going to return 1,2,3,4,5,6,7... and so on. This is because the `_i` field is a piece of mutable state and it is modified every time we request a hash code. Now, I assume no sane person would write code like this, but on the other hand, the language does not restrict it. So assuming such an evil class would come to existence in a codebase that uses the `DateWithZone`, let's see what could be the consequences.

First, let's imagine someone doing the following:

```
var date = new DateWithZone(new EvilZoneId());

//...

DoSomething(date.GetHashCode());
DoSomething(date.GetHashCode());
DoSomething(date.GetHashCode());
```

Note that the user of the `DateWithZone` instance uses its hash code, but the `GetHashCode()` operation of this class is implemented as:

```
public override int GetHashCode()
{
  return (_zoneId != null ? _zoneId.GetHashCode() : 0);
}
```

So it uses the hash code of the zone id, which, in our example, is of class `EvilZoneId` which is mutable. As a consequence, our instance of `DateWithZone` ends up being mutable as well.

This example shows a trivial and not too believable case of `GetHashCode()` because I wanted to show you that even empty classes have some methods that can be overridden to make the objects mutable. To make sure the class cannot be subclassed by a mutable class, we would have to either make all methods `sealed` (including those inherited from `Object`) or, better, make the class `sealed`. Another observation that can be made is that if our `ZoneId` was an abstract class with at least one abstract method, we would have no chance of ensuring immutability of its implementations, as abstract methods by definition exist to be implemented in subclasses, so we cannot make an abstract method or class `sealed`.

Another way of preventing mutability by subclasses is making the class constructor `private`. Classes with private constructors can still be subclassed, but only by nested classes, so there is a way for the author of the original class to control the whole hierarchy and make sure no operation mutates any state.

There are more gotchas (e.g. a similar one applied to generic types), but I'll leave them for another time.

Handling of variability

As in ordinary objects, there can be some variability in the world of values. For example, money can be dollars, pounds, zlotys (Polish money), euros, etc. Another example of something that can be modeled as a value is path values (you know, `C:\Directory\file.txt` or `/usr/bin/sh`) – there can be absolute paths, relative paths, paths to files and paths pointing to directories, we can have Unix paths and Windows paths.

Contrary to ordinary objects, however, where we solved variability by using interfaces and different implementations (e.g. we had an `Alarm` interface with implementing classes such as `LoudAlarm` or `SilentAlarm`), in the world values we do it differently. Taking the alarms I just mentioned as an example, we can say that the different kinds of alarms varied in how they fulfilled the responsibility of signaling that they were turned on (we said they responded to the same message with – sometimes entirely – different behaviors). Variability in the world of values is typically not behavioral in the same way as in the case of objects. Let's consider the following examples:

1. Money can be dollars, pounds, zlotys, etc., and the different kinds of currencies differ in what exchange rates are applied to them (e.g. "how many dollars do I get from 10 Euros and how many from 10 Pounds?"), which is not a behavioral distinction. Thus, polymorphism does not fit this case.
2. Paths can be absolute and relative, pointing to files and directories. They differ in what operations can be applied to them. E.g. we can imagine that for paths pointing to files, we can have an operation called `GetFileName()`, which doesn't make sense for a path pointing to a directory. While this is a behavioral distinction, we cannot say that "directory path" and a "file path" are variants of the same abstraction - rather, that are two different abstractions. Thus, polymorphism does not seem to be the answer here either.
3. Sometimes, we may want to have a behavioral distinction, like in the following example. We have a value class representing product names and we want to write in several different formats depending on the situation.

How do we model this variability? I usually consider three basic approaches, each applicable in different contexts:

- implicit - which would apply to the money example,
- explicit - which would fit the case of paths nicely,
- delegated - which would fit the case of product names.

Let me give you a longer description of each of these approaches.

Implicit variability

Let's go back to the example of modeling money using value objects[94]. Money can have different currencies, but we don't want to treat each currency in any special way. The only things that are impacted by currency are rates by which we exchange them for other currencies. We want the rest of our program to be unaware of which currency it's dealing with at the moment (it may even work with several values, each of different currency, at the same time, during one calculation or another business operation).

This leads us to make the differences between currencies implicit, i.e. we will have a single type called `Money`, which will not expose its currency at all. We only have to tell what the currency is when we create an instance:

```
Money tenPounds = Money.Pounds(10);
Money tenBucks = Money.Dollars(10);
Money tenYens = Money.Yens(10);
```

and when we want to know the concrete amount in a given currency:

```
//doesn't matter which currency it is, we want dollars.
decimal amountOfDollarsOnMyAccount = mySavings.AmountOfDollars();
```

other than that, we are allowed to mix different currencies whenever and wherever we like[95]:

```
Money mySavings =
  Money.Dollars(100) +
  Money.Euros(200) +
  Money.Zlotys(1000);
```

This approach works under the assumption that all of our logic is common for all kinds of money and we don't have any special piece of logic just for Pounds or just for Euros that we don't want to pass other currencies into by mistake[96].

To sum up, we designed the `Money` type so that the variability of currency is implicit - most of the code is simply unaware of it and it is gracefully handled under the hood inside the `Money` class.

Explicit variability

There are times, however, when we want the variability to be explicit, i.e. modeled using different types. Filesystem paths are a good example.

For starters, let's imagine we have the following method for creating backup archives that accepts a destination path (for now as a string - we'll get to path objects later) as its input parameter:

[94] This example is loosely based on Kent Beck's book Test-Driven Development By Example.

[95] I could use extension methods to make the example even more idiomatic, e.g. to be able to write `5.Dollars()`, but I don't want to go to far in the land of idioms specific to any language, because my goal is an audience wider than just C# programmers.

[96] I am aware that this example looks a bit naive - after all, adding money in several currencies would imply they need to be converted to a single currency and the exchange rates would then apply, which could make us lose money. Kent Beck acknowledged and solved this problem in his book Test-Driven Development By Example - be sure to take a look at his solution if you're interested.

```
void Backup(string destinationPath);
```

This method has one obvious drawback - its signature doesn't tell anything about the characteristics of the destination path, which begs some questions:

- Should it be an absolute path, or a relative path. If relative, then relative to what?
- Should the path contain a file name for the backup file, or should it be just a directory path and a filename will be given according to some kind of pattern (e.g. a word "backup" + the current timestamp)?
- Or maybe the file name in the path is optional and if none is given, then a default name is used?

These questions suggest that the current design doesn't convey the intention explicitly enough. We can try to work around it by changing the name of the parameter to hint the constraints, like this:

```
void Backup(string absoluteFilePath);
```

but the effectiveness of that is based solely on someone reading the argument name and besides, before a path (passed as a string) reaches this method, it is usually passed around several times and it's very hard to keep track of what is inside this string, so it becomes easy to mess things up and pass e.g. a relative path where an absolute one is expected. The compiler does not enforce any constraints. More than that, one can pass an argument that's not even a path, because a `string` can contain any arbitrary content.

It looks to me like a good situation to introduce a value object, but what kind of type or types should we introduce? Surely, we could create a single type called `Path`[97] that would have methods like `IsAbsolute()`, `IsRelative()`, `IsFilePath()` and `IsDirectoryPath()` (i.e. it would handle the variability implicitly), which would solve (only - we'll see that shortly) one part of the problem - the signature would be:

```
void Backup(Path absoluteFilePath);
```

and we would not be able to pass an arbitrary string, only an instance of a `Path`, which may expose a factory method that checks whether the string passed is in a proper format:

```
//the following could throw an exception
//because the argument is not in a proper format
Path path = Path.Value(@"C:\C:\C:\//\/\/");
```

Such a factory method could throw an exception at the time of path object creation. This is important - previously, when we did not have the value object, we could assign garbage to a string, pass it between several objects and get an exception from the `Backup()` method. Now, that we modeled paths as value objects, there is a high probability that the `Path` type will be

[97] This is what Java did. I don't declare that Java designers made a bad decision - a single `Path` class is probably much more versatile. The only thing I'm saying is that this design is not optimal for our particular scenario.

used as early as possible in the chain of calls. Thanks to this and the validation inside the factory method, we will get an exception much closer to the place where the mistake was made, not at the end of the call chain.

So yeah, introducing a general `Path` value object might solve some problems, but not all of them. Still, the signature of the `Backup()` method does not signal that the path expected must be an absolute path to a file, so one may pass a relative path or a path to a directory, even though only one kind of path is acceptable.

In this case, the varying properties of paths are not just an obstacle, a problem to solve, like in case of money. They are the key differentiating factor in choosing whether a behavior is appropriate for a value or not. In such a case, it makes a lot of sense to create several different value types, each representing a different set of path constraints.

Thus, we may decide to introduce types like[98]:

- `AbsoluteFilePath` - representing an absolute path containing a file name, e.g. `C:\Dir\file.txt`
- `RelativeFilePath` - representing a relative path containing a file name, e.g. `Dir\file.txt`
- `AbsoluteDirPath` - representing an absolute path not containing a file name, e.g. `C:\Dir\`
- `RelativeDirPath` - representing a relative path not containing a file name, e.g. `Dir\`

Having all these types, we can now change the signature of the `Backup()` method to:

```
void Backup(AbsoluteFilePath path);
```

Note that we don't have to explain the constraints with the name of the argument - we can just call it `path` because the type already says what needs to be said. And by the way, no one will be able to pass e.g. a `RelativeDirPath` now by accident, not to mention an arbitrary string.

Making variability among values explicit by creating separate types usually leads us to introduce some conversion methods between these types where such conversion is legal. For example, when all we've got is an `AbsoluteDirPath`, but we still want to invoke the `Backup()` method, we need to convert our path to an `AbsoluteFilePath` by adding a file name, that can be represented by a value objects itself (let's call its class `FileName`). In C#, we can use operator overloading for some of the conversions, e.g. the `+` operator would do nicely for appending a file name to a directory path. The code that does the conversion would then look like this:

```
AbsoluteDirPath dirPath = ...
...
FileName fileName = ...
...
//'+' operator is overloaded to handle the conversion:
AbsoluteFilePath filePath = dirPath + fileName;
```

Of course, we create conversion methods only where they make sense in the domain we are modeling. We wouldn't put a conversion method inside `AbsoluteDirectoryPath` that would combine it with another `AbsoluteDirectoryPath`[99].

[98] for reference, please take a look at https://www.nuget.org/packages/AtmaFilesystem/

[99] frankly, as in the case of money, the vision of paths I described here is a bit naive. Still, this naive view may be all we need in our particular case.

Delegated variability

Finally, we can achieve variability by delegating the varying behavior to an interface and have the value object accept an implementation of the interface as a method parameter. An example of this would be the `Product` class from the previous chapter that had the following method declared:

```
1 public string ToString(Format format);
```

where `Format` was an interface and we passed different implementations of this interface to this method, e.g. `ScreenFormat` or `ReportingFormat`. Note that having the `Format` as a method parameter instead of e.g. a constructor parameter allows us to uphold the value semantics because `Format` is not part of the object but rather a "guest helper". Thanks to this, we are free from dilemmas such as "is the name 'laptop' formatted for screen equal to 'laptop' formatted for a report?"

Summing up the implicit vs explicit vs delegated discussion

Note that while in the first example (the one with money), making the variability (in currency) among values implicit helped us achieve our design goals, in the path example it made more sense to do exactly the opposite - to make the variability (in both absolute/relative and to file/to directory axes) as explicit as to create a separate type for each combination of constraints.

If we choose the implicit approach, we can treat all variations the same, since they are all of the same type. If we decide on the explicit approach, we end up with several types that are usually incompatible and we allow conversions between them where such conversions make sense. This is useful when we want some pieces of our program to be explicitly compatible with only one of the variations.

I must say I find delegated variability a rare case (formatting the conversion to string is a typical example) and throughout my entire career, I had maybe one or two situations where I had to resort to it. However, some libraries use this approach and in your particular domain or type of application, such cases may be much more typical.

Special values

Some value types have values that are so specific that they have their own names. For example, a string value consisting of `""` is called "an empty string". `2,147,483,647` is called "a maximum 32 bit integer value". These special values make their way into value objects design. For example, in C#, we have `Int32.MaxValue` and `Int32.MinValue` which are constants representing a maximum and minimum value of 32-bit integer and `string.Empty` representing an empty string. In Java, we have things like `Duration.ZERO` to represent a zero duration or `DayOfWeek.MONDAY` to represent a specific day of the week.

For such values, the common practice I've seen is making them globally accessible from the value object classes, as is done in all the above examples from C# and Java. This is because values are immutable, so the global accessibility doesn't hurt. For example, we can imagine `string.Empty` implemented like this:

```
public sealed class String //... some interfaces here
{
  //...
  public const string Empty = "";
  //...
}
```

The additional `const` modifier ensures no one will assign any new value to the `Empty` field. By the way, in C#, we can use `const` only for types that have literal values, like a string or an `int`. For our custom value objects, we will have to use a `static readonly` modifier (or `static final` in the case of Java). To demonstrate it, let's go back to the money example from this chapter and imagine we want to have a special value called `None` to symbolize no money in any currency. As our `Money` type has no literals, we cannot use the `const` modifier, so instead, we have to do something like this:

```
public class Money
{
  //...

  public static readonly
    Money None = new Money(0, Currencies.Whatever);

  //...
}
```

This idiom is the only exception I know from the rule I gave you several chapters ago about not using static fields at all. Anyway, now that we have this `None` value, we can use it like this:

```
if(accountBalance == Money.None)
{
  //...
}
```

Value types and Tell Don't Ask

When talking about the "web of objects" metaphor, I stressed that objects should be told what to do, not asked for information. I also wrote that if a responsibility is too big for a single object to handle, it shouldn't try to achieve it alone, but rather delegate the work further to other objects by sending messages to them. I mentioned that preferably we would like to have mostly `void` methods that accept their context as arguments.

What about values? Does that metaphor apply to them? And if so, then how? And what about Tell Don't Ask?

First of all, values don't appear explicitly in the web of objects metaphor, at least they're not "nodes" in this web. Although in almost all object-oriented languages, values are implemented

using the same mechanism as objects - classes[100], I treat them as somewhat different kind of construct with their own set of rules and constraints. Values can be passed between objects in messages, but we don't talk about values sending messages by themselves.

A conclusion from this may be that values should not be composed of objects (understood as nodes in the "web"). Values should be composed of other values (as our `Path` type had a `string` inside), which ensures their immutability. Also, they can occasionally accept objects as parameters of their methods (like the `ProductName` class from the previous chapter that had a method `ToString()` accepting a `Format` interface), but this is more of an exception than a rule. In rare cases, I need to use a collection inside a value object. Collections in Java and C# are not typically treated as values, so this is kind of an exception from the rule. Still, when I use collections inside value objects, I tend to use the immutable ones, like ImmutableList[101].

If the above statements about values are true, then it means values simply cannot be expected to conform to Tell Don't Ask. Sure, we want them to encapsulate domain concepts, to provide higher-level interface, etc., so we struggle very hard for the value objects not to become plain data structures like the ones we know from C, but the nature of values is rather as "intelligent pieces of data" rather than "abstract sets of behaviors".

As such, we expect values to contain query methods (although, as I said, we strive for something more abstract and more useful than mere "getter" methods most of the time). For example, you might like the idea of having a set of path-related classes (like `AbsoluteFilePath`), but in the end, you will have to somehow interact with a host of third-party APIs that don't know anything about those classes. Then, a `ToString()` method that just returns internally held value will come in handy.

Summary

This concludes my writing on value objects. I never thought there would be so much to discuss as to how I believe they should be designed. For readers interested in seeing a state-of-the-art case study of value objects, I recommend looking at Noda Time[102] (for C#) and Joda Time[103] (for Java) libraries (or Java 8 new time and date API[104]).

[100] C# has structs, which can sometimes come in handy when implementing values, even though they have some constraints (see https://stackoverflow.com/questions/333829/why-cant-i-define-a-default-constructor-for-a-struct-in-net). Also, since more recent releases, C# has records which can be sometimes used to cut some of the boilerplate.

[101] https://msdn.microsoft.com/en-us/library/dn467185(v=vs.111).aspx

[102] https://nodatime.org/

[103] http://www.joda.org/joda-time/

[104] http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html

Part 3: TDD in Object-Oriented World

> ## Status: under development
>
> I am in progress of writing this part. Still, several chapters are already available for reading and pretty stable. I look forward to receiving your feedback!

So far, we've talked a lot about the object-oriented world, consisting of objects that exhibit the following properties:

1. Objects send messages to each other using interfaces and according to protocols. As long as these interfaces and protocols are adhered to by the recipients of the messages, the sender objects don't need to know who exactly is on the other side to handle the message. In other words, interfaces and protocols allow decoupling senders from the identity of their recipients.
2. Objects are built with Tell Don't Ask heuristic in mind, so that each object has its own responsibility and fulfills it when it's told to do something, without revealing the details of how it handles this responsibility,
3. Objects, their interfaces and protocols, are designed with composability in mind, which allows us to compose them as we would compose parts of sentences, creating small higher-level languages, so that we can reuse the objects we already have as our "vocabulary" and add more functionality by combining them into new "sentences".
4. Objects are created in places well separated from the places that use those objects. The place of object creation depends on the object lifecycle - it may be e.g. a factory or a composition root.

The world of objects is complemented by the world of values that exhibit the following characteristics:

1. Values represent quantities, measurements and other discrete pieces of data that we want to name, combine, transform and pass along. Examples are dates, strings, money, time durations, path values, numbers, etc.
2. Values are compared based on their data, not their references. Two values containing the same data are considered equal.

3. Values are immutable - when we want to have a value like another one, but with one aspect changed, we create a new value containing this change based on the previous value and the previous value remains unchanged.
4. Values do not (typically) rely on polymorphism - if we have several value types that need to be used interchangeably, the usual strategy is to provide explicit conversion methods between those types.

There are times when choosing whether something should be an object or a value poses a problem (I ran into situations when I modeled the same concept as a value in one application and as an object in another), so there is no strict rule on how to choose and, additionally, different people have different preferences.

This joint world is the world we are going to fit mock objects and other TDD practices into in the next part.

I know we have put TDD aside for such a long time. Believe me that this is because I consider understanding the concepts from part 2 crucial to getting mocks right.

Mock objects are not a new tool, however, there is still a lot of misunderstanding of what their nature is and where and how they fit best into the TDD approach. Some opinions went as far as to say that there are two styles of TDD: one that uses mocks (called "mockist TDD" or "London style TDD") and another without them (called "classic TDD" or "Chicago style TDD"). I don't support this division. I like very much what Nat Pryce wrote about it[105]:

> (...) I argue that there are not different kinds of TDD. There are different design conventions, and you pick the testing techniques and tools most appropriate for the conventions you're working in.

The explanation of the "design conventions" that mocks were born from required putting you through so many pages about a specific view on object-oriented design. This is the view that mock objects as a tool and as a technique were chosen to support. Talking about mock objects out of the context of this view would make me feel like I'm painting a false picture.

After reading part 3, you will understand how mocks fit into test-driving object-oriented code, how to make Statements using mocks maintainable and how some of the practices I introduced in the chapters of part 1 apply to mocks. You will also be able to test-drive simple object-oriented systems.

[105] https://groups.google.com/forum/#!msg/growing-object-oriented-software/GNS8bQ93yOo/GViu-YvWCEoJ

Mock Objects as a testing tool

Remember one of the first chapters of this book, where I introduced mock objects and mentioned that I had lied to you about their true purpose and nature? Now that we have a lot more knowledge of object-oriented design (at least on a specific, opinionated view on it), we can truly understand where mocks come from and what they are for.

In this chapter, I won't say anything about the role of mock objects in test-driving object-oriented code yet. For now, I want to focus on justifying their place in the context of testing objects written in the style that I described in part 2.

A backing example

For the need of this chapter, I will use one toy example. Before I describe it, I need you to know that I don't consider this example a showcase for mock objects. Mocks shine where there are domain-driven interactions between objects and this example is not like that - the interactions here are more implementation-driven. Still, I decided to use it anyway because I consider it something easy to understand and good enough to discuss some mechanics of mock objects. In the next chapter, I will use the same example as an illustration, but after that, I'm dropping it and going into more interesting stuff.

The example is a single class, called `DataDispatch`, which is responsible for sending received data to a channel (represented by a `Channel` interface). The `Channel` needs to be opened before the data is sent and closed after. `DataDispatch` implements this requirement. Here is the full code for the `DataDispatch` class:

```
1  public class DataDispatch
2  {
3    private Channel _channel;
4
5    public DataDispatch(Channel channel)
6    {
7      _channel = channel;
8    }
9
10   public void Dispatch(byte[] data)
11   {
12     _channel.Open();
13     try
14     {
15       _channel.Send(data);
16     }
17     finally
```

```
18      {
19        _channel.Close();
20      }
21    }
22  }
```

The rest of this chapter will focus on dissecting the behaviors of `DataDispatch` and their context. I will start describing this context by looking at the interface used by `DataDispatch`.

Interfaces

As shown above, `DataDispatch` depends on a single interface called `Channel`. Here is the full definition of this interface:

```
public interface Channel
{
  void Open();
  void Send(byte[] data);
  void Close();
}
```

An implementation of `Channel` is passed into the constructor of `DataDispatch`. In other words, `DataDispatch` can be composed with anything that implements `Channel` interface. At least from the compiler's point of view. This is because, as I mentioned in the last part, for two composed objects to be able to work together successfully, interfaces are not enough. They also have to establish and follow a protocol.

Protocols

Note that when we look at the `DataDispatch` class, there are two protocols it has to follow. I will describe them one by one.

Protocol between `DataDispatch` and its user

The first protocol is between `DataDispatch` and the code that uses it, i.e. the one that calls the `Dispatch()` method. Someone, somewhere, has to do the following:

```
dataDispatch.Send(messageInBytes);
```

or there would be no reason for `DataDispatch` to exist. Looking further into this protocol, we can note that `DataDispatch` does not require too much from its users – it doesn't have any kind of return value. The only feedback it gives to the code that uses it is rethrowing any exception raised by a channel, so the user code must be prepared to handle the exception. Note that `DataDispatch` neither knows nor defines the kinds of exceptions that can be thrown. This is the responsibility of a particular channel implementation. The same goes for deciding under which condition should an exception be thrown.

Protocol between `DataDispatch` and `Channel`

The second protocol is between `DataDispatch` and `Channel`. Here, `DataDispatch` will work with any implementation of `Channel` the allows it to invoke the methods of a `Channel` specified number of times in a specified order:

1. Open the channel – once,
2. Send the data – once,
3. Close the channel – once.

Whatever actual implementation of `Channel` interface is passed to `DataDispatch`, it will operate on the assumption that this indeed is the count and order in which the methods will be called. Also, `DataDispatch` assumes it is required to close the channel in case of error while sending data (hence the `finally` block wrapping the `Close()` method invocation).

Two conversations

Summing it up, there are two "conversations" a `DataDispatch` object is involved in when fulfilling its responsibilities – one with its user and one with a dependency passed by its creator. We cannot specify these two conversations separately as the outcome of each of these two conversations depends on the other. Thus, we have to specify the `DataDispatch` class, as it is involved in both of these conversations at the same time.

Roles

Our conclusion from the last section is that the environment in which behaviors of `DataDispatch` take place is comprised of three roles (arrows show the direction of dependencies, or "who sends messages to whom"):

```
1  User -> DataDispatch -> Channel
```

Where `DataDispatch` is the specified class and the rest is its context (`Channel` being the part of the context `DataDispatch` depends on. As much as I adore context-independence, most classes need to depend on some kind of context, even if to a minimal degree).

Let's use this environment to define the behaviors of `DataDispatch` we need to specify.

Behaviors

The behaviors of `DataDispatch` defined in terms of this context are:

1. Dispatching valid data:

```
GIVEN User wants to dispatch a piece of data
AND a DataDispatch instance is connected to a Channel
  that accepts such data
WHEN the User dispatches the data via the DataDispatch instance
THEN the DataDispatch object should
  open the channel,
  then send the User data through the channel,
  then close the channel
```

1. Dispatching invalid data:

```
GIVEN User wants to dispatch a piece of data
AND a DataDispatch instance is connected to a Channel
  that rejects such data
WHEN the User dispatches the data via the DataDispatch instance
THEN the DataDispatch object should report to the User
  that data is invalid
AND close the connection anyway
```

For the remainder of this chapter, I will focus on the first behavior as our goal for now, is not to create a complete Specification of DataDispatch class, but rather to observe some mechanics of mock objects as a testing tool.

Filling in the roles

As mentioned before, the environment in which the behavior takes place looks like this:

```
User -> DataDispatch -> Channel
```

Now we need to say who will play these roles. I marked the ones we don't have filled yet with question marks (?):

```
User? -> DataDispatch? -> Channel?
```

Let's start with the role of DataDispatch. Probably not surprisingly, it will be filled by the concrete class `DataDispatch` – after all, this is the class that we specify.

Our environment looks like this now:

```
User? -> DataDispatch (concrete class) -> Channel?
```

Next, who is going to be the user of the `DataDispatch` class? For this question, I have an easy answer – the Statement body is going to be the user – it will interact with `DataDispatch` to trigger the specified behaviors. This means that our environment looks like this now:

```
Statement body -> DataDispatch (concrete class) -> Channel?
```

Now, the last element is to decide who is going to play the role of a channel. We can express this problem with the following, unfinished Statement (I marked all the current unknowns with a double question mark: `??`):

```
[Fact] public void
ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch()
{
  //GIVEN
  Channel channel = ??; //what is it going to be?
  var dispatch = new DataDispatch(channel);
  var data = Any.Array<byte>();

  //WHEN
  dispatch.ApplyTo(data);

  //THEN
  ?? //how to specify DataDispatch behavior?
}
```

As you see, we need to pass an implementation of `Channel` to a `DataDispatch`, but we don't know what this channel should be. Likewise, we have no good idea of how to specify the expected calls and their order.

From the perspective of `DataDispatch`, it is designed to work with everything that implements the `Channel` interface and follows the protocol, so there is no single "privileged" implementation that is more appropriate than others. This means that we can pretty much pick and choose the one we like best. Which one do we like best? The one that makes writing the specification easiest, of course. Ideally, we'd like to pass a channel that best fulfills the following requirements:

1. Adds as little side effects of its own as possible. If a channel implementation used in a Statement added side effects, we would never be sure whether the behavior we observe when executing our Specification is the behavior of `DataDispatch` or maybe the behavior of the particular `Channel` implementation that is used in this Statement. This is a requirement of trust – we want to trust our Specification that it specifies what it says it does.
2. Is easy to control – so that we can easily make it trigger different conditions in the object we are specifying. Also, we want to be able to easily verify how the specified object interacts with it. This is a requirement of convenience.
3. Is quick to create and easy to maintain – because we want to focus on the behaviors we specify, not on maintaining or creating helper classes. This is a requirement of low friction.

There is a tool that fulfills these three requirements better than others I know of and it's called a mock object. Here's how it fulfills the mentioned requirements:

1. Mocks add almost no side effects of its own. Although they do have some hardcoded default behaviors (e.g. when a method returning `int` is called on a mock, it returns `0` by default), but these behaviors are as default and meaningless as they can possibly be. This allows us to put more trust in our Specification.
2. Mocks are easy to control - every mocking library comes provided with an API for defining pre-canned method call results and for verification of received calls. Having such API provides convenience, at least from my point of view.
3. Mocks can be trivial to maintain. While you can write your own mocks (i.e. your own implementation of an interface that allows setting up and verifying calls), most of us use libraries that generate them, typically using a reflection feature of a programming language (in our case, C#). Typically, mock libraries free us from having to maintain mock implementations, lowering the friction of writing and maintaining our executable Statements.

So let's use a mock in place of `Channel`! This makes our environment of the specified behavior look like this:

```
Statement body -> DataDispatch (concrete class) -> Mock Channel
```

Note that the only part of this environment that comes from production code is the `DataDispatch`, while its context is Statement-specific.

Using a mock channel

I hope you remember the NSubstitute library for creating mock objects that I introduced way back at the beginning of the book. We can use it now to quickly create an implementation of `Channel` that behaves the way we like, allows easy verification of protocol and between `Dispatch` and `Channel` and introduces the minimal number of side effects.

By using this mock to fill the gaps in our Statement, this is what we end up with:

```
[Fact] public void
ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch()
{
  //GIVEN
  var channel = Substitute.For<Channel>();
  var dispatch = new DataDispatch(channel);
  var data = Any.Array<byte>();

  //WHEN
  dispatch.ApplyTo(data);

  //THEN
  Received.InOrder(() =>
  {
```

```
15      channel.Open();
16      channel.Send(data);
17      channel.Close();
18    };
19  }
```

previously, this Statement was incomplete, because we lacked the answer to the following two questions:

1. Where to get the channel from?
2. How to verify `DataDispatch` behavior?

I answered the question of "where to get the channel from?" by creating it as a mock:

```
1  var channel = Substitute.For<Channel>();
```

Then the second question: "how to verify DataDispatch behavior?" was answered by using the NSubstitute API for verifying that the mock received three calls (or three messages) in a specific order:

```
1  Received.InOrder(() =>
2  {
3    channel.Open();
4    channel.Send(data);
5    channel.Close();
6  };
```

The consequence is that if I rearrange the order of the messages sent to `Channel` in the implementation of the `ApplyTo()` method from this one:

```
1  public void Dispatch(byte[] data)
2  {
3    _channel.Open();
4    _channel.Send(data);
5    _channel.Close();
6  }
```

to this one (note the changed call order):

```
public void Dispatch(byte[] data)
{
  _channel.Send(data); //before Open()!
  _channel.Open();
  _channel.Close();
}
```

The Statement will turn false (i.e. will fail).

Mocks as yet another context

What we did in the above example was to put our `DataDispatch` in a context that was most trustworthy, convenient and frictionless for us to use in our Statement.

Some say that specifying object interactions in the context of mocks is "specifying in isolation" and that providing such mock dependencies is "isolating" the class from its "real" dependencies. I don't identify with this point of view very much. From the point of view of a specified class, mocks are yet another context – they are neither better, nor worse, they are neither more nor less real than other contexts we want to put our `Dispatch` in. Sure, this is not the context in which it runs in production, but we may have other situations than mere production work – e.g. we may have a special context for demos, where we count sent packets and show the throughput on a GUI screen. We may also have a debugging context that in each method, before passing the control to production code, writes a trace message to a log. The `DataDispatch` class may be used in the production code in several contexts at the same time. We may dispatch data through the network, to a database and a file all at the same time in our application and the `DataDispatch` class may be used in all these scenarios, each time connected to a different implementation of `Channel` and used by a different piece of code.

Summary

The goal of this chapter was only to show you how mock objects fit into testing code written in a "tell don't ask" style, focusing on roles, responsibilities, behaviors, interfaces, and protocols of objects. This example was meant as something you could easily understand, not as a showcase for TDD using mocks. For one more chapter, we will work on this toy example and then I will try to show you how I apply mock objects in more interesting cases.

Test-first using mock objects

Now that we saw mocks in action and placed them in the context of a specific design approach, I'd like to show you how mock objects are used when employing the test-first approach. To do that, I'm going to reiterate the example from the last chapter. I already mentioned how this example is not particularly strong in terms of showcasing the power of mock objects, so I won't repeat myself here. In the next chapter, I will give you an example I consider more suited.

How to start? – with mock objects

You probably remember the chapter "How to start?" from part 1 of this book. In that chapter, I described the following ways to kick-start writing a Statement before the actual implementation is in place:

1. Start with a good name.
2. Start by filling the GIVEN-WHEN-THEN structure with the obvious.
3. Start from the end.
4. Start by invoking a method if you have one.

Pretty much all of these strategies work equally well with Statements that use mock objects, so I'm not going to describe them in detail again. In this chapter, I will focus on one of the strategies: "Start by invoking a method if you have one" as it's the one I use most often. This is driven not only by my choice to use mock objects, but also by the development style I tend to use. This style is called "outside-in" and all we need to know about it, for now, is that following it means starting the development from the inputs of the system and ending on the outputs. Some may consider it counter-intuitive as it means we will write classes collaborating with classes that don't exist yet. I will give you a small taste of it (together with a technique called "interface discovery") in this chapter and will expand on these ideas in the next one.

Responsibility and Responsibility

In this chapter, I will be using two concepts that, unfortunately, happen to share the same name: "responsibility". One meaning of responsibility was coined by Rebecca Wirfs-Brock[106] to mean "an obligation to perform a task or know certain information", and the other by Robert C. Martin to mean "a reason to change". To avoid this ambiguity, I will try calling the first one "obligation" and the second one "purpose" in this chapter.

The relationship between the two can be described by the following sentences:

1. A class has obligations towards its clients.

[106] http://www.wirfs-brock.com/PDFs/PrinciplesInPractice.pdf

2. The obligations are what the class "promises" to do for its clients.
3. The class does not have to fulfill the obligations alone. Typically, it does so with help from other objects – its collaborators. Those collaborators, in turn, have their obligations and collaborators.
4. Each of the collaborators has its purpose - a role in fulfilling the main obligation. The purpose results from the decomposition of the main obligation.

Channel and DataDispatch one more time

Remember the example from the last chapter? Imagine we are in a situation where we already have the `DataDispatch` class, but its implementation is empty – after all, this is what we're going to test-drive.

So for now, the `DataDispatch` class looks like this

```
public class DataDispatch
{
 public void ApplyTo(byte[] data)
 {
  throw new NotImplementedException();
 }
}
```

Where did I get this class from in this shape? Well, let's assume for now that I am in the middle of development and this class is a result of my earlier TDD activities (after reading this and the next chapter, you'll hopefully have a better feel on how it happens).

The first behavior

A TDD cycle starts with a false Statement. What behavior should it describe? I'm not sure yet, but, as I already know the class that will have the behaviors that I want to specify, plus it only has a single method (`ApplyTo()`), I can almost blindly write a Statement where I create an object of this class and invoke the method:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 var dispatch = new DataDispatch();

 //WHEN
 dispatch.ApplyTo(); //TODO doesn't compile

 //THEN
 Assert.True(false); //TODO state expectations
}
```

Note several things:

1. I'm now using a placeholder name for the Statement and I added a TODO item to my list to correct it later when I define the purpose and behavior of `DataDispatch`.
2. According to its signature, the `ApplyTo()` method takes an argument, but I didn't provide any in the Statement. For now, I don't want to think too hard, I just want to brain-dump everything I know.
3. the `//THEN` section is empty for now – it only has a single assertion that is designed to fail when the execution flow reaches it (this way I protect myself from mistakenly making the Statement true until I state my real expectations). I will define the `//THEN` section once I figure out what is the purpose that I want to give this class and the behavior that I want to specify.
4. If you remember the `Channel` interface from the last chapter, then imagine that it doesn't exist yet and that I don't even know that I will need it. I will "discover" it later.

Leaning on the compiler

So I did my brain dump. What do I do now? I don't want to think too hard yet (time will come for that). First, I reach for the feedback to my compiler – maybe it can give me some hints on what I am missing?

Currently, the compiler complains that I invoke the `ApplyTo()` method without passing any argument. What's the name of the argument? As I look up the signature of the `ApplyTo()` method, it looks like the name is `data`:

```
public void ApplyTo(byte[] data)
```

Hmm, if it's data it wants, then let's pass some data. I don't want to decide what it is yet, so I will act *as if I had* a variable called `data` and just write its name where the argument is expected:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 var dispatch = new DataDispatch();

 //WHEN
 dispatch.ApplyTo(data); //TODO still doesn't compile

 //THEN
 Assert.True(false); //TODO state expectations
}
```

The compiler gives me more feedback – it says my `data` variable is undefined. It might sound funny (as if I didn't know!), but this way I come one step further. Now I know I need to define this `data`. I can use a "quick fix" capability of my IDE to introduce a variable. E.g. in Jetbrains IDEs (IntelliJ IDEA, Resharper, Rider...) this can be done by pressing `ALT + ENTER` when the cursor is on the name of the missing variable. The IDE will create the following declaration:

```
byte[] data;
```

Note that the IDE guessed the type of the variable for me. How did it know? Because the definition of the method where I try to pass it already has the type declared:

```
public void ApplyTo(byte[] data)
```

Of course, the declaration of `data` that my IDE put in the code will still not compile because C# requires variables to be explicitly initialized. So the code should look like this:

```
byte[] data = ... /* whatever initialization code*/;
```

Turning the brain on – what about data?

It looks like I can't continue my brain-dead parade anymore. To decide how to define this data, I have to turn my thought processes on and decide what exactly is the obligation of the `ApplyTo()` method and what does it need the `data` for. After some thinking, I decide that applying data dispatch should send the data it receives. But... should it do all the work, or maybe delegate some parts? There are at least two subtasks associated with sending the data:

1. The raw sending logic (laying out the data, pushing it e.g. through a web socket, etc.)
2. Managing the connection lifetime (deciding when it should be opened and when closed, disposing of all the allocated resources, even in the face of an exception that may occur while sending).

I decide against putting the entire logic in the `DataDispatch` class, because:

1. It would have more than one purpose (as described earlier) – in other words, it would violate the Single Responsibility Principle.
2. I am mentally unable to figure out how to write a false Statement for so much logic before the implementation. I always treat it as a sign that I'm trying to put too much of a burden on a single class[107].

Introducing a collaborator

Thus, I decide to divide and conquer, i.e. find `DataDispatch` some collaborators that will help it achieve its goal and delegate parts of the logic to them. After some consideration, I conclude that the purpose of `DataDispatch` should be managing the connection lifetime. The rest of the logic I decide to delegate to a collaborator role I call a `Channel`. The process of coming up with collaborator roles and delegating parts of specified class obligations to them is called *interface discovery*.

Anyway, since my `DataDispatch` is going to delegate some logic to the `Channel`, it has to know it. Thus, I'll connect this new collaborator to the `DataDispatch`. A `DataDispatch` will not work without a `Channel`, which means I need to pass the channel to `DataDispatch` as a constructor parameter. It's tempting me to forget TDD, just go to the implementation of this constructor and add a parameter there, but I resist. As usual, I start my changes from the Statement. Thus, I change the following code:

[107] more on this in further chapters.

```
//GIVEN
var dispatch = new DataDispatch();
```

to:

```
//GIVEN
var dispatch = new DataDispatch(channel); //doesn't compile
```

I use a `channel` variable *as if* it was already defined in the Statement body and *as if* the constructor already required it. Of course, none of these is true yet. This leads my compiler to give me more compile errors. For me, this is a valuable source of feedback which I need to progress further. The first thing the compiler tells me to do is to introduce a `channel` variable. Again, I use my IDE to generate it for me. This time, however, the result of the generation is:

```
object channel;
```

The IDE could not guess the correct type of `channel` (which would be `Channel`) and made it an `object`, because I haven't created the `Channel` type yet.

First, I'll introduce the `Channel` interface by changing the declaration `object channel;` into `Channel channel;`. This will give me another compile error, as the `Channel` type does not exist. Thankfully, creating it is just one IDE click away (e.g. in Resharper, I place my cursor at the non-existent type, press `ALT + ENTER` and pick an option to create it as an interface.). Doing this will give me:

```
public interface Channel
{

}
```

which is enough to get past this particular compiler error, but then I get another one – nothing is assigned to the `channel` variable. Again, I have to turn my thinking on again. Luckily, this time I can lean on a simple rule: in my design, `Channel` is a role and, as mentioned in the last chapter, I use mocks to play the roles of my collaborators. So the conclusion is to use a mock. By applying this rule, I change the following line:

```
Channel channel;
```

to:

```
var channel = Substitute.For<Channel>();
```

The last compiler error I need to address to fully introduce the `Channel` collaborator is to make the `DataDispatch` constructor accept the channel as its argument. For now `DataDispatch` uses an implicit parameterless constructor. I generate a new one, again, using my IDE. I go to the place where the constructor is called with the channel as argument and tell my IDE to correct the constructor signature for me based on this usage. This way I get a constructor code inside the `DataDispatch` class:

```
public DataDispatch(Channel channel)
{

}
```

Note that the constructor doesn't do anything with the channel yet. I could create a new field and assign the channel to it, but I don't need to do that yet, so I decide I can wait a little bit longer.

Taking a bird's-eye view on my Statement, I currently have:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 byte[] data; // does not compile yet
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Assert.True(false); //TODO state expectations
}
```

This way, I defined a `Channel` collaborator and introduced it first in my Statement, and then in the production code.

Specifying expectations

The compiler and my TODO list point out that I still have three tasks to accomplish for the current Statement:

- define `data` variable,
- name my Statement and
- state my expectations (the `THEN` section of the Statement)

I can do them in any order I see fit, so I pick the last task from the list - stating the expected behavior.

To specify what is expected from `DataDispatch`, I have to answer myself four questions:

1. What are the obligations of `DataDispatch`?
2. What is the purpose of `DataDispatch`?
3. Who are the collaborators that need to receive messages from `DataDispatch`?

4. What is the behavior of `DataDispatch` that I need to specify?

My answers to these questions are:

1. `DataDispatch` is obligated to send data as long as it is valid. In case of invalid data, it throws an exception. That's two behaviors. As I only specify a single behavior per Statement, I need to pick one of them. I pick the first one (which I will call "the happy path" from now on), adding the second one to my TODO list:

```
1 //TODO: specify a behavior where sending data
2 //      through a channel raises an exception
```

2. The purpose of `DataDispatch` is to manage connection lifetime while sending data received via the `ApplyTo()` method. Putting it together with the answer to the last question, what I would need to specify is how `DataDispatch` manages this lifetime during the "happy path" scenario. The rest of the logic which I need to fulfill the obligation of `DataDispatch` is outside the scope of the current Statement as I decided to push it to collaborators.
3. I already defined one collaborator and called it `Channel`. As mentioned in the last chapter, in unit-level Statements, I fill my collaborators' roles with mocks and specify what messages they should receive. Thus, I know that the `THEN` section will describe the messages that the `Channel` role (played by a mock object) is expected to receive from my `DataDispatch`.
4. Now that I know the scenario, the purpose and the collaborators, I can define my expected behavior in terms of those things. My conclusion is that I expect `DataDispatch` to properly manage (purpose) a `Channel` (collaborator) in a "happy path" scenario where the data is sent without errors (obligation). As channels are typically opened before they are used and are closed afterwards, then what my `DataDispatch` is expected to do is to open the channel, then push data through it, and then close it.

How to implement such expectations? Implementation-wise, what I expect is that `DataDispatch`:

- makes correct calls on the `Channel` collaborator (open, send, close)
- with correct arguments (the received data)
- in the correct order (cannot e.g. call close before open)
- correct number of times (e.g. should not send the data twice)

I can specify that using NSubstitute's `Received.InOrder()` syntax. I will thus use it to state that the three methods are expected to be called in a specific order. Wait, what methods? After all, our `Channel` interface looks like this:

```
1 public interface Channel
2 {
3
4 }
```

so there are no methods here whatsoever. The answer is – just like I discovered the need for the `Channel` interface and then brought it to life afterward, I now discovered that I need three methods: `Open()`, `Send()` and `Close()`. The same way as I did with the `Channel` interface, I will use them in my Statement *as if* they existed:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 byte[] data; // does not compile yet
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Received.InOrder(() =>
 {
  channel.Open(); //doesn't compile
  channel.Send(data); //doesn't compile
  channel.Close(); //doesn't compile
 });
}
```

and then pull them into existence using my IDE and its shortcut for generating missing classes and methods. This way, I get:

```
public interface Channel
{
 void Open();
 void Send(byte[] data);
 void Close();
}
```

Now I have only two things left on my list – giving the Statement a good name and deciding what the `data` variable should hold. I'll go with the latter as it is the last thing that prevents the compiler from compiling and running my Statement and I expect it will give me more useful feedback.

The `data` variable

What should I assign to the `data` variable? Time to think about how much the `DataDispatch` needs to know about the data it pushes through the channel. I decide that `DataDispatch` should work with any data – its purpose is to manage the connection after all – it does not need to read or manipulate the data to do this. Someone, somewhere, probably needs to validate this data, but I decide that if I added validation logic to the `DataDispatch`, it would break the single-purposeness. So I push validation further to the `Channel` interface, as the decision to accept the data or not depends on the actual implementation of sending logic. Thus, I define the `data` variable in my Statement as just `Any.Array<byte>()`:

```
[Fact] public void
ShouldXXXXXXXXXYYY() //TODO give a better name
{
 //GIVEN
 var data = Any.Array<byte>();
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Received.InOrder(() =>
 {
  channel.Open();
  channel.Send(data);
  channel.Close();
 });
}
```

Good name

The Statement now compiles and runs (it is currently false, of course, but I'll get to that), so what I need is to give this Statement a better name. I'll go with `ShouldSendDataThroughOpenChannelThenCloseWhenAsked` This was the last TODO on the Specification side, so let's see the full Statement code:

```
[Fact] public void
ShouldSendDataThroughOpenChannelThenCloseWhenAskedToDispatch()
{
 //GIVEN
 var data = Any.Array<byte>();
 var channel = Substitute.For<Channel>();
 var dispatch = new DataDispatch(channel);

 //WHEN
 dispatch.ApplyTo(data);

 //THEN
 Received.InOrder(() =>
 {
  channel.Open();
  channel.Send(data);
  channel.Close();
 });
}
```

Failing for the correct reason

The Statement I just wrote can now be evaluated and, as expected, it is false. This is because the current implementation of the `ApplyTo` method throws a `NotImplementedException`:

```
public class DataDispatch
{
 public DataDispatch(Channel channel)
 {

 }

 public void ApplyTo(byte[] data)
 {
  throw new NotImplementedException();
 }
}
```

What I'd like to see before I start implementing the correct behavior is that the Statement is false because assertions (in this case – mock verifications) fail. So the part of the Statement that I would like to see throwing an exception is this one:

```
Received.InOrder(() =>
{
 channel.Open();
 channel.Send(data);
 channel.Close();
});
```

but instead, I get an exception as early as:

```
//WHEN
dispatch.ApplyTo(data);
```

To make progress past the `WHEN` section, I need to push the production code a little bit further towards the correct behavior, but only as much as to see the expected failure. Thankfully, I can achieve it easily by going into the `ApplyTo()` method and removing the `throw` clause:

```
public void ApplyTo(byte[] data)
{

}
```

This alone is enough to see the mock verification making my Statement false. Now that I can see that the Statement is false for the correct reason, my next step is to put the correct implementation to make the Statement true.

Making the Statement true

I start with the `DataDispatch` constructor, which currently takes a `Channel` as a parameter, but doesn't do anything with it:

```
public DataDispatch(Channel channel)
{

}
```

I want to assign the channel to a newly created field (I can do this using a single command in most IDEs). The code then becomes:

```
private readonly Channel _channel;

public DataDispatch(Channel channel)
{
 _channel = channel;
}
```

This allows me to use the `_channel` in the `ApplyTo()` method that I'm trying to implement. Remembering that my goal is to open the channel, push the data and close the channel, I type:

```
public void ApplyTo(byte[] data)
{
 _channel.Open();
 _channel.Send(data);
 _channel.Close();
}
```

To tell you the truth, sometimes before writing the correct implementation, I play a bit, making the Statement wrong in several ways, just to see if I can correctly guess the reason why the Statement will turn false and to make sure the error messages are informative enough. For example, I may only implement opening the channel at first and observe whether the Statement is still false and if the reason for that changes the way I expect. Then I may add sending the data, but pass something other than `_data` to the `Send()` method (e.g. a `null`), etc. This way, I "test my test", not only for correctness (whether it will fail for the right reason) but also for diagnostics (will it give me enough information when it fails?). Finally, this is also a way I learn about how my test automation tools inform me of issues in such cases.

Second behavior – specifying an error

The first Statement is implemented, so time for the second one – remember I put it on the TODO list a while ago so that I don't forget about it:

```
//TODO: specify a behavior where sending data
//      through a channel raises an exception
```

This behavior is that when the sending fails with an exception, the user of `DataDispatch` should receive an exception and the connection should be safely closed. Note that the notion of what "closing the connection" means is delegated to the `Channel` implementations, so when specifying the behaviors of `DataDispatch` I only need to care whether `Channel`'s `Close()` method is invoked correctly. The same goes for the meaning of "errors while sending data" – this is also the obligation of `Channel`. What we need to specify about `DataDispatch` is how it handles the sending errors regarding its user and its `Channel`.

Starting with a good name

This time, I choose the strategy of starting with a good name, because I feel I have a much better understanding of what behavior I need to specify than with my previous Statement. I pick the following name to state the expected behavior:

```
public void
ShouldRethrowExceptionAndCloseChannelWhenSendingDataFails()
{
 //...
}
```

Before I start dissecting the name into useful code, I start by stating the bleedy obvious (note that I'm mixing two strategies of starting from false Statement now – I didn't say you can't do that now, did I?). Having learned a lot by writing and implementing the previous Statement, I know for sure that:

1. I need to work with `DataDispatch` again.
2. I need to pass a mock of `Channel` to `DataDispatch` constructor.
3. `Channel` role will be played by a mock object.
4. I need to invoke the `ApplyTo()` method.
5. I need some kind of invalid data (although I don't know yet what to do to make it "invalid").

I write that down as code:

```
public void
ShouldRethrowExceptionAndCloseChannelWhenSendingDataFails()
{
 //GIVEN
 var channel = Substitute.For<Channel>();
 var dataDispatch = new DataDispatch(channel);
 byte[] invalidData; //doesn't compile

 //WHEN
 dataDispatch.ApplyTo(invalidData);

 //THEN
 Assert.True(false); //no expectations yet
}
```

Expecting that channel is closed

I also know that one aspect of the expected behavior is closing the channel. I know how to write this expectation – I can use the `Received()` method of NSubstitute on the channel mock. This will, of course, go into the `//THEN` section:

```
 //THEN
 channel.Received(1).Close(); //added
 Assert.True(false); //not removing this yet
}
```

I used `Received(1)` instead of just `Received()`, because attempting to close the channel several times might cause trouble, so I want to be explicit on the expectation that the `DataDispatch` should close the channel exactly once. Another thing – I am not removing the `Assert.True(false)` yet, as the current implementation already closes the channel and so the Statement could become true if not for this assertion (if it compiled, that is). I will remove this assertion only after I fully define the behavior.

Expecting exception

Another thing I expect `DataDispatch` to do in this behavior is to rethrow any sending errors, which are reported as exceptions thrown by `Channel` from the `Send()` method.

Typically, I rarely write Statements about rethrown exceptions, but here I have no choice – if I don't catch the exception in my Statement, I won't be able to evaluate whether the channel was closed or not, since the uncaught exception will stop executing the Statement.

To specify that I expect an exception in my Statement, I need to use a special assertion called `Assert.Throws<>()` and pass the code that should throw the exception as a lambda:

```
 //WHEN
Assert.Throws<Exception>(() =>
  dataDispatch.ApplyTo(invalidData));
```

Defining invalid data

My compiler shows me that the `data` variable is undefined. OK, now the time has come to define *invalid data*.

First of all, remember that `DataDispatch` cannot tell the difference between valid and invalid data - this is the purpose of the `Channel` as each `Channel` implementation might have different criteria for data validation. In my Statement, I use a mock to play the channel role, so I can just tell my mock that it should treat the data I define in my Statement as invalid. Thus, the value of the `data` itself is irrelevant as long as I configure my `Channel` mock to act as if it was invalid. This means that I can just define the `data` as any byte array:

```
var invalidData = Any.Array<byte>();
```

I also need to write down the assumption of how the `channel` will behave given this data:

```
//GIVEN
...
var exceptionFromChannel = Any.Exception();
channel.When(c => c.Send(invalidData)).Throw(exceptionFromChannel);
```

Note that the place where I configure the mock to throw an exception is the `//GIVEN` section. This is because any predefined mock behavior is my assumption. By pre-canning the method outcome, in this case, I say "given that channel for some reason rejects this data".

Now that I have the full Statement code, I can get rid of the `Assert.True(false)` assertion. The full Statement looks like this:

```
public void
ShouldRethrowExceptionAndCloseChannelWhenSendingDataFails()
{
 //GIVEN
 var channel = Substitute.For<Channel>();
 var dataDispatch = new DataDispatch(channel);
 var data = Any.Array<byte>();
 var exceptionFromChannel = Any.Exception();

 channel.When(c => c.Send(data)).Throw(exceptionFromChannel);

 //WHEN
 var exception = Assert.Throws<Exception>(() =>
  dataDispatch.ApplyTo(invalidData));
```

```
15
16  //THEN
17  Assert.Equal(exceptionFromChannel, exception);
18  channel.Received(1).Close();
19 }
```

Now, it may look a bit messy, but given my toolset, this will have to do. The Statement will now turn false on the second assertion. Wait, the second? What about the first one? Well, the first assertion says that an exception should be rethrown and methods in C# rethrow the exception by default, not requiring any implementation on my part[108]. Should I just accept it and go on? Well, I don't want to. Remember what I wrote in the first part of the book – we need to see each assertion fail at least once. An assertion that passes straight away is something we should be suspicious about. What I need to do now is to temporarily break the behavior so that I can see the failure. I can do that in (at least) two ways:

1. By going to the Statement and commenting out the line that configures the `Channel` mock to throw an exception.
2. By going to the production code and surrounding the `channel.Send(data)` statement with a try-catch block.

Either way would do, but I typically prefer to change the production code and not alter my Statements, so I chose the second way. By wrapping the `Send()` invocation with `try` and empty `catch`, I can now observe the assertion fail, because an exception is expected but none comes out of the `dataDispatch.ApplyTo()` invocation. Now I'm ready to undo my last change, confident that my Statement describes this part of the behavior well and I can focus on the second assertion, which is:

```
1 channel.Received(1).Close();
```

This assertion fails because my current implementation of the `ApplyTo()` method is:

```
1 _channel.Open();
2 _channel.Send(data);
3 _channel.Close();
```

and an exception thrown from the `Send()` method interrupts the processing, instantly exiting the method, so `Close()` is never called. I can change this behavior by using `try-finally` block to wrap the call to `Send()`[109]:

[108] that's why typically I don't specify that something should rethrow an exception – I do it this time because otherwise, it would not let me specify how `DataDispatch` uses a `Channel`.

[109] of course, the idiomatic way to do it in C# would be to use the `IDisposable` interface and a `using` block (or `IAsyncDisposable` and `await using` in the case of `async` calls).

```
_channel.Open();
try
{
 _channel.Send(data);
}
finally
{
 _channel.Close();
}
```

This makes my second Statement true and concludes this example. If I were to go on, my next step would be to implement the newly discovered `Channel` interface, as currently, it has no implementation at all.

Summary

In this chapter, I delved into writing mock-based Statements and developing classes in a test-first manner. I am not showing this example as a prescription or any kind of "one true way" of test-driving such implementation - some things could've been done differently. For example, there were many situations where I got several TODO items pointed out by my compiler or my false Statement. Depending on many factors, I might've approached them in a different order. Or in the second behavior, I could've defined `data` as `Any.Array<byte>()` right from the start (and left a TODO item to check on it later and confirm whether it can stay this way) to get the Statement to a compiling state quicker.

Another interesting point was the moment when I discovered the `Channel` interface – I'm aware that I slipped over it by saying something like "we can see that the class has too many purposes, then magic happens and then we've got an interface to delegate parts of the logic to". This "magic happens" or, as I mentioned, "interface discovery", is something I will expand on in the following chapters.

You might've noticed that this chapter was longer than the last one, which may lead you to a conclusion that TDD complicates things rather than simplifying them. There were, however, several factors that made this chapter longer:

1. In this chapter, I specified two behaviors (a "happy path" plus error handling), whereas in the last chapter I only specified one (the "happy path").
2. In this chapter, I designed and implemented the `DataDispatch` class and discovered the `Channel` interface whereas in the last chapter they were given to me right from the start.
3. Because I assume the test-first way of writing Statements may be less familiar to you, I took my time to explain my thought process in more detail.

So don't worry – when one gets used to it, the process I described in this chapter typically takes several minutes at worst.

Test-driving at the input boundary

In this chapter, we'll be joining Johnny and Benjamin again as they try to test-drive a system starting from its input boundaries. This will hopefully demonstrate how abstractions are pulled from need and how roles are invented at the boundary of a system. The further chapters will explore the domain model. This example makes several assumptions:

1. In this story, Johnny is a super-programmer, who never makes mistakes. In real-life TDD, people make mistakes and correct them, sometimes they go back and forth thinking about tests and design. Here, Johnny gets everything right the first time. Although I know this is a drop on realism, I hope that it will help my readers in observing how some TDD mechanics work. This is also why Johnny and Benjamin will not need to refactor anything in this chapter.
2. There will be no Statements written on higher than unit level. This means that Johnny and Benjamin will do TDD using unit-level Statements only. This is why they will need to do some things they could avoid if they could write a higher-level Statement. A separate part of this book will cover working with different levels of Statements at the same time.
3. This chapter (and several next ones) will avoid the topic of working with any I/O, randomness, and other hard-to-test stuff. For now, I want to focus on test-driving pure code-based logic.

With this out of our way, let's join Johnny and Benjamin and see what kind of issue they are dealing with and how they try to solve it using TDD.

Fixing the ticket office

Johnny: What do you think about trains, Benjamin?

Benjamin: Are you asking because I was traveling by train yesterday to get here? Well, I like it, especially after some of the changes that happened over the recent years. I truly think that today's railways are modern and passenger-friendly.

Johnny: And about the seat reservation process?

Benjamin: Oh, that… I mean, why didn't they still automate the process? Is this even thinkable that in the 21st century I cannot reserve a seat through the internet?

Johnny: I kinda hoped you'd say that because our next assignment is to do exactly that.

Benjamin: You mean reserving seats through the internet?

Johnny: Yes, the railroad company hired us.

Benjamin: You're kidding me, right?

Johnny: No, I'm telling the truth.

Benjamin: No way.

Johnny: Take your smartphone and check your e-mail. I already forwarded the details to you.

Benjamin: Hey, that looks legit. Why didn't you tell me earlier?

Johnny: I'll explain on the way. Come on, let's go.

Initial objects

Benjamin: do we have any sort of requirements, stories or whatever to work with?

Johnny: Yes, I'll explain some as I walk you through the input and output data structures. It will be enough to get us going.

Request

Johnny: Somebody's already written the part that accepts an HTTP request and maps it to the following structure:

```
public class ReservationRequestDto
{
  public readonly string TrainId;
  public readonly uint SeatCount;

  public ReservationRequestDto(string trainId, uint seatCount)
  {
    TrainId = trainId;
    SeatCount = seatCount;
  }
}
```

Benjamin: I see... Hey, why does the `ReservationRequestDto` name has `Dto` in it? What is it?

Johnny: The suffix `Dto` means that this class represents a Data Transfer Object (in short, DTO)[110]. Its role is just to transfer data across the process boundaries.

Benjamin: So you mean it is just needed to represent some kind of XML or JSON that is sent to the application?

Johnny: Yes, you could say that. The reason people typically place `Dto` in these names is to communicate that these data structures are special - they represent an outside contract and cannot be freely modified like other objects.

Benjamin: Does it mean that I can't touch them?

Johnny: It means that if you did touch them, you'd have to make sure they are still correctly mapped from outside data, like JSON or XML.

[110]Patterns of enterprise application architecture, M. Fowler.

Benjamin: So I'd better not mess with them. Cool, and what about the train ID?

Johnny: It represents a train. The client application that uses the backend that we'll write will understand these IDs and use them to communicate with us.

Benjamin: Cool, what's next?

Johnny: The client application tells our service how many seats it needs to reserve, but doesn't say where. This is why there's only a `seatCount` parameter. Our service must determine which seats to pick.

Benjamin: So if a couple wants to reserve two seats, they can be in different coaches?

Johnny: Yes, however, there are some preference rules that we need to code in, like, if we can, a single reservation should have all seats in the same coach. I'll fill you in on the rules later.

Response

Benjamin: Do we return something to the client app?

Johnny: Yes, we need to return a response, which, no surprise, is also modeled in the code as a DTO. This response represents the reservation made:

```
public class ReservationDto
{
  public readonly string TrainId;
  public readonly string ReservationId;
  public readonly List<TicketDto> PerSeatTickets;

  public ReservationDto(
    string trainId,
    List<TicketDto> perSeatTickets,
    string reservationId)
  {
    TrainId = trainId;
    PerSeatTickets = perSeatTickets;
    ReservationId = reservationId;
  }
}
```

Benjamin: I see that there's a train ID, which is… the same as the one in the request, I suppose?

Johnny: Right. It's used to correlate the request and the response.

Benjamin: …and there is a reservation ID - does our service generate that?

Johnny: Correct.

Benjamin: but the `PerSeatTickets` field… it is a list of `TicketDto`, which as I understand is one of our custom types. Where is it?

Johnny: I forgot to show it to you. `TicketDto` is defined as:

```
public class TicketDto
{
  public readonly string Coach;
  public readonly int SeatNumber;

  public TicketDto(string coach, int seatNumber)
  {
    Coach = coach;
    SeatNumber = seatNumber;
  }
}
```

so it has a coach name and a seat number, and we have a list of these in our reservation.

Benjamin: So a single reservation can contain many tickets and each ticket is for a single place in a specific coach, right?

Johnny: Yes.

Ticket Office class

Benjamin: The classes we just saw - are they representations of some kind of JSON or XML?

Johnny: Yes, but we don't have to write the code to do it. As I mentioned earlier, someone already took care of that.

Benjamin: Lucky us.

Johnny: Our work starts from the point where the deserialized data is passed to the application as a DTO. The request entry point is in a class called `TicketOffice`:

```
[SomeKindOfController]
public class TicketOffice
{
    [SuperFrameworkMethod]
    public ReservationDto MakeReservation(ReservationRequestDto requestDto)
    {
        throw new NotImplementedException("Not implemented");
    }
}
```

Johnny: I see that the class is decorated with attributes specific to a web framework, so we will probably not implement the use case directly in the `MakeReservation` method to avoid coupling our use case logic to the code that has to meet the requirements of a specific framework.

Benjamin: So what you're saying is that you're trying to keep the `TicketOffice` class away from the application logic as much you can?

Johnny: Yes. I only need it to wrap all the data into appropriate abstractions which I design to match my preferences, not the framework. Then, in these abstractions, I can freely code my solution the way I like.

Bootstrap

Benjamin: Are we ready to go?

Johnny: Typically, if I were you, I would like to see one more place in the code.

Benjamin: Which is..?

Johnny: The composition root, of course.

Benjamin: Why would I like to see a composition root?

Johnny: The first reason is that it is very close to the entry point of the application, so it is a chance for you to see how the application manages its dependencies. The second reason is that each time we will be adding a new class that has the same lifespan as the application, we will need to go to the composition root and modify it. Sooo I'd probably like to know where it is and how I should work with it.

Benjamin: I thought I could find that later, but while we're at it, can you show me the composition root?

Johnny: Sure, it's here, in the `Application` class:

```
1  public class Application
2  {
3    public static void Main(string[] args)
4    {
5      new WebApp(
6          new TicketOffice()
7      ).Host();
8    }
9  }
```

Benjamin: Good to see it doesn't require us to use any fancy reflection-based mechanism for composing objects.

Johnny: Yes, we're lucky about that. We can just create the objects with the `new` operator and pass them to the framework. Another important piece of information is that we have a single `TicketOffice` instance to handle all the requests. This means that we cannot store any state related to a single request inside this instance as it will come into conflict with other requests.

Writing the first Statement

Johnny: Anyway, I think we're ready to start.

Benjamin: But where do we start from? Should we write some kind of a class called "Reservation" or "Train" first?

Johnny: No, what we will do is we will start from the inputs and work our way towards the inside of the application. Then, if necessary, to the outputs.

Benjamin: I don't think I understand what you're talking about. Do you mean this "outside-in" approach that you mentioned yesterday?

Johnny: Yes, and don't worry if you didn't get what I said, I will explain as we go. For now, the only thing I mean by it is that we will follow the path of the request as it comes from the outside of our application and start implementing in the first place where the request is not handled as it should be. Specifically, this means we start at:

```
[SomeKindOfController]
public class TicketOffice
{
    [SuperFrameworkMethod]
    public ReservationDto MakeReservation(ReservationRequestDto requestDto)
    {
        throw new NotImplementedException("Not implemented");
    }
}
```

Benjamin: Why?

Johnny: Because this is the place nearest to the request entry point where the behavior differs from the one we expect. As soon as the request reaches this point, its handling will stop and an exception will be thrown. We need to alter this code if we want the request to go any further.

Benjamin: I see... so... if we didn't have the request deserialization code in place already, we'd start there because that would be the first place where the request would get stuck on its way towards its goal, right?

Johnny: Yes, you got it.

Benjamin: And... we start with a false Statement, no?

Johnny: Yes, let's do that.

First Statement skeleton

Benjamin: Don't tell me anything, I'll try doing it myself.

Johnny: Sure, as you wish.

Benjamin: The first thing I need to do is to add an empty Specification for the `TicketOffice` class:

```
public class TicketOfficeSpecification
{
    //TODO add a Statement
}
```

Then, I need to add my first Statement. I know that in this Statement, I need to create an instance of the `TicketOffice` class and call the `MakeReservation` method, since it's the only method in this class and it's not implemented.

Johnny: so what strategy do you use for starting with a false Statement?

Benjamin: "invoke a method when you have one", as far as I remember.

Johnny: So what's the code going to look like?

Benjamin: for starters, I will do a brain dump just as you taught me. After stating all the bleedy obvious facts, I get:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var ticketOffice = new TicketOffice();

  //WHEN
  ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.True(false);
}
```

Johnny: Good... my... apprentice...

Benjamin: What?

Johnny: Oh, nevermind... anyway, the code doesn't compile now, since this line:

```
ticketOffice.MakeReservation(requestDto);
```

uses a variable `requestDto` that does not exist. Let's generate it using our IDE!

Benjamin: By the way, I wanted to ask about this exact line. Making it compile is something we need to do to move on. Weren't we supposed to add a `TODO` comment for things we need to get back to? Just as we did with the Statement name, which was:

```
public void ShouldXXXXX() //TODO better name
```

Johnny: My opinion is that this is not necessary, because the compiler, by failing on this line, has already created a sort of TODO item for us, just not on our TODO list but the compiler error log. This is different than e.g. a need to change a method's name, which the compiler will not remind us about.

Benjamin: So my TODO list is composed of compile errors, false Statements, and the items I manually mark as `TODO`? Is this how I should understand it?

Johnny: Exactly. Going back to the `requestDto` variable, let's create it.

Benjamin: Sure. It came out like this:

```
ReservationRequestDto requestDto;
```

We need to assign something to the variable.

Johnny: Yes, and since it's a DTO, it is certainly not going to be a mock.

Benjamin: You mean we don't mock DTOs?

Johnny: No, there's no need. DTOs are, by definition, data structures and mocking involves polymorphism which applies to behavior rather than data. Later I will explain it in more detail. For now, just accept my word on it.

Benjamin: Sooo... if it's not going to be a mock, then let's generate an anonymous value for it using the `Any.Instance<>()` method.

Johnny: That is exactly what I would do.

Benjamin: So I will change this line:

```
ReservationRequestDto requestDto;
```

to:

```
var requestDto = Any.Instance<ReservationRequestDto>();
```

Setting up the expectation

Johnny: Yes, and now the Statement compiles, and it seems to be false. This is because of this line:

```
Assert.True(false);
```

Benjamin: so we change this `false` to `true` and we're done here, right?

Johnny: ...Huh?

Benjamin: Oh, I was just pulling your leg. What I really wanted to say is: let's turn this assertion into something useful.

Johnny: Phew, don't scare me like that. Yes, this assertion needs to be rewritten. And it so happens that when we look at the following line:

```
ticketOffice.MakeReservation(requestDto);
```

it doesn't make any use of the return value of `MakeReservation()` while it's evident from the signature that its return type is a `ReservationDto`. Look:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
```

In our Statement, we don't do anything with it.

Benjamin: Let me guess, you want me to go to the Statement, assign this return value to a variable and then assert its equality to... what exactly?

Johnny: For now, to an expected value, which we don't know yet, but we will worry about it later when it really blocks us.

Benjamin: This is one of those situations where we need to imagine that we already have something we don't, right?. Ok, here goes:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var ticketOffice = new TicketOffice();

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  //doesn't compile - we don't have expectedReservationDto yet:
  Assert.Equal(expectedReservationDto, reservationDto);
}
```

There, I did what you asked. So please explain to me now how did it get us any closer to our goal?

Johnny: We transformed our problem from "what assertion to write" into "what is the reservation that we expect". This is indeed a step in the right direction.

Benjamin: Please enlighten me then - what is "the reservation that we expect"?

Johnny: For now, the Statement is not compiling at all, so to go any step further, we can just introduce an `expectedReservationDto` as an anonymous object. Thus, we can just write in the `GIVEN` section:

```
var expectedReservationDto = Any.Instance<ReservationDto>();
```

and it will make the following code compile:

```
//THEN
Assert.Equal(expectedReservationDto, reservationDto);
```

Benjamin: But this assertion will fail anyway...

Johnny: That's still better than not compiling, isn't it?

Benjamin: Well, if you put it this way... Now our problem is that the expected value from the assertion is something the production code doesn't know about - this is just something we created in our Statement. This means that this assertion does not assert the outcome of the behavior of the production code. How do we solve this?

Johnny: This is where we need to exercise our design skills to introduce some new collaborators. This task is hard at the boundaries of application logic where we need to draw the collaborators not from the domain, but rather think about design patterns that will allow us to reach our goals. Every time we enter our application logic, we do so with a specific use case in mind. In this particular example, our use case is "making a reservation". A use case is typically represented by either a method in a facade[111] or a command object[112]. Commands are a bit more complex but more scalable. If making a reservation was our only use case, it probably wouldn't make sense to use it. But as we already have more high priority requests for features, I believe we can assume that commands will be a better fit.

Benjamin: So you propose to use a more complex solution - isn't that "big design up front"?

Johnny: I believe that it isn't. Remember I'm using just *a bit* more complex solution. The cost of implementation is only a bit higher as well as the cost of maintenance in case I'm wrong. If for some peculiar reason someone says tomorrow that they don't need the rest of the features at all, the increase in complexity will be negligible taking into account the small size of the overall codebase. If, however, we add more features, then using commands will save us some time in the longer run. Thus, given what I know, I am not adding this to support speculative new features, but to make the code easier to modify in the long run[113]. I agree though that choosing just enough complexity for a given moment is a difficult task[114].

Benjamin: I still don't get how introducing a command is going to help us here. Typically, a command has an `Execute()` method that doesn't return anything. How then will it give us the response that we need to return from the `MakeReservation()`? And also, there's another issue: how is this command going to be created? It will probably require the request passed as one of its constructor parameters. This means we cannot just pass the command to the `TicketOffice`'s constructor as the first time we can access the request is when the `MakeReservation()` method is invoked.

Johnny: Yes, I agree with both of your concerns. Thankfully, when you choose to go with commands, typically there are standard solutions to the problems you mentioned. The commands can be created using factories and they can convey their results using a pattern called *collecting parameter*[115] - we will pass an object inside the command to gather all the events from handling the use case and then be able to prepare a response for us.

Introducing a reservation in progress collaborator

Johnny: Let's start with the collecting parameter, which will represent a domain concept of a reservation in progress. We need it to collect the data necessary to build a response DTO. Thus, what we currently know about it is that it's going to be converted to a response DTO at the

[111] https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-facade-pattern
[112] https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-command-pattern
[113] https://martinfowler.com/bliki/Yagni.html
[114] https://www.youtube.com/watch?v=aCLBd3a1rwk
[115] Refactoring to Patterns, Joshua Kerievsky

very end. All of the three objects: the command, the collecting parameter, and the factory, are collaborators, so they will be mocks in our Statement.

Benjamin: Ok, lead the way.

Johnny: Let's start with the `GIVEN` section. Here, we need to say that we assume that the collecting parameter mock, let's call it `reservationInProgress` will give us the `expectedReservationDto` (which is already defined in the body of the Statement) when asked:

```
//GIVEN
//...
reservationInProgress.ToDto().Returns(expectedReservationDto);
```

Of course, we don't have the `reservationInProgress` yet, so we need to introduce it. As I explained earlier, this needs to be a mock, because otherwise, we wouldn't be able to call `Returns()` on it:

```
///GIVEN
var reservationInProgress = Substitute.For<ReservationInProgress>();
//...
reservationInProgress.ToDto().Returns(expectedReservationDto);
//...
```

Now, the Statement does not compile because the `ReservationInProgress` interface that I just used in the mock definition is not introduced yet.

Benjamin: In other words, you just discovered that you need this interface.

Johnny: Exactly. What I'm currently doing is I'm pulling abstractions and objects into my code as I need them. And my current need is to have the following interface in my code:

```
public interface ReservationInProgress
{

}
```

Now, the Statement still doesn't compile, because there's this line:

```
reservationInProgress.ToDto().Returns(expectedReservationDto);
```

which requires the `ReservationInProgress` interface to have a `ToDto()` method, but for now, this interface is empty. After adding the required method, it looks like this:

```
public interface ReservationInProgress
{
  ReservationDto ToDto();
}
```

and the Statement compiles again, although it is still a false one.

Benjamin: Ok. Now give me a second to grasp the full Statement in its current state.

Johnny: Sure, take your time, this is how it currently looks like:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var ticketOffice = new TicketOffice();
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();

  reservationInProgress.ToDto().Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservationDto, reservationDto);
}
```

Introducing a factory collaborator

Benjamin: I think I managed to catch up. Can I grab the keyboard?

Johnny: I was about to suggest it. Here.

Benjamin: Thanks. Looking at this Statement, we have this `ReservationInProgress` all setup and created, but this mock of ours is not passed to the `TicketOffice` at all. So how should the `TicketOffice` use our pre-configured `reservationInProgress`?

Johnny: Remember our discussion about separating object use from construction?

Benjamin: I guess I know what you're getting at. The `TicketOffice` should somehow get an already created `ReservationInProgress` object from the outside. It can get it e.g. through a constructor or from a factory.

Johnny: Yes, and if you look at the lifetime scope of our `TicketOffice`, which is created once at the start of the application, it can't accept a `ReservationInProgress` through a constructor, because every time a new reservation request is made, we want to have a new `ReservationInProgress`, so passing it through a `TicketOffice` constructor would force us to

create a new `TicketOffice` every time as well. Thus, the solution that better fits our current situation is…

Benjamin: A factory, right? You're suggesting that instead of passing a `ReservationInProgress` through a constructor, we should rather pass something that knows how to create `ReservationInProgress` instances?

Johnny: Exactly.

Benjamin: How to write it in the Statement?

Johnny: First, write only what you need. The factory needs to be a mock because we need to configure it so that when asked, it returns our `ReservationInProgress` mock. So let's write that return configuration first, pretending we already have the factory available in our Statement body.

Benjamin: Let me see… that should do it:

```
1 //GIVEN
2 ...
3 reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);
```

Johnny: Nice. Now the code does not compile, because we don't have a `reservationInProgressFactory`. So let's create it.

Benjamin: And, as you said earlier, it should be a mock object. Then this will be the definition:

```
1 var reservationInProgressFactory
2   = Substitute.For<ReservationInProgressFactory>();
```

For the need of this line of code, I pretended that I have an interface called `ReservationInProgressFactory`, and, let me guess, you want me to introduce this interface now?

Johnny: (smiles)

Benjamin: Alright. Here:

```
1 public interface ReservationInProgressRepository
2 {
3
4 }
```

And now, the compiler tells us that we don't have the `FreshInstance()` method, so let me introduce it:

```
public interface ReservationInProgressRepository
{
    ReservationInProgress FreshInstance();
}
```

Good, the compiler doesn't complain anymore, but the Statement fails with a `NotImplementedException`.

Johnny: Yes, this is because the current body of the `MakeReservation()` method of the `TicketOffice` class looks like this:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
{
  throw new NotImplementedException("Not implemented");
}
```

Expanding the ticket office constructor

Benjamin: So should we implement this now?

Johnny: We still have some stuff to do in the Statement.

Benjamin: Like..?

Johnny: For example, the `ReservationInProgressFactory` mock that we just created is not passed to the `TicketOffice` constructor yet, so there is no way for the ticket office to use this factory.

Benjamin: Ok, so I'll add it. The Statement will change in this place:

```
var reservationInProgressFactory
  = Substitute.For<ReservationInProgressFactory>();
var ticketOffice = new TicketOffice();
```

to:

```
var reservationInProgressFactory
  = Substitute.For<ReservationInProgressFactory>();
var ticketOffice = new TicketOffice(reservationInProgressFactory);
```

and a constructor needs to be added to the `TicketOffice` class:

```
public TicketOffice(ReservationInProgressFactory reservationInProgressFactory)
{

}
```

Johnny: Agreed. And, we need to maintain the composition root which just stopped compiling. This is because the constructor of a `TicketOffice` is invoked there and it needs an update as well:

```
public class Application
{
  public static void Main(string[] args)
  {
    new WebApp(
        new TicketOffice(/* compile error - instance missing */)
    ).Host();
  }
}
```

Benjamin: But what should we pass? We have an interface, but no class of which we could make an instance.

Johnny: We need to create the class. Typically, if I have an idea about the name of the required class, I create the class by that name. If I don't have any idea on how to call it yet, I can call it e.g. `TodoReservationInProgressFactory` and leave a TODO comment to get back to it later. For now, we just need this class to compile the code. It's still out of the scope of our current Statement.

Benjamin: So maybe We could pass a `null` so that we have `new TicketOffice(null)`?

Johnny: We could, but that's not my favorite option. I typically just create the class. One of the reasons is that the class will need to implement an interface to compile and then we will need to introduce a method which will, by default, throw a `NotImplementedException` and these exceptions will end up on my TODO list as well.

Benjamin: Ok, that sounds reasonable to me. So this line:

```
new TicketOffice(/* compile error - instance missing */)
```

becomes:

```
new TicketOffice(
    new TodoReservationInProgressFactory()) //TODO change the name
```

And it doesn't compile, because we need to create this class so let me do it:

```
public class TodoReservationInProgressFactory : ReservationInProgressFactory
{
}
```

Johnny: It still doesn't compile, because the interface `ReservationInProgressFactory` has some methods that we need to implement. Thankfully, we can do this with a single IDE command and get:

```
public class TodoReservationInProgressFactory : ReservationInProgressFactory
{
    public ReservationInProgress FreshInstance()
    {
        throw new NotImplementedException();
    }
}
```

and, as I mentioned a second ago, this exception will end up on my TODO list, reminding me that I need to address it.

Now that the code compiles again, let's backtrack to the constructor of `TicketOffice`:

```
public TicketOffice(ReservationInProgressFactory reservationInProgressFactory)
{

}
```

here, we could already assign the constructor parameter to a field, but it's also OK to do it later.

Benjamin: Let's do it later, I wonder how far we can get delaying work like this.

Introducing a command collaborator

Johnny: Sure. So let's take a look at the Statement we're writing. It seems we are missing one more expectation in our `THEN` section. if you look at the Statement's full body as it is now:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var reservationInProgressFactory
    = Substitute.For<ReservationInProgressFactory>();
  var ticketOffice = new TicketOffice(reservationInProgressFactory);
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();

  reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);
  reservationInProgress.ToDto().Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservation, reservationDto);
}
```

the only interaction between a `TicketOffice` and `ReservationInProgress` it describes is the former calling the `ToDto` method on the latter. So the question that we need to ask ourselves now is "how will the instance of `ReservationInProgress` know what `ReservationDto` to create when this method is called?".

Benjamin: Oh right... the `ReservationDto` needs to be created by the `ReservationInProgress` based on the current state of the application and the data in the `ReservationRequestDto`, but the `ReservationInProgress` knows nothing about any of these things so far.

Johnny: Yes, filling the `ReservationInProgress` is one of the responsibilities of the application we are writing. If we did it all in the `TicketOffice` class, this class would surely have too much to handle and our Statement would grow immensely. So we need to push the responsibility of handling our use case further to other collaborating roles and use mocks to play those roles here.

Benjamin: So what do you propose?

Johnny: Remember our discussion from several minutes ago? Usually, when I push a use case-related logic to another object at the system boundary, I pick from among the Facade pattern and the Command pattern. Facades are simpler but less scalable, while commands are way more scalable and composable but a new command object needs to be created by the application each time a use case is triggered and a new command class needs to be created by a programmer when support for a new use case is added to the system.

Benjamin: I already know that you prefer commands.

Johnny: Yeah, bear with me if only for the sake of seeing how commands can be used here. I am sure you could figure out the Facade option by yourself.

Benjamin: What do I type?

Johnny: Let's assume we have this command and then let's think about what we want our `TicketOffice` to do with it.

Benjamin: We want the `TicketOffice` to execute the command, obviously..?

Johnny: Right, let's write this in a form of expectation.

Benjamin: Ok, I'd write something like this:

```
1 reservationCommand.Received(1).Execute(reservationInProgress);
```

I already passed the `reservationInProgress` as the command will need to fill it.

Johnny: Wait, it so happens that I prefer another way of passing this `reservationInProgress` to the `reservationCommand`. Please, for now, make the `Execute()` method parameterless.

Benjamin: As you wish, but I thought this would be a good place to pass it.

Johnny: It might look like it, but typically, I want my commands to have parameterless execution methods. This way I can compose them more freely, using patterns such as decorator[116].

Benjamin: As you wish. I reverted the last change and now the `THEN` section looks like this:

[116]https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-decorator-pattern

```
//THEN
reservationCommand.Received(1).Execute();
Assert.Equal(expectedReservationDto, reservationDto);
```

and it doesn't compile of course. `reservationCommand` doesn't exist, so I need to introduce it. Of course, the type of this variable doesn't exist as well – I need to pretend it does. And, I already know it should be a mock since I specify that it should have received a call to its `Execute()` method.

Johnny: (nods)

Benjamin: In the `GIVEN` section, I'll add the `reservationCommand` as a mock:

```
var reservationCommand = Substitute.For<ReservationCommand>();
```

For the sake of this line, I pretend that I have an interface called `ReservationCommand`, and now I need to create it to make the code compile.

```
public interface ReservationCommand
{

}
```

but that's not enough, because in the Statement, I expect to receive a call to an `Execute()` method on the command and there's no such method. I can fix it by adding it on the command interface:

```
public interface ReservationCommand
{
  void Execute();
}
```

and now everything compiles again.

Introducing a command factory collaborator

Johnny: Sure, now we need to figure out how to pass this command to the `TicketOffice`. As we discussed, by nature, a command object represents, well, an issued command, so it cannot be created once in the composition root and then passed to the constructor, because then:

1. it would essentially become a facade,
2. we would need to pass the `reservationInProgress` to the `Execute()` method which we wanted to avoid.

Benjamin: Wait, don't tell me... you want to add another factory here?

Johnny: Yes, that's what I would like to do.

Benjamin: But… that's the second factory in a single Statement. Aren't we, like, overdoing it a little?

Johnny: I understand why you feel that way. Still, this is a consequence of my design choice. We wouldn't need a command factory if we went with a facade. In simple apps, I just use a facade and do away with this dilemma. I could also drop the use of the collecting parameter pattern and then I wouldn't need a factory for reservations in progress, but that would mean I would not resolve the command-query separation principle violation and would need to push this violation further into my code. To cheer you up, this is an entry point for a use case where we need to wrap some things in abstractions, so I don't expect this many factories in the rest of the code. I treat this part as a sort of adapting layer which protects me from everything imposed by outside of my application logic which I don't want to deal with inside of my application logic.

Benjamin: I will need to trust you on that. I hope it makes things easier later because for now… ugh…

Johnny: Let's introduce the factory mock. Of course, before I define it, I want to use it first in the Statement to feel a need for it. I will expand the `GIVEN` section with an assumption that a factory, asked for a command, returns out `reservationCommand`:

```
//GIVEN
...
commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
```

This doesn't compile because we have no `commandFactory` yet.

Benjamin: Oh, I can see that the factory's `CreateNewReservationCommand()` is where you decided to pass the `reservationInProgress` that I wanted to pass to the `Execute()` method earlier. Clever. By leaving the command's `Execute()` method parameterless, you made it more abstract and made the interface decoupled from any particular argument types. On the other hand, the command is created in the same scope it is used, so there is literally no issue with passing all the parameters through the factory method.

Johnny: That's right. We now know we need a factory, plus that it needs to be a mock since we configure it to return a command when it is asked for one. So let's declare the factory, pretending we have an interface for it:

```
//GIVEN
...
var commandFactory = Substitute.For<CommandFactory>();
...
commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
```

Benjamin: …and the `CommandFactory` interface doesn't exist, so let's create it:

```
public interface CommandFactory
{

}
```

and let's add the missing `CreateNewReservationCommand` method:

```
public interface CommandFactory
{
  ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress);
}
```

Benjamin: Now the code compiles and looks like this:

```
[Fact]
public void ShouldXXXXX() //TODO better name
{
  //WHEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var commandFactory = Substitute.For<CommandFactory>();
  var reservationInProgressFactory
    = Substitute.For<ReservationInProgressFactory>();
  var ticketOffice = new TicketOffice(reservationInProgressFactory);
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();
  var reservationCommand = Substitute.For<ReservationCommand>();

  commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
  reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);
  reservationInProgress.ToDto().Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservationDto, reservationDto);
  reservationCommand.Received(1).Execute();
}
```

Benjamin: I can see that the command factory is not passed anywhere from the Statement - the `TicketOffice` doesn't know about it.

Johnny: Yes, and, lucky us, a factory is something that can have the same lifetime scope as the `TicketOffice` since to create the factory, we don't need to know anything about a request for

reservation. This is why we can safely pass it through the constructor of `TicketOffice`. Which means that these two lines:

```
var commandFactory = Substitute.For<CommandFactory>();
var ticketOffice = new TicketOffice(reservationInProgressFactory);
```

will now look like this:

```
var commandFactory = Substitute.For<CommandFactory>();
var ticketOffice = new TicketOffice(
    reservationInProgressFactory, commandFactory);
```

A constructor with such a signature does not exist, so to make this compile, we need to add a parameter of type `CommandFactory` to the constructor:

```
public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{

}
```

which forces us to add a parameter to our composition root. So this part of the composition root:

```
new TicketOffice(
    new TodoReservationInProgressFactory()) //TODO change the name
```

becomes:

```
new TicketOffice(
    new TodoReservationInProgressFactory(), //TODO change the name
    new TicketOfficeCommandFactory())
```

which in turn forces us to create the `TicketOfficeCommandFactory` class:

```
public class TicketOfficeCommandFactory : CommandFactory
{
  public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
      throw new NotImplementedException();
  }
}
```

Benjamin: Hey, this time you gave the class a better name than the previous factory which was called `TodoReservationInProgressFactory`. Why didn't you want to leave it for later this time?

Johnny: This time, I think I have a better idea of how to name this class. Typically, I name concrete classes based on something from their implementation and I find the names hard to find when I don't have this implementation yet. This time I believe I have a name that can last a bit, which is also why I didn't leave a TODO comment next to this name. Still, further work can invalidate my naming choice, and I will be happy to change the name when the need arises. For now, it should suffice.

Giving the Statement a name

Johnny: Anyway, getting back to the Statement, I think we've got it all covered. Let's just give it a good name. Looking at the assertions:

```
1 reservationCommand.Received(1).Execute();
2 Assert.Equal(expectedReservationDto, reservationDto);
```

I think we can say:

```
1 public void
2 ShouldExecuteReservationCommandAndReturnResponseWhenAskedToMakeReservation()
```

Benjamin: Just curious... Didn't you tell me to watch out for the "and" word in Statement names and that it may suggest something is wrong with the scenario.

Johnny: Yes, and in this particular case, there *is* something wrong - the class `TicketOffice` violates the command-query separation principle. This is also why the Statement looks so messy. For this class, however, we don't have a big choice since our framework requires this kind of method signature. That's also why we are working so hard in this class to introduce the collecting parameter and protect the rest of the design from the violation.

Benjamin: I hope the future Statements will be easier to write than this one.

Johnny: Me too.

Making the Statement true - the first assertion

Johnny: Let's take a look at the code of the Statement:

```
[Fact] public void
ShouldExecuteReservationCommandAndReturnResponseWhenAskedToMakeReservation()
{
  //GIVEN
  var requestDto = Any.Instance<ReservationRequestDto>();
  var commandFactory = Substitute.For<CommandFactory>();
  var reservationInProgressFactory
    = Substitute.For<ReservationInProgressFactory>();
  var reservationInProgress = Substitute.For<ReservationInProgress>();
  var expectedReservationDto = Any.Instance<ReservationDto>();
  var reservationCommand = Substitute.For<ReservationCommand>();

  var ticketOffice = new TicketOffice(
    reservationInProgressFactory,
    commandFactory);

  reservationInProgressFactory.FreshInstance()
    .Returns(reservationInProgress);
  commandFactory.CreateNewReservationCommand(requestDto, reservationInProgress)
    .Returns(reservationCommand);
  reservationInProgress.ToDto()
    .Returns(expectedReservationDto);

  //WHEN
  var reservationDto = ticketOffice.MakeReservation(requestDto);

  //THEN
  Assert.Equal(expectedReservationDto, reservationDto);
  reservationCommand.Received(1).Execute();
}
```

Johnny: I think it is complete, but we won't know that until we see the assertions failing and then passing. For now, the implementation of the `MakeReservation()` method throws an exception and this exception makes our Statement stop at the `WHEN` stage, not even getting to the assertions.

Benjamin: But I can't just put the right implementation in yet, right? This is what you have always told me.

Johnny: Yes, ideally, we should see the assertion errors to gain confidence that the Statement *can* turn false when the expected behavior is not in place.

Benjamin: This can only mean one thing – return `null` from the `TicketOffice` instead of throwing the exception. Right?

Johnny: Yes, let me do it. I'll just change this code:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
{
  throw new NotImplementedException("Not implemented");
}
```

to:

```
public ReservationDto MakeReservation(ReservationRequestDto requestDto)
{
  return null;
}
```

Now I can see that the first assertion:

```
Assert.Equal(expectedReservationDto, reservationDto);
```

is failing, because it expects an `expectedReservationDto` from the `reservationInProgress` but the `reservationInProgress` itself can only be received from the factory. The relevant lines in the Statement that say this are:

```
reservationInProgressFactory.FreshInstance()
    .Returns(reservationInProgress);
reservationInProgress.ToDto()
    .Returns(expectedReservationDto);
```

Let's just implement the part that is required to pass this first assertion. To do this, I will need to create a field in the `TicketOffice` class for the factory and initialize it with one of the constructor parameters. So this code:

```
public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{

}
```

becomes:

```
private readonly ReservationInProgressFactory _reservationInProgressFactory;

public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{
  _reservationInProgressFactory = reservationInProgressFactory;
}
```

Benjamin: Couldn't you just introduce the other field as well?

Johnny: Yes, I could and I usually do that. But since we are training, I want to show you that we will be forced to do so anyway to make the second assertion pass.

Benjamin: Go on.

Johnny: Now I have to modify the `MakeReservation()` method by adding the following code that creates the reservation in progress and makes it return a DTO:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
return reservationInProgress.ToDto();
```

Benjamin: ...and the first assertion passes.

Johnny: Yes, and this is clearly not enough. If you look at the production code, we are not doing anything with the `ReservationRequestDto` instance. Thankfully, we have the second assertion:

```
reservationCommand.Received(1).Execute();
```

and this one fails. To make it pass, We need to create a command and execute it.

Making the Statement true – the second assertion

Benjamin: Wait, why are you calling this an "assertion"? There isn't a word "assert" anywhere in this line. Shouldn't we just call it "mock verification" or something like that?

Johnny: I'm OK with "mock verification", however, I consider it correct to call it an assertion as well, because, in essence, that's what it is – a check that throws an exception when a condition is not met.

Benjamin: OK, if that's how you put it...

Johnny: anyway, we still need this assertion to pass.

Benjamin: Let me do this. So to create a command, we need the command factory. This factory is already passed to `TicketOffice` via its constructor, so we need to assign it to a field because as of now, the constructor looks like this:

```
public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{
  _reservationInProgressFactory = reservationInProgressFactory;
}
```

After adding the assignment, the code looks like this:

```
private readonly CommandFactory _commandFactory;

public TicketOffice(
    ReservationInProgressFactory reservationInProgressFactory,
    CommandFactory commandFactory)
{
  _reservationInProgressFactory = reservationInProgressFactory;
  _commandFactory = commandFactory;
}
```

and now in the `MakeReservation()` method, I can ask the factory to create a command:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
return reservationInProgress.ToDto();
```

But this code is not complete yet – I still need to execute the created command like this:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

Johnny: Great! Now the Statement is true.

Benjamin: Wow, this isn't a lot of code for such a big Statement that we wrote.

Johnny: The real complexity is not even in the lines of code, but the number of dependencies that we had to drag inside. Note that we have two factories in here. Each factory is a dependency and it creates another dependency. This is better visible in the Statement than in the production method and this is why I find it a good idea to pay close attention to how a Statement is growing and using it as a feedback mechanism for the quality of my design. For this particular class, the design issue we observe in the Statement can't be helped a lot since, as I mentioned, this is the boundary where we need to wrap things in abstractions.

Benjamin: You'll have to explain this bit about design quality a bit more later.

Johnny: Sure. Tea?

Benjamin: Coffee.

Johnny: Whatever, let's go.

Summary

This is how Johnny and Benjamin accomplished their first Statement using TDD and mock with an outside-in design approach. What will follow in the next chapter is a small retrospective with comments on what these guys did. One thing I'd like to mention now is that the outside-in approach does not rely solely on unit-level Statements, so what you saw here is not the full picture. We will get to that soon.

Test-driving at the input boundary – a retrospective

A lot of things happened in the last chapter and I feel some of them deserve a deeper dive. The purpose of this chapter is to do a small retrospective of what Johnny and Benjamin did when test-driving a controller-type class for the train reservation system.

Outside-in development

Johnny and Benjamin started their development almost entirely at the peripherals of the system, at the beginning of the flow of control. This is typical of the outside-in approach to software development. It took me a while to get used to it. To illustrate it, let's consider a system of three classes, where `Object3` depends on `Object2` and `Object2` depends on `Object1`:

```
Object3 -> Object2 -> Object1
```

Before adopting the outside-in approach, my habit was to start with the objects at the end of the dependency chain (which would typically be at the end of the control flow as well) because I had everything I needed to run and check them. Looking at the graph above, I could develop and run the logic in `Object1` because it did not require dependencies. Then, I could develop `Object2` because it depended on `Object1` that I already created and I could then do the same with `Object3` because it depended only on `Object2` that I already had. At any given time, I could run everything I created up to that point.

The outside-in approach contradicted this habit of mine, because the objects I had to start with were the ones that had to have dependencies and these dependencies did not exist yet. Looking at the example above, I would need to start with `Object3`, which could not be instantiated without `Object2`, which, in turn, could not be instantiated without `Object1`.

"If this feels difficult, then why bother?" you might ask. My reasons are:

1. By starting from the inputs and going inside, I allow my interfaces and protocols to be shaped by use cases rather than by the underlying technology. That does not mean I can always ignore the technology stuff, but I consider the use case logic to be the main driver. This way, my protocols tend to be more abstract which in turn enforces higher composability.
2. Every line of code I introduce is there because the use case needs it. Every method, every interface, and class exists because there already exists someone who needs it to perform its obligations. This way, I ensure I only implement the stuff that's needed and that it is shaped the way the users find it comfortable to use. Before adpoting this approach, I would start from the inside of the system and I would design classes by guessing how they would be used and I would later often regret these guesses, because of the rework and complexity they would create.

For me, these pros outweigh the cons. Moreover, I found I can mitigate the uncomfortable feeling of starting from the inputs ("there is nothing I can fully run") with the following practices:

1. Using TDD with mocks – TDD encourages every little piece of code to be executed well before the whole task completion. Mock objects serve as the first collaborators that allow this execution to happen. To take advantage of mocks, at least in C#, I need to use interfaces more liberally than in a typical development approach. A property of interfaces is that they typically don't contain implementation[117], so they can be used to cut the dependency chain. In other words, whereas without interfaces, I would need all three: `Object1`, `Object2` and `Object3` to instantiate and use `Object3`, I could alternatively introduce an interface that `Object3` would depend on and `Object2` would implement. This would allow me to use `Object3` before its concrete collaborators exist, simply by supplying a mock object as its dependency.
2. Slicing the scope of work into smaller vertical parts (e.g. scenarios, stories, etc.) that can be implemented faster than a full-blown feature. We had a taste of this in action when Johnny and Benjamin were developing the calculator in one of the first chapters of this book.
3. Not starting with a unit-level Statement, but instead, writing on a higher-level first (e.g. end-to-end or against another architectural boundary). I could then make this bigger Statement work, then refactor the initial objects out of this small piece of working code. Only after having this initial structure in place would I start using unit-level Statements with mocks. This approach is what I will be aiming at eventually, but for this and several furter chapters, I want to focus on the mocks and object-oriented design, so I left this part for later.

Workflow specification

The Statement about the controller is an example of what Amir Kolsky and Scott Bain call a workflow Statement[118]. This kind of Statement describes how a specified unit of behavior (in our case, an object) interacts with other units by sending messages and receiving answers. In Statements specifying workflow, we document the intended purpose and behaviors of the specified class in terms of its interaction with other roles in the system. We use mock objects to play these roles by specifying the return values of some methods and asserting that other methods are called.

For example, in the Statement Johnny and Benjamin wrote in the last chapter, they described how a command factory reacts when asked for a new command and they also asserted on the call to the command's `Execute()` method. That was a description of a workflow.

Should I verify that the factory got called?

You might have noticed in the same Statement that some interactions were verified (using the `.Received()` syntax) while some were only set up to return something. An example of the latter is a factory, e.g.

[117] though both C# and Java in their current versions allow putting logic in interfaces. Still, due to convention and some constraints, interfaces in those languages are still considered as beings without implementation.

[118] http://www.sustainabletdd.com/2012/02/testing-best-practices-test-categories.html

```
reservationInProgressFactory.FreshInstance().Returns(reservationInProgress);`
```

You may question why Johnny and Benjamin did not write something like `reservationInProgressFactory.Received().FreshInstance()` at the end.

One reason is that a factory resembles a function – it is not supposed to have any visible side-effects. As such, calling the factory is not a goal of the behavior I specify – it will always be only a mean to an end. For example, the goal of the behavior Johnny and Benjamin specified was to execute the command and return its result. The factory was brought to existence to make getting there easier.

Also, Johnny and Benjamin allowed the factory to be called many times in the implementation without altering the expected behavior in the Statement. For example, if the code of the `MakeReservation()` method they were test-driving did not look like this:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

but like this:

```
// Repeated multiple times:
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
reservationInProgress = _reservationInProgressFactory.FreshInstance();
reservationInProgress = _reservationInProgressFactory.FreshInstance();
reservationInProgress = _reservationInProgressFactory.FreshInstance();

var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

then the behavior of this method would still be correct. Sure, it would do some needless work, but when writing Statements, I care about externally visible behavior, not lines of production code. I leave more freedom to the implementation and try not to overspecify.

On the other hand, consider the command – it is supposed to have a side effect, because I expect it to alter some kind of reservation registry in the end. So if I sent the `Execute()` message more than once like this:

```
var reservationInProgress = _reservationInProgressFactory.FreshInstance();
var reservationCommand = _commandFactory.CreateNewReservationCommand(
  requestDto, reservationInProgress);
reservationCommand.Execute();
reservationCommand.Execute();
reservationCommand.Execute();
reservationCommand.Execute();
reservationCommand.Execute();
return reservationInProgress.ToDto();
```

then it could possibly alter the behavior – maybe by reserving more seats than the user requested, maybe by throwing an error from the second `Execute()`... This is why I want to strictly specify how many times the `Execute()` message should be sent:

```
reservationCommand.Received(1).Execute();
```

The approach to specifying functions and side-effects I described above is what Steve Freeman and Nat Pryce call "Allow queries; expect commands"[119]. According to their terminology, the factory is a "query" – side-effect-free logic returning some kind of result. The `ReservationCommand`, on the other side, is a "command" - not producing any kind of result, but causing a side-effect like a state change or an I/O operation.

Data Transfer Objects and TDD

While looking at the initial data structures, Johnny and Benjamin called them Data Transfer Objects.

A Data Transfer Object is a pattern to describe objects responsible for exchanging information between processes[120]. As processes cannot really exchange objects, the purpose of DTOs is to be serialized into some kind of data format and then transferred in that form to another process which deserializes the data. So, we can have DTOs representing output that our process sends out and DTOs representing input that our process receives.

As you might have seen, DTOs are typically just data structures. That may come as surprising, because for several chapters now, I have repeatedly stressed how I prefer to bundle data and behavior together. Isn't this breaking all the principles that I mentioned?

My response to this would be that exchanging information between processes is where the mentioned principles do not apply and that there are some good reasons why data is exchanged between processes.

1. It is easier to exchange data than to exchange behavior. If I wanted to send behavior to another process, I would have to send it as data anyway, e.g. in a form of source code. In such a case, the other side would need to interpret the source code, provide all the dependencies, etc. which could be cumbersome and strongly couple implementations of both processes.

[119] Steve Freeman, Nat Pryce, Growing Object Oriented Software Guided By Tests
[120] Patterns of Enterprise Application Architecture, Martin Fowler

2. Agreeing on a simple data format makes creating and interpreting the data in different programming languages easier.
3. Many times, the boundaries between processes are designed as functional boundaries at the same time. In other words, even if one process sends some data to another, both of these processes would not want to execute the same behavior on the data.

These are some of the reasons why processes send data to each other. And when they do, they typically bundle the data into bigger structures for consistency and performance reasons.

DTOs vs value objects

While DTOs, similarly to value objects, carry and represent data, their purpose and design constraints are different.

1. Values have value semantics, i.e. they can be compared based on their content. This is one of the core principles of their design. DTOs don't need to have value semantics (if I add value semantics to DTOs, I do it because I find it convenient for some reason, not because it's a part of the domain model).
2. DTOs must be easily serializable and deserializable from some kind of data exchange format (e.g. JSON or XML).
3. Values may contain behavior, even quite complex (an example of this would be the `Replace()` method of the `String` class), while DTOs typically contain no behavior at all.
4. Despite the previous point, DTOs may contain value objects, as long as these value objects can be reliably serialized and deserialized without loss of information. Value objects don't contain DTOs.
5. Values represent atomic and well-defined concepts (like text, date, money), while DTOs mostly function as bundles of data.

DTOs and mocks

As we observed in the example of Johnny and Benjamin writing their first Statement, they did not mock DTOs. It's a general rule – a DTO is a piece of data, it does not represent an implementation of an abstract protocol nor does it benefit from polymorphism the way objects do. Also, it is typically far easier to create an instance of a DTO than to mock it. Imagine we have the following DTO:

```
public class LoginDto
{
  public LoginDto(string login, string password)
  {
    Login = login;
    Password = password;
  }

  public string Login { get; }
  public string Password { get;}
}
```

An instance of this class can be created by typing:

```
var loginDto = new LoginDto("James", "007");
```

If we were to create a mock, we would probably need to extract an interface:

```
public class ILoginDto
{
  public string Login { get; }
  public string Password { get; }
}
```

and then write something like this in our Statement:

```
var loginDto = Substitute.For<ILoginDto>();
loginDto.Login.Returns("James");
loginDto.Password.Returns("Bond");
```

Not only is this more verbose, it also does not buy us anything. Hence my advice:

> Do not try to mock a DTO in your Statements. Create the real thing.

Creating DTOs in Statements

As DTOs tend to bundle data, creating them for specific Statements might be a chore as there might sometimes be several fields we would need to initialize in each Statement. How do I approach creating instances of DTOs to avoid this? I summarized my advice on dealing with this as the priority-ordered list below:

1. Limit the reach of your DTOs in the production code

As a rule of thumb, the fewer types and methods know about them, the better. DTOs represent an external application contract. They are also constrained by some rules mentioned earlier (like the ease of serialization), so they cannot evolve the same way normal objects do. Thus, I try to limit the number of objects that know about DTOs in my application to a necessary minimum. I use one of the two strategies: wrapping or mapping.

When wrapping, I have another object that holds a reference to the DTO and then all the other pieces of logic interact with this wrapping object instead of directly with a DTO:

```
var user = new User(userDto);
//...
user.Assign(resource);
```

I consider this approach simpler but more limited. I find that it encourages me to shape the domain objects similarly to how the DTOs are designed (because one object wraps one DTO). I usually start with this approach when the external contract is close enough to my domain model and move to the other strategy – mapping – when the relationship starts getting more complex.

When mapping, I unpack the DTO and pass specific parts into my domain objects:

```
var user = new User(
  userDto.Name,
  userDto.Surname,
  new Address(
    userDto.City,
    userDto.Street));
//...
user.Assign(resource);
```

This approach requires me to rewrite data into new objects field by field, but in exchange, it leaves me more room to shape my domain objects independent of the DTO structure[121]. In the example above, I was able to introduce an `Address` abstraction even though the DTO does not have an explicit field containing the address.

How does all of this help me avoid the tediousness of creating DTOs? Well, the fewer objects and methods know about a DTO, the fewer Statements will need to know about it as well, which leads to fewer places where I need to create and initialize one.

2. Use constrained non-determinism if you don't need specific data

In many Statements where I need to create DTOs, the specific values held inside it don't matter to me. I care only about *some* data being there. For situations like this, I like using constrained non-determinism. I can just create an anonymous instance and use it, which I find easier than assigning field by field.

As an example, look at the following line from the Statement Johnny and Benjamin wrote in the last chapter:

```
var requestDto = Any.Instance<ReservationRequestDto>();
```

In that Statement, they did not need to care about the exact values held by the DTO, so they just created an anonymous instance. In this particular case, using constrained non-determinism not only simplified the creation of the DTO, but it even allowed them to completely decouple the Statement from the DTO's structure.

[121] https://martinfowler.com/eaaCatalog/mapper.html

3. Use patterns such as factory methods or builders

When all else fails, I use factory methods and test data builders to ease the pain of creating DTOs to hide away the complexity and provide some good default values for the parts I don't care about.

A factory method can be useful if there is a single distinguishing factor about the particular instance that I want to create. For example:

```
public UserDto AnyUserWith(params Privilege[] privileges)
{
  var dto = Any.Instance<UserDto>()
    .WithPropertyValue(user => user.Privileges, privileges);
  return dto;
}
```

This method creates any user with a particular set of privileges. Note that I utilized constrained non-determinism as well in this method, which spared me some initialization code. If this is not possible, I try to come up with some sort of "safe default" values for each of the fields.

I like factory methods, but the more flexibility I need, the more I gravitate towards test data builders[122].

A test data builder is a special object that allows me to create an object with some default values, but allows me to customize the default recipe according to which the object is created. It leaves me much more flexibility with how I set up my DTOs. The typical syntax for using a builder that you can find on the internet[123] looks like this:

```
var user = new UserBuilder().WithName("Johnny").WithAge("43").Build();
```

Note that the value for each field is configured separately. Typically, the builder holds some kind of default values for the fields I don't specify:

```
//some safe default age will be used when not specified:
var user = new UserBuilder().WithName("Johnny").Build();
```

I am not showing an example implementation on purpose, because one of the further chapters will include a longer discussion of test data builders.

Using a `ReservationInProgress`

A controversial point of the design in the last chapter might be the usage of a `ReservationInProgress` class. The core idea of this abstraction is to collect the data needed to produce a result. To introduce this object, we needed a separate factory, which made the design more complex. Thus, some questions might pop into your mind:

[122] http://www.natpryce.com/articles/000714.html
[123] https://blog.ploeh.dk/2017/08/15/test-data-builders-in-c/

1. What exactly is `ReservationInProgress`?
2. Is the `ReservationInProgress` really necessary and if not, what are the alternatives?
3. Is a separate factory for `ReservationInProgress` needed?

Let's try answering them.

What exactly is `ReservationInProgress`?

As mentioned earlier, the intent for this object is to collect information about processing a command, so that the issuer of the command (in our case, a controller object) can act on that information (e.g. use it to create a response) when processing finishes. Speaking in patterns language, this is an implementation of a Collecting Parameter pattern[124].

There is something I often do, but I did not put in the example for the sake of simplicity. When I implement a collecting parameter, I typically make it implement two interfaces – one more narrow and the other one – wider. Let me show them to you:

```
public interface ReservationInProgress
{
   void Success(SomeData data);
   //... methods for reporting other events
}

public interface ReservationInProgressMakingReservationDto
  : ReservationInProgress
{
  ReservationDto ToDto();
}
```

The whole point is that only the issuer of the command can see the wider interface (`ReservationInProgressMakingReservationDto`) and when it passes this interface down the call chain, the next object only sees the methods for reporting events (`ReservationInProgress`). This way, the wider interface can even be tied to a specific technology, as long as the narrower one is not. For example, If I needed a JSON string response, I might do something like this:

```
public interface ReservationInProgressMakingReservationJson
  : ReservationInProgress
{
  string ToJsonString();
}
```

and only the command issuer (in our case, the controller object) would know about that. The rest of the classes using the narrower interface would interact with it happily without ever knowing that it is meant to produce a JSON output.

[124] https://wiki.c2.com/?CollectingParameter

Is `ReservationInProgress` necessary?

In short – no, although I find it useful. There are at least several alternative designs.

First of all, we might decide to return from the command's `Execute()` method. Then, the command would look like this:

```
public interface ReservationCommand
{
  public ReservationDto Execute();
}
```

This would do the job for the task at hand but would make the `ReservationCommand` break the command-query separation principle, which I like to uphold as much as I can. Also, the `ReservationCommand` interface would become much less reusable. If our application was to support different commands, each returning a different type of result, we could not have a single interface for all of them. This, in turn, would make it more difficult to decorate the commands using the decorator pattern. We might try to fix this by making the command generic:

```
public interface ReservationCommand<T>
{
  public T Execute();
}
```

but this still leaves a distinction between `void` and non-`void` commands (which some people resolve by parameterizing would-be `void` commands with `bool` and always returning `true` at the end).

The second option to avoid a collecting parameter would be to just let the command execute and then obtain the result by querying the state that was modified by the command (similarly to a command, a query may be a separate object). The code of the `MakeReservation()` would look somewhat like this:

```
var reservationId = _idGenerator.GenerateId();
var reservationCommand = _factory.CreateNewReservationCommand(
  requestDto, reservationId);
var reservationQuery = _factory.CreateReservationQuery(reservationId);

reservationCommand.Execute();
return reservationQuery.Make();
```

Note that in this case, there is nothing like "result in progress", but on the other hand, we need to generate the id for the command, as the query must use the same id. This approach might appeal to you if:

1. You don't mind that if you store your changes in a database or an external service, your logic might need to call it twice – once during the command, and again during the query.

2. The state that is modified by the command is queryable (i.e. a potential destination API for data allows executing both commands and queries on the data). It's not always a given.

There are more options, but I'd like to stop here as this is not the main concern of this book.

Do we need a separate factory for `ReservationInProgress`?

This question can be broken into two parts:

1. Can we use the same factory as for commands?
2. Do we need a factory at all?

The answer to the first one is: it depends on what the `ReservationInProgress` is coupled to. In this specific example, it is just creating a DTO to be returned to the client. In such a case, it does not necessarily need any knowledge of the framework that is being used to run the application. This lack of coupling to the framework would allow me to place creating `ReservationInProgress` in the same factory. However, if this class needed to decide e.g. HTTP status codes or create responses required by a specific framework or in a specified format (e.g. JSON or XML), then I would opt, as Johnny and Benjamin did, for separating it from the command factory. This is because the command factory belongs to the world of application logic and I want my application logic to be independent of the framework, the transport protocols, and the payload formats that I use.

The answer to the second question (whether we need a factory at all) is: it depends whether you care about specifying the controller behavior on the unit level. If yes, then it may be handy to have a factory to control the creation of `ReservationInProgress`. If you don't want to (e.g. you drive this logic with higher-level Specification, which we will talk about in one of the next parts), then you can decide to just create the object inside the controller method or even make the controller itself implement the `ReservationInProgress` interface (though if you did that, you would need to make sure a single controller instance is not shared between multiple requests, as it would contain mutable state).

Should I specify the controller behavior on the unit level?

As I mentioned, it's up to you. As controllers often serve as adapters between a framework and the application logic, they are constrained by the framework and so there is not that much value in driving their design with unit-level Statements. It might be more accurate to say that these Statements are closer to *integration Specification* because they often describe how the application is integrated into a framework.

An alternative to specifying the integration on the unit level could be driving it with higher-level Statements (which I will be talking about in detail in the coming parts of this book). For example, I might write a Statement describing how my application works end-to-end, then write a controller (or another framework adapter) code without a dedicated unit-level Statement for it and then use unit-level Statements to drive the application logic. When the end-to-end Statement passes, it means that the integration that my controller was meant to provide, works.

There are some preconditions for this approach to work, but I will cover them when talking about higher-level Statements in the further parts of this book.

Interface discovery and the sources of abstractions

As promised, the earlier chapter included some interface discovery, even though it wasn't a typical way of applying this approach. This is because, as I mentioned, the goal of the Statement was to describe integration between the framework and the application code. Still, writing the Statement allowed Johnny and Benjamin to discover what their controller needs from the application to pass all the necessary data to it and to get the result without violating the command-query separation principle.

The different abstractions Johnny pulled into existence were driven by:

- his need to convert to a different programming style (command-query separation principle, "tell, don't ask"…),
- his knowledge about design patterns,
- his experience with similar problems.

Had he lacked these things, he would have probably written the simplest thing that would come to his mind and then changed the design after learning more.

Do I need all of this to do TDD?

The last chapter might have left you confused. If you are wondering whether you need to use controllers, collecting parameters, commands and factories in order to do TDD, than my answer is *not necessarily*. Moreover, there are many debates on the internet whether these particular patterns are the right ones to use and even I don't like using controllers with problems such as this one (although I used it because this is what many web frameworks offer as a default choice).

I needed all of this stuff to create an example that resembles a real-life problem. The most important lesson though is that, while making the design decisions was up to Johnny and Benjamin's knowledge and experience, writing the Statement code before the implementation *informed* these design decisions, helping them distribute the responsibilities across collaborating objects.

What's next?

This chapter hopefully connected all the missing dots of the last one. The next chapter will focus on test-driving object creation.

Test-driving object creation

In this chapter, we join Johnny and Benjamin as they continue their outside-in TDD journey. In this chapter, they will be test-driving a factory.

Johnny: What's next on our TODO list?

Benjamin: There are two instances of `NotImplementedException`, both from factories that we discovered when implementing the last Statement.

The first factory is for creating instances of `ReservationInProgress` – remember? You gave it a bogus name on purpose. Look:

```
1  public class TodoReservationInProgressFactory : ReservationInProgressFactory
2  {
3    public ReservationInProgress FreshInstance()
4    {
5     throw new NotImplementedException();
6    }
7  }
```

Johnny: Oh, yeah, that one…

Benjamin: and the second factory is for creating commands – it's called `TicketOfficeCommandFactory`:

```
1  public class TicketOfficeCommandFactory : CommandFactory
2  {
3    public ReservationCommand CreateNewReservationCommand(
4      ReservationRequestDto requestDto,
5      ReservationInProgress reservationInProgress)
6    {
7     throw new NotImplementedException();
8    }
9  }
```

Johnny: Let's do this one.

Benjamin: Why this one?

Johnny: No big reason. I just want to dig into the domain logic as soon as possible.

Benjamin: Right. We don't have a Specification for this class, so let's create one.

```
public class TicketOfficeCommandFactorySpecification
{

}
```

Johnny: Go on, I think you will be able to write this Statement mostly without my help.

Benjamin: The beginning looks easy. I'll add a new Statement which I don't know yet what I'm going to call.

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateXXXXXXXXX() //TODO rename
 {
  Assert.True(false);
 }
}
```

Now, as you've told me before, I already know what class I am describing – it's `TicketOfficeCommandFactory`. It implements an interface that I discovered in the previous Statement, so it already has a method: `CreateNewReservationCommand`. So in the Statement, I'll just create an instance of the class and call the method, because why not?

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateXXXXXXXXX() //TODO rename
 {
  //GIVEN
  var factory = new TicketOfficeCommandFactory();

  //WHEN
  factory.CreateNewReservationCommand();

  //THEN
  Assert.True(false);
 }
}
```

This doesn't compile. To find out why, I look at the signature of the `CreateNewReservationCommand` method, and I see that it not only returns a command, but also accepts two arguments. I need to take that into account in the Statement:

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateXXXXXXXXX() //TODO rename
 {
  //GIVEN
  var factory = new TicketOfficeCommandFactory();

  //WHEN
  var command = factory.CreateNewReservationCommand(
    Any.Instance<ReservationRequestDto>(),
    Any.Instance<ReservationInProgress>());

  //THEN
  Assert.True(false);
 }
}
```

I used `Any.Instance<>()` to generate anonymous instances of both the DTO and the `ReservationInProgress`.

Johnny: That's exactly what I'd do. I'd assign them to variables only if I needed to use them somewhere else in the Statement.

Benjamin: The Statement compiles. I still have the assertion to write. What exactly should I assert? The command I'm getting back from the factory has only a `void Execute()` method. Everything else is hidden.

Johnny: Typically, there isn't much we can specify about the objects created by factories. In this case, we can only state the expected type of the command.

Benjamin: Wait, isn't this already in the signature of the factory? Looking at the creation method, I can already see that the returned type is `ReservationCommand`... wait!

Johnny: I thought you'd notice that. `ReservationCommand` is an interface. But an object needs to have a concrete type and this type is what we need to specify in this Statement. We don't have this type yet. We are on the verge of discovering it, and when we do, some new items will be added to our TODO list.

Benjamin: I'll call the class `NewReservationCommand` and modify my Statement to assert the type. Also, I think I know now how to name this Statement:

```
public class TicketOfficeCommandFactorySpecification
{
 [Fact]
 public void ShouldCreateNewReservationCommandWhenRequested()
 {
  //GIVEN
  var factory = new TicketOfficeCommandFactory();

  //WHEN
  var command = factory.CreateNewReservationCommand(
    Any.Instance<ReservationRequestDto>(),
    Any.Instance<ReservationInProgress>());

  //THEN
  Assert.IsType<NewReservationCommand>(command);
 }
}
```

Johnny: Now, let's see – the Statement looks like it's complete. You did it almost without my help.

Benjamin: Thanks. The code does not compile, though. The `NewReservationCommand` type does not exist.

Johnny: Let's introduce it then:

```
public class NewReservationCommand
{

}
```

Benjamin: Shouldn't it implement the `ReservationCommand` interface?

Johnny: We don't need to do that yet. The compiler only complained that the type doesn't exist.

Benjamin: The code compiles now, but the Statement is false because the `CreateNewReservationCommand` method throw a `NotImplementedException`:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   throw new NotImplementedException();
  }
```

Johnny: Remember what I told you the last time?

Benjamin: Yes, that the Statement needs to be false for the right reason.

Johnny: Exactly. `NotImplementedException` is not the right reason. I will change the above code to return null:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   return null;
  }
```

Now the Statement is false because this assertion fails:

```
Assert.IsType<NewReservationCommand>(command);
```

Benjamin: it complains that the command is null.

Johnny: So the right time came to put in the correct implementation:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
  {
   return new NewReservationCommand();
  }
```

Benjamin: But this doesn't compile – the `NewReservationCommand` doesn't implement the `ReservationCommand` interface, I told you this before.

Johnny: and the compiler forces us to implement this interface:

```
public class NewReservationCommand : ReservationCommand
{

}
```

The code still doesn't compile, because the interface has an `Execute()` method we need to implement in the `NewReservationCommand`. Any implementation will do as the logic of this method is outside the scope of the current Statement. We only need to make the compiler happy. Our IDE can generate the default method body for us. This should do:

```
public class NewReservationCommand : ReservationCommand
{
 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

The `NotImplementedException` will be added to our TODO list and we'll specify its behavior with another Statement later.

Benjamin: Our current Statement seems to be true now. I'm bothered by something, though. What about the arguments that we are passing to the `CreateNewReservationCommand` method? There are two:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto, //first argument
    ReservationInProgress reservationInProgress) //second argument
  {
   return new NewReservationCommand();
  }
```

and they are both unused. How come?

Johnny: We could've use them and pass them to the command. Sometimes I do this though the Statement does not depend on this, so there is no pressure. I chose not to do it this time to show you that we will need to pass them anyway when we specify the behaviors of our command.

Benjamin: let's say you are right and when we specify the command behavior, we will need to modify the factory to pass these two, e.g. like this:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto, //first argument
    ReservationInProgress reservationInProgress) //second argument
  {
   return new NewReservationCommand(requestDto, reservationInProgress);
  }
```

but no Statement says which values should be used, so I will as well be able to cheat by writing something like:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto, //first argument
    ReservationInProgress reservationInProgress) //second argument
  {
   return new NewReservationCommand(null, null);
  }
```

and all Statements will still be true.

Johnny: That's right. Though there are some techniques to specify that on unit level, I'd rather rely on higher-level Statements to guarantee we pass the correct values. I will show you how to do it another time.

Benjamin: Thanks, seems I'll just have to wait.

Johnny: Anyway it looks like we're done with this Statement, so let's take a short break.

Benjamin: Sure!

Test-driving object creation – a retrospective

In the last chapter, Johnny and Benjamin specified behavior of a factory, by writing a Specification Statement about object creation. This chapter will hopefully answer some questions you may have about what they did and what you should be doing in similar situations.

Limits of creation specification

Object creation might mean different things, but in the style of design I am describing, it is mostly about creating abstractions. These abstractions usually encapsulate a lot, exposing only narrow interfaces. An example might be the `ReservationCommand` from our train reservation system – it contains a single method called `Execute()`, that returns nothing. The only thing Johnny and Benjamin could specify about the created command in the Statement was its concrete type. They could not enforce the factory method arguments being passed into the created instance, so their implementation just left that away. Still, at some point they would need to pass the correct arguments or else the whole logic would not work.

To enforce the Statement to be false when correct parameters are not passed, they could try using techniques like reflection to inspect the created object graph and verify whether it contains the right objects, but that would soon turn the specification into a fragile mirror of the production code. The object composition is a mostly declarative definition of behavior, much the same as HTML is a declarative description of a GUI. We should specify behavior rather than the structure of what makes the behavior possible.

So how do we know whether the command was created exactly as we intended? Can we specify this at all?

I mentioned that the object composition is meant to provide some kind of higher-level behavior – one resulting from how the objects are composed. The higher-level Statements can describe that behavior. For example, when we specify the behavior of the whole component, These Statements will indirectly force us to get the object composition right – both in our factories and at the composition root level.

For now, I'll leave it at that, and I'll go over higher-level Specification in the further parts of the book.

Why specify object creation?

So if we rely on the higher-level Specification to enforce the correct object composition, why do we write a unit-level Specification for object creation? The main reason is: progress. Looking back at the Statement Johnny and Benjamin wrote – it forced them to create a class implementing

the `ReservationCommand` interface. This class then needed to implement the interface method in a way that would satisfy the compiler. Just to remind you, it ended up like this:

```
public class NewReservationCommand : ReservationCommand
{
 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

This way, they got a new class to test-drive, and the `NotImplementedException` that appeared in the generated `Execute` method, landed on their TODO list. As their next step, they could pick this TODO item and begin working on it.

So as you can see, a Statement specifying the factory's behavior, although not perfect as a test, allowed continuing the flow of TDD.

What do we specify in the creational Statements?

Johnny and Benjamin specified what should be the type of the created object. Indirectly, they also specified that the created object should not be a null. Is there anything else we might want to include in a creational specification?

Choice

Sometimes, factories make a decision on what kind of object to return. In such cases, we might want to specify the different choices that the factory can make. For example if a factory looks like this:

```
public class SomeKindOfFactory
{
 public Command CreateFrom(string commandName)
 {
  if(commandName == "command1")
  {
   return new Command1();
  }
  if(commandName == "command2")
  {
   return new Command2();
  }
  else
  {
   throw new InvalidCommandException(commandName);
```

```
16    }
17   }
18  }
```

then we would need three Statements to specify its behavior:

1. For the scenario where an instance of `Command1` is returned.
2. For the scenario where an instance of `Command2` is returned.
3. For the scenario where an exception is thrown.

Collection content

Likewise, if our factory needs to return a collection - we need to state what we expect it to contain. Consider the following Statement about a factory that needs to create a sequence of processing steps:

```
1  [Fact] public void
2  ShouldCreateProcessingStepsInTheRightOrder()
3  {
4   //GIVEN
5   var factory = new ProcessingStepsFactory();
6
7   //WHEN
8   var steps = factory.CreateStepsForNewItemRequest();
9
10  //THEN
11  Assert.Equal(3, steps.Count);
12  Assert.IsType<ValidationStep>(steps[0]);
13  Assert.IsType<ExecutionStep>(steps[1]);
14  Assert.IsType<ReportingStep>(steps[2]);
15 }
```

It says that the created collection should contain three items in a specific order and states the expected types of those items.

Last but not least, sometimes we specify the creation of value objects

Value object creation

Value objects are typically created using factory methods. Depending on whether the constructor of a such value object is public or not, we can accommodate these factory methods differently in our Specification.

Value object with a public constructor

When a public constructor is available, we can state the expected equality of an object created using a factory method to another object created using the constructor:

```
[Fact] public void
ShouldBeEqualToIdWithGroupType()
{
 //GIVEN
 var groupName = Any.String();

 //WHEN
 var id = EntityId.ForGroup(groupName);

 //THEN
 Assert.Equal(new EntityId(groupName, EntityTypes.Group), id);
}
```

Value object with a non-public constructor

When factory methods are the only means of creating value objects, we just use these methods in our Statements and either state the expected data or expected behaviors. For example, the following Statement says that an absolute directory path should become an absolute file path after appending a file name:

```
[Fact]
public void ShouldBecomeAnAbsoluteFilePathAfterAppendingFileName()
{
 //GIVEN
 var dirPath = AbsoluteDirectoryPath.Value("C:\\");
 var fileName = FileName.Value("file.txt");

 //WHEN
 AbsoluteFilePath absoluteFilePath = dirPath.Append(fileName);

 //THEN
 Assert.Equal(
     AbsoluteDirectoryPath.Value("C:\\file.txt"),
     absoluteFilePath);
}
```

Note that the Statement creates the values using their respective factory methods. This is inevitable as these methods are the only way the objects can be created.

Validation

The factory methods for value objects may have some additional behavior like validation – we need to specify it as well. For example, the Statement below says that an exception should be thrown when I try to create an absolute file path from a null string:

```
[Fact]
public void ShouldNotAllowToBeCreatedWithNullValue()
{
 Assert.Throws<ArgumentNullException>(() => AbsoluteFilePath.Value(null));
}
```

Summary

That was a quick ride through writing creational Statements. Though not thorough as tests, they create needs to new abstractions, allowing us to continue the TDD process. When we start our TDD process from higher-level Statements, we can defer specifying factories and sometimes even avoid it altogether.

Test-driving application logic

Johnny: What's next on our TODO list?

Benjamin: We have...

- a single reminder to change the name of the factory that creates `ReservationInProgress` instances.
- Also, we have two places where a `NotImplementedException` suggests there's something left to implement.
  - The first one is the mentioned factory.
  - The second one is the `NewReservationCommand` class that we discovered when writing a Statement for the factory.

Johnny: Let's do the `NewReservationCommand`. I suspect the complexity we put inside the controller will pay back here and we will be able to test-drive the command logic on our own terms.

Benjamin: Can't wait to see that. Here's the current code of the command:

```
public class NewReservationCommand : ReservationCommand
{
 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

Johnny: So we have the class and method signature. It seems we can use the usual strategy of writing new Statement.

Benjamin: You mean "start by invoking a method if you have one"? I thought of the same. Let me try it.

```
public class NewReservationCommandSpecification
{
 [Fact] public void
 ShouldXXXXXXXXXXX() //TODO change the name
 {
  //GIVEN
  var command = new NewReservationCommand();

  //WHEN
```

```
10    command.Execute();
11
12    //THEN
13    Assert.True(false); //TODO unfinished
14   }
15  }
```

This is what I could write almost brain-dead. I just took the parts we have and put them in the Statement.

The assertion is still missing - the one we have is merely a placeholder. This is the right time to think what should happen when we execute the command.

Johnny: To come up with that expected behavior, we need to look back to the input data for the whole use case. We passed it to the factory and forgot about it. So just as a reminder - here's how it looks like:

```
1  public class ReservationRequestDto
2  {
3    public readonly string TrainId;
4    public readonly uint SeatCount;
5
6    public ReservationRequestDto(string trainId, uint seatCount)
7    {
8      TrainId = trainId;
9      SeatCount = seatCount;
10   }
11 }
```

The first part is train id – it says on which train we should reserve the seats. So we need to somehow pick a train from the fleet for reservation. Then, on that train, we need to reserve as many seats as the customer requests. The requested seat count is the second part of the user request.

Benjamin: Aren't we going to update the data in some kind of persistent storage? I doubt that the railways company would want the reservation to disappear on application restart.

Johnny: Yes, we need to act as if there was some kind of persistence.

Given all of above, I can see two new roles in our scenario:

1. A fleet - from which we pick the train and where we save our changes
2. A train - which is going to handle the reservation logic.

Both of these roles need to be modeled as mocks, because I expect them to play active roles in this scenario.

Let's expand our Statement with our discoveries.

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var command = new NewReservationCommand(fleet);

 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount);
  fleet.UpdateInformationAbout(train);
 };
}
```

Benjamin: I can see there are many things missing here. For instance, we don't have the `train` and `fleet` variables.

Johnny: We created a need for them. I think we can safely introduce them into the Statement now.

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var fleet = Substitute.For<TrainFleet>();
 var train = Substitute.For<ReservableTrain>();
 var command = new NewReservationCommand(fleet);

 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount);
  fleet.UpdateInformationAbout(train);
 };
}
```

Benjamin: I see two new types: `TrainFleet` and `ReservableTrain`.

Johnny: They symbolize the roles we just discovered.

Benjamin: Also, I can see that you used `Received.InOrder()` from NSubstitute to specify the expected call order.

Johnny: That's because we need to reserve the seats before we update the information in some kind of storage. If we got the order wrong, the change could be lost.

Benjamin: But something is missing in this Statement. I just looked at the outputs our users expect:

```
public class ReservationDto
{
  public readonly string TrainId;
  public readonly string ReservationId;
  public readonly List<TicketDto> PerSeatTickets;

  public ReservationDto(
    string trainId,
    List<TicketDto> perSeatTickets,
    string reservationId)
  {
    TrainId = trainId;
    PerSeatTickets = perSeatTickets;
    ReservationId = reservationId;
  }
}
```

That's a lot of info we need to pass back to the user. How exactly are we going to do that when the `train.ReserveSeats(seatCount)` call you invented is not expected to return anything?

Johnny: Ah, yes, I almost forgot - we've got the `ReservationInProgress` instance that we passed to the factory, but not yet to the command, right? The `ReservationInProgress` was invented exactly for this purpose - to gather the information necessary to produce a result of the whole operation. Let me just quickly update the Statement:

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var fleet = Substitute.For<TrainFleet>();
 var train = Substitute.For<ReservableTrain>();
 var reservationInProgress = Any.Instance<ReservationInProgress>();
 var command = new NewReservationCommand(fleet, reservationInProgress);

 fleet.Pick(trainId).Returns(train);

 //WHEN
```

```
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount, reservationInProgress);
  fleet.UpdateInformationAbout(train);
 };
}
```

Now the `ReserveSeats` method accepts `reservationInProgress`.

Benjamin: Why are you passing the `reservationInProgress` further to the `ReserveSeats` method?

Johnny: The command does not have the necessary information to fill the `reservationInProgress` once the reservation is successful. We need to defer it to the `ReservableTrain` implementations to further decide the best place to do that.

Benjamin: I see. Looking at the Statement again - we're missing two more variables – `trainId` and `seatCount` – and not only their definitions, but also we don't pass them to the command at all. They are only present in our assumptions and expectations.

Johnny: Right, let me correct that.

```
[Fact] public void
ShouldReserveSeatsInSpecifiedTrainWhenExecuted()
{
 //GIVEN
 var fleet = Substitute.For<TrainFleet>();
 var train = Substitute.For<ReservableTrain>();
 var trainId = Any.String();
 var seatCount = Any.UnsignedInt();
 var reservationInProgress = Any.Instance<ReservationInProgress>();
 var command = new NewReservationCommand(
   trainId,
   seatCount,
   fleet,
   reservationInProgress);

 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
```

```
24    train.ReserveSeats(seatCount, reservationInProgress);
25    fleet.UpdateInformationAbout(train);
26   };
27  }
```

Benjamin: Why is `seatCount` a `uint`?

Johnny: Look it up in the DTO - it's a `uint` there. I don't see the need to redefine that here.

Benjamin: Fine, but what about the `trainId` - it's a `string`. Didn't you tell me we need to use domain-related value objects for concepts like this?

Johnny: Yes, and we will refactor this `string` into a value object, especially that we have a requirement that train id comparisons should be case-insensitive. But first, I want to finish this Statement before I go into defining and specifying a new type. Still, we'd best leave a TODO note to get back to it later:

```
1   var trainId = Any.String(); //TODO extract value object
```

So far so good, I think we have a complete Statement. Want to take the keyboard?

Benjamin: Thanks. Let's start implementing it then. First, I will start with these two interfaces:

```
1  var fleet = Substitute.For<TrainFleet>();
2  var train = Substitute.For<ReservableTrain>();
```

They don't exist, so this code doesn't compile. I can easily fix this by creating the interfaces in the production code:

```
1  public interface TrainFleet
2  {
3  }
4
5  public interface ReservableTrain
6  {
7  }
```

Now for this part:

```
1  var command = new NewReservationCommand(
2     trainId,
3     seatCount,
4     fleet,
5     reservationInProgress);
```

It doesn't compile because the command does not accept any constructor parameters yet. Let's create a fitting constructor, then:

```
public class NewReservationCommand : ReservationCommand
{
 public NewReservationCommand(
  string trainId,
  uint seatCount,
  TrainFleet fleet,
  ReservationInProgress reservationInProgress)
 {

 }

 public void Execute()
 {
  throw new NotImplementedException();
 }
}
```

Our Statement can now invoke this constructor, but we broke the `TicketOfficeCommandFactory` which also creates a `NewReservationCommand` instance. Look:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
 {
  //Stopped compiling:
  return new NewReservationCommand();
 }
```

Johnny: We need to fix the factory the same way we needed to fix the composition root when test-driving the controller. Let's see... Here:

```
public ReservationCommand CreateNewReservationCommand(
    ReservationRequestDto requestDto,
    ReservationInProgress reservationInProgress)
 {
  return new NewReservationCommand(
   requestDto.TrainId,
   requestDto.SeatCount,
   new TodoTrainFleet(), // TODO fix name and scope
   reservatonInProgress
   );
 }
```

Benjamin: The parameter passing looks straightforward to me except the `TodoTrainFleet()` – I already know that the name is a placeholder - you already did something like that earlier. But what about the lifetime scope?

Johnny: It's also a placeholder. For now, I want to make the compiler happy, at the same time keeping existing Statements true and introducing a new class – `TodoTrainFleet` – that will bring new items to our TODO list.

Benjamin: New TODO items?

Johnny: Yes. Look – the type `TodoTrainFleet` does not exist yet. I'll create it now:

```
public class TodoTrainFleet
{

}
```

This doesn't match the signature of the command constructor, which expects a `TrainFleet`, so I need to make `TodoTrainFleet` implement this interface:

```
public class TodoTrainFleet : TrainFleet
{

}
```

Now I am forced to implement the methods from the `TrainFleet` interface. Although this interface doesn't define any methods yet, we already discovered two in our Statement, so it will shortly need to get them to make the compiler happy. They will both contain code throwing `NotImplementedException`, which will land on the TODO list.

Benjamin: I see. Anyway, the factory compiles again. We still got this part of the Statement left:

```
 fleet.Pick(trainId).Returns(train);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  train.ReserveSeats(seatCount, reservationInProgress);
  fleet.UpdateInformationAbout(train);
 };
```

Johnny: That's just introducing three methods. You can handle it.

Benjamin: Thanks. The first line is `fleet.Pick(trainId).Returns(train)`. I'll just generate the `Pick` method using my IDE:

```
public interface TrainFleet
{
 ReservableTrain Pick(string trainId);
}
```

The `TrainFleet` interface is implemented by the `TodoTrainFleet` we talked about several minutes ago. It needs to implement the `Pick` method as well or else it won't compile:

```
public class TodoTrainFleet : TrainFleet
{
 public ReservableTrain Pick(string trainId)
 {
   throw new NotImplementedException();
 }
}
```

This `NotImplementedException` will land on our TODO list just as you mentioned. Nice!

Then comes the next line from the Statement: `train.ReserveSeats(seatCount, reservationInProgress)` and I'll generate a method signature out of it the same as from the previous line.

```
public interface ReservableTrain
{
 void ReserveSeats(uint seatCount, ReservationInProgress reservationInProgress);
}
```

`ReservableTrain` interface doesn't have any implementations so far, so nothing more to do with this method.

The last line: `fleet.UpdateInformationAbout(train)` which needs to be added to the `TrainFleet` interface:

```
public interface TrainFleet
{
 ReservableTrain Pick(string trainId);
 void UpdateInformationAbout(ReservableTrain train);
}
```

Also, we need to define this method in the `TodoTrainFleet` class:

```
public class TodoTrainFleet : TrainFleet
{
 public ReservableTrain Pick(string trainId)
 {
   throw new NotImplementedException();
 }

 void UpdateInformationAbout(ReservableTrain train)
 {
   throw new NotImplementedException();
 }
}
```

Johnny: This `NotImplementedException` will be added to the TODO list as well, so we can revisit it later. It looks like the Statement compiles and, as expected, is false, but not for the right reason.

Benjamin: Let me see... yes, a `NotImplementedException` is thrown from the command's `Execute()` method.

Johnny: Let's get rid of it.

Benjamin: Sure. I removed the `throw` and the method is empty now:

```
 public void Execute()
 {

 }
```

The Statement is false now because the expected calls are not matched.

Johnny: Which means we are finally ready to code some behavior into the `NewReservationCommand` class. First, let's assign all the constructor parameters to fields – we're going to need them.

Benjamin: Here:

```
public class NewReservationCommand : ReservationCommand
{
 private readonly string _trainId;
 private readonly uint _seatCount;
 private readonly TraingFleet _fleet;
 private readonly ReservationInProgress _reservationInProgress;

 public NewReservationCommand(
  string trainId,
  uint seatCount,
  TrainFleet fleet,
  ReservationInProgress reservationInProgress)
 {
  _trainId = trainId;
```

```
15    _seatCount = seatCount;
16    _fleet = fleet;
17    _reservationInProgress = reservationInProgress;
18   }
19
20   public void Execute()
21   {
22    throw new NotImplementedException();
23   }
24  }
```

Johnny: Now, let's add the calls expected in the Statement, but in the opposite order.

Benjamin: To make sure the order is asserted correctly in the Statement?

Johnny: Exactly.

Benjamin: Ok.

```
1  public void Execute()
2  {
3   var train = _fleet.Pick(_trainId);
4   _fleet.UpdateInformationAbout(train);
5   train.ReserveSeats(seatCount);
6  }
```

The Statement is still false, this time because of the wrong call order. Now that we have confirmed that we need to make the calls in the right order, I suspect you want me to reverse it, so...

```
1  public void Execute()
2  {
3   var train = _fleet.Pick(_trainId);
4   train.ReserveSeats(seatCount, reservationInProgress);
5    _fleet.UpdateInformationAbout(train);
6  }
```

Johnny: Exactly. The Statement is now true. Congratulations!

Benjamin: Now that I look at this code, it's not protected from any kind of exceptions that might be thrown from either the `_fleet` or the `train`.

Johnny: Add that to the TODO list - we will have to take care of that, sooner or later. For now, let's take a break.

Summary

In this chapter, Johnny and Benjamin used interface discovery again. They used some technical and some domain-related reasons to create a need for new abstractions and design their communication protocols. These abstractions were then pulled into the Statement.

Remember Johnny and Benjamin extended effort when test-driving the controller. This effort paid off now - they were free to shape abstractions mostly outside the constraints imposed by a specific framework.

This chapter does not have a retrospective companion chapter like the previous ones. Most of the interesting stuff that happened here was already explained earlier.

Test-driving value objects

In this chapter, we skip further ahead in time. Johnny and Benjamin just extracted a value object type for a train ID and started the work to further specify it.

Initial value object

Johnny: Oh, you're back. The refactoring is over – we've got a nice value object type extracted from the current code. Here's the source code of the `TrainId` class:

```
public class TrainId
{
 private readonly string _value;

 public static TrainId From(string trainIdAsString)
 {
  return new TrainId(trainIdAsString);
 }

 private TrainId(string value)
 {
  _value = value;
 }

 public override bool Equals(object obj)
 {
  return _value == ((TrainId) obj)._value;
 }

 public override string ToString()
 {
  return _value;
 }
}
```

Benjamin: Wait, but we don't have any Specification for this class yet. Where did all of this implementation come from?

Johnny: That's because while you were drinking your tea, I extracted this type from an existing implementation that was already a response to false Statements.

Benjamin: I see. So we didn't mock the `TrainId` class in other Statements, right?

Johnny: No. This is a general rule – we don't mock value objects. They don't represent abstract, polymorphic behaviors. For the same reasons we don't create interfaces and mocks for `string` or `int`, we don't do it for `TrainId`.

Value semantics

Benjamin: So, given the existing implementation, is there anything left for us to write?

Johnny: Yes. I decided that this `TrainId` should be a value object and my design principles for value objects demand it to provide some more guarantees than the ones resulting from a mere refactoring. Also, don't forget that the comparison of train ids needs to be case-insensitive. This is something we've not specified anywhere.

Benjamin: You mentioned "more guarantees". Do you mean equality?

Johnny: Yes, C# as a language expects equality[125] to follow certain rules. I want this class to implement them. Also, I want it to implement `GetHashCode` properly[126] and make sure instances of this class are immutable. Last but not least, I'd like the factory method `From` to throw an exception when null or empty string are passed. Neither of these inputs should lead to creating a valid ID.

Benjamin: Sounds like a lot of Statements to write.

Johnny: Not really. Add the validation and lowercase comparison to TODO list. In the meantime – look – I downloaded a library that will help me with the immutability and equality part. This way, all I need to write is:

```
[Fact] public void
ShouldHaveValueObjectSemantics()
{
 var trainIdString = Any.String();
 var otherTrainIdString = Any.OtherThan(trainIdString);
 Assert.HasValueSemantics<TrainId>(
   new Func<TrainId>[]
   {
    () => TrainId.From(trainIdString)
   },
   new Func<TrainId>[]
   {
    () => TrainId.From(otherTrainIdString);
   },
 );
}
```

This assertion accepts two arrays:

[125] https://docs.microsoft.com/en-us/dotnet/api/system.object.equals
[126] https://docs.microsoft.com/en-us/dotnet/api/system.object.gethashcode

- the first array contains factory functions that create objects that should be equal to each other. For now, we only have a single example, because I didn't touch the lowercase vs uppercase issue. But when I do, the array will contain more entries to stress that ids created from the same string with different letter casing should be considered equal.
- the second array contains factory functions that create example objects which should be considered not equal to any of the objects generated by the "equal" factory functions. There is also a single example here as `TrainId`'s `From` method has a single argument, so the only way one instance can differ from another is by being created with a different value of this argument.

After evaluating this Statement, I get the following output:

```
- TrainId must be sealed, or derivatives will be able to override GetHashCode() w\
ith mutable code.

- a.GetHashCode() and b.GetHashCode() should return same values for equal objects.

- a.Equals(null) should return false, but instead threw System.NullReferenceExcep\
tion: Object reference not set to an instance of an object.

- '==' and '!=' operators are not implemented
```

Benjamin: Very clever. So these are the rules that our `TrainId` doesn't follow yet. Are you going to implement them one by one?

Johnny: Hehe, no. The implementation would be so dull that I'd either use my IDE to generate the necessary implementation or, again, use a library. Lately, I prefer the latter. So let me just download a library called Value[127] and use it on our `TrainId`.

First off, the `TrainId` needs to inherit from `ValueType` generic class like this:

```
public class TrainId : ValueType<TrainId>
```

This inheritance requires us to implement the following "special" method:

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 throw new NotImplementedException();
}
```

Benjamin: Weird, what does that do?

Johnny: It's how the library automates equality implementation. We just need to return an array of values we want to be compared between two instances. As our equality is based solely on the `_value` field, we need to return just that:

[127] https://www.nuget.org/packages/Value/

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 yield return _value;
}
```

We also need to remove the existing Equals() method.

Benjamin: Great, now the only reason for the assertion to fail is:

```
- TrainId must be sealed, or derivatives will be able to override GetHashCode() w\
ith mutable code.
```

I am impressed that this Value[128] library took care of the equality methods, equality operators, and `GetHashCode()`.

Johnny: Nice, huh? Ok, let's end this part and add the `sealed` keyword. The complete source code of the class looks like this:

```
public sealed class TrainId : ValueType<TrainId>
{
 private readonly string _value;

 public static TrainId From(string trainIdAsString)
 {
  return new TrainId(trainIdAsString);
 }

 private TrainId(string value)
 {
  _value = value;
 }

 protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
 {
  yield return _value;
 }

 public override string ToString()
 {
  return _value;
 }
}
```

Benjamin: And the Statement is true.

[128] https://www.nuget.org/packages/Value/

Case-insensitive comparison

Johnny: What's left on our TODO list?

Benjamin: Two items:

- Comparison of train ids should be case-insensitive.
- `null` and empty string should not be allowed as valid train ids.

Johnny: Let's do the case-insensitive comparison, it should be relatively straightforward.

Benjamin: Ok, you mentioned that we would need to expand the Statement you wrote, by adding another "equal value factory" to it. Let me try. What do you think about this?

```
[Fact] public void
ShouldHaveValueSemantics()
{
 var trainIdString = Any.String();
 var otherTrainIdString = Any.OtherThan(trainIdString);
 Assert.HasValueSemantics<TrainId>(
   new Func<TrainId>[]
   {
    () => TrainId.From(trainIdString.ToUpper())
    () => TrainId.From(trainIdString.ToLower())
   },
   new Func<TrainId>[]
   {
    () => TrainId.From(otherTrainIdString);
   },
 );
}
```

How about that? From what you explained to me, I understand that by adding a second factory to the first array, I say that both instances should be treated as equal - the one with lowercase string and the one with uppercase string.

Johnny: Exactly. Now, let's make the Statement true. Fortunately, we can do this by changing the `GetAllAttributesToBeUsedForEquality` method from:

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 yield return _value;
}
```

to:

```
protected override IEnumerable<object> GetAllAttributesToBeUsedForEquality
{
 yield return _value.ToLower();
}
```

Aaand done! The assertion checked `Equals`, equality operators and `GetHashCode()` and everything seems to be working. We can move on to the next item on our TODO list.

Input validation

Johnny: Let's take care of the `From` method - it should disallow `null` input – we expect an exception when we pass a `null` inside.

For now, the method looks like this:

```
public static TrainId From(string trainIdAsString)
{
 return new TrainId(trainIdAsString);
}
```

Benjamin: OK, let me write a Statement about the expected behavior:

```
[Fact] public void
ShouldThrowWhenCreatedWithANullInput()
{
  Assert.Throws<ArgumentNullException>(() => TrainId.From(null));
}
```

That was easy, huh?

Johnny: Thanks. The Statement is currently false because it expects an exception but nothing is thrown. Let's make it true by implementing the `null` check.

```
public static TrainId From(string trainIdAsString)
{
 if(trainIdAsString == null)
 {
  throw new ArgumentNullException(nameof(trainIdAsString));
 }
 return new TrainId(trainIdAsString);
}
```

Benjamin: Great, it worked!

Summary

Johnny and Benjamin have one more behavior left to specify – throwing an exception when empty string is passed to the factory method – but following them further won't probably bring us any new insights. Thus, I'd like to close this chapter. Before I do, several points of summary of what to remember about when test-driving value objects:

1. Value objects are often (though not always) refactored retroactively from existing code. In such a case, they will already have some coverage from specifications of classes that use these value objects. You don't need to specify the covered behaviors again in the value object specification. Though I almost always do it, Johnny and Benjamin chose not to and I respect their decision.
2. Even if we choose not to write Statements for behaviors already covered by existing specification, we still need to write additional Statements to ensure a type is a proper value object. These Statements are not driven by existing logic, but by our design principles.
3. There are many conditions that apply to equality and hash codes of value objects. Instead of writing tests for these behaviors for every value object, I advise to use a generic custom assertion for this. Either find a library that already contains such an assertion, or write your own.
4. When implementing equality and hash code methods, I, too, advise to strongly consider using some kind of helper library or at least generating them using an IDE feature.
5. We don't ever mock value objects in Specifications of other classes. In these Specifications, we use the value objects in the same way as `int`s and `string`s.

Reaching the web of objects boundaries

When doing the outside-in interface discovery and implementing collaborator after collaborator, there's often a point when we reach the boundaries of our application, which means we must execute some kind of I/O operation (like calling external API via HTTP) or use a class that is part of some kind of third-party package (e.g. provided by our framework of choice). In these places, the freedom with which we could shape our object-oriented reality is hugely constrained by these dependencies. Even though we still drive the behaviors using our intention, we need to be increasingly aware that the direction we take has to match the technology with which we communicate with the outside world at the end.

There are multiple examples of resources that lay on the boundaries of our web of objects. Files, threads, clock, database, communication channels (e.g. http, service bus, websockets), graphical user interfaces… these resources are used to implement mechanisms for things that play roles in our design. We need to somehow reach a consensus between what these implementation mechanisms require to do their job and what is the domain role they need to fit.

In this chapter, I describe several examples of reaching this consensus with some guidance of how to approach them. The gist of this guidance is to hide such dependencies below interfaces that model the roles we need.

What time is it?

Consider getting the current time.

Imagine our application manages sessions that expire at one point. We model the concept of a session by creating a class called `Session` and allow querying it whether it's expired or not. To answer that question, the session needs to know its expiry time and the current time, and calculate their difference. In such a case, we can model the source of current time as some kind of "clock" abstraction. Here are several examples of how to do that.

A query for the current time with a mock framework

We can model the clock as merely a service that delivers current time (e.g. via some kind of `.CurrentTime()` method) The `Session` class is thus responsible for doing the calculation. An example Statement describing this behavior could look like this:

```
[Fact] public void
ShouldSayItIsExpiredWhenItsPastItsExpiryDate()
{
 //GIVEN
 var expiryDate = Any.DateTime();
 var clock = Substitute.For<Clock>();
 var session = new Session(expiryDate, clock);

 clock.CurrentTime().Returns(expiryDate + TimeSpan.FromTicks(1));

 //WHEN
 var isExpired = session.IsExpired();

 //THEN
 Assert.True(isExpired);
}
```

I stubbed my clock to return a specific point in time to make executing the Statement deterministic and to describe a specific behavior, which is indication of session expiry.

More responsibility allocation in the clock abstraction

Our previous attempt at the clock abstraction was just to abstract away the services granted by a real time source ("what's the current time?"). We can make this abstraction fit our specific use case better by giving it a `IsPast()` method that will accept an expiry time and just tell us whether it's past that time or not. One of the Statements using the `IsPast()` method could look like this:

```
[Fact] public void
ShouldSayItIsExpiredWhenItsPastItsExpiryDate()
{
 //GIVEN
 var expiryDate = Any.DateTime();
 var clock = Substitute.For<Clock>();
 var session = new Session(expiryDate, clock);

 clock.IsPast(expiryDate).Returns(true);

 //WHEN
 var isExpired = session.IsExpired();

 //THEN
 Assert.True(isExpired);
}
```

This time, I am also using a mock to stand in for a `Clock` implementation.

The upside of this is that the abstraction is more tuned to our specific need. The downside is that the calculation of the time difference between current and expiry time is done in a class that will be at least very difficult to describe with a deterministic Statement as it will rely on the system clock.

A fake clock

If the services provided by an interface such as `Clock` are simple enough, we can create our own implementation for use in Statements. Such implementation would take away all the pain of working with a real external resource and instead give us a simplified or more controllable version. For example, a fake implementation of the `Clock` interface might look like this:

```
class SettableClock : Clock
{
 public DateTime _currentTime = DateTime.MinValue;

 public void SetCurrentTime(DateTime currentTime)
 {
  _currentTime = currentTime;
 }

 public DateTime CurrentTime()
 {
  return _currentTime;
 }
}
```

This implementation of `Clock` has not only a getter for the current time (which is inherited from the `Clock` interface), but it also has a setter. The "current time" as returned by the `SettableClock` instance does not come from system clock, but can instead be set to a specific, deterministic value. Here's how the use of `SettableClock` looks like in a Statement:

```
[Fact] public void
ShouldSayItIsExpiredWhenItsPastItsExpiryDate()
{
 //GIVEN
 var expiryDate = Any.DateTime();
 var clock = new SettableClock();
 var session = new Session(expiryDate, clock);

 clock.SetCurrentTime(expiryDate + TimeSpan.FromSeconds(1));

 //WHEN
 var isExpired = session.IsExpired();

 //THEN
```

```
15  Assert.True(isExpired);
16 }
```

I could also do the second version of the clock - the one with `IsPast()` method - in a very similar way. I would just need to put some extra intelligence into the `SettableClock`, duplicating tiny bits of real `Clock` implementation. In this case, it's not a big issue, but there can be cases when this can be an overkill. For a fake to be warranted, the fake implementation must be much, much simpler than the real implementation that we intend to use.

An advantage of a fake is that it can be reused (e.g. a reusable settable clock abstraction can be found in Noda Time and Joda Time libraries). A disadvantage is that the more sophisticated code sits inside the fake, the more probable it is that it will contain bugs. If a very intelligent fake is still worth it despite the complexity (and I've seen several cases when it was), I'd consider writing some Specification around the fake implementation.

Other usages of this approach

The same approach as we used with a system clock can be taken with other sources of non-deterministic values, e.g. random value generators.

Timers

Timers are objects that allow deferred or periodic code execution, usually in an asynchronous manner.

In C# these days, almost everything asynchronous is done using `Task` class, and `async-await` keywords. Despite that, I decided to leave them out of this chapter to make it better understandable to non-C# users. So forgive me for a little less realistic example and I hope you can map that to your modern code.

Typically, a timer usage is composed of three stages:

1. Scheduling the timer expiry for a specific point in time, passing it a callback to execute on expiry. Usually a timer can be told whether to continue the next cycle after the current one is finished, or stop.
2. When the timer expiry is scheduled, it runs asynchronously in some kind of background thread.
3. When the timer expires, the callback that was passed during scheduling is executed and the timer either begins another cycle, or stops.

From the point of view of our collaboration design, the important parts are point 1 and 3. Let's tackle them one by one.

Scheduling a periodic task

First let's imagine we create a new session, add it to some kind of cache and set a timer expiring every 10 seconds to check whether privileges of the session owner are still valid. A Statement for that might look like this:

```
[Fact] public void
ShouldAddCreatedSessionToCacheAndScheduleItsPeriodicPrivilegesRefreshWhenExecuted\
()
{
 //GIVEN
 var sessionData = Any.Instance<SessionData>();
 var id = Any.Instance<SessionId>();
 var session = Any.Instance<Session>();
 var sessionFactory = Substitute.For<SessionFactory>();
 var cache = Substitute.For<SessionCache>();
 var periodicTasks = Substitute.For<PeriodicTasks>();
 var command = new CreateSessionCommand(id, sessionFactory, cache, sessionData);

 sessionFactory.CreateNewSessionFrom(sessionData).Returns(session);

 //WHEN
 command.Execute();

 //THEN
 Received.InOrder(() =>
 {
  cache.Add(id, session);
  periodicTasks.RunEvery(TimeSpan.FromSeconds(10), session.RefreshPrivileges);
 });
}
```

Note that I created a `PeriodicTasks` interface to model an abstraction for running... well... periodic tasks. This seems like a generic abstraction and might be made a bit more domain-oriented if needed. For our toy example, it should do. The `PeriodicTask` interface looks like this:

```
interface PeriodicTasks
{
 void RunEvery(TimeSpan period, Action actionRanOnExpiry);
}
```

In the Statement above, I only specified that a periodic operation should be scheduled. Specifying how the `PeriodicTasks` implementation carries out its job is out of scope.

The `PeriodicTasks` interface is designed so that I can pass a method group instead of a lambda, because requiring lambdas would make it harder to compare arguments between expected and actual invocations in the Statement. So if I wanted to schedule a periodic invocation of a method that has an argument (say, a single `int`), I would add a `RunEvery` method that would look like this:

```
interface PeriodicTasks
{
 void RunEvery(TimeSpan period, Action<int> actionRanOnExpiry, int argument);
}
```

or I could make the method generic:

```
interface PeriodicTasks
{
 void RunEvery<TArg>(TimeSpan period, Action<TArg> actionRanOnExpiry, TArg argume\
nt);
}
```

If I needed different sets of arguments, I could just add more methods to the `PeriodicTasks` interface, provided I don't couple it to any specific domain-related class (if I needed that, I'd rather split `PeriodicTasks` into several more domain-specific interfaces).

Expiry

Specifying a case when the timer expires and the scheduled task needs to be executed is even easier. We just need to do what the timer code would do - invoke the scheduled code from our Statement. For my session example, assuming that an implementation of the `Session` interface is a class called `UserSession`, I could the following Statement to describe a behavior where losing privileges to access session content leads to event being generated:

```
[Fact] public void
ShouldNotifyObserverThatSessionIsClosedOnPrivilegesRefreshWhenUserLosesAccessToSe\
ssionContent()
{
 //GIVEN
 var user = Substitute.For<User>();
 var sessionContent = Any.Instance<SessionContent>();
 var sessionEventObserver = Substitute.For<SessionEventObserver>();
 var id = Any.Instance<SessionId>();
 var session = new UserSession(id, sessionContent, user, sessionEventObserver);

 user.HasAccessTo(sessionContent).Returns(false);

 //WHEN
 session.RefreshPrivileges();

 //THEN
 sessionEventObserver.Received(1).OnSessionClosed(id);
}
```

For the purpose of this example, I am skipping other Statements (for example you might want to specify a behavior when the `RefreshPrivileges` is called multiple times as that's what the timer is ultimately going to do) and multithreaded access (timer callbacks are typically executed from other threads, so if they access mutable state, this state must be protected from concurrent modification).

Threads

I usually see threads used in two situations:

1. To run several tasks in parallel. For example, we have several independent, heavy calculations and we want to do them at the same time (not one after another) to make the execution of the code finish earlier.
2. To defer execution of some logic to an asynchronous background job. This job can still be running even after the method that started it finishes execution.

Parallel execution

In the first case – one of parallel execution – multithreading is an implementation detail of the method being called by the Statement. The Statement itself doesn't have to know anything about it. Sometimes, though, it needs to know that certain operations might not execute in the same order every time.

Consider the an example, where we evaluate payment for multiple employees. Each evaluation is a costly operation, so implementation-wise, we want to do them in parallel. A Statement describing such operation could look like this:

```
public void [Fact]
ShouldEvaluatePaymentForAllEmployees()
{
 //GIVEN
 var employee1 = Substitute.For<Employee>();
 var employee2 = Substitute.For<Employee>();
 var employee3 = Substitute.For<Employee>();
 var employees = new Employees(employee1, employee2, employee3);

 //WHEN
 employees.EvaluatePayment();

 //THEN
 employee1.Received(1).EvaluatePayment();
 employee2.Received(1).EvaluatePayment();
 employee3.Received(1).EvaluatePayment();
}
```

Note that the Statement doesn't mention multithreading at all, because that's the implementation detail of the `EvaluatePayment` method on `Employees`. However, note also that the Statement doesn't specify the order in which the payment is evaluated for each employee. Part of that is because the order doesn't matter and the Statement accurately describes that. If, however, the Statement specified the order, it would not only be overspecified, but also could be evaluated as true or false non-deterministically. That's because in the case of parallel execution, the order in which the methods would be called on each employee could be different.

Background task

Using threads to run background tasks is like using timers that run only once. Just use a similar approach to timers.

Others

There are other dependencies like I/O devices, random value generators or third-party SDKs (e.g. an SDK for connecting to a message bus), but I won't go into them as the strategy for them is the same - don't use them directly in your domain-related code. Instead, think about what role they play domain-wise and model that role as an interface. Then use the problematic resource inside a class implementing this interface. Such class will be covered by another kind of Specification which I will cover further in the book.

What's inside the object?

What are object's peers?

So far I talked a lot about objects being composed in a web and communicating by sending messages to each other. They work together as *peers*.

The name peer comes from the objects working on equal footing – there is no hierarchical relationship between peers. When two peers collaborate, none of them can be called an "owner" of the other. They are connected by one object receiving a reference to another object - its "address".

Here's an example

```
1  class Sender
2  {
3   public Sender(Recipient recipient)
4   {
5    _recipient = recipient;
6    //...
7   }
8  }
```

In this example, the `Recipient` is a peer of `Sender` (provided `Recipient` is not a value object or a data structure). The sender doesn't know:

- the concrete type of `recipient`
- when `recipient` was created
- by whom `recipient` was created
- how long will `recipient` object live after the `Sender` instance is destroyed
- which other objects collaborate with `recipient`

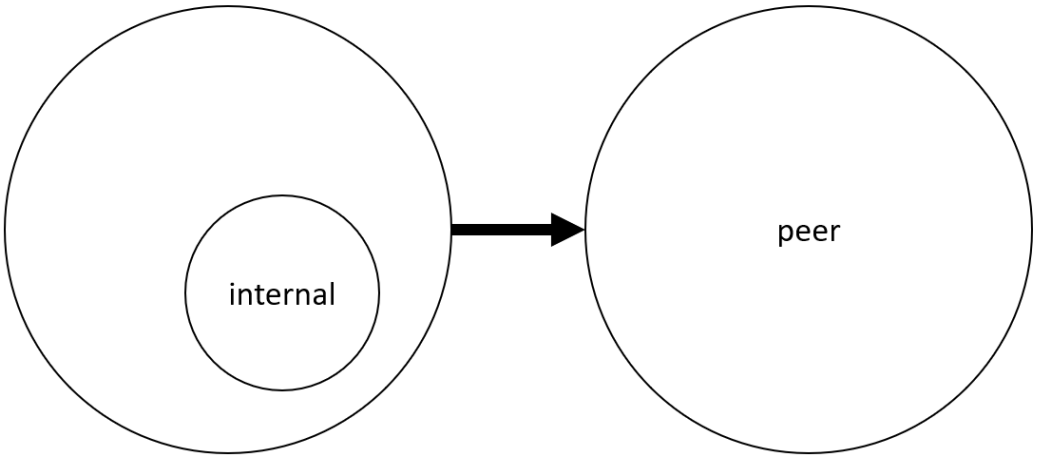
What are object's internals?

Not every object is part of the web. I already mentioned value objects and data structures. They might be part of the public API of a class but are not its peers.

Another useful category are internals – objects created and owned by other objects, tied to the lifespans of their owners.

Internals are encapsulated inside its owner objects, hidden from the outside world. Exposing internals should be considered a rare exception rather than a rule. They are a part of why their

owner objects behave the way they do, but we cannot pass our own implementation to customize that behavior. This is also why we can't mock the internals - there is no extension point we can use. Luckily we also don't want to. Meddling into the internals of an object would be breaking encapsulation and coupling to the things that are volatile in our design.



An internal vs a peer

Internals can be sometimes passed to other objects without breaking encapsulation. Let's take a look at this piece of code:

```
public class MessageChain
{
 private IReadOnlyList<Message> _messages = new List<Message>();
 private string _recipientAddress;

 public MessageChain(string recipientAddress)
 {
  _recipientAddress = recipientAddress;
 }

 public void Add(string title, string body)
 {
  _messages.Add(new Message(_recipientAddress, title, body));
 }

 public void Send(MessageDestination destination)
 {
  destination.SendMany(_messages);
 }
}
```

In this example, the `_messages` object is passed to the `destination`, but the `destination` doesn't know where it got this list from, so this interaction doesn't necessarily expose structural details of the `MessageChain` class.

The distinction between peers and internals was introduced by Steve Freeman and Nat Pryce in their book Growing Object-Oriented Software Guided By Tests.

Examples of internals

How to discover that an object should be an internal rather than a peer? Below, I listed several examples of categories of objects that can become internals of other objects. I hope these examples help you train the sense of identifying the internals in your design.

Primitives

Consider the following `CountingObserver` toy class that counts how many times it was notified and when a threshold is reached, passes that notification further:

```
class ObserverWithThreshold : Observer
{
 private int _count = 0;
 private Observer _nextObserver;
 private int _threshold;

 public ObserverWithThreshold(int threshold, Observer nextObserver)
 {
  _threshold = threshold;
  _nextObserver = nextObserver;
 }

 public void Notify()
 {
  _count++;
  if(_count > _threshold)
  {
    _count = 0;
    _nextObserver.Notify();
  }
 }
}
```

The `_count` field is owned and maintained by a `CountingObserver` instance. It's invisible outside an `ObserverWithThreshold` object.

An example Statement of behavior for this class could look like this:

```
[Fact] public void
ShouldNotifyTheNextObserverWhenItIsNotifiedMoreTimesThanTheThreshold()
{
 //GIVEN
 var nextObserver = Substitute.For<Observer>();
 var observer = new ObserverWithThreshold(2, nextObserver);

 observer.Notify();
 observer.Notify();

 //WHEN
 observer.Notify();

 //THEN
 nextObserver.Received(1).Notify();
}
```

The current notification count is not exposed outside of the ObserverWithThreshold object. I am only passing the threshold that the notification count needs to exceed. If I really wanted, I could, instead of using an int, introduce a collaborator interface for the counter, e.g. called `Counter`, give it methods like `Increment()` and `GetValue()` and the pass a mock from the Statement, but for me, if all I want is counting how many times something is called, I'd rather make it a part of the class implementation to do the counting. It feels simpler if the counter is not exposed.

Value object fields

The counter from the last example was already a value, but I thought I'd mention about richer value objects.

Consider a class representing a commandline session. It allows executing commands within the scope of a working directory. An implementation of this class could look like this:

```
public class CommandlineSession
{
 private AbsolutePath _workingDirectory;

 public CommandlineSession(PathRoot root)
 {
  _workingDirectory = root.AsAbsolutePath();
 }

 public void Enter(DirectoryName dirName)
 {
  _workingDirectory = _workingDirectory.Append(dirName);
 }

```

```
15  public void Execute(Command command)
16  {
17   command.ExecuteIn(_workingDirectory);
18  }
19  //...
20 }
```

This `CommandlineSession` class has a private field called `_workingDirectory`, representing the working directory of the current commandline session. Even though the initial value is based on the constructor argument, the field is managed internally from there on and only passed to a command so that it knows where to execute. An example Statement for the behavior of the `CommandLineSession` could look like this:

```
1  [Fact] public void
2  ShouldExecuteCommandInEnteredWorkingDirectory()
3  {
4   //GIVEN
5   var pathRoot = Any.Instance<PathRoot>();
6   var commandline = new CommandLineSession(pathRoot);
7   var subDirectoryLevel1 = Any,Insytance<DirectoryName>();
8   var subDirectoryLevel2 = Any,Insytance<DirectoryName>();
9   var subDirectoryLevel3 = Any,Insytance<DirectoryName>();
10  var command = Substitute.For<Command>();
11
12  commandline.Enter(subDirectoryLevel1);
13  commandline.Enter(subDirectoryLevel2);
14  commandline.Enter(subDirectoryLevel3);
15
16  //WHEN
17  commandLine.Execute(command);
18
19  //THEN
20  command.Received(1).Execute(
21   AbsolutePath.Combine(
22    pathRoot,
23    subDirectoryLevel1,
24    subDirectoryLevel2,
25    subDirectoryLevel3));
26 }
```

Again, I don't have any access to the internal `_workingDirectory` field. I can only predict its value and create an expected value in my Statement. Note that I am not even using the same methods to combine the paths in both the Statement and the production code - while the production code is using the `Append()` method, my Statement is using a static `Combine` method on an `AbsolutePath` type. This shows that my Statement is oblivious to how exactly the internal state is managed by the `CommandLineSession` class.

Collections

Raw collections of items (like lists, hashsets, arrays etc.) aren't typically viewed as peers. Even if I write a class that accepts a collection interface (e.g. `IList` in C#) as a parameter, I never mock the collection interface, but rather, use one of the built-in collections.

Here's an example of a `InMemorySessions` class initializing and utilizing a collection:

```
public class InMemorySessions : Sessions
{
 private Dictionary<SessionId, Session> _sessions
  = new Dictionary<SessionId, Session>();
 private SessionFactory _sessionFactory;

 public InMemorySessions(SessionFactory sessionFactory)
 {
  _sessionFactory = sessionFactory;
 }

 public void StartNew(SessionId id)
 {
  var session = _sessionFactory.CreateNewSession(id);
  session.Start();
  _sessions[id] = session;
 }

 public void StopAll()
 {
  foreach(var session in _sessions.Values())
  {
   _session.Stop();
  }
 }
 //...
}
```

The dictionary used here is not exposed at all to the external world. It's only used internally. I can't pass a mock implementation and even if I could, I'd rather leave the behavior as owned by the `InMemorySessions`. An example Statement for the `InMemorySessions` class demonstrated how the dictionary is not visible outside the class:

```
[Fact] public void
ShouldStopAddedSessionsWhenAskedToStopAll()
{
 //GIVEN
 var sessionFactory = Substitute.For<SessionFactory>();
 var sessions = new InMemorySessions(sessionFactory);
 var sessionId1 = Any.Instance<SessionId>();
 var sessionId2 = Any.Instance<SessionId>();
 var sessionId3 = Any.Instance<SessionId>();
 var session1 = Substitute.For<Session>();
 var session2 = Substitute.For<Session>();
 var session3 = Substitute.For<Session>();

 sessionFactory.CreateNewSession(sessionId1)
  .Returns(session1);
 sessionFactory.CreateNewSession(sessionId2)
  .Returns(session2);
 sessionFactory.CreateNewSession(sessionId3)
  .Returns(session3);

 sessions.StartNew(sessionId1);
 sessions.StartNew(sessionId2);
 sessions.StartNew(sessionId3);

 //WHEN
 sessions.StopAll();

 //THEN
 session1.Received(1).Stop();
 session2.Received(1).Stop();
 session3.Received(1).Stop();
}
```

Toolbox classes and objects

Toolbox classes and objects are not really abstractions of any specific problem domain, but they help make the implementation more concise, reducing the number of lines of code I have to write to get the job done. One example is a C# `Regex` class for regular expressions. Here's an example of a line count calculator that utilizes a `Regex` instance to count the number of lines in a piece of text:

```
class LocCalculator
{
 private static readonly Regex NewlineRegex
  = new Regex(@"\r\n|\n", RegexOptions.Compiled);

 public uint CountLinesIn(string content)
 {
  return NewlineRegex.Split(contentText).Length;
 }
}
```

Again, I feel like the knowledge on how to split a string into several lines should belong to the `LocCalculator` class. I wouldn't introduce and mock an abstraction (e.g. called a `LineSplitter` unless there were some kind of domain rules associated with splitting the text). An example Statement describing the behavior of the calculator would look like this:

```
[Fact] public void
ShouldCountLinesDelimitedByCrLf()
{
 //GIVEN
 var text = $"{Any.String()}\r\n{Any.String()}\r\n{Any.String()}";
 var calculator = new LocCalculator();

 //WHEN
 var lineCount = calculator.CountLinesIn(text);

 //THEN
 Assert.Equal(3, lineCount);
}
```

The regular expression object is nowhere to be seen - it remains hidden as an implementation detail of the `LocCalculator` class.

Some third-party library classes

Below is an example that uses a C# fault injection framework, Simmy. The class decorates a real storage class and allows to configure throwing exceptions instead of talking to the storage object. The example might seem a little convoluted and the class isn't production-ready anyway. Note that a lot of classes and methods are only used inside the class and not visible from the outside.

```
public class FaultInjectablePersonStorage : PersonStorage
{
 private readonly PersonStorage _storage;
 private readonly InjectOutcomePolicy _chaosPolicy;

 public FaultInjectablePersonStorage(bool injectionEnabled, PersonStorage storage)
 {
  _storage = storage;
  _chaosPolicy = MonkeyPolicy.InjectException(with =>
    with.Fault(new Exception("thrown from exception attack!"))
     .InjectionRate(1)
     .EnabledWhen((context, token) => injectionEnabled)
  );
 }

 public List<Person> GetPeople()
 {
  var capturedResult = _chaosPolicy.ExecuteAndCapture(() => _storage.GetPeople());
  if (capturedResult.Outcome == OutcomeType.Failure)
  {
   throw capturedResult.FinalException;
  }
  else
  {
   return capturedResult.Result;
  }
 }
}
```

An example Statement could look like this:

```
[Fact] public void
ShouldReturnPeopleFromInnerInstanceWhenTheirRetrievalIsSuccessfulAndInjectionIsDi\
sabled()
{
 //GIVEN
 var innerStorage = Substitute.For<PersonStorage>();
 var peopleInInnerStorage = Any.List<Person>();
 var storage = new FaultInjectablePersonStorage(false, innerStorage);

 innerStorage.GetPeople().Returns(peopleFromInnerStorage);

 //WHEN
 var result = storage.GetPeople();

 //THEN
```

```
16  Assert.Equal(peopleInInnerStorage, result);
17 }
```

and it has no trace of the Simmy library.

Summary

In this chapter, I argued that not all communication between objects should be represented as public protocols. Instead, some of it should be encapsulated inside the object. I also provided several examples to help you find such internal collaborators.

Design smells visible in the Specification

In the earlier chapters, I stressed many times how mock objects can and should be used as a design tool, helping flesh out the protocols between collaborating objects. This is because something I call *Test-design approach mismatch* exists. The way I chose to phrase it is:

"Test automation approach and design approach need to live in symbiosis - build on a similar sets of assumptions and towards the same goals. If they do, they reinforce each other. If they don't, there will be friction between them."

I find this to be universally true. For example, if I test an asynchronous application as if I was testing a synchronous application or if I try to test JSON web API using a clicking tool, my code will either not work correctly or look weird. I will probably need to put extra work to compensate for the mismatch between the testing and design approach. For this reason some who prefer much different design approaches consider mock objects a smell[129].

I am seeing it on the unit level as well. In this book, I am using the specification terminology instead of testing terminology. Thus I can say that by writing my Specification on unit level and using mock objects in my Statements, I expect my design to have certain properties and I find these properties desirable. If my code does not show these properties, it will make my life harder. The other side of this coin is that if I train myself to recognize the situations where the design and specification approaches diverge and patterns by which it happens, I can use this knowledge to improve my design.

In this chapter, I present some of the smells that can be seen in the Statements but are in reality design issues. All of these smells come from the community and you can find them catalogued in other places.

Design smells' catalog

Specifying the same behavior in many places

Sometimes, when I write a Statement, I have a deja vu and start thinking "I remember specifying a behavior like this already". I will feel this especially when that other time is not so long ago, maybe even a couple of minutes.

Consider the following Statement describing a reporting rule generation logic:

[129] https://medium.com/javascript-scene/mocking-is-a-code-smell-944a70c90a6a

```
[Fact] public void
ShouldReportAllEmployeesWhenExecuted()
{
  //GIVEN
  var employee1 = Substitute.For<Employee>();
  var employee2 = Substitute.For<Employee>();
  var employee3 = Substitute.For<Employee>();
  var allEmployees = new List<Employee>() {employee1, employee2, employee3};
  var reportBuilder = Substitute.For<ReportBuilder>();
  var employeeReportingRule = new EmployeeReport(allEmployees, reportBuilder);

  //WHEN
  employeeReportingRule.Execute();

  //THEN
  Received.InOrder(() =>
  {
    reportBuilder.AddHeader();
    employee1.AddTo(report);
    employee2.AddTo(report);
    employee3.AddTo(report);
    reportBuilder.AddFooter();
  });
}
```

Now, another Statement for a different rule:

```
[Fact] public void
ShouldReportTotalCost()
{
  //GIVEN
  var calculatedTotalCost = Any.Integer();
  var reportBuilder = Substitute.For<ReportBuilder>();
  var costReportingRule = new EmployeeReport(totalCost, reportBuilder);

  //WHEN
  employeeReportingRule.Execute();

  //THEN
  Received.InOrder(() =>
  {
    reportBuilder.AddHeader();
    reportBuilder.AddTotalCost(totalCost);
    reportBuilder.AddFooter();
  });
}
```

And note that I have to specify again that a header and a footer is added to the report. This might mean there is redundancy in the design and maybe every report or at least most of them, need a header and a footer. This observation, amplified of the feeling of pointlessness from describing the same behavior several times, may lead me to try to extract some logic into a separate object, maybe a decorator, that will add footer before the main part of the report and footer at the end.

When this is not a problem

The trick with recognizing redundancy in the design is to detect whether the behavior in both places will change for the same reasons and in the same way. This is usually true if both places duplicate the same domain rule. But consider these two functions:

```
double CmToM(double value)
{
  return value/100;
}

double PercentToFraction(double value)
{
  return value/100;
}
```

While the implementation is the same, the domain rule is different. In such situations, we're not talking about redundancy, but coincidence. Also see Vladimir Khorikov's excellent post on this topic[130].

Redundancy in the code is often referred to as "DRY (don't repeat yourself) violation". Remember that DRY is about domain rules, not code.

Many mocks

So far, I have advertised using mocks in modelling roles in the Statements. So, mocking is good, right? Well, not exactly. Through mock objects, we can see how the class we specify interacts with its context. If the interactions are complex, then the Statements will show that. This isn't something we should ignore. Quite the contrary, the ability to see this problem through our Statements is one of the main reasons we use mocks.

One of the symptoms of the design issues is that we have too many mocks. We can feel the pain either when writing the Statement ("oh no, I need another mock") or when reading ("so many mocks, I can't see where something real happens").

Doing too much

It may be that a class needs lots of dependencies because it does too much. It may do validation, read configuration, page through persistent records, handle errors etc. The problem with this is that when the class is coupled to too many things, there are many reasons for the class to change.

A remedy for this could be to extract parts of the logic into separate classes so that our specified class does less with less peers.

[130] https://enterprisecraftsmanship.com/posts/dry-damp-unit-tests/

Mocks returning mocks returning mocks

Mocks returning mocks returning mocks typically mean that our code is not built around the tell don't ask heuristic and that the specified class is coupled to too many types.

Many unused mocks

If we are passing many mocks in each Statement only to fill the parameter list, it may be a sign of poor cohesion.

Having many unused mocks is OK in facade classes, because the reasons for facades to exist is to simplify access to a subsystem and they typically hold references to many objects, doing little with each of them.

Mock stubbing and verifying the same call

Consider an example where our application manages some kind of resource and each requester obtains a "slot" in this resource. The resource is represented by the following interface:

```
interface ISharedResource
{
  public SlotId AssignSlot();
}
```

when specifying the behaviors of any class that uses this resource, will use a mock:

```
var resource = Substitute.For<ISharedResource>();
```

we want to describe that a slot reservation is attempted in a shared resource only once, so we will have the following call on our mock:

```
resource.Received(1).AssignSlot();
```

But we will also need to describe what happens when the slot is successfully assigned or when the assignment fails, so we need to configure the mock to return something:

```
resource.AssignSlot().Returns(slotId);
```

Having both setting up a return value and verification of the call to same method in a Statement is a sign we are violating the command-query separation principle. This will limit our ability to elevate polymorphism in implementations of this interface.

But what can we do about this?

pass a continuation

We may pass a more full-featured role that will continue the flow. Below is an example of passing a cache where the assigned slot will be saved if the assignment is successful:

```
resource.Received(1).AssignSlot(cache);
```

pass a callback

we may create a special role for a recipient of the result and pass it inside the `AssignSlot` method, then make it act on the result:

```
Received.InOrder(() =>
{
  resource.AssignSlot(assignSlotResponse);
  assignSlotResponse.Received(1).DisplayOn(screen);
})
```

pass a the needed collaborators through the constructor

Sometimes we may completely remove the notion of result from the interface and make passing it further an implementation detail. Here is the Statement where we don't specify anything about handling a potential result:

```
resource.Received(1).AssignSlot();
```

the implementations are still free to do what they want (or nothing at all) depending on assignment success or failure. One such implementation might have a constructor that looks like this:

```
public CacheBackedSharedResource(IAssignmentCache cache) { ... }
```

and may use its constructor argument to delegate some of the logic there.

When is this not a problem?

TODO: I/O boundary - sometimes easier to return something than to pass a collaborator through an architectural boundary.

Trying to mock a third-party interface

The classic book on TDD, Growing Object-Oriented Software Guided By Tests, suggests to not mock types you don't own. Because mocks are a design tool, we only want to create mocks of types we can control and which represent abstractions. Otherwise we are tied by the definition of the interface "as is" and cannot improve our Statements by improving the design.

Does it mean that if I have a `File` class, I can just wrap it in a thin interface and create an `IFile` interface to enable mocking and I'm good? If the constraint I place on this interface is that it's 1:1 to the type I don't own, then it's back to square one - I still have an interface which is defined in a way that may make writing Statements with mocks hard and there is nothing I can do about it.

So what should I do when I am close to the I/O boundary? I should use a different type of Statements, which I will show you in one of the next parts of this book.

What about logging?

The GOOS book also discusses this topic - loggers from libraries are often ubiquitous all over many codebases. TODO https://learn.microsoft.com/en-us/dotnet/core/extensions/logger-message-generator and Support class.

Blinking tests because of generated data

This specific smell is not related to mock objects, but rather to using constrained non-determinism.

When a Statement becomes unstable due to data being generated as different values on each execution, this might mean that the behavior is too reliant of data and lacks abstraction. The code might look like this:

```
1  if(customer.Accounts.First(a => a.IsPrimary).IsActive)
2  {
3    customer.Status = CustomerStatus.Active;
4  }
```

Note that code checks the `IsActive` flag (a boolean) of the first account set as primary (again, using a boolean `IsPrimary` flag). Booleans are especially vulnerable to non-deterministic generation because they have two possible values, each of which is typically in a different equivalence class. And here we have two. So I would expect Statements that use generated data to fail regularly in a non-deterministic way.

One of the possible solutions here could be to create an abstraction over the customer data structure and use a mock instead of a generated data structure. Something like this:

```
1  if(customer.HasActiveAccount())
2  {
3    customer.Activate();
4  }
```

This solution is the naive, textbook solution. In reality, we might even end up rethinking the whole concept of activation and it could lead us to inventing new interesting roles.

Many times, I encountered people using data generators such as `Any` as a free pass to pass monstrous and complicated data structures around (because "why not? I can generate it with a single call to `Any.Instance<>()`"). However, I believe they should be working in the opposite direction - by introducing the notion of non-determinism, they force me to be more paranoid about the context in which my object works to keep "moving parts" of it to a minimum.

TODO: example

TODO: lack of abstraction

TODO: do I need this?

Too fine-grained role separation

can we introduce too many roles? Probably.

For example, we might have a cache class playing three roles: `ItemWriter` for saving items, `ItemReader` for reading items and `ExpiryCheck` for checking if a specific item is expired. While I consider striving for more fine-grained roles to be a good idea, if all are used in the same place, we now have to create three mocks for what seems like a consistent set of obligations scattered across three different interfaces.

General description

what does smell mean? why test smells can mean code smells?

Flavors

- testing the same thing in many places
  - Redundancy
  - When this is not a problem - decoupling of different contexts, no real redundancy
- Many mocks/ Bloated constructor (GOOS) / Too many dependencies (GOOS)
  - Coupling
  - Lack of abstractions
  - Not a problem when: facades
- Mock stubbing and verifying the same call
  - Breaking CQS
  - When not a problem: close to I/O boundary
- Blinking test (Any + boolean flags and ifs on them, multiple ways of getting the same information)
  - too much focus on data instead of abstractions
  - lack of data encapsulation
  - When not a problem: when it's only a test problem (badly written test). Still a problem but not design problem.
- Excessive setups (sustainabletdd) - both preparing data & preparing the unit with additional calls
  - chatty protocols
  - issues with cohesion
  - coupling (e.g. to lots of data that is needed)
  - When not a problem: higher level tests (e.g. configuration upload) - maybe this still means bad higher level architecture?
- Combinatorial explosion (TODO: confirm name)
  - cohesion issues?
  - too low abstraction level (e.g. ifs to collection)

  - When not a problem: decision tables (domain logic is based on many decisions and we already decoupled them from other parts of logic)
- Need to assert on private fields
  - Cohesion
- pass-through tests (test simple forwarding of calls)
  - too many layers of abstraction.
  - Also happens in decorators when decorated interfaces are too wide
- Set up mock calls consistency (many ways to get the same information, the tested code can use either, so we need to setup multiple calls to behave consistently)
  - Might mean too wide interface with too many options
  - or insufficient level of abstraction
- overly protective test (sustainabletdd) - checking other behaviors than only the intended one out of fear
  - lack of cohesion
  - hidden coupling?
- Having to mock/prepare framework/library objects (e.g. HttpRequest or sth.)
  - Don't mock what you don't own?
  - In domain code - coupling to tech details
- Liberal matchers (lost control over how objects flow through methods)
  - cohesion
  - chatty protocols
  - separate usage from construction
- Encapsulating the protocol (hiding mock configuration and verification behind helper methods because they are too complex or repeat in each test)
  - Cohesion (the repeated code might be moved to a separate class)
- I need to mock object I can't replace (GOOS)
- Mocking concrete classes (GOOS)
- Mocking value-like objects (GOOS)
  - making objects instead of values
- Confused object (GOOS)
- Too many expectations (GOOS)
- Many tests in one (hard to setup)
- also look here: https://www.yegor256.com/2018/12/11/unit-testing-anti-patterns.html

THIS IS ALL I HAVE FOR NOW. WHAT FOLLOWS IS RAW, UNORDERED MATERIAL THAT'S NOT YET READY TO BE CONSUMED AS PART OF THIS TUTORIAL

Mock objects as a design tool

Responsibility-Driven Design

TDD with mocks follows RDD, so what? Model roles. Everything else might go into values and internals. But there is already a metaphor of the web

Ok, so maybe a CRC card?

Not every class is part of the web. One example is value objects. Another example are simple data structures, or DTOs.

Guidance of test smells

Long Statements

Lots of stubbing

Specifying private members

Revisiting topics from chapter 1

Constrained non-determinism in OO world

Passive vs active roles

Behavioral boundaries

Triangulation

Maintainable mock-based Statements

Setup and teardown

Refactoring mock code

until you pop it out through the constructor, it's object's private business.

mocks rely on the boundaries being stable. If wrong on this, tests need to be rewritten, but the feedbak from tests allows stabilizing the boundaries further. And there are not that many tests to change as we test small pieces of code.

well-designed interactions should limit the impact of change

Part 4: Application architecture

On stable/architectural boundaries

Ports and adapters

Physical separation of layers

“Screaming” architecture

What goes into application?

Application and other layers

Services, entities, interactors, domain etc. - how does it match?

What goes into ports?

Data transfer objects

Ports are not a layer

Part 5: TDD on application architecture level

Designing automation layer

Adapting screenplay pattern

code in terms of intention (when knowing more about intention) refactor the domain-specific API (when knowing more about underlying technology)

Driver

reusing the composition root

Separate start method

Fake adapters

They include port-specific setup and assertions.

Create a new fake adapter per each call.

Using fakes

For threads and e.g. databases - simpler objects with partially defined behavior

Actors

Where do assertions go? into the actors or context?

How to manage per-actor context (e.g. each actor has its own sent & received messages stored)

These are not actors as in actor model

Data builders

nesting builders, builders as immutable objects.

Further Reading

Motivation – the first step to learning TDD

- Fearless Change: Patterns for Introducing New Ideas by Mary Lynn Manns Ph.D. and Linda Rising Ph.D. is worth looking at.
- Resistance Is Not to Change[131] by Al Shalloway

The Essential Tools

- Gerard Meszaros has written a long book about using the XUnit family of test frameworks, called XUnit Test Patterns[132]. This book also explains a lot of philosophy behind these tools.

Value Objects

- Ken Pugh has a chapter devoted to value objects in his book *Prefactoring* (the name of the chapter is *Abstract Data Types*).
- *Growing Object Oriented Software Guided By Tests* contains some examples of using value objects and some strategies on refactoring towards them.
- Value object discussion[133] on C2 wiki.
- Martin Fowler's bliki mentions[134] value objects. They are also one of the patterns in his book Patterns of Enterprise Application Architecture[135]
- Arlo Beshele describes[136] how he uses value objects (described under the name of *Whole Value*) much more than I do in this book, presenting an alternative design style that is closer to functional that the one I write about.
- Implementation Patterns[137] book by Kent Beck includes value object as one of the patterns.

[131] http://www.netobjectives.com/blogs/resistance-not-change
[132] http://xunitpatterns.com/
[133] http://c2.com/cgi/wiki?ValueObject
[134] https://martinfowler.com/bliki/ValueObject.html
[135] https://martinfowler.com/books/eaa.html
[136] http://arlobelshee.com/the-no-mocks-book/
[137] https://isbnsearch.org/isbn/9780321413093