Grzegorz Gałęzowski

Test-Driven Development

Extensive Tutorial

Test-Driven Development: Extensive Tutorial

Grzegorz Gałęzowski

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

This version was published on 2016-03-30

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

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:

https://twitter.com/search?q=#tddebookxt

Contents

Dedications

Ad Deum qui laetificat iuventutem meam.

To my beloved wife Monika.

Thanks!

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

- Brad Appleton
- Chris Kucharski
- Daniel Dec
- Daniel Żołopa (cover image)
- Donghyun Lee
- Marek Radecki
- Martin Moene
- Michael Whelan
- 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
- Ken Pugh
- Kent Beck
- Mark Seemann
- Martin Fowler
- Nat Pryce
- Philip Schwarz
- Robert C. Martin
- Scott Bain
- Steve Freeman

Part 1: Just the basics

In this part I introduce the basic TDD philosophy and practices, without going much into advanced aspects like applying TDD to object-oriented systems where multiple objects collaborate (which is a topic of part 2). 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 quite 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 a TDD enthusiast. I believe TDD is a huge improvement over other software development methodologies I have used to deliver quality software. I believe this is true not only for me, but for many other software developers. Which makes me question, why don’t more people learn and use TDD as their software delivery methodology of choice? In my professional life, I haven’t seen the adoption rate to be big enough to justify the claim that TDD is currently in the mainstream.

You already have my respect for deciding to pick up a book, rather than building your understanding of TDD on the foundation of urban legends and your own imagination. I am honored and happy 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 really hope you will read this book from cover to cover, I am aware it doesn’t always happen. That makes me want to ask you an important question that may help you determine whether you really 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. A few years ago, I had an apprentice who wanted to learn TDD. Together we started working on a small project to let him grasp the necessary skills through practice, with me sitting next to him, providing guidance. He showed up three or four times, then he resigned, having “more urgent things to do” and “no time”. Since then, he has not progressed in his understanding or utilization of TDD at all. Even today, I sometimes wonder what was his motivation and why it somehow burned out.

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 whatever reasons, e.g. getting a promotion at work, gaining a certificate, adding something to CV, or just “staying up to date” with recent hypes. Unfortunately, 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 really 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 draw conclusions 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, however, they miss a lot of the essence of TDD. If TDD does not bring the benefits we imagine it might bring, dissapointment may creep in. I heard such disappointed practitioners saying “I don’t really 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

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

integration tests, smoke tests, sanity tests, exploration tests, etc…?”. Many times, I saw TDD getting abandoned before even being understood.

Is learning TDD a high priority for you? Are you determined to try it out and really 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? Ok, I’m just teasing, however, as some say, TDD is “easy to learn, hard to master”[2], so without some guts to move on, it will be hard. Especially since my plan is to introduce the content slowly and gradually, so that you can get 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

[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.

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 down. These in-the-air techniques were called "juggles". There were magazines that 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 really challenging. And while I think such exercises are very useful (in fact, I do a lot of them), I find an immense benefit in understand 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 really 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 really far back then from knowing traffic regulations and following road signs, but that didn't really 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 in order 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 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 commandline, multiplies them and prints the result on the console. The code is pretty straightforward:

```
public static void Main(string[] args)
{
  try
  {
    int firstNumber = Int32.Parse(args[0]);
    int secondNumber = Int32.Parse(args[1]);

    var result =
      new Multiplication(firstNumber, secondNumber).Perform();

    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 then 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);
  }
}
```

Looks simple, right? 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)
```

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

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 equality not for integers, but for 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 commandline 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:

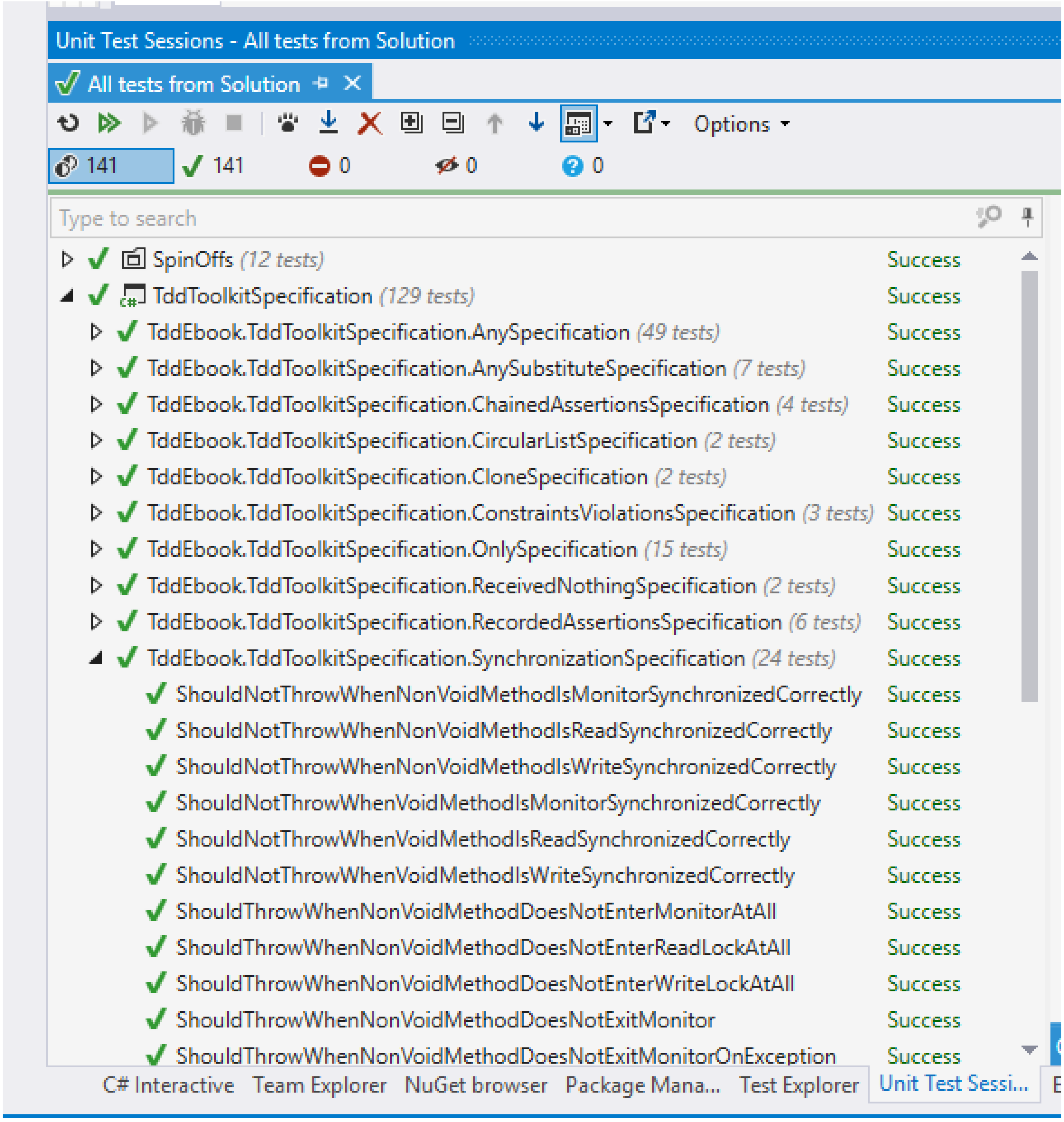
```
[Fact] public void
Multiplication_ShouldResultInAMultiplicationOfTwoPassedNumbers()
{
  //Assuming...
  var multiplication = new Multiplication(3,7);

  //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 replacements that are far better, 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 own 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] http://xunit.github.io/



```
▷ ✓ SpinOffs (12 tests)                                                              Success
◢ ✓ TddToolkitSpecification (129 tests)                                              Success
   ▷ ✓ TddEbook.TddToolkitSpecification.AnySpecification (49 tests)                  Success
   ▷ ✓ TddEbook.TddToolkitSpecification.AnySubstituteSpecification (7 tests)         Success
   ▷ ✓ TddEbook.TddToolkitSpecification.ChainedAssertionsSpecification (4 tests)     Success
   ▷ ✓ TddEbook.TddToolkitSpecification.CircularListSpecification (2 tests)          Success
   ▷ ✓ TddEbook.TddToolkitSpecification.CloneSpecification (2 tests)                 Success
   ▷ ✓ TddEbook.TddToolkitSpecification.ConstraintsViolationsSpecification (3 tests) Success
   ▷ ✓ TddEbook.TddToolkitSpecification.OnlySpecification (15 tests)                 Success
   ▷ ✓ TddEbook.TddToolkitSpecification.ReceivedNothingSpecification (2 tests)       Success
   ▷ ✓ TddEbook.TddToolkitSpecification.RecordedAssertionsSpecification (6 tests)    Success
   ◢ ✓ TddEbook.TddToolkitSpecification.SynchronizationSpecification (24 tests)      Success
        ✓ ShouldNotThrowWhenNonVoidMethodIsMonitorSynchronizedCorrectly             Success
        ✓ ShouldNotThrowWhenNonVoidMethodIsReadSynchronizedCorrectly                Success
        ✓ ShouldNotThrowWhenNonVoidMethodIsWriteSynchronizedCorrectly               Success
        ✓ ShouldNotThrowWhenVoidMethodIsMonitorSynchronizedCorrectly                Success
        ✓ ShouldNotThrowWhenVoidMethodIsReadSynchronizedCorrectly                   Success
        ✓ ShouldNotThrowWhenVoidMethodIsWriteSynchronizedCorrectly                  Success
        ✓ ShouldThrowWhenNonVoidMethodDoesNotEnterMonitorAtAll                      Success
        ✓ ShouldThrowWhenNonVoidMethodDoesNotEnterReadLockAtAll                     Success
        ✓ ShouldThrowWhenNonVoidMethodDoesNotEnterWriteLockAtAll                    Success
        ✓ ShouldThrowWhenNonVoidMethodDoesNotExitMonitor                            Success
        ✓ ShouldThrowWhenNonVoidMethodDoesNotExitMonitorOnException                 Success
```

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 youreslf 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 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`). In addition, 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 really hard that this is some serious domain logic. Here’s the code for this class:

```
public class OrderProcessing
{
  OrderDatabase _orderDatabase; //OrderDatabase is an interface
  Log _log;

  //we get the database object  from outside the class:
  public OrderProcessing(
    OrderDatabase database,
    Log log)
  {
    _orderDatabase = database;
    _log = log;
  }

  //other code...

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

  //other code...
}
```

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:

```
[Fact] public void
ShouldInsertNewOrderToDatabaseWhenOrderIsPlaced()
{
  //GIVEN
  var orderDatabase = new MySqlOrderDatabase(); //uses real database
  orderDatabase.Connect();
  orderDatabase.Clean(); //clean up after potential previous tests
  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);
}
```

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 make an assertion 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 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 in a different way. 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 like 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? Susprise, surprise, there is one! It's called **Autofixture**[5]. It is an example of so-called anonymous values generator (although its creator likes to say that it is also an implementation of 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 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 in hope 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 drown.

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 course book 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 counter-intuitive 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:

```
var reporting = new ReportingFeature();
var anyPowerUser = Any.Of(Users.Admin, Users.Auditor);
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 a notion of users and privileges.
3. We need to support a concept of 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 requiremnts 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 feelings is that maybe a different name would be better - hence the term *specification*.

The discovery of 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 analysing 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 – my hope is 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”

[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!

False Statement
will be used instead of “failing test”

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 other spoken language. However, our Specifications differ from them in at least few 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 our 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?

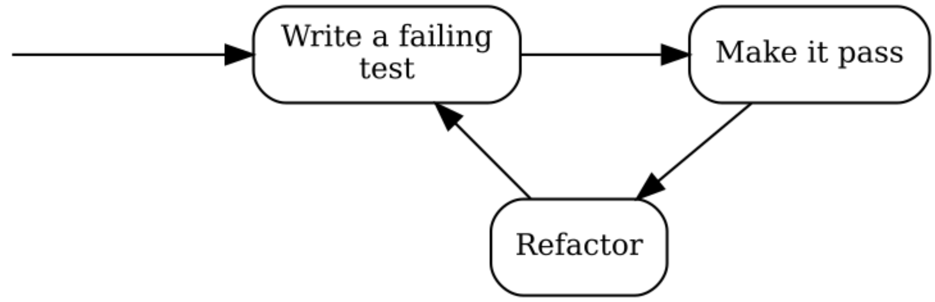
One of the best known thing about TDD is that a failing test for a behavior of a piece of code is written before this behavior is implemented. This concept is often called "test-first development" and seems controversial to many.

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 really know whether we are done implementing our changes or not? This is because in order 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.

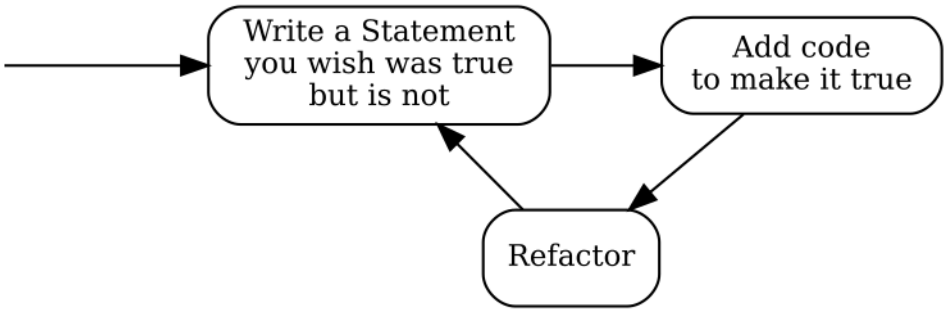
Another thing I mentioned in the previous chapter is that we approach writing a Specification of executable Statements differently from writing a textual design or requirements specification: even though a behavior is implemented after its Specification is ready, we do not write the Specification entirely 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 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 really 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 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 of 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 really 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 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 the 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.

The reason I didn't see my mistake was because I was running more than once 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 work flow: 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 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 test 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 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 as an argument and checks whether the time slot (whatever it is) is above a value specified in a constant called TimeSlot.MaxAllowed (so it's a constant defined in a TimeSlot class). If it is, then the validation returns false, if it's not, then it returns true.

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:

[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 from that rule. This will be explained in part 2.

```
[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* a 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 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 discovering 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...-->
  </section>
</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:

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

And goes on with the scenario. When we execute it, it passes – cool… not. Ok, what's wrong with this? At the 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:

```
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 despite the fact that 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 a requirements or 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 writing my executable Specification. By observing myself and other developers, I came to a conclusion 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 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 its’s very likely to get dropped, especially in 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 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 – in order 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.

[8] http://www.confreaks.com/videos/759-rubymidwest2011-keynote-architecture-the-lost-years

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).
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 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 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 did not 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 saw 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 are gonna 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 really hope you did not fall asleep while reading it. To tell you the truth, we need to grasp much more theory until we are really able to write real-world applications using TDD. To compensate for this somehow, I propose we make a side trip from the trail and try what we already learned on 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. Do not 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 nameless ninja was to real martial arts fight, but will show you some of the elements of 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 details, 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?

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, 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 do not get over-excited.

Benjamin: Sure.

Act 2: The Customer

Johnny: Hi, Jane, how are you?

Jane: Thanks, I am fine, how about you?

Johnny: Same here, thanks. 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 was running on the previous phone and I cannot find it for my current device.

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

Jane: That is right. I checked them all and none has exactly the same behavior as the one I was using for my tax calculations. You know, the program was like right hand to me and it had some really nice shortcuts that made my life easier.

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

Jane: Yes.

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 am so used to my calculator application 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 a simple 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 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 implement addition, we will have to implement displaying digits on the screen as you enter them. Is that correct?

Jane: Yes, I want the display stuff to work as well – it is the most basic feature, 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** tax payer **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** tax payer, **I want** the calculator to display the values I enter
3. **In order to** calculate the sum of my different incomes, **As a** tax payer **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 do not think I have any corrections to make.

Johnny: Now we will 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 am in. What do I do?

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

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

Jane: I do not think there is much to talk about. If you display "0", I will be happy. That is all.

Johnny: Let's write down this example:

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 is what these examples are for – they make our thinking concrete. As Ken Pugh says: "Often the complete understanding of a concept does not occur until someone tries to use the concept". We would normally put it on a TODO list, because it is part of a different story, but as we are done with this one, let's move straight to the story about displaying entered digits. How about it, Jane?

Jane: Agree.

In order to see what numbers I am currently operating on, As a tax payer, 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 have only "0" on the display?

Jane: Just one "0" should be displayed

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 is 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 am beginning to understand why you like working with examples!

Johnny: Good. Is there anything else, Jane?

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

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

Johnny: Is the following all 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 do not mind. I do not 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 just asking for result 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 situation at all. How about that, Jane?

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

Johnny: See? That is why I do not 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 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 am 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: Boy, I do not 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:

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{

}

}
```

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. 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 is a statement of behavior, isn't it?

Johnny: That is right. Now, the second trick I can sell to you is that if you do not know what code to start 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?

```
public class CalculatorSpecification
{

[Fact] public void
ShouldDisplay0WhenCreated()
{
 Assert.Equal("0", displayedResult);
}

}
```

Johnny: Precisely.

Benjamin: But that does not even compile. What use is it?

Johnny: The code not compiling is the feedback that you needed to proceed. While previously you did not 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 form 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 and this is 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: Now the code still does not compile, because I do not have the `Calculator` class 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 will add it.

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

Johnny: Hey hey, not so fast!

Benjamin: What?

Johnny: You already provided an implementation that will make our current Statement true. Remember its name? `ShouldDisplay0WhenCreated` – and that is 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 evaluate to false. Now we are 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 is 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 then evaluate to 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: Looks like there are quite a few benefits. Still, I will have to get used to this kind of working.

Johnny: Do not worry, it is central to TDD, so you will grasp it in no time. Now, before we proceed to 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 constant in one place will make it more maintainable, but I think the Statement in its current form is weaker than before. We could change the `InitialValue` to anything and the Statement would still be true, since it does not force this constant to be "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 is 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 will 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 am glad that you got the part about naming and writing a Statement. There is 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 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 is pretty self-evident that the sum is a concatenation of entered digits, we do not want anyone who will be reading this Specification in the future to have to guess.
3. The Statement suggests that what you wrote is sufficient for any value, which isn't true, since the behavior for "0" is different (no matter how many times we enter "0", the result is just "0")

Hence, I propose the following:

```
[Fact] public void
ShouldDisplayAllEnteredDigitsThatAreNotLeadingZeroes()
{
 //GIVEN
 var calculator = new Calculator();
 var nonZeroDigit = Any.Besides(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 am lost! Can you explain what is 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 `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.Besides()` and `Any.Of()`? What do they do?

Johnny: They are methods from a small utility library I am using when writing unit-level Specifications. `Any.Besides()` returns any value from enumeration besides the one supplied as an argument. Hence, the call `Any.Besides(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 when I create the values this way, I state explicitly that this behavior occurs when first digit is non-zero. This technique of using generated values instead of literals has its own principles, but let's leave it for later. I promise to give you a detailed lecture on it. Agree?

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 understand what is going on – 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 "331", 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:

```
public enum DigitKeys
{
 Zero = 0,
 TODO1, //TODO
 TODO2, //TODO
 TODO3, //TODO
 TODO4, //TODO
}
```

Benjamin: What is with all those bogus values? Should we not just enter all the digits we support?

Johnny: Nope, not yet. We still do not have a Statement which would say what digits are supported and thus 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, should it not?

Benjamin: It is difficult to agree with, especially since I am used to writing unit tests, not Statements, and, in unit tests, I wanted to verify what I was unsure of.

Johnny: I will try to give you more arguments later. For now, just bear with me and note that adding such Statement will be almost effortless.

Benjamin: OK.

Johnny: Now for the implementation. Just to remind you – up to now, it looked like this:

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

This is not enough to support displaying multiple digits (as we saw, because the Statement saying they should be supported was evaluated to false). So let's evolve the code to handle this case:

```
public class Calculator
{
 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.Besides(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 do not 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.Besides(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, did we not?

Johnny: Not exactly. With moves such as this one, I like to be extra careful. Let's comment out the body of the `Enter()` method and see if this Statement can still be made false by the implementation:

```
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 are pretty sure that it works OK. Let's uncomment the lines we just commented out and move forward.

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 previous Statement.

Benjamin: Let me try… ok, here it is:

```
[Fact] public void
ShouldDisplayOnlyOneZeroDigitWhenItIsTheOnlyEnteredDigitEvenIfItIsEnteredMult\
ipleTimes()
{
 var zero = DigitKeys.Zero;
 var calculator = new Calculator();

 calculator.Enter(zero);
 calculator.Enter(zero);
 calculator.Enter(zero);

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

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

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

Johnny: That would be wise thing to do. When a Statement is true without requiring you to change any production code, it is always suspicious. Just like you said, we have to change production code for a second to force this Statement to be 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:

```
public class Calculator
{
 int _result = 0;
 string _fakeResult = "0"; //+

 public void Enter(DigitKeys digit)
 {
  _result *= 10;
  _result += (int)digit;
  if(digit == DigitKeys.Zero) //+
  {  //+
   _fakeResult += "0";  //+
  } //+
 }

 public string Display()
 {
  if(_result == 0)  //+
  {  //+
   return _fakeResult;  //+
  }  //+
  return _result.ToString();
 }
}
```

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 is 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 actually planned to make this chapter longer, and cover all the other operations, but I fear I would make this too long and bore you. 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, do not worry – I do not 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 previous 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 sorted out. We will pick up all those questions and explain them one by one. Here they are:

- 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 input of a specified behavior?
- Why and how to use the Any class?
- What code to extract from a Statement to shared utility method?
- Why such a strange approach to create enumerated constants?

A lot of questions, isn’t it? It is unfortunate that 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 is easy to say: "You know how to write code, you know how to write a unit 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, do not worry – you are 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

It may sound obvious, but some people are having serious trouble describing the behavior they expect from their code. If you can name the behavior, it is a great starting point.

I know not everybody pays attention to naming their Statements, mainly because the Statements are considered second-level citizens – as long as they run and "prove the code does not 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

Many people do not really care about how their Statements are named. This is a symptom of treating the Specification as garbage or leftovers – this approach is dangerous, because it leads to bad, unmaintainable Specifications that look more like lumps of code put together accidentally in a haste than that they resemble living documentation. Imagine that your Specification consists of names like this:

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

and try for yourself 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 really 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 when you modify the functionality it specifies?
5. If your changes in production code make a Statement evaluate to 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 behavior when adding to a Specification that was originally 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:

```
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 does not contain the name of the method that triggers the behavior, because what is specified is not a single method, but the behavior itself. 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 because you can derive such a description from thinking about the responsibilities of a class without the need to know any of its method signature or the code that is inside the class. Hence, this is something you can come up with before implementing – you just need to know why you created this class and build on this knowledge.
2. The name is relatively long. Really, really, **really** do not worry about it. As long as you are describing a single behavior, it’s fine. I know people usually are hesitant to give long names to Statements, because they try to apply the same rules to those names as to method names in production code (and in production code a long method name can be a sign that the method has too many responsibilities). Let me make it clear – these two cases are different. In 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. Sure, we could put all the information in a comment instead of the Statement name and leave the name short, like this:

```
[Fact]
//Notifies listeners that server
//is unavailable when cannot connect
//to its monitoring port
public void Statement_002()
{
  //...
}
```

There are two downsides to this. We now have to add extra information (`Statement_002`) specifically for the compiler, because every method needs to have a name – there is usually no value a human could derive from such a name. The second downside is that when the Statement evaluates to 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. It is really better to receive a report like:

`notifiesListenersThatServerIsUnavailableWhenCannotConnectToItsMonitoringPort: FAILED`

because then a lot of information about the Statement that fails is available from the test runner window.

3. Using a name that describes a single behavior allows you to track quickly why the Statement is false when it is. Suppose a Statement is true when you start refactoring, but at one point it starts being evaluated as false and the report in the runner looks like this: `TrySendingHttpRequest: FAILED` – it tells you that an attempt is made to send a HTTP request, but, for instance, does not tell you whether the object you specified is the sender that should try to send this request under some circumstances, or if it is the receiver that should handle such a request properly. To learn what went wrong, you have to go to the Statement body and scan its source code. Now compare the Statement name to the following one: `ShouldRespondWithAnAckWheneverItReceivesAHttpRequest`. Now when it evaluates to false, you can tell what is 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.

My favourite convention

There are many conventions for naming Statements appropriately. My favorite is the one developed by Dan North[9], where each Statement name begin 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:

[9] http://dannorth.net/introducing-bdd/

```
public class SortingOperationSpecification
{
 [Fact] public void
 ShouldSortAllItemsInAscendingOrderWhenPerformed()
 {
 }
}
```

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[10].

When devising a name, it is important to put the main focus on what result or action is expected from an object. If you do not do that, it'll quickly become troublesome. As an example, one of my colleagues was specifying a class `UserId` and wrote the following name for the Statement about the comparison of two identifiers:

`EqualOperationShouldPassForTwoInstancesWithTheSameUserName()`.

Note that this is not 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 a Statement, this name should be something more like:

`ShouldReportThatItIsEqualToAnotherIdThatHasTheSameUserName()`.

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

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 the name really not 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 does 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 the name is 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.

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

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

This is a technique that is applicable when you come up with a GIVEN-WHEN-THEN structure for the Statement or a good name for it (a GIVEN-WHEN-THEN structure can be easily derived from a good name and vice versa). Anyway, this technique is about taking the GIVEN, WHEN and THEN parts and translating them into code in an almost literal, brute-force way, and then adding all the missing pieces that are required for the code to compile and run.

Example

Let’s try this on a simple problem of comparing two users. 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

The first line:

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

can be translated literally to code like this:

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

Then the second line:

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

can be written as:

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

Great! 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 we have made the translation, let's summarize it and see what is 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 will not 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 do not have compiling code, filling the gaps 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 follows:

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

   ```
   public class User {}
   ```

Putting it all together:

```
[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 is complete!

Start from the end

This is a technique that I suggest to people that seem to have absolutely no idea how to start. I got it from Kent Beck's book Test Driven Development by Example. It seems funny initially, but it is quite powerful. The trick is to write the Statement 'backwards', i.e. starting with what the Statement asserts to be true (in terms of the GIVEN-WHEN-THEN structure, we would say that we start with our THEN).

This works, because many times we are quite sure of what the outcome of the behavior should be, but are unsure of how to get there.

Example

Imagine we are writing a class for granting access to a reporting functionality that 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 and, starting backwards, 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 are to answer the question: how do I know whether the access is granted or not? The answer: it is the result of authorization of the allowed role. Ok, so let's just write it down, ignoring everything that stands in our way:

```
//WHEN
var accessGranted
 = authorization.AccessToReportingIsGrantedTo(
  roleAllowedToUseReporting);
```

Resist the urge to define a class or variable as soon as it seems needed, as that will only 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.

Note that we do not know what `roleAllowedToUseReporting` is, neither do we know what `authorization` is, but that did not stop us from writing this line. Also, the `AccessToReportingIsGrantedTo()` method is just taken off the top of our head. It is 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` from, but it also makes us ask further questions:

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

As for `authorization`, we do not have anything specific to say about it other than that it is an object of a class that we do not yet have. To proceed, let's pretend that we have such a class. How do we call it? The instance name is `authorization`, so it is quite straightforward to name the class `Authorization` and instantiate it in the simplest way we can think of:

```
//GIVEN
var authorization = new Authorization();
```

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. So:

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

And so, working our way backwards, we have arrived at the final solution:

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

 //WHEN
 var accessGranted = authorization
  .AccessToReportingIsGrantedTo(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

Sometimes, we have to add a new class that must implement an existing interface. The interface imposes what methods the new class must support. If this point is already decided, we can start our Statement by first calling the method and then discovering what we need to supply.

Example

Suppose we have an application that, among other things, handles importing an existing database 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:

```
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 importing a database. One of the things we need to do is to 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 class `FriendlyMessages` and implement this interface in a new class that is 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. When we implement the class, we start with the following:

```
public class TrialVersionMessages : Messages
{
 public string HoldOnASecondWhileWeImportYourDatabase(string userName)
 {
   return null;
 }
}
```

Now, we are ready to start writing a Statement. Assuming we do not know where to start, we just start with creating an object and invoking the method that needs to be implemented:

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

 //WHEN
 trialMessages.HoldOnASecondWhileWeImportYourDatabase();

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

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 would otherwise be evaluated as true and we might not notice that it's incomplete. However, as it stands the Statement does not compile anyway, because the method `HoldOnASecondWhileWeImportYourDatabase` takes a string argument and we didn't pass any. This makes us ask the question what this argument is and what its role is in the behavior triggered by method `HoldOnASecondWhileWeImportYourDatabase`. It looks like it is a user name and we want it to be somewhere in the result of the method. 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 evaluated as 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();

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

 //THEN
 var expectedMessage =
  string.Format("{0}, better get some pocket money!", userName);

 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 `ShouldYieldAMessageSayingThatFeatureIsLockedWhenAskedForImportDatabaseMessage`.

Summary

When you are stuck and do not know how to start, the techniques from this chapter may help to get you on your way. Note that the examples given are simplistic and assume that there is only one object that takes some kind of input parameter and returns a well defined result. However, this is not how most of the object-oriented world is built. There we have many objects that communicate with other objects, send messages, invoke methods on each other and that often do not return a value. Don’t worry, all of these techniques also work there 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). Here, I’ve tried to keep it simple.

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

Is there really a commonality between analysis and TDD?

From Wikipedia:

> 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 has to fulfill two conditions:

1. Be a process of breaking a complex topic into smaller parts
2. Allow gaining a better understanding of such broken topics

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 technique called Acceptance Test-Driven Development. The process followed by the three characters fulfilled both mentioned conditions for a process to be 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 (we will talk about 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 assert its result. This makes us think: what should this method be called? What are the scenarios supported? What do I need to get out of it? How should 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. This is how we gain a better understanding of the topic we are analyzing, which makes TDD fulfill the first of the two requirements for it to be an analysis method.

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

If you go back to our example, you will note that both when talking to a customer and when writing code, Johnny and Benjamin used a TODO list. This list was first filled with whatever scenarios they came up with, but later, they would add a smaller unit of work. This is one way complex topics are decomposed into smaller items that all land on the TODO list (the other way is mocking, but we will not get into that yet). Thanks to this, we can focus on one thing at a time, crossing off item after item from the list after it is done. If we learn something new or encounter new issue that needs our attention, we can add it to the TODO list and get back to it later, for now continuing our work on the current point of focus.

An example TODO list from the middle of implementation may look like this (do not read through it, I put it here only to give you a glimpse):

1. — Create 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 items are already crossed out as done, while other remain undone and waiting to be addressed.

Ok, that is it for the discussion. Now that we are sure that TDD is about analysis, let's focus on the tools we can use to aid and inform it. You already saw both of them in this book, now we are going to have a closer look.

Gherkin

Hungry? Too bad, because the Gkerkin I am gonna talk 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 it.

In fact we already used the Gherkin 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 of three parts:

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

In other words, the emphasis is on causality in a given context.

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

```
Given a bag of tea costs $20
When I buy two of them
Then I should be charged $30 due to promotion
```

And here is one for unit-level (note 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 the same artifact (If this does not ring a bell, look at tools like SpecFlow or Cucumber or FIT to get some examples), on the unit level the description is usually not written down literally, but in form of code only. Still, this structure 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 makes them more readable. So most of my unit-level Statements follow this template:

```
[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.

TODO list… again!

As we said previously, a TODO list is a repository for anything that comes to our mind when writing or thinking about a Statement, but is not a part o the current Statement we are writing. We do not want to forget it, neither do we 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.

Suppose you are writing a piece of small logic that allows user access when he is an employee of a zoo, but denies access if he is a guest of the zoo. Then, after starting writing a Statement it gets to you that any employee can be a guest as well – for example, he might choose to visit the zoo with his family during his vacation. Still, the two previous rules hold, so not to allow this case to distract you, you quickly add 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 you are finished, you can always come back to the list and pick item to do next.

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 the TODO list?

Everything that you need addressed but is not part of the current Statement. Those items may be related to implementing unimplemented methods, to add whole functionalities (such items are usually followed my more fine-grained sub tasks as soon as you start implementing the item), there 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 you get carried away too much in coding that you forget to eat, you can even add a reminder ("TODO: get something to eat!"). The list is yours!

How to pick items from TODO list?

Which item to choose from the TODO list when you have several? 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 big impact on design or behavior of the system, I tend to pick such items first. An example of such item is that you start implementing validation of a request that arrives to your application and want to return different error depending on which part of the request is wrong. Then, during the development, you discover that more than one part of the request can be wrong at a time and you have to answer yourself a question: which error code should be returned in such 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 out things, I tend to take on some small and easy items to feel 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.

Where to put a TODO list?

There are two ways of maintaining a TODO list. The first one is on a sheet of paper, which is nice, but requires you to take your hands off the keyboard, grab a pen or pencil and then get back to coding every time you think of something. Also, the only way a TODO item written on a sheet of paper can tell you which place in code it is related to, is (obviously) by its text.

Another alternative is to use a TODO list functionality built-in into an IDE. Most IDEs, such as Visual Studio (and Resharper plugin has its own enhanced version), Xamarin Studio or eclipse-based IDEs have such functionality. The rules are simple – you put special comments in the code and a special view in your IDE aggregates them for you, allowing you to navigate to each. Such lists are great because:

1. They do not force you to take your hands off keyboard to add an item to the list.
2. You can put such item in a certain place in the code where is makes sense and then navigate back to it with a click of a mouse. This, apart from other advantages, lets you write shorter notes than if you had to do it on paper. For example, a TODO item saying “TODO: what if it throws an exception?” looks out of place on the paper, but when added as a comment to your code in the right place, it is mighty sufficient.
3. Many TODO lists automatically add items for certain things. E.g. in C#, when you are yet to implement a method that was automatically generated by a tool, its body usually consists of a line that throws a `NotImplementedException` exception. Guess what

– `NotImplementedException` occurences are added to the TODO list automatically, so we do not have to manually add items to the TODO list for implementing the methods where they occur.

Expanded TDD process with a TODO list

In one of the previous chapters, I introduced you to the basic TDD process that contained three steps: write unfulfilled Statement, fulfill it and refactor the code. TODO list adds new steps to this process leading to the following list of steps:

1. Examine TODO list and pick an item that makes most sense to implement next
2. Write unfulfilled Statement
3. Make it unfulfilled for the right reason
4. Fulfill the Statement and make sure all already fulfilled Statements are still fulfilled
5. Cross out the item from TODO list
6. Repeat until no item is left on the TODO list

Of course, we are free to add new items to the TODO list as we make progress with the existing ones and at the beginning of each cycle the list is re-evaluated to choose the most important item to implement next taking into account what was added during the previous cycle.

Potential downsides

There are also some downsides. The biggest is that people often add TODO items for other means than to support TDD and they never go back to such items. Some people joke that a TODO left in the code means "Once, I wanted to…". Anyway, such items may pollute your TDD-related TODO list with so much cruft that your own items are barely findable. To work around it, I tend to use a different tag than TODO (many IDEs let you define your own tags, or support multiple tag types out of the box. E.g. with Resharper, I like to use "bug" tag, because this is something no one would leave in the code) and filter by it. Another options is, of course, getting rid of the leftover TODO items – if no one addressed it for a year, then probably no one ever will.

Another downside is that when you work with multiple workspaces/solutions, your IDE will gather TODO items only from current solution/workspace, so you will have few TODO lists – one per workspace or solution. Fortunately, this isn't usually a big deal.

Developing a TDD style and Constrained Non-Determinism

In one of the first chapters, I introduced to you the idea of anonymous values generator idea, which I have wrapped in a static class called `Any`. Throughout the next chapters, you have seen me using it quite extensively in many of the Statements I wrote.

The time has come to explain a little bit more carefully what principles lie under this technique and tool. I will also use this technique as a case study to show you how one develops a style of Test-Driven Development.

A style?

Yep. Why am I wasting your time writing about style instead of giving you the hardcore technical details? The answer is simple. Before I started writing this tutorial, I read four or five books solely on TDD and maybe two others that contain chapters on TDD. All of this sums 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 exactly the same sets of techniques for test-driving their code! I mean, sometimes, when you look at the techniques they are suggesting, two authorities contradict each other. As each authority has their followers, it is not uncommon to observe and take part in discussions about whether this or that technique is better than a competing one or which one leads to trouble in the long run.

I did this, too. I also tried to understand how come people praise techniques I KNEW were wrong and led to disaster. Then, Finally, I got it. I understood that it is not a "technique A vs. technique B" debate. There are certain sets of techniques that work together and choosing one technique leaves us with issues we have to resolve by adopting other techniques. This is how a style is created.

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 unit tests 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 oversimplified example shows a Statement where it is not explicitly stated what is the relationship between input and output:

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

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

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

Although the relationship can be guessed quite easily, remember it is just an example. 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_HOST_NAME"? Will the name be "backup_backup_.zip", or just "backup_.zip"? Also, is this object responsible for any validation of passed string?

This makes me invent a first technique to provide my Statements with better support for the principle I believe in.

First technique: Anonymous Input

I can wrap the actual value "MY_HOST_NAME" with a method and give it a name that better documents the constraints imposed on it by the specified functionality. In our case, we can pass whatever string we want (the object is not responsible for input validation), so we will name our method `AnyString()`:

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

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

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

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

By using **anonymous input**, we provide a 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. `AnyAlphaNumericString()` when I need a string that does not contain any characters other than letters and digits. Note that this technique **is applicable only when the particular value of the variable is not important, but rather its "trait"**. Taking authorization as an example, when a certain behavior occurs only when the input value is `Users.Admin`, there is **no sense** making it anonymous. On the other hand, for a behavior that occurs for all values other than `Users.Admin`, it makes sense to use a method like `AnyUserOtherThan(Users.Admin)` or even `AnyNonAdminUser()`.

Now that the Statement itself is freed from the knowledge of the concrete value of `hostName` variable, the concrete value of "backup_MY_HOST_NAME.zip" looks kind of weird. There is no clear indication of the kind of relationship between input and output and whether there is any at all (one may reason whether the output is always the same string or maybe it depends on the string length). It is unclear which part is added by the production code and which part depends on the input we pass to the method. 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(
    string.Format("backup_{0}.zip", hostName),
    name);
}
public string AnyString()
{
  return "MY_HOST_NAME";
}
```

This looks more like a part of 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 host name".

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

Third technique: Distinct Generated Values

Let's assume that some time after our initial version is shipped, we are asked to make the backup feature applied locally per user only for this user's data. As the customer does not want to confuse files from different users, we are asked to add name of the user doing backup to the backup file name. Thus, the new format is "backup_H_U.zip", where H is still the host name 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(string.Format(
    "backup_{0}_{1}.zip", hostName, userName),
    name);
}

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

Now, we can clearly 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 values are applied and that they are applied in the correct places. For example, the Statement will be evaluated to true when user name is used instead of host name in specified production code. This means that if we still want to use the anonymous input effectively, we have to make the two values distinct, e.g. like this:

```
[Fact] public void
ShouldCreateBackupFileNameContainingPassedHostNameAndUserName()
{
  //GIVEN
  var hostName = AnyString();
  var userName = AnyString2();
  var fileNamePattern = new BackupFileNamePattern();

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

  //THEN
  Assert.Equal(string.Format(
    "backup_{0}_{1}.zip", hostName, userName),
    name);
}

public string AnyString()
{
  return "MY_HOST_NAME";
}

public string AnyString2()
{
  return "MY_USER_NAME";
}
```

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

```
[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(string.Format(
    "backup_{0}_{1}.zip", hostName, userName),
    name);
}

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

This time, we are not returning an understandable string, but rather a guid, which gives us the fairly strong guarantee of generating distinct value each time. The string not being understandable (contrary to something like “MY_HOST_NAME”) 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 need a value of a particular type, we get something different (if possible) than the last time and each 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 version number of our application as well. Remembering that we want to use Derived Values, we will not hardcode the version number into our Statement. Instead, we are going to use a constant that is already defined somewhere else in the application (this way we also avoid duplication of this version number across the application):

```
[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(
    string.Format(
      "backup_{0}_{1}_{2}.zip",
      hostName, userName, Version.Number),
    name);
}

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

Note that I didn't use the literal constant value, but rather, the value inside the `Version.Number` constant. This allows us to use derived value, but leaves us a little worried about whether the value of the constant is correct – after all, we are using it for creation of our expected value, but it is a part of production code – i.e. is something that should be specified itself!

To keep everyone happy, we write a single Statement just for the constant to specify what the value should be:

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

By doing so, we make the value in the production code just echo what is in our executable Specification, which we can fully trust.

Summary of the example

In this example, I tried to show you how a style can evolve from the principles you value when doing TDD. I did so for two reasons:

1. To introduce to you a set of techniques I personally use and recommend and to do it in a fluent and logical way.
2. To help you better communicate with people that are using different styles. Instead of just throwing "you are doing it wrong" at them, try to understand their principles and how their techniques of choice support those principles.

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

Anonymous Input
moving the output out of the Statement code and hide it behind a method that to emphasize the constrain on the data used rather than what is its value

Derived Values
defining expected output in terms of the input to document the relationship between input and output

Distinct Generated Values
When using Anonymous Input, 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
Write a separate Statement for a constant and use the constant instead of its literal value in all other Statements to create a Derived Value.

Constrained non-determinism

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

1. Values are anonymous i.e. we do not 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 in order to provide values that are not special in any way (e.g. when generating integers, we do not allow generating '0' as it is usually a special-case-value) and that are not "evil" (e.g. for integers, we generate small positive values first and go with bigger numbers only when we run out of those small ones).

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[11]. Here is a shortest possible snippet to generate an anonymous integer using AutoFixture:

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

I, after Amir Kolsky and Scott Bain, like to use 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[12].

Summary

That was a long ride, wasn't it? 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.

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

[12] https://github.com/grzesiek-galezowski/tdd-toolkit

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

Ha, now I have to admit that I deferred for a long time to answer a pretty fundamental question: what should be the scope of a single Statement? If I put the whole system together, can I write a Statement for its behavior? Or maybe the other way round – there should be a Statement for each method of each class, including the private ones? Well, first thing I want to explain is that there are multiple levels we can write our Statements on. This varies depending on the TDD authority, but in this book, we will cover two of such levels – unit level and acceptance level. For now, let's stick to the unit level, which is what we have done so far anyway. The time will come for the rest. This is, however, a good moment to stop and consider the "scope" of a single unit-level Statement in TDD. Is it method scope? Class scope? Feature scope?

Let's try to answer the question by examining some TDD unit-level Statements:

Is it class scope?

Let's see the first example and try to answer this question:

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

  //WHEN
  var exceptionThrown = Assert.Throws<CustomException>(
    () => validation.Perform(emptyString)
  );

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

This is an example of a well-written unit-level Statement. Ok, so let's see… how many real classes take part in this spec? Three: a string, an exception and the validation. So the class scope is not the most accurate description.

Or a method scope?

So, maybe the scope covers a single method, meaning a Statement always exercises one method of a specified object?

Let's consider the following example:

```
[Fact] public void
ShouldBeFulfilledWhenEventOccursThreeTimes()
{
  //GIVEN
  var rule = new FullQueueRule();
  rule.Queued();
  rule.Queued();

  //WHEN
  rule.Queued();

  //THEN
  Assert.True(rule.IsFulfilled());
}
```

Count with me: how many methods are called? Depending on how we count, it is two (`Queued()` and `IsFulfilled()`) or four (`Queued(), Queued(), Queued(), IsFulfilled()`). In any case, not one. So it is not method scope either.

It is the scope of class behavior!

The proper answer is: behavior! Each TDD Statement specifies a single behavior. I like how Amir Kolsky and Scott Bain[13] phrase it, by saying that each unit-level Statement should "introduce a behavioral distinction not existing before".

It may look that "behavior" scope is broader than method or class-level scope, since such Statement can span multiple classes and multiple methods. This is only partially true. That is because e.g. Statements with method scope can span multiple behaviors (which, by the way, is a sign of poorly written Statement). Let's take a look at an example:

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

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

  //WHEN
  var resultForLength3
    = bufferSizeRule.CanHandle(Any.StringOfLength(3));
  //THEN
  Assert.True(resultForLength3);

  //WHEN again?
  var resultForLength4
    = bufferSizeRule.CanHandle(Any.StringOfLength(4))
  //THEN again?
  Assert.False(resultForLength4);

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

Note that it specifies three (or two – depending on how you count) behaviors: acceptance of string of allowed size, refusal of handling string above the allowed size and a special case of null string. As I said – this is an antipattern and is sometimes called a “check-it-all test”. The issue with this kind of Statement is that it can be evaluated to false for at least two reasons – when the allowed string size changes and when null handling is done in another way. Also, xUnit tools by default stop execution on first error, so, assuming that the first assertion fails, we will not know the outcome of the next assertion unless we fix the previous one (does that mean we have to use a single `Assert` per Statement? We will take care of this question later. For now, let the answer be: “not necessarily”).

On the other hand, Statements with behavior scope do not necessary have to be broader than those with class scope. Let’s take the following example that proves it:

```
[Fact] public void
ShouldReportItIsStartedAndItDoesNotYetTransmitVoiceWhenItStarts()
{
  //GIVEN
  var call = new DigitalCall();
  call.Start();

  //WHEN
  var callStarted = call.IsStarted;

  //THEN
  Assert.True(callStarted);

  //WHEN-THEN
  Assert.Throws<Exception>(
    () => call.Transmit(Any.InstanceOf<Frame>())
  );
}
```

Again, there are two behaviors here: reporting the call status after start and not being able to transmit frames after start. That is why this test should be split into two.

How to catch that you are writing a Statement about two or more behaviors rather than one? First, take a look at the test name – if it looks strange and contains some “And” or “Or” words, it may (but does not have to) be about more than one behavior. Another way is to write the description of a behavior in a Given-When-Then way. If you have more than one item in the “When” section or the structure is not Given-When-Then, but rather a “Given-When-Then-When-Then” – that is also a signal.

Specifying Boundaries and Conditions

A Disclaimer

Before I begin, I have to admit that this chapter is mostly based on the material from a series of posts by Scot Bain and Amir Kolsky from the blog Sustainable Test-Driven Development and their upcoming book by the same title. I like their idea of boundaries so much that I just follow the guidelines they outlined. This chapter is going to be a rephrase of these guidelines. I placed it here so that you have all the important topics covered in one place, but I encourage you to read the original blog posts on this subject on http://www.sustainabletdd.com/ (and the upcoming book).

Sometimes, anonymous value is not enough

When we specify a behavior, there are times when this behavior should be the same no matter what arguments we pass to the constructor or invoked methods. An example would be an addition of two numbers – whatever numbers we would supply, the answer would always be a sum of those numbers:

```
[Fact] public void
ShouldCalculateTheSumOfTwoNumbers()
{
  //GIVEN
  var a = Any.Integer();
  var b = Any,Integer();

  //WHEN
  var result = new Sum().Of(a, b);

  //THEN
  Assert.Equal(a + b, result);
}
```

In this case, the integer numbers can really be "any" – the described relationship between input and output is independent of the actual values we use. As indicated in one of the previous chapters, this is the canonical case where Constrained Non-Determinism applies.

Sometimes, however, objects exhibit different behaviors based on what is passed to their constructor and to their methods (OK, we also have static methods and singletons. Longer discussion on those will be included in one of the next chapters – for now we can safely ignore them). For example, in our application we may have a licensing policy where a feature is allowed to use only if the license is valid, and denied after it has expired. Another example would be that some shops are open from 10:00 to 18:00, 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.

In such cases, Scott and Amir offer us other guidelines for choosing input values.

Exceptions to the rule

There are times, when a Statement is true for every value except one (or more) explicitly specified. For example, in Poland, high school students are graded for exams between “mediocre” and “very good”. Every grade means the exam is passed except for “mediocre” grade which means the exam is failed. If we imagine we have to specify an object that decides whether the exam is passed based on rating, we would have to write two Statements: one for the mediocre rating and other for all the others.

Here is the Statement for mediocre grade (let’s imagine that all grades are members of an enum):

```
[Fact] public void
ShouldNotPassTheExamWhenGradeIsMediocre()
{
  //GIVEN
  var decision = new PassOrNotDecision();

  //WHEN
  var examPassed = decision.MakeBasedOn(Grades.Mediocre);

  //THEN
  Assert.False(examPassed);
}
```

Note that here, we used the literal value of `Grades.Mediocre`. This is because no other value gives the same behavior as this one, so generating the value would not make any sense.

The second Statement is for all the other cases. Here, we are going to use another method of the `Any` class for generating any enum member other than specified:

```
[Fact] public void
ShouldPassTheExamWhenGradeIsOtherThanMediocre()
{
  //GIVEN
  var decision = new PassOrNotDecision();

  //WHEN
  var examPassed
    = decision.MakeBasedOn(Any.Besides(Grades.Mediocre));

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

Here, `Any.Besides()` takes care of the enum value generation, producing a nice, readable code as a side effect.

The example shown above assumes there is only one exception to the rule (the mediocre grade). However, this concept can be scaled up to more values, as long as it is a finished, discrete set. If there are multiple exceptional values that produce the same behavior, a single Statement is sufficient to cover them all (using `Any` class and making the following call for exception Statement: `Any.Of(value1, value2)` and the following for the rest: `Any.OtherThan(value1, value2)`). However, when there are multiple exceptions to the rule and each one triggers a different behavior, each one deserves its own Statement.

Rules valid within boundaries

Sometimes, a behavior varies around a numerical boundary. The simplest example would be a set of rules on how to calculate an absolute value of a number:

1. for any X less than `0`, the result is -X (e.g. absolute value of `-1.5` is `1.5`)
2. for any X greater or equal to `0`, the result is X (e.g. absolute value of `3` is `3`).

As you can see, there is a boundary between the two behaviors and the right edge of the boundary is `0`. Why do I say "right edge"? That is because the boundary always has two edges and there's a length between them. If we assume we are talking about the mentioned absolute value calculation and that our numerical domain is that of integer numbers, we can as well use `-1` as edge value and say that:

1. for any X less or equal to `-1`, the result is -X (e.g. absolute value of `-1.5` is `1.5`)
2. for any X greater than `-1`, the result is X (e.g. absolute value of `3` is `3`).

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 case of our example, the length between `-1` (left edge – last negated number) and `0` (right edge – first non-negated) is `1`.

Now, imagine that we are not talking about integer values anymore, but about floating point values. Then the right edge value would still be 0. But what about left edge? It would not be possible for it to stay -1, because the rule applies to e.g. -0.9 as well. So what is the correct right edge value and the correct length of the boundary? Would the length be 0.1? Or 0.001? Or maybe 0.00000001? This is harder to answer and depends heavily on the context, but it is something that must be answered for each particular case – this way we know what kind of precision is expected of us. In our Specification, we have to document the boundary length somehow.

So the next topic is: how to describe the boundary length with Statements? To illustrate this, I want to show you two Statements assuming we're implementing the mentioned absolute value calculation for integers. The first Statement is for values smaller than 0 and we want to use the left edge value here like this:

```
[Fact] public void
ShouldNegateTheNumberWhenItIsLessThan0()
{
  //GIVEN
  var function = new AbsoluteValueCalculation();
  var lessThan0 = 0 - 1;

  //WHEN
  var result = function.PerformFor(lessThan0);

  //THEN
  Assert.Equal(-lessThan0, result);
}
```

And the next Statement for values at least 0 and we want to use the right edge value:

```
[Fact] public void
ShouldNotNegateTheNumberWhenItIsGreaterOrEqualTo0()
{
  //GIVEN
  var function = new AbsoluteValueCalculation();
  var moreOrEqualTo0 = 0;

  //WHEN
  var result = function.PerformFor(moreOrEqualTo0);

  //THEN
  Assert.Equal(moreOrEqualTo0, result);
}
```

There are two things to note about these examples. The first one is that we don't use any kind of `Any` methods. We explicitly take the edges, because they're the numbers that most strictly define the boundary. This way we document the boundary length.

It is important to understand why we are not using methods like `Any.IntegerGreaterOrEqualTo(0)`, even though we do use `Any` in case when we have no boundary. This is because in the latter case, no value is better than the other in any particular way. In case of boundaries, however, the edge values are better in that they more strictly define the boundary and drive the right implementation.

The second thing to note is the usage of literal constant `0` in the above example. In one of the previous chapter, I showed you a technique called **Constant Specification**, where we write an explicit Statement about the value of the named constant and use the named constant everywhere else instead of its literal. So why did i not use this technique?

The only reason is that this might have looked a little bit silly with such extremely trivial example as calculating absolute value. 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
ShouldIncludeSmallestValueNotToNegateSetToZero()
{
  Assert.Equal(0, Constants.SmallestValueNotToNegate);
}
```

Now we have got everything we need to rewrite the two Statements we wrote earlier. The first would look like this:

```
[Fact] public void
ShouldNegateTheNumberWhenItIsLessThanSmallestValueNotToNegate()
{
  //GIVEN
  var function = new AbsoluteValueCalculation();
  var lastNumberToNegate
    = Constants.SmallestValueNotToNegate - 1;

  //WHEN
  var result = function.PerformFor(lastNumberToNegate);

  //THEN
  Assert.Equal(-lastNumberToNegate, result);
}
```

And the second Statement for values at least `0`:

```
[Fact] public void
ShouldNotNegateNumberWhenItIsGreaterOrEqualToSmallestValueNotToNegate()
{
  //GIVEN
  var function = new AbsoluteValueCalculation();

  //WHEN
  var result = function.PerformFor(
    Constants.SmallestValueNotToNegate
  );

  //THEN
  Assert.Equal(
    Constants.SmallestValueNotToNegate, result);
}
```

As you can see, the first Statement contains the following expression: `Constants.SmallestValueNotToNegate - 1`, where `1` is the exact length of the boundary. Like I said, the situation is so trivial that it may look silly and funny, however, in real life scenarios, this is a great technique to apply anytime, anywhere.

Boundaries may look like they apply only to integer input, but they occur at many other places. There are boundaries associated with date/time (e.g. an action is performed again when time from last action is at least 30 seconds – the decision would need to be made whether we need precision in seconds or maybe in ticks), to 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) etc.

Combination of boundaries – ranges

So, what about a behavior that is valid in a range? Let's assume that we live in a country where a citizen can get a driving license only after their 17th birthday, but before 65th (the government decided that people after 65 have worse sight and it's safer not to give them new driving licenses). Let's also assume that we try to develop a class that answers the question whether we can apply for driving license and the return values of this query are as follows:

1. Age < 17 – returns enum value `QueryResults.TooYoung`
2. 17 <= age >= 65 – returns enum value `QueryResults.AllowedToApply`
3. Age > 65 – returns enum value `QueryResults.TooOld`

Now, remember I told you that we specify the behaviors near boundaries? This, however, when applied to the situation I just described, would give us 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, we would describe the behavior where the query should return `AllowedToApply` value twice, which effectively means that we would copy-paste the Statement and change just one value. How do we deal with this? Thankfully, we have a solution available:

Most xUnit frameworks provide some kind of data-driven test functionality, which we can use to write parameterized Statements. The functionality basically means that we can write the code of the Statement once, but make the xUnit framework invoke it twice with different sets of input values. Let's see an example in XUnit.net:

```
[Theory]
[InlineData(17, QueryResults.TooYoung)]
[InlineData(18, QueryResults.AllowedToApply)]
[InlineData(65, QueryResults.AllowedToApply)]
[InlineData(66, QueryResults.TooOld)]
public void ShouldYieldResultForAge(int age, QueryResults expectedResult)
{
  //GIVEN
  var query = new DrivingLicenseQuery();

  //WHEN
  var result = query.ExecuteFor(age);

  //THEN
  Assert.Equal(expectedResult, result);
}
```

This way, there is only one Statement executed four times. The case of `AllowedToApply` is still evaluated twice for both edge cases (so there is more time spent on executing it, which for small cases is not an issue), but the code maintenance issue is gone – we don't have to copy-paste the code to specify both edges of the behavior.

Note that we're quite lucky because the specified logic is strictly functional (i.e. returns different results based on different inputs). Thanks to this, 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 where we can set alarm time. The class allows us to set hour between `0` and `24`, but otherwise throws an exception.

While some xUnit frameworks, like NUnit, allow us to handle both cases with one Statement by writing something like this:

```
//NOTE: this is an example in NUnit framework!
[TestCase(Hours.Min, Result=Hours.Min)]
[TestCase(Hours.Max, Result=Hours.Max)]
[TestCase(Hours.Min-1, ExpectedException = typeof(OutOfRangeException))]
[TestCase(Hours.Max+1, ExpectedException = typeof(OutOfRangeException))]
public int
ShouldBeAbleToSetHourBetweenMinAndMaxButNotOutsideThatRange(
  int inputHour)
{
  //GIVEN
  var clock = new Clock();
  clock.SetAlarmHour(inputHour);

  //WHEN
  var setHour = clock.GetAlarmHour();

  //THEN
  return setHour;
}
```

Others, like XUnit.NET, don't (not that it's a defect of the framework, it's just that the philosophy behind those two is a little bit different and that some features have hidden price attached to their usage which some frameworks are willing to pay, while others aren't). Thus, we have to solve it by writing two parameterized Statements – one where a value is returned (for valid cases) and one where 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)
  );
}
```

Summary

In this chapter, we covered specifying numerical boundaries with a minimal amount of code, so that the Specification is more maintainable and runs fast. 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 mocks.

Triangulation

A disclaimer

The first occurrence of the term triangulation I know about is in Kent Beck's book Test-Driven Development: By Example[14].

As one of the last topics of the core TDD techniques that do not require us to delve into the object-oriented world, I'd like to show you triangulation.

Triangulation is often described as the most conservative of three approaches of test-driving implementation. These approaches are:

1. Type the obvious implementation
2. Fake it ('til you make it)
3. Triangulate

All of these techniques are simple (triangulation being a little more complex), so I'll show you all of them one by one, putting more emphasis on triangulation:

Type the obvious implementation

The first of the three techniques is just writing an obvious implementation in response to a Statement. If the implementation is simple, this approach makes a lot of sense. Let's take a trivial example of adding two numbers:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var sum = new Sum();

  //WHEN
  var result = sum.Of(3,5);

  //THEN
  Assert.Equal(8, result);
}
```

You may remember I told you that usually we write the simplest production code that would make the Statement true. This rule would encourage us to just return 8 from the Of method, because it would be sufficient to make the Statement true. Instead, we can decide that the logic is so obvious, that we can just write it based on this one Statement:

[14] http://www.pearsonhighered.com/educator/product/Test-Driven-Development-By-Example/9780321146533.page

```
public class Sum
{
  public int Of(int a, int b)
  {
    return a+b;
  }
}
```

and this is it.

Note that I didn’t use Constrained Non-Determinism here, because its use enforces using “Just write obvious implementation” technique. In fact, most (if not all) Statements we wrote so far in previous chapters, uses this approach because of this fact. Let’s take a look at how the above Statement would look if we used Constrained Non-Determinism:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var a = Any.Integer();
  var b = Any.Integer();
  var sum = new Sum();

  //WHEN
  var result = sum.Of(a,b);

  //THEN
  Assert.Equal(a + b, result);
}
```

Here, we don’t have any choice. The most obvious implementation that would make this Statement true is the correct implementation. We are unable to return some constant value as we previously could (but we did not), because we just don’t know what the expected result is and it is strictly dependent on the input values which we don’t know as well.

Fake it (‘til you make it)

This technique is kind of funny. I do not recall myself ever using it, but it is so interesting 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.

There are two core things that we need to pay attention when using these technique:

1. Start with the simplest implementation possible (i.e. fake it), which usually is returning a literal constant. Then gradually transform the code of both Statement and implementation using variables.

2. When doing it, rely on your sense of duplication between Statement and (fake) implementation.

Let's apply Fake It to the same addition example as before (I promise, for triangulation, I will give you better one). The Statement looks the same as before:

```
[Fact] public void
ShouldAddTwoNumbersTogether()
{
  //GIVEN
  var sum = new Sum();

  //WHEN
  var result = sum.Of(3,5);

  //THEN
  Assert.Equal(8, result);
}
```

Only this time, we are going to use the simplest implementation possible. As I wrote, this simplest implementation is almost always to return a constant:

```
public class Sum
{
  public int Of(int a, int b)
  {
    return 8;
  }
}
```

The Statement evaluates to true (green) now, though the implementation is obviously wrong. BUT, the TDD cycle is not over yet. Remember that as soon as the Statement is true, refactoring phase kicks in. This is something we were ignoring silently for now (and here we will only slightly lick it). We can use the refactoring phase to remove duplication between the Statement and it's implementing 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 a sum:

```
public class Sum
{
  public int Of(int a, int b)
  {
    return 3 + 5;
  }
}
```

Note the smart trick I did. I changed duplication between implementation and expected result to duplication between implementation and input values of the Statement. After all, 3 and 5 are the exact values I used in the Statement, right? This kind of duplication is different in that it can be removed using variables (this applies not only to input variables, but basically anything we have access to prior to triggering specified behavior – constructor parameters, fields etc. in contrast to result which we normally do not know until we invoke the behavior). The duplication of number 3 can be eliminated this way:

```
public class Sum
{
  public int Of(int a, int b)
  {
    return a + 5;
  }
}
```

and we have just the number 5 duplicated, because we used variable to transfer the value of 3 from Statement method to the `Of` implementation, so we have it in one place now. let's do the same with 5:

```
public class Sum
{
  public int Of(int a, int b)
  {
    return a + b;
  }
}
```

And that's it. 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

As I wrote, triangulation is the most conservative technique, because following it involves the tiniest possible steps to arrive at the right solution. The technique is called triangulation by analogy to radar triangulation[15] 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 do not 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. Basically, it says:

1. **Indirect measurement**: derive the design from few known examples of its desired external behavior by looking at what varies in these examples and making this variability into something more general
2. **Using at least two sources of information**: start with the simplest possible implementation and make it more general **only** when you have two or more different examples (i.e. Statements that describe the desired functionality for specific 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.

Usually, when TDD is showcased on simple examples, triangulation is the primary technique used, so many novices mistakenly believe TDD is all about triangulation. This isn’t true, although triangulation is important.

Example

Suppose we want to write a logic that creates an aggregate sum of the list. Let’s assume that we have no idea how to design the internals of our custom list class so that it fulfills its responsibility. Thus, we start with the simplest example of calculating a sum of 0 elements:

```
[Fact] public void
ShouldReturn0AsASumOfNoElements()
{
  //GIVEN
  var listWithAggregateOperations
    = new ListWithAggregateOperations();

  //WHEN
  var result = listWithAggregateOperations.SumOfElements();

  //THEN
  Assert.Equal(0, result);
}
```

[15] http://encyclopedia2.thefreedictionary.com/radar+triangulation

Remember we want to write just enough code to make the Statement true. We can achieve it with just returning 0 from the `SumOfElements` method:

```
public class ListWithAggregateOperations
{
  public int SumOfElements()
  {
    return 0;
  }
}
```

This is not yet the implementation we are happy with, which makes us add another Statement – this time for a single element:

```
[Fact] public void
ShouldReturnTheSameElementAsASumOfSingleElement()
{
  //GIVEN
  var singleElement = Any.Integer();
  var listWithAggregateOperations
    = new ListWithAggregateOperations(singleElement);

  //WHEN
  var result = listWithAggregateOperations.SumOfElements();

  //THEN
  Assert.Equal(singleElement, result);
}
```

The naive implementation can be as follows:

```
public class ListWithAggregateOperations
{
  int _element = 0;

  public ListWithAggregateOperations()
  {
  }

  public ListWithAggregateOperations(int element)
  {
    _element = element;
  }

  public int SumOfElements()
  {
    return _element;
  }
}
```

We have two examples, so let's check whether we can generalize now. We could try to get rid of the two constructors now, but let's wait just a little bit longer and see if this is the right path to go (after all, I told you that we need **at least** two examples).

Let's add third example then. What would be the next more complex one? Note that the choice of next example is not random. Triangulation is about considering the axes of variability. If you carefully read the last example, you probably noticed that we already skipped one axis of variability – the value of the element. We used `Any.Integer()` where we could use a literal value and add a second example with another value to make us turn it into variable. This time, however, I decided to **type the obvious implementation**. The second axis of variability is the number of elements. The third example will move us further along this axis – so it will use two elements instead of one or zero. This is how it looks like:

```
[Fact] public void
ShouldReturnSumOfTwoElementsAsASumWhenTwoElementsAreSupplied()
{
  //GIVEN
  var firstElement = Any.Integer();
  var secondElement = Any.Integer();
  var listWithAggregateOperations
    = new ListWithAggregateOperations(firstElement, secondElement);

  //WHEN
  var result = listWithAggregateOperations.SumOfElements();

  //THEN
  Assert.Equal(firstElement + secondElement, result);
}
```

And the naive implementation will look like this:

```
public class ListWithAggregateOperations
{
  int _element1 = 0;
  int _element2 = 0;

  public ListWithAggregateOperations()
  {
  }

  public ListWithAggregateOperations(int element)
  {
    _element1 = element;
  }

  //added
  public ListWithAggregateOperations(int element1, int element2)
  {
    _element1 = element1;
    _element2 = element2;
  }

  public int SumOfElements()
  {
    return _element1 + _element2; //changed
  }
}
```

After adding and implementing the third example, the variability of elements count becomes obvious. Now that we have three examples, we see even more clearly that we have redundant constructors and redundant fields for each element in the list and if we added a fourth example for three elements, we'd have to add another constructor, another field and another element of the sum computation. Time to generalize!

How do we encapsulate the variability of the element count so that we can get rid of this redundancy? A collection! How do we generalize the addition of multiple elements? A foreach loop through the collection! Thankfully, C# supports `params` keyword, so let's use it to remove the redundant constructor like this:

```
public class ListWithAggregateOperations
{
  int[] _elements;

  public ListWithAggregateOperations(params int[] elements)
  {
    _elements = elements;
  }

  public int SumOfElements()
  {
    //changed
    int sum = 0;
    foreach(var element in _elements)
    {
      sum += element;
    }
    return sum;
  }
}
```

While the first Statement (“no elements”) seems like a special case, the remaining two – for one and two elements – seem to be just two variations of the same behavior (“some elements”). Thus, it is a good idea to make a more general Statement that describes this logic to replace the two examples. After all, we don’t want more than one failure for the same reason. So as the next step, I will write a Statement to replace these examples (I leave them in though, until I get this one to evaluate to true).

```
[Fact]
public void
ShouldReturnSumOfAllItsElementsWhenAskedForAggregateSum()
{
  //GIVEN
  var firstElement = Any.Integer();
  var secondElement = Any.Integer();
  var thirdElement = Any.Integer();
  var listWithAggregateOperations
    = new ListWithAggregateOperations(
        firstElement,
        secondElement,
        thirdElement);

  //WHEN
  var result = listWithAggregateOperations.SumOfElements();

  //THEN
  Assert.Equal(
   firstElement +
   secondElement +
   thirdElement, result);
}
```

This Statement uses three values rather than zero, one or two as in the examples we had. When I need to use collections with deterministic size (and I do prefer to do it everywhere where using collection with non-deterministic size would force me to use a `for` loop in my Statement), I pick 3, which is the number I got from Mark Seemann and the rationale is that 3 is the smallest number that has distinct head, tail and middle element. One or two elements seem like a special case, while three sounds generic enough.

One more thing we can do is to ensure that we didn’t write a **false positive**, i.e. a Statement that is always true due to being badly written. In other words, we need to ensure that the Statement we just wrote will ever evaluate to false if the implementation is wrong. As we wrote it after the implementation is in place, we do not have this certainty.

What we will do is to modify the implementation slightly to make it badly implemented and see how our Statement will react (we expect it to evaluate to false):

```
public int SumOfElements()
{
  //changed
  int sum = 0;
  foreach(var element in _elements)
  {
    sum += element;
  }
  return sum + 1; //modified with "+1"!
}
```

When we do this, we can see our last Statement evaluate to false with a message like "expected 21, got 22". We can now undo this one little change and go back to correct implementation.

The examples ("zero elements", "one element" and "two elements") still evaluate to true, but it's now safe to remove the last two, leaving only the Statement about a behavior we expect when we calculate sum of no elements and the Statement about N elements we just wrote.

And voilà! We have arrived at the final, generic solution. Note that the steps we took were tiny – so you might get the impression that the effort was not worth it. Indeed, this example was only to show the mechanics of triangulation – in real life, if we encountered such simple situation we'd know straight away what the design would be and we'd start with the general Statement straight away and just type in the obvious implementation. Triangulation shows its power in more complex problems with multiple design axes and where taking tiny steps helps avoid "analysis paralysis".

Summary

As I stated before, triangulation is most useful when you have no idea how the internal design of a piece of functionality will look like (e.g. even if there are work-flows, they cannot be easily derived from your knowledge of the domain) and it's not obvious along which axes your design must provide generality, but you are able to give some examples of the observable behavior of that functionality given certain inputs. These are usually situations where you need to slow down and take tiny steps that slowly bring you closer to the right design and functionality – and that's what triangulation is for!

Part 2: Test-Driven Development in Object-Oriented World

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 a real 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 everything bad in the world, and a lot of opinions in between those. 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 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[16]: "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. That is why not all chapters in this part are specifically about TDD, but some are 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 will focus on is object composability.

[16] https://groups.google.com/d/msg/growing-object-oriented-software/rwxCURI_3kM/2UcNAlF_Jh4J

Teaching one thing at a time

For several next chapters, 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 makes 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 layed the groundwork for mock objects, I’ll re-introduce TDD and write lots of tests. Please trust me and be patient.

After reading part 2, 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.

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 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 by 10%. But to encourage behaviors that give an employee a higher pay grade, such 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()
  {
    _repository.Dispose();
  }
}
```

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 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:

```
public class CompanyPolicies : IDisposable
{
  readonly SqlRepository _repository
    = 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:

```
public class CompanyPolicies : IDisposable
{
  readonly Repository _repository;

  public CompanyPolicies()
  {
    if(...)
    {
      _repository = new SqlRepository();
    }
    else
    {
      _repository = new NoSqlRepository();
    }
  }
```

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 makes the choice 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 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 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: Ha ha! 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, with the exception of 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 itself does not have to be disposable – 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
  }

  //no Dispose() method anymore
}
```

Benjamin: Cool. So, what now? Seems we have the `CompanyPolicies` class depending on repository abstraction instead of an actual implementation, like SQL repository. My guess is that 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.

As I said, the code mentioned has some serious flaws. For now, Johnny and Benjamin did not encounter a desperate need to address those flaws, but this is going to change in the next chapter.

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 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 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:

```
public class CompanyPolicies
{
  readonly Repository _repository;

  public CompanyPolicies(Repository repository)
  {
    _repository = repository;
  }

  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();
    }
  }
}
```

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 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 repository for retrieving the contractors.

Johnny: In addition, 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 is it there by 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 for 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 this behind more abstract methods that hide what actually happens, but 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 employee's methods in the current implementation?

Benjamin: Actually, no. 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 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 makes the decision on their behalf. This is because if an API of an abstraction is built around answering to 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 on 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 specific goal in mind. My hope is 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[17], when we wanted to execute a 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:

```
void triggerAlarm(Alarm* alarm)
{
  if(alarm->kind == LOUD_ALARM)
  {
    playLoudSound(alarm);
  }
  else if(alarm->kind == SILENT_ALARM)
  {
    notifyPolice(alarm);
  }
}
```

The code above makes decision based on the alarm kind which is embedded in the alarm structure:

```
struct Alarm
{
  int kind;
  //other data
};
```

[17] 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.

If the alarm kind is the loud one, it executes behavior associated with loud alarm. If this is a silent alarm, the behavior for silent alarms is executed. This seems to work. Unfortunately, if we 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 few 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 let’s bind behaviors with data. As a matter of fact, the whole paradigm of procedural programming is about separating behaviors 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 behaviors are separated, we need to query the data each time we want to pick a behavior based on it. That's why we have the 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 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 have no knowledge whatsoever about the real class of the alarm they are dealing with, yet are able to use the alarm just fine only by knowing a common abstract type – `Alarm`. If we are able to 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[18]. We saw this already as Johnny was explaining composability to

[18] More on context-independence and what these "special places" are, in the next chapters.

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 police for help instead, and a loud alarm is desired as well.

Note that besides just triggering loud or silent alarm, we have a requirement for 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;
  }

  public void Trigger()
  {
    if(/* is day */)
    {
      _dayAlarm.Trigger();
    }
    else
    {
      _nightAlarm.Trigger();
    }
  }

  public void Disable()
  {
    _dayAlarm.Disable();
    _nightAlarm.Disable();
  }
}
```

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:

```
public class HybridAlarm : Alarm
{
  private readonly Alarm _alarm1;
  private readonly Alarm _alarm2;

  public HybridAlarm(
    Alarm alarm1,
    Alarm alarm2)
  {
    _alarm1 = alarm1;
    _alarm2 = alarm2;
  }

  public void Trigger()
  {
    _alarm1.Trigger();
    _alarm2.Trigger();
  }

  public void Disable()
  {
    _alarm1.Disable();
    _alarm2.Disable();
  }
}
```

Using these two classes along with already existing alarms, we can implement the requirements by composing instances of those classes like this:

```
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 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:

```
new DayNightSwitchAlarm(
  new SilentAlarm("222-333-444"),
  new LoudAlarm());
```

which would mean triggering silent alarm during a day and loud one during 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 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 other code than 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 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.

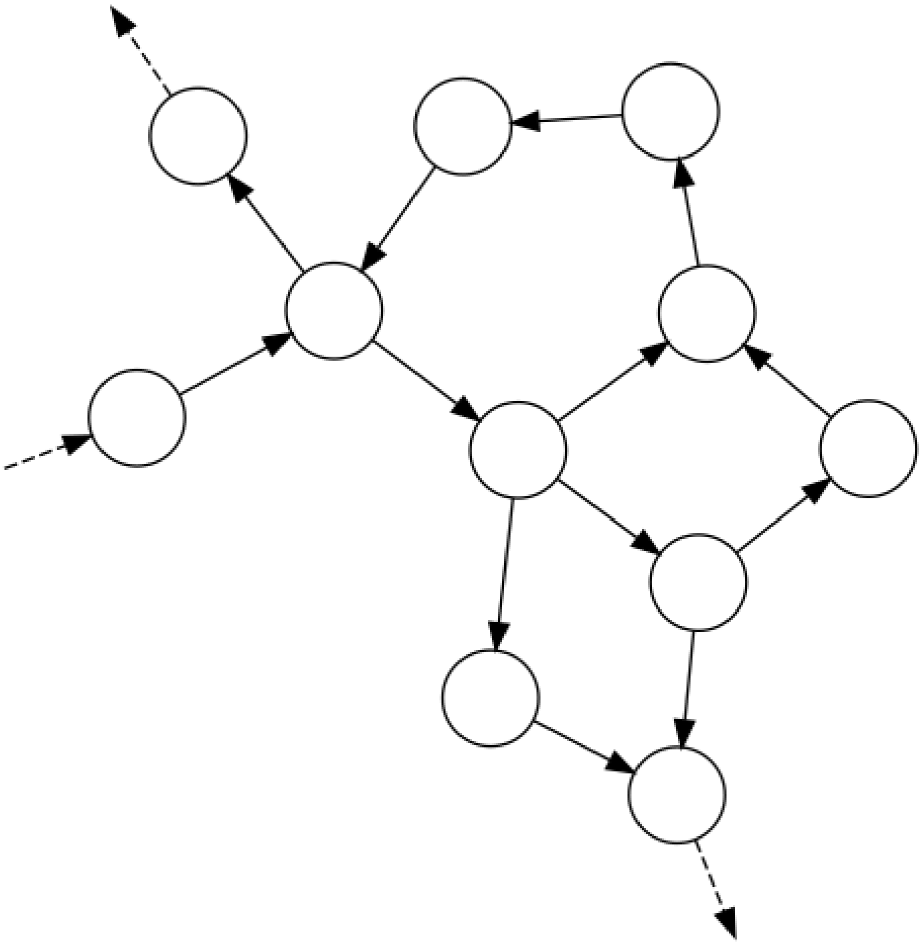
In the next chapter, we will take a closer look at composing objects itself.

Web, messages and protocols

In the previous chapter, we talked a little bit about why composability is valuable, now let's flesh a little bit of terminology to get more precise understanding.

So, again, what does it mean to compose objects?

Basically it means that an object has obtained a reference to another object and is able to 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 object-oriented world, we call a reference (and in languages such as C++, references are just that – addresses in memory).
3. A communication between sender and recipients has to obey certain **protocol**. For example, a 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 do not 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 that, when triggered, makes all lifts go to bottom floor, opens them and then disables 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.

First, we do not want the alarm to have to distinguish between an automatic and a mechanical lift – this would only add complexity to alarm system, especially that there are plans to add a third kind of lift – a more modern one, so if we made the alarm aware of the different kinds, 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.

```
public interface Lift
{
  ...
}

public class AutoLift : Lift
{
  ...
}

public class MechanicalLift : Lift
{
  ...
}
```

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 them e.g. through a constructor:

```
public class Alarm
{
  private readonly IEnumerable<Lift> _lifts;

  //obtain "addresses" through 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();
}
```

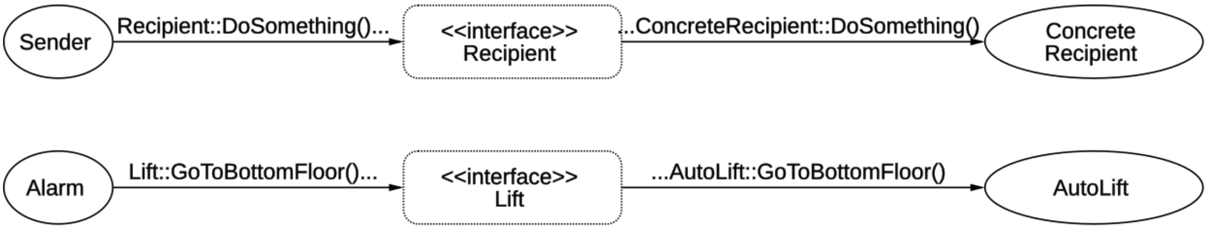
and, as a matter of fact, it sends all these messages when triggered. The `Trigger()` method on the alarm looks like this:

```
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 is sending (controlling 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 **recipients** – they do not know who they got 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 own way of reacting and `MechanicalLift` has its own), what kind of the message they received (a lift does a different thing when asked to go to 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 does, in fact, exist, it is sufficient to say that we could even write an implementation of `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 not the `Alarm`'s responsibility.


Sender
Recipient::DoSomething()...
<<interface>> Recipient
...ConcreteRecipient::DoSomething()
Concrete Recipient
Alarm
Lift::GoToBottomFloor()...
<<interface>> Lift
...AutoLift::GoToBottomFloor()
AutoLift

Sender, interface, and recipient

Ok, I hope we got that part straight. Time for some new requirements. It has been 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 `Lift`:

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).

Here is an exemplary code of `Alarm` handling the malfunction reports in its `Trigger()` method:

```
public void Trigger()
{
  foreach(var lift in _lifts)
  {
    try
    {
      lift.GoToBottomFloor();
      lift.OpenDoor();
      lift.DisablePower();
    }
    catch(LiftUnoperationalException e)
    {
      report.ThatCannotSecure(lift);
    }
  }
}
```

This is a second example of a **protocol** existing between `Alarm` and `Lift` that must be adhered to by both sides.

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. but:

- This is the way object-oriented programming inventors[19] have thought about object-oriented systems,
- It will prove useful as I explain making connections between objects and achieving strong composability in the next chapters.

[19] http://c2.com/cgi/wiki?AlanKayOnMessaging

Composing a web of objects

Three important questions

Ok, I told you that such a thing as a web of objects exists, that there are connections, protocols and such, but there is 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 not able to build a web, we do not have a web. In addition, this is a question that is a little more tricky that you may think and it contains three other questions that we need to answer:

1. When are objects composed (i.e. when connections are made)?
2. How does an object obtain a reference to another one in the web (i.e. how connections made)?
3. Where are objects composed (i.e. where connections are made)?

For now, you may have some trouble understanding the difference between those questions, but the good news is that they are the topic of this chapter, so I hope we will have that cleared shortly. Let's go!

A preview

Before we take a deep dive, let's try to answer these three questions for a really simple example code of a console application:

```
public static void Main(string[] args)
{
  var sender = new Sender(new Recipient());

  sender.Work();
}
```

And here are the answers to our questions:

1. When are objects composed? Answer: up-front, during application startup.
2. How does an object (`Sender`) obtain a reference to another one (`Recipient`)? Answer: the reference is obtained as a constructor parameter.
3. Where are objects composed? Answer: at application entry point (`Main()` method)

Depending on circumstances, we have different sets of best answers. To find them, let's take the questions on one by one.

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 this way until the application finishes executing (unless we are doing a web app – then most of the important stuff happens "per request"). Let's call this part the **static part** of the web.

Apart from that, there's a **dynamic part** – a part that undergoes constant changes – objects are created, destroyed, connected temporarily, and then disconnected. 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 composed up-front (because they do not exist yet), 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 do not have these objects up-front and they often do not live as long as the application itself. All of these objects come and go, making temporary connections.
2. There are objects that have a life span as long as the application itself, but are connected only for the needs of a single interaction (e.g. when one object is passed to a method of another as an argument) or at some point during the application runtime.

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 it is passed inside a message sent to another object (i.e. passed as method parameter).

How does a sender obtain a reference to a recipient (i.e. how connections are made)?

There are few ways this can happen, each of them useful in certain circumstances. These ways are:

1. Receive as constructor parameter
2. Receive inside a message (i.e. as a method parameter)
3. Receive in response to message (i.e. as method return value)
4. Register a recipient with already created sender

Let's have a closer look at what each of them is about and which one to choose in what circumstances.

Receive as constructor parameter

Two objects can be composed by passing one into the 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. By separating object use from construction, we end up with the code that creates a `Sender` being in a totally different place than the code that uses it. And, as `Recipient` is passed to `Sender` during its creation, it is the only place external to the `Sender` that needs to know that `Sender` uses `Recipient`. The part of code that uses `Sender` is not aware at all that `Sender` stores a reference to `Recipient` inside it. This basically means that when `Sender` is used, e.g. like this:

```
sender.DoSomething();
```

the `Sender` may then react by sending message to `Recipient`, but the code invoking the `DoSomething()` method is completely unaware of that – it is hidden. This is good, because "what you hide, you can change"[20] – e.g. if we decide that the `Sender` needs not use the `Recipient` to do its duty, the code that uses `Sender` does not need to change at all – it still looks the same as before:

```
sender.DoSomething();
```

All we have to change is the composition code to remove the `Recipient`:

[20] I got this saying from Amir Kolsky and Scott Bain

```
//no need to pass a reference to Recipient anymore
new Sender();
```

and the Sender class itself will work in a different way.

Communication of intent: required recipient

Another advantage of the constructor approach is that if a reference to Recipient is required for a Sender to work correctly and it does not make sense to create a Sender without a Recipient, the signature of the constructor makes it explicit – the compiler will not let us create a Sender without passing *something* as a Recipient.

Where to apply

Passing into constructor is a great solution in cases we want to compose sender with a recipient permanently (i.e. for the lifetime of 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 Recipient must be usable at least as long as Sender is usable. In other words, the following is nonsense:

```
sender = new Sender(recipient);
recipient.Dispose(); //but sender is unaware of it
                     //and may still use recipient in:
sender.DoSomething();
```

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 case, the objects are most often composed temporarily, just for the time of execution of this single method:

```
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 `Recipient`, in this case the user of `Sender` is explicitly responsible for supplying a `Recipient`. It may look like the coupling of user to `Recipient` is a disadvantage, but there are scenarios where it is actually **required** for code using `Sender` to be able to provide its own `Recipient` – it lets us 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, the constructor approach is better as it removes the then unnecessary coupling between code using `Sender` and a `Recipient`.

Receive in response to a message (i.e. as method return value)

This method of composing objects relies on an intermediary object – often an implementation of a factory pattern[21] – 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[22] as well (also, later in this chapter, I'll cover some aspects of 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 constructor (an approach we already discussed). For example:

```
var sender = new Sender(recipientFactory);
```

The factory can then be used by the `Sender` at will to get a hold of new recipients:

[21] http://www.netobjectives.com/PatternRepository/index.php?title=TheAbstractFactoryPattern

[22] http://en.wikipedia.org/wiki/Creational_pattern

```
public class Sender
{
  //...

  public void DoSomething()
  {
    //ask the factory for a recipient:
    var recipient = _recipientFactory.CreateRecipient();

    //use the recipient:
    recipient.DoSomethingElse();
  }
}
```

Where to apply

This kind of composition is beneficial when a new recipient is needed each time `DoSomething()` is called. In this sense it may look much like in case of 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 from 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 the factories we considered had creation methods with empty parameter list, but this is not a requirement of any sort - I just wanted to make the examples simple, so I left out everything that was not helpful in making 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[23], caches[24], builders[25], collections[26]. While they are all important topics (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 in depth on them.

Register a recipient with already created sender

This means passing a recipient to an **already created** sender (contrary to passing as constructor parameter where recipient was passed **during** creation) as a parameter of a method that stores the reference for later use. This may be a "setter" method, although I do not like naming it according to the convention "setWhatever()" – after Kent Beck[27] 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.

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 immediately used (and then forgotten), but rather only 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 with 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 solve the problem by introducing an observer[28] registration mechanism 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:

[23] http://martinfowler.com/eaaCatalog/repository.html

[24] http://en.wikipedia.org/wiki/Cache_(computing)

[25] http://www.blackwasp.co.uk/Builder.aspx

[26] If you never used collections before and you are not a copy-editor, then you are probably reading the wrong book :-)

[27] Kent Beck, Implementation Patterns

[28] http://www.oodesign.com/observer-pattern.html

```
public interface TemperatureObserver
{
  void NotifyOn(
    Temperature currentValue,
    Temperature meanValue);
}
```

Now we are 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

  public void Run()
  {
    while(/* needs to run */)
    {
      var currentValue = /* get current value somehow */;
      _meanValue = /* update mean value somehow */;

      _observer.NotifyOn(currentValue, _meanValue);

      WaitUntilTheNextMeasurementTime();
    }
  }
}
```

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 an ugly null check inside the `Run()` method). We have already seen such "null objects"[29] 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

[29] 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)

application e.g. by a network message 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 do not 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:

```
public class TemperatureSensor
{
  private TemperatureObserver _observer
    = new NullObserver(); //ignores reported values

  //... ... ...

  public void FromNowOnReportTo(TemperatureObserver observer)
  {
    _observer = observer;
  }

  //... ... ...
}
```

This lets us overwrite the observer with a new one should we ever need to do it. Note that, as I mentioned, this is the place where 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, do not 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. Placing a recipient in a constructor signature is effectively saying that “I will not work without it”. Let’s practice – just look at how the following class members signatures talk to you:

```
public class Sender
{
  //"I will not work without a Recipient1"
  public Sender(Recipient1 recipient1) {...}

  //"I will do fine without Recipient2 but you
  //can overwrite the default here to take advantage
  //of some features"
  public void Register(Recipient2 recipient2) {...}
}
```

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 screen, saved to database, 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 observers list:

```
...
foreach(var observer in _observers)
{
  observer.NotifyOn(currentValue, meanValue);
}
...
```

Another, more flexible option, is to use something like we did in the previous chapter with a `HybridAlarm` (remember? It was an alarm aggregating other alarms) – i.e. instead of introducing a collection in the sensor, we can create a special kind of observer – a "broadcasting observer" that would itself hold collection of other observers (hurrah composability!) and broadcast the values to them every time it itself receives those values:

```
public class BroadcastingObserver
  : TemperatureObserver
{
  private readonly
    TemperatureObserver[] _observers;

  public BroadcastingObserver(
    params TemperatureObserver[] observers)
  {
    _observers = observers;
  }

  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(
      new DisplayingObserver(),
      new StoringObserver(),
      new CalculatingObserver());

...

//registration:
sensor.FromNowOnReportTo(broadcastingObserver);
```

The additional benefit of modeling broadcasting as an observer is that it would let us change the broadcasting policy without touching either the sensor code or the other observers. For example, we might introduce `ParallelBroadcastObserver` that would notify each observer asynchronously instead of sequentially and put it to use by changing the composition code only:

```
//now using parallel observer
var broadcastingObserver
  = new ParallelBroadcastObserver( //change here!
      new DisplayingObserver(),
      new StoringObserver(),
      new CalculatingObserver());

sensor.FromNowOnReportTo(broadcastingObserver);
```

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 above 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 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

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 `Level`s and each of them that contain 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:

```
public static void Main(string[] args)
{
  var game = new Game();

  game.Play();
}
```

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? So now we will 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 instantiation of their castles out. As 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? The only problem is that 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 do the same to it that we did to the level classes – move the creation of levels out of it:

```
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 what we are going to do:

```
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 of time). 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. Shouldn't we fix this as well?

The answer is "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 (we call such object **a factory**, as you already know), but it's a dead end, because it would leave us with the question: where do we create the factory?
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 is important is that we reached a place where the web of objects is created using 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**[30].

We say that composition root is "as close as possible" to 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[31].

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

As I previously said, it is not 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 factory is and what are its 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.

Emerging factory – example

Consider the following code that receives a frame from the network (as raw data), then packs it into an object, validates and applies to the system:

[30] http://blog.ploeh.dk/2011/07/28/CompositionRoot/

[31] For details, check Dependency Injection in .NET by Mark Seemann.

```
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 the new instances when requested, hence the name **factory**. The factory itself can be passed through 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, the factory itself looks like this:

```
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
    {
      throw
        new InvalidRequestException(frame.Type);
    }
  }
}
```

And this is it. We have a factory now and the way we got to this point is by trying to be true to the principle of separating use from construction.

Now that we are through with the example, we are ready for some more general explanation on factories.

Reasons to use factories

As you saw in the example, factories are objects responsible for creating other objects. They are used to achieve the separation of object constructions from their use when not all of the context necessary to create an object is known up-front. We pass the part of the context we know up-front (so called **global context**) in the factory via its constructor and supply the rest that becomes available later (so called **local context**) in a form of factory method parameters when it becomes available:

```
var factory = new Factory(globalContextKnownUpFront);

//...

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):

```
var factory = new Factory(globalContext);

//...

//we need a fresh instance
factory.CreateInstance();

//...

//we need another fresh instance
factory.CreateInstance();
```

In the above example, two independent instances are created, even though both are created in an identical way (there is no local context that would differ them).

Both these reasons were present in our example:

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[32]). More often, however, when talking about simple factories, we 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 does 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)

In order for you 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". So, here are these benefits:

Factories allow creating objects polymorphically (encapsulation of type)

Each time we invoke a `new` operator, we have to put a name of a concrete type next to it:

[32]

A. Shalloway et al., Essential Skills For The Agile Developer.

```
new List<int>(); //OK!
new IList<int>(); //won't compile...
```

This means that whenver we want to use the class that does this instantiation with another concrete object (e.g. a sorted list), 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`).

Factories do not have this defficiency. Because we get objects from factories 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 is typical for a 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 abstract class. But it does not 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 some 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)
  {
    switch(rawData.MessageType)
    {
      case Messages.SessionInit:
        return new SessionInit(rawData);
      case Messages.SessionEnd:
        return new SessionEnd(rawData);
      case Messages.SessionPayload:
        return new SessionPayload(rawData);
      default:
        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 type of message:

```
var message = _messageFactory.NewInstanceFrom(rawData);
message.ValidateUsing(_primitiveValidations);
message.ApplyTo(_sessions);
```

Note that the above code does not need to change in case we want to add a new type of message that is compatible with the existing flow of processing messages[33]. The only place we need to modify in such case is the factory. For example, imagine we decided to add a session refresh message. The modified factory would look like this:

```
public class Version1ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    switch(rawData.MessageType)
    {
      case Messages.SessionInit:
        return new SessionInit(rawData);
      case Messages.SessionEnd:
        return new SessionEnd(rawData);
      case Messages.SessionPayload:
        return new SessionPayload(rawData);
      case Messages.SessionRefresh: //new message type!
        return new SessionRefresh(rawData);
      default:
        throw new UnknownMessageException(rawData);
    }
  }
}
```

[33] although it does need to change when the rule “first validate, then apply to sessions” changes

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 is less code to change when 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) [34] – 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 `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 constructors is that they are composable. This allows replacing the rule used to create objects with another one, by replacing one factory implementation with another.

In the example from the previous section, we examined a situation where we extended the existing factory with a `SessionRefresh` message. This was done with assumption that we do not need the previous version of the factory. But consider a situation where we need both versions of the behavior and want to be able to use the old version sometimes, and other times the new one. The "version 1" of the factory (the old one) would look like this:

```
public class Version1ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    switch(rawData.MessageType)
    {
      case Messages.SessionInit:
        return new SessionInit(rawData);
      case Messages.SessionEnd:
        return new SessionEnd(rawData);
      case Messages.SessionPayload:
        return new SessionPayload(rawData);
      default:
        throw new UnknownMessageException(rawData);
    }
  }
}
```

and the "version 2" (the new one) would be:

[34] Note that this is an application of Gang of Four guideline: "encapsulate what varies".

```
//note that now it is a version 2 protocol factory
public class Version2ProtocolMessageFactory
  : MessageFactory
{
  public Message NewInstanceFrom(MessageData rawData)
  {
    switch(rawData.MessageType)
    {
      case Messages.SessionInit:
        return new SessionInit(rawData);
      case Messages.SessionEnd:
        return new SessionEnd(rawData);
      case Messages.SessionPayload:
        return new SessionPayload(rawData);
      case Messages.SessionRefresh: //new message type!
        return new SessionRefresh(rawData);
      default:
        throw new UnknownMessageException(rawData);
    }
  }
}
```

Depending on what the user chooses in the configuration, we give them either a version 1 protocol support which does not support session refreshing, or a version 2 protocol support that does. Assuming the configuration is only read once during the application start, we may have the following code in our composition root:

```
MessageFactory messageFactory = configuration.Version == 1 ?
  new Version1ProtocolMessageFactory() :
  new Version2ProtocolMessageFactory() ;

var messageProcessing = new MessageProcessing(messageFactory);
```

The above code composes a `MessageProcessing` instance with either a `Version1ProtocolMessageFactory` or a `Version2ProtocolMessageFactory`, depending on the configuration.

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, we will be able to switch the rules of object creation without having to modify these objects.

Factories can hide some of the created object dependencies (encapsulation of global context)

Let's consider another simple example. We have an application that, again, can process messages. One of the things that is done with those messages is saving them in a database and another is validation. The processing of 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.Vaidate();
    message.Persist();

    //... other actions
  }
}
```

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 a 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:

```
public class MessageFactory
{
  private DataDestination _database;
  private ValidationRules _validation;

  public MessageFactory(
    DataDestination database,
    ValidationRules validation)
  {
    _database = database;
    _validation = validation;
  }

  public Message CreateFrom(MessageData data)
  {
    return
      new Message(data, _database, _validation);
  }
}
```

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.Vaidate();
    message.Persist();

    //... other actions
  }
}
```

So, instead of `DataDestination` and `ValidationRules` interfaces, the `MessageProcessing` depends only on the factory. This may not sound as 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 needs to be said 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). 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 which 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 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 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 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 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 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 loose readability.

One of the ways to do this is to use a factory[35]. We can create such 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:

[35] There are simple ways, yet none is as flexible as using factories.

```
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;
    }
    finally
    {
      saveFile.Close();
    }
  }
}
```

Now we can use the factory in the place where we are notified of the user choice. Remember? This was the place:

```
public void OnNewGame()
{
  //...
}

public void OnContinue(PathToFile savedGameFilePath)
{
  //...
}
```

When we fill the method bodies with the factory usage, the code ends up like this:

```
public void OnNewGame()
{
  var game = _gameFactory.NewGame();
  game.Start();
}

public void OnContinue(PathToFile savedGameFilePath)
{
  var game = _gameFactory.GameSavedIn(savedGameFilePath);
  game.Start();
}
```

Note that using factory helps make 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 the meaning of these terms 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 introducing terminology that can be changed by changing 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[36]. Usually it is understood as code duplication, but I consider “conceptual duplication”

[36] A. Shalloway et al., Essential Skills For The Agile Developer.

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, this is not redundancy, because the two methods represent different concepts that would change for different reasons. Even if we were to extract “common logic” from the two methods, the only sensible name we could come up with would be something like `MultiplyBy100()` which wouldn’t add any value at all.

Note that up to now, we considered four things factories encapsulate about creation of objects:

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 case, any change to how instances are created would mean a change to all classes needing 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 already convinced you that this is a pretty darn beneficial concept for such a reasonably low cost.

Summary

In this chapter, I tried to show you a variety of ways of composing objects together. Do not worry if you feel overwhelmed, for the most part, just remember to follow the principle of separating use from construction and you will be fine.

The rules outlined here apply to the overwhelming part of the objects in our application. Wait, did I say overwhelming? Not all? So there are exceptions? Yes, there are and we'll talk about them shortly, 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 the higher composability. There are numerous factors influencing 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 receipient like this:

```
public Sender(Recipient recipient)
{
  this._recipient = recipient;
}
```

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:

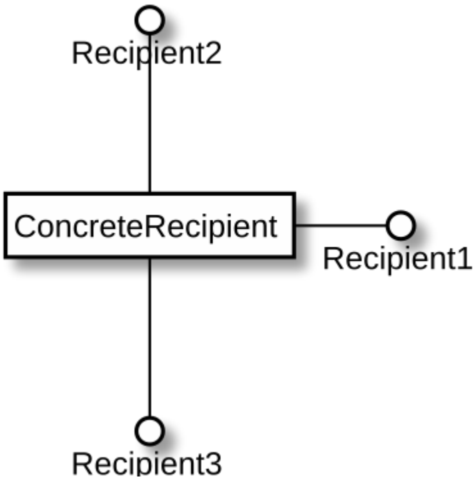
1. The recipient class may have some real dependencies. For example, if our `Recipient` depends on Windows Communication Foundation (WCF) stack, then all classes depending directly on `Recipient` will indirectly depend on WCF, including our `Sender`. The more damaging version of this problem is where 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 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?"

Actually, 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 removes 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 point of view of the sender. 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 do something to do somebody, but to indicate what happened (that's why we call them events, after all…). This fits well 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 the 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 a coherent abstractions and convey roles in the communication. This improves readability and clarity.

Small interfaces

Ok, so we said that he 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 in regard to this:

All other things equal, smaller interfaces (i.e. with less methods) are easier to implement than bigger interfaces.

The obvious conclusion from this is that if we want to have really strong composability, our "slots", i.e. interfaces, have to be as small as possible (but not smaller – see 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:

```
public void Process(Recipient recipient)
{
  recipient.DoSomething();
  recipient.DoSomethingElse();
}
```

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:

```
public class ImplementingObject
: InterfaceForSender1,
  InterfaceForSender2,
  InterfaceForSender3
{ ... }
```

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[37].

[37] http://docs.google.com/a/cleancoder.com/viewer?a=v&pid=explorer&chrome=true&srcid=0BwhCYaYDn8EgOTViYjJhYzMtMzYxMC00MzFjLWJjMzYtOGJiMDc

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:

```
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 NotifyingAdminComands : 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: `OrganizationalStructureAdminCommands`, 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 `NotifyingAdminComands` 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 enything 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 basically 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 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 users database on local system or 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 are more free to develop different implementations of this method. In addition, each implementation of the `WriteTo` method is more useful as it can be reused with different kinds of `ProducOutputs`.

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 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. As a matter of fact, 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 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:

```
if(recipient.ExtractStatusCodeFrom(response) == -1)
{
  observer.NotifyErrorOccured();
}
```

This design is a bit simplistic, but never mind. Its role is to make a certain point. Whoever the `recipient` is, it is expected to report 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 be mistakenly recognize this as 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:

```
public class WrongRecipient : Recipient
{
  public int ExtractStatusFrom(Response response)
  {
    if( /* success */ )
    {
      return -1; // but -1 is for errors!
    }
    else
    {
      return 1; // -1 should be used!
    }
  }
}
```

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, 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.

```
connection.Open();
connection.Send(data);
connection.Close();
```

Assuming the above `connection` is an implementation of `Connection` interface, if we were to implement it like this:

```
public class WrongConnection : Connection
{
  public void Open()
  {
    // imagine implementation
    // for *closing* the connection is here!!
  }

  public void Close()
  {
    // imagine implementation for
    // *opening* the connection is here!!
  }
}
```

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 certain rules that restrict the way two objects can communicate. Both sender and recipient of a message must adhere to the rules, or the 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 or 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 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 there are some heuristics that let us get closer to these qualities:

Craft messages to reflect 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, methods should reflect the intention of senders rather than capabilities of recipients.

As an example, let's look at a code for logging in that uses an instance of an `AccessGuard` class:

```
accessGuard.SetLogin(login);
accessGuard.SetPassword(password);
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 observer pattern, we don't want to say: `SetObserver(screen)`, but rather something like `FromNowOnReportCurrentWeatherTo(screen)`.

The issue of naming can be summarized as this: 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[38]: if I told you "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.

[38] 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 becomes 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 usually, 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 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 word, 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:

```
taxExpert.PayAnnualIncomeTax(
  ourIncomeDocuments,
  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:

```
taxExpert.PayAnnualIncomeTax(
  joansIncomeDocuments,
  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 basically means that each object, as an expert in its job, is not doing what is not its job, but instead relying on 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 with 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 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 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 the 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);
```

Bank is probably not a piece of data. Rather, I would imagine 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` is a piece of behavior, not data, and it itself 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 already said, there are places where Tell Don't Ask does not apply. Here are some examples from 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 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"[39]

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:

```
Radio radio = radioRepository().GetRadio(12);
var userName = radio.GetUsers().First().GetName();
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:

```
Radio radio = radioRepository().GetRadio(12);
radio.AddPrimaryUserNameTo(primaryUsersList);
```

It does not have any of the weaknesses of the previous example. Thus, it is more stable in 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?

[39] This topic is outside the scope of the book, but you can take a look at: M. Fowler, Domain-Specific Languages, Addison-Wesley 2010

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. You see, I started learning programming using structural 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 really 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 other 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:

```
public interface Session
{
  string GetOwner();
  string GetTarget();
  DateTime GetExpiryTime();
}
```

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 that:

```
// 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, 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, composability of these three classes is pretty bad, because they will have to change a lot just 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, 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[40]. If one is to make 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 `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 ugly getters from the `Session` and introduce new method called `DumpInto()` that will take a `Destination` interface implementation as a parameter:

```
public interface Session
{
  void DumpInto(Destination destination);
}
```

The implementation of `Session`, e.g. a `RealSession` can pass all fields into this destination like so:

[40] This is sometimes called Feature Envy. It means that a class is more interested in other class' data than in 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 a 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);
  }

  public void AcceptExpiryTime(DateTime expiryTime)
  {
    _row.SetCellContentFor("expiryTime", expiryTime);
  }

  public void Done()
  {
    _table.Add(_row);
  }
}
```

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:

```
public class Session
{
  string GetOwner();
  string GetTarget();
  DateTime GetExpiryTime();
}
```

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:

```
public class TimedSession : Session
{
  //...

  public void DumpInto(Destination destination)
  {
    if(!IsExpired())
    {
      destination.AcceptOwner(this.owner);
      destination.AcceptTarget(this.target);
      destination.AcceptExpiryTime(this.expiryTime);
      destination.Done();
    }
  }

  //...
}
```

and there is no need to change any other code to get this working[41].

Another added bonus of this situation that `Session` does not have to return anything from methods is that we are free to apply proxy and decorator patterns more freely to the `Session`. For example, we may have hidden sessions, that are not displayed/stored/sent at all, but retain the rest of the session functionality. We may implement the feature using a proxy, that forwards all messages it receives to the original, wrapped `Session` object, but discards the `DumpInto()` calls:

```
public class HiddenSession : Session
{
  private Session _innerSession;

  public HiddenSession(Session innerSession)
  {
    _innerSession = innerSession;
  }

  public void DoSomethig()
  {
    // forward the message to wrapped instance:
    _innerSession.DoSomething();
  }

  //...

  public void DumpInto(Destination destination)
  {
    // discard the message - do nothing
  }

  //...
}
```

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

[41] 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.

an extreme example, let's take the following interface:

```
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 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 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. Obviously, 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[42], which basically says[43]:

> 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 favourite web search engine (my recent search yielded over 74 thousands 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[44] 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.

[42] 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 ;-).

[43] http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

[44] https://stackoverflow.fogbugz.com/default.asp?W29030

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):

```
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 as for changing 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 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 Single Reponsibility 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. 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[45] 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 I said, may be spotted from code or, in some cases, known up-front from our experience as developers or domain knowledge).

The mutual relationship between Single Responsibility Principle and composability

The reason I am writing all this is that responsibilities 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’s our real goal. 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.[46].

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 UNIX commandline to pass 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 directory), `sort` (sorts passed input) and `more` (allows comfortably viewing on the screen input that takes more than one screen) into a pipeline:

[45] Provided we are not using a measure such as story points.

[46] 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 responsibilities of a class, not responsibilities of its API.

```
ls | sort | more
```

Which displays sorted content of current directory for 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 actually 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[47]. In our code, 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:

```
public class OutboundSmtpMessage
{
  //... other code

  private Encoding _encoding = new QuotedPrintableEncoding();

  //... other code
}
```

Note that each message has its own encoding objects, 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 own encoding object, since an encoding is 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.

[47] SMTP stands for Simple Mail Transfer Protocol and is a standard protocol for sending and receiving e-mail. You can read more on Wikipedia.

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:

```
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 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 message 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 static field. How do we do it? How do we achieve sharing of encodings without the constraints of 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 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 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:
      return new StmpMessage(content, _quotedPrintable);
    }
    else
    {
      //each message gets the same instance of encoding:
      return new StmpMessage(content, _base64);
    }
  }
}
```

The performance and memory saving is not exactly as big as when using a static field (e.g. each `OutboundSmtpMessage` instance uses 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 does 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 in nature 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 compositon 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 aspect 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 small amount of 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, few chapters back we dealt with dumping sessions and I showed you that such dump method may be implemented like this:

```
public class RealSession : Session
{
  //...

  public void DumpInto(Destination destination)
  {
    destination.AcceptOwner(this.owner);
    destination.AcceptTarget(this.target);
    destination.AcceptExpiryTime(this.expiryTime);
    destination.Done();
  }

  //...
}
```

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 make decision about changing application behavior when we do not understand it? And no longer than a paragraph ago we came to conclusion that just reading a class after class is not enough. We have 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 together, passing a real collaborators to each object and showing 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:

```
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()
    )
  )
);
```

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, 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 overall application description: readability. Even though the composition *is* the description of the system, it does not 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*[48], 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:

[48] 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 few places where we create `SilentAlarm`. Let's move 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 `news` in favor of something that's 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 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 functionality is to trigger both alarms when it is triggered itself. 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 loose 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 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 chaing 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 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 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, what this thing we invented really does is allowing us to combine words into *sentences*.
3. Although we are quite liberal in defining behaviors for alarms, there are some rules as 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 a 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*[49], which, by the way, is a *higher-level language*, meaning we describe our software on a higher level of abstraction.

The significance of 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"?

[49] M. Fowler, Domain-Specific Languages, Addison-Wesley 2010.

My main answer is: to deal with with complexity more effectively.

What't complexity? For our purpose we can approximate it as a 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 so larger than we are able to 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 higher level of abstraction – this is what we did in this chapter so far. The advantage is that it allows us to reduce complexity of our domain, by creating a 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 develoment 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 a good material to being structured like a language. Despite these difficulties, I think it's well worth the effort. Programming on 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. It is true that I 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 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 because 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[50]

I already briefly noted this. While it may be tempting to invent a single DSL to describe 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 a real composition. Not all parts would lend themselves to shape this way, at least not instantly. What starts off 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 are able to do so.

As Nat Pryce puts it[51], it's all about:

> (…) 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:

[50] 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.

[51] http://www.natpryce.com/articles/000783.html

```
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[52] 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:

```
new HybridAlarm(
  new SilentAlarm("222-333-444"),
  new LoudAlarm()
)
```

look like this:

[52] M. Fowler, Domain-Specific Languages, Addison-Wesley 2010.

```
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[53]:

1. Put the methods in the class that performs the composition.
2. Put the methods in superclass.

Apart from that, we can choose between:

1. Making the factory methods static.
2. Making the factory methods non-static.

[53] 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.

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 which we can inherit from. Also, a combination of the two approaches is possible, where superclass contains a more general method, while composing class wraps it with another method that adjusts the creation to the current context. By the way, remember that in most languages, we can inherit from a single class only – thus, putting methods for each language in a separate superclass forces us to distribute compositiion 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 constructor:

```
public AlarmComposition(
  HiddenDependency hiddenDependency)
{
  _hiddenDependency = hiddenDependency;
}
```

This way, I can hide it from the main composition code. This is 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 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 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 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 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"[54].

[54] 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 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 values. 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 in a different way.

What is a value?

In short, values are usually seen as immutable quantities, measurements[55] 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, 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. Some examples include: date and time (which nowadays is usually a part of 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…

[55] 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 a few 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 promotion 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, a notebook 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 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 current situation, these changes are really 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 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 sentnced 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 occurences 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 an 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 humouristic "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 of an 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[56].

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 product name.

So let's say we want to add comparison with letter case ignored. Using this option would lead us to find all places where we do something like this:

```
if(productName == productName2)
{
..
```

or

[56] All engineering decisions are trade offs anyway, so I should really 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[57] 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 determine equality based on hash code, not plain comparisons of data) 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. In fact, 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:

[57] 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 is now hidden inside the newly created static `AreEqual()` method. This method has become the only place that has knowledge of 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 `strings`s 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 I said, except from the 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[58] 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[59] from the outside world. The only available way to operate

[58] and, for C#, overriding equality operators (== and `!=`) is probably a good idea as well, not to mention `GetHashCode()` (See https://msdn.microsoft.com/en-us/library/vstudio/7h9bszxx(v=vs.100).aspx)

[59] 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 :-).

on this data is through the `ProductName`'s public interface (which exposes only those methods that we 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 name 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 ; after each method declaration. I did this because it would be a lot of code to grasp otherwise and I don't necessary 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:

```
//This is the class we created and used
//in the previous chapter
public class ProductName
  : IEquatable<ProductName>
{
  // Hidden data:
  private string _value;

  // Constructor - hidden as well:
  private ProductName(string value);

  // Static method for creating new instances:
  public static ProductName For(string value);

  // Overridden version of ToString()
  // from Object class
  public override string ToString();

  // Non-standard version of ToString().
  // I will explain its purpose later
  public string ToString(Format format);

  // For value types, we need to implement all the equality
  // methods and operators, plus GetHashCode():
  public override bool Equals(Object other);
  public bool Equals(ProductName other);
  public override int GetHashCode();
  public static bool operator ==(ProductName a, ProductName b);
  public static bool operator !=(ProductName a, ProductName b);
}
```

Using the comments, I divided the class into sections and will describe them in order.

Hidden data

The actual data is private:

```
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 with 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:

```
private ProductName(string value)
{
  _value = value;
}
```

and you probably wonder why. I'd like to decompose the question further decomposed 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[60]. 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[61]. One can argue that the following:

```
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:

[60] 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.

[61] This is literally 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.

```
// 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 favourite 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 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 case where we have different ways of initializing an object that 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 constructo that uses the common initialization
public ProductName(string model, string onfiguration)
 : 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 totally separate set of fields in each constructor:

```
public ProductName(string value)
{
  _value = value;
}

public ProductName(string model, string onfiguration)
 //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 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 very simple 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 for just assigning fields? The answer to this questions 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 values of both product names to the constant value of `"Bundle: "`. The result of such 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, thease 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 a validation on its own. Such validation may be more detailed and helpful as it is in a factory method made for 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, is sometimes may be a good idea would be 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 maye 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 personal 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, want it to use the only the "safe" methods that validate input and raise errors, not the constructor.

Separating use from construction[62]

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:

```
public static ProductName For(string value)
{
  //validations skipped for brevity
  return new ProductName(value);
}
```

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 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 database, the database API has no idea about `ProductName`, but rather accepts library types such as strings, numbers etc. In such 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);
```

[62] A. Shalloway et al., Essential Skills For The Agile Developer.

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 witj 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 (with the exception of 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 is mostly C#-specific, so I won't go into the details here, but you can always look it up in the documentation[63].

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 intentify 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.

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 with out `ProductName` class has paid off.

[63] https://msdn.microsoft.com/en-us/library/vstudio/ms131187%28v=vs.100%29.aspx

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`:

```
private ProductName(string value, string config)
{
  _value = value;
  _config = config;
}
```

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 to add additional validations to the factory method:

```
public static ProductName CombinedOf(string value, string config)
{
  if(string.IsNullOrWhiteSpace(value))
  {
    throw new ArgumentException(
          "Product names must be human readable!");
  }
  else if(string.IsNullOrWhiteSpace(config))
  {
    throw new ArgumentException(
          "Configs must be human readable!");
  }
  else
  {
    return new ProductName(value, config);
  }
}
```

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.

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

Aspects of value objects design

There are few aspects of design of value objects that I still need to talk about.

Immutability

I already said 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 hard constraint. One way or another, immutability makes an awful lot of sense for value objects. Allow me to outline just three reasons I think immutability is 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). Not that many people are aware of how hash maps work. the thing is that we act as if we were indexing something using an object, but the truth is, we are using hash codes in such cases. Let's imagine we have a dictionary indexed by instances of some elusive type, called `ValueObject`:

```
Dictionary<ValueObject, AnObject> _objects;
```

and that we are allowed to change the state of its object, e.g. by using a method `SetName()`. Now, it is a shame to admit, but there was a time when I thought that doing this:

```
ValueObject val = ValueObject.With("name");
_objects[val] = new SomeObject();

// we are mutating the state:
val.SetName("name2");

var objectIAddedTwoLinesAgo = _objects[val];
```

would give me access to the original object I put into the dictionary with `val` as a key. The impression was caused by the code `_objects[val] = new SomeObject();` looking as if I indexed the dictionary with an object, where in reality, the dictionary was merely taking the `val` to calculate its hash code and use this as a real key.

This is the reason why the above code would throw an exception, because by changing the state of the `val` with the statement: `val.SetName("name2");`, I also changed its calculated hash code, so the second time I did `_objects[val]`, I was accessing an entirely different index of the dictionary than when I did it the first time.

As it is quite common situation that value objects end up as keys inside dictionaries, it is better to leave them immutable to avoid nasty surprises.

Accidental modification by foreign code

If you have ever programmed in Java, you have to remember its `Date` class, that did behave like a value, but was mutable (with methods like `setMonth()`, `setTime()`, `setHours()` etc.).

Now, value objects are different from normal objects in that they tend to be passed a lot to many subroutines or accessed from getters. Many Java programmers did this kind of error when allowing access to a `Date` field:

```
public class ObjectWithDate {

  private final Date _date = new Date();

  //...

  public Date getDate() {
    //oops...
    return _date;
  }
}
```

It was so funny, because every user of such objects could modify the internal data like this:

```
ObjectWithDate o = new ObjectWithDate();

o.getDate().setTime(10000);
```

The reason this was happening was that the method `getDate()` returned a reference to a mutable object, so by calling the getter, we would get access to internal field.

As it was most of the time against the intention of the developers, it forced them to manually creating a copy each time they were returning a date:

```
public Date getDate() {
  return (Date)_date.clone();
}
```

which was easy to forget and may have introduced a performance penalty on cloning objects each time, even when the code that was calling the getter had no intention of modifying the date.

And that's not all, folks - after all, I said to avoid getters, so we should not have this problem, right? Well, no, because the same applies when our class passes the date somewhere like this:

```
public void dumpInto(Destination destination) {
  return destination.write(_date);
}
```

In case of this `dumpInto()` method, the `Destination` is allowed to modify the date it receives anyway it likes, which, again, was usually against developers' intention.

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 intention.

Thread safety

Mutable values cause issues when they are shared by threads, because such objects can be changed by few threads at the same time, which can cause data corruption. I stressed a few times already that value objects tend to be created many times in many places and passed along inside methods or returned as results a lot. Thus, this is a real danger. Sure, we could lock each method such a mutable value object, but then, the performance penalty could be severe.

On the other hand, when an object is immutable, there are no multithreading concerns. After all, no one is able to modify the state of an object, so there is no possibility for concurrent modifications causing data corruption. This is one of the reasons why functional languages, where data is immutable by default, gain a lot of attention in domains where running many threads is necessary.

If not mutability, then what?

There, I hope I convinced you that immutability is a great choice for value objects and nowadays, when we talk about values, we mean immutable ones. But one question remains unanswered: what about a situation when I really want to have:

- a number that is greater by three than another number?
- a date that is later by five days than another date?
- a path to a file in a directory that I already have?

If I cannot 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 state we are expecting. The old value remains unmodified.

Just to give you three examples, 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 have a date later by five days than another date:

```
string oldDate = DateTime.Now;
string newString = oldDate + TimeSpan.FromDays(5);
//oldDate is unchanged, newDate is later by 5 days
```

When I want to make a path to a file in a directory from a path to the directory[64]:

```
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, any time we want to have a value based on a previous value, instead of modifying the previous object, we create a new object with desired state.

Implicit vs. explicit handling of variability (TODO check vs with or without a dot)

As in ordinary objects, there can be some variability in values. For example, we can have money, which includes dollars, pounds, zlotys (Polish money), euros etc. Another example of something that can be modelled as a value are paths (you know, `C:\Directory\file.txt` or `/usr/bin/sh`) - here, we can have absolute paths, relative paths, paths to files and paths pointing to directories.

Contrary to ordinary objects, however, where we solved variability by using interfaces and different implementations (e.g. we had `Alarm` interface with implementing classes like `LoudAlarm` or `SilentAlarm`), in values we do it differenly. This is because the variability of values is not behavioral. Whereas the different kinds of alarms varied in how they fulfilled the responsibility of signaling they were turned on (we said they responded to the same message with, sometimes entirely different, behaviors), the diffenrent 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 Punds?"), which is not a behavioral distinction. Likewise, paths differ in what kinds of 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 does not make sense for a path pointing to a directory).

So, assuming the differences are important, how do we handle them? There are two basic approaches that I like calling implicit and explicit. Both are useful in certain contexts, depending on what exactly we want to model, so both demand an explanation.

[64] this example uses a library called Atma Filesystem: TODO hyperlink to nuget

Implicit variability

Let's imagine we want to model money using value objects[65]. Money can have different currencies, but we don't want to treat each currency in a special way. The only things that are impacted by currency are exhange rtates to other currencies. Other than this, we want every part of logic that works with money to work with each currency.

This leads us to making the differencies 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 the currency 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:

```
decimal amountOfDollarsOnMyAccount = mySavings.AmountOfDollars();
```

other than that, we are allowed to mix different currencies whenever and wherever we like[66]:

```
Money mySavings =
  Money.Dollars(100) +
  Money.Euros(200) +
  Money.Zlotys(1000);
```

And this is good, assuming all of our logic is common for all kinds of money and we do not have any special logic just for Pounds or just for Euros that we don't want to pass other currencies into by mistake.

Here, 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 usnig different types. Filesystem paths are a good example. Let's imagine the following method for creating a backup archives that accepts a destination path (for now as a string) as its input parameter:

```
void Backup(string destinationPath);
```

[65] This example is loosely based on Kent Beck's book Test-Driven Development By Example. based on TODO add reference to Kent's book

[66] 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.

This method has one obvious drawback - its signature does not tell anything about the characteristics of the destination path - is it an absolute path, or a relative path (and 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 file name is given according to some pattern (e.g. given on current date)? Or maybe file name in the path is optional and if none is given, then a default name is used? A lot of questions, isn't it?

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 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's 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. Besides that, one can pass an argument that's not even a path inside, because a string can contain any arbitrary content.

Looks like this is 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` 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 a proper path:

```
//the following throws exception because string is not proper path
Path path = Path.Value(@"C:\C:\C:\C:\//\/\/");
```

and throws an exception in place of path 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, when we have a value object, there is a high probability thet it will be used as early as possible in the chain of calls, and if we try to create a path with wrong arguments, 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.

In this case, the varying properties of paths are not just an obstacle, a problem to solve, like in case of money. They are they key differentiating factor in choosing whether a behavior is appropriate for a value or not. In such case, it makes a lot of sense to create several different value types for path, each representing a different set of constraints.

Thus, we may decide to introduce types like[67]:

[67] for reference, please take a look at TODO hyperlink

- `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 do not have to explain the constraints in the argument name - 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 a string.

Another property of making variability among values explicit is that some methods for conversions should be provided. 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 it a `FileName`). The code that does the conversion then looks like this:

```
//below dirPath is an instance of AbsoluteDirPath:
AbsoluteFilePath filePath = dirPath + FileName.Value("backup.zip");
```

Of course, we create conversion methods only where a conversion makes sense.

And that's it for the path example.

Summing up the implicit vs. explicit discussion

Note that while in the previous example (the one with money), making the variability (in currency) among values implicit helped us achieve our design goals, in this 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 path, we can treat all variations the same, since they are all of the same type. If we decide on the explicit path, we end up with several types that are usually incompatible and we allow conversions between them where such conversions make sense.

Special values

Some value types has values that are so specific asto have their own name. For example, a string value consisting of `""` is called an empty string. A XXXXXXXX (TODO put a maximum 32 bit integer here) is called "maximum 32 bit integer value".

For example, in C#, we have `Int32.Max` and `Int32.Min` which are constants representing a maximum and minimum value of 32 bit integer. `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 week.

For such values, is makes a lot of sense to make them globally accessible from the value object classes, as is done in all the above examples from C# and Java library. This is because they are immutable, so the global accessibility does not cause any hurt. For example, we can imagine `string.Empty` implemented like this:

```
public class string
{
  //...
  public const string Empty = "";
  //...
}
```

The additional `const` modifier ensures no one will assign any new value to the `Empty` field. By the way, we can use `const` only for types that have literal values, like a string. For many others, we will have to use a `static readonly` (or `final static` in case of Java) modifier. 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 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 said that if a responsibility is too big for a single object to handle, it does not try to achieve it alone, but rather distribute the work further to other objects by sending messages to them. I said that preferrably (TODO check spelling) 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 do not belong to the web of objects metaphor, alhough in almost all object-oriented languages, values are implemented using the same mechanism as objects - a class[^csharpstructs]. Values can be passed between objects in messages, but we don't talk about values sending messages.

A conclusion from this is that values cannot be composed of objects. Values can be composed of values (as our `Path` type had a `string` inside), which ensures their immutability. Also, they can occasionally can objects as parameters of their methods (remember the `ProductName` class from previous chapters? I had a method `ToString()` accepting a `Format` interface), but this is more of an exception than a rule.

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 be encapsulate domain concepts, to provide higher-level interface etc., so we do **not** want values to become plain data structures like the ones we know from C, but the nature of values is to transfer pieces of data.

As such, we expect values to contain a lot of 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

[^csharpstructs] C# has structs, which can sometimes come in handy when implementing values, especially starting from C# 5.0 where they got a bit more powerful.

TODO check whether the currencies are written uppercase in Kent's book

An object-oriented approach summary

Where are we now?

So far, we talked a lot about the object-oriented world, consisting of objects, that:

- send messages to each other using interfaces and according to protocols. Thanks to this, objects could send messages, without knowing who exactly is on the other side to handle the message
- are built around the Tell Don't Ask heuristic, so that each object has its own responsibility and handles it when its told, without conveying to anyone how it is handling the responsibility
- are built around the quality of composaibility, which lets us compose them as we would compose parts of sentences, creating higher-level languages, so that we can reuse the objects we already have as our "vocabulary" and add new functionality by combining them into new "sentences".
- are created in places well separated from the places that use those objects, depending on their lifecycle, e.g. factories and composition root.

and of values that:

- represent quantities, measurements and other discrete pieces of data that we want to name, combine with each other, transform and pass along. Examples are: dates, strings, money, time durations, path values, numbers, etc.
- are compared based on their data, not their references. Two values containing the same data are considered equal.
- are immutable - when we want to have a value like another one, but with one aspect changed, we create another value based on the previous value and the previous value remains unchanged.
- do not 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, so there is no strict rule on how to choose and different people have different preferences.

This is the world we are going to fit mock objects and other TDD practices into.

So, tell me again, why are we here?

I hope you're not mad at me because we put aside TDD for such a long time. Believe me that understanding the concepts from all the chapters from part 2 up to now is 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 "cassic TDD" or "Chicago style TDD"). Personally, I don't support this division. I like very much what Nat Pryce said about it[68]:

> (…) 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.

I hope now you understand why i put you through so many pages talking 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 not make too much sense.

[68] TODO add reference and change it into a url instead of footnote.

Mock Objects as a testing tool

Remember the beginning of this book, where I introduced mock objects and said that I lied to you about their true purpose and nature? Now that we have a lot more knowledge about the view on object-oriented design, we can truly understand where mocks come from and what thay are for.

In this chapter, I will not yet say anything about the role of mock objects in test-driving object-oriented code, just justify their place in the context of testing objects.

A backing example

Believe me I tried to write this chapter without leaning on any particular example, but the outcome was so dry and abstract, that I decided it could really use a backing example.

So for the needs of this chapter, I will use a single class, called `DataDispatch`, which has the responsibility of sending data to a destination (modeled using a `Destination` interface) which needs to be opened before the sending operation and closed after sending. Its design is very naive, but that's the purpose - to not let the example itself get in the way of explaining mock objects.

Anyway, the `DataDispatch` class is defined like this:

```
public class DataDispatch
{
  Destination _destination;

  public DataDispatch(Destination destination)
  {
    _destination = destination;
  }

  public void Dispatch(byte[] data)
  {
    _destination.Open();
    _destination.Send(data);
    _destination.Close();
  }
}
```

And the `Destination` interface is defined like this:

```
public interface Destination
{
  void Open();
  void Send(byte[] data);
  void Close();
}
```

Now we are ready to introduce mocks! Let's go!

Specifying protocols

I hope in previous chapters, I succeeded in making my point that protocols are very important. Our goal is to design them so that we can reuse them in different contexts. Thus, it makes a lot of sense to specify (remember, that's the word we are using for "test") whether an object adheres to its part of the protocol. For example, our `DataDispatch` must first open a destination, then send the data and at last, close the connection. If we rely on these calls being made in this order when we write our implementations of the `Destination` interface, we'd better specify what calls they expect to receive from `DataDispatch` and in which order, using executable Statements.

Remember from the previous chapters then I told you that we strive for context-independence when designing objects? This is true, however, it's impossible most of the time to attain complete context-independence. In case of `DataDispatch`, it knows very little of its context, which id a `Destination`, but nevertheless, it needs to know *something* about it. Thus, when writing a Statement, we need to pass an object of a class implementing `Destination` into `DataDispatch`. But which context should we use?

In other words, we can express our problem with the following, unfinished Statement (I marked all the unknowns with a double question mark: `??`):

```
[Fact] public void
ShouldSendDataToOpenedDestinationThenCloseWhenAskedToDispatch()
{
  //GIVEN
  var destination = ??;
  var dispatch = new DataDispatch(destination);
  var data = Any.Array<byte>();

  //WHEN
  dispatch.ApplyTo(data);

  //THEN
  ??
}
```

As you see, we need to pass a `Destination` to a `DataDispatch`, but we don't know what that destination should be. Likewise, we have no good idea of how to specify the expected calls and their orders.

From the perspective of `DataDispatch`, it is designed to work with different destinations, so no context is more appropriate than other. This means that we can pick and choose the one we like. Ideally, we'd like to pass a context that best fulfills the following requirements:

1. Does not add side effects of its own - when we are specifying a protocol of an object, we want to be sure that what we are making assertions on are the actions of this object itself, not its context. This is a requirement of trust - you want to trust your specifications that they are specifying what they say they do.
2. Is easy to control - so that we can easily make it trigger different behaviors in the object we are specifying, Also, we want to be able to easily verify how the specified object interacts with its context. 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 context. Also, we don't want to write special Statements for the behaviors of this context. This is a requirement of low friction.

There is a tool that fulfills these three requirements - you guessed it - mock objects!

Using a mock destination

I hope you remember the NSubstitute library for creating mocks objects that we introduce way back at the beginning of the book. We can use it now to quickly create an implementation of `Destination` that behaves the way we like, allows easy verification of protocol and between `Dispatch` and `Destination` and introduces as minimal number of side effects as possible.

Filling the gap, this is what we get:

```
[Fact] public void
ShouldSendDataToOpenedDestinationThenCloseWhenAskedToDispatch()
{
  //GIVEN
  var destination = Substitute.For<Destination>;
  var dispatch = new DataDispatch(destination);
  var data = Any.Array<byte>();

  //WHEN
  dispatch.ApplyTo(data);

  //THEN
  Received.InOrder(() =>
  {
    destination.Open();
    destination.Send(data);
    destination.Close();
  };
}
```

There, it did the trick![69] The only thing that might seem new to you is this:

```
Received.InOrder(() =>
{
  destination.Open();
  destination.Send(data);
  destination.Close();
};
```

What it does is checking whether the `Destination` got the messages (remember? Objects send messages to each other) in the right order. If we changed the implementation of the `ApplyTo()` method from this one:

```
public void Dispatch(byte[] data)
{
  _destination.Open();
  _destination.Send(data);
  _destination.Close();
}
```

to this one (note the changed call order):

```
public void Dispatch(byte[] data)
{
  _destination.Send(data);
  _destination.Open();
  _destination.Close();
}
```

The Statement will turn false (i.e. will fail).

Mocks as yet another context

What we have done in the above example was to put our `DataDispatch` in a context that was most convenient for us to use in our Statement.

Some say that specifying object interactions in a context consisting of mocks is "specifying in isolation" and that providing such mock context is "isolating". I don't like this point of view very much. From the point of view of a specified object, mocks are just another context - they are not better, nor worse, not more or 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 a production code, writes a trace message to a log.

[69] By the way, note that this protocol we are specifying is very naive, since we assume that sending data through destination will never throw any exception.

Summary

Now, wasn’t this a painless introduction to mock objects! In the next chapters, we will examine how mock objects help in test-driven development.

Further Reading

The Essential Tools

- Gerard Meszaros has written a long book about using the XUnit family of test frameworks, called XUnit Test Patterns[70]. 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[71] on C2 wiki.
- Martin Fowler's bliki mentions[72] value objects. They are also one of the patterns in his book Patterns of Enterprise Application Architecture[73]
- Arlo Beshele describes[74] 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[75] book by Kent Beck includes value object as one of the patterns.

[70] http://xunitpatterns.com/

[71] http://c2.com/cgi/wiki?ValueObject

[72] http://martinfowler.com/bliki/ValueObject.html

[73] http://martinfowler.com/books/eaa.html

[74] http://arlobelshee.com/the-no-mocks-book/

[75] http://www.isbnsearch.org/isbn/0321413091