Clean Architectures in Python

Chapter 3 - A basic example

Joshua/WOPR: Wouldn't you prefer a good game of chess?
David: Later. Let's play Global Thermonuclear War.

Wargames, 1983

The goal of the "Rent-o-Matic" project is to create a simple search engine for a room renting company. Objects in the dataset (rooms) are described by some attributes and the search engine shall allow the user to set some filters to narrow the search.

A room is stored in the system through the following values:

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

The description is purposely minimal so that we can focus on the architectural problems and how to solve them. The concepts that I will show are then easily extendable to more complex cases.

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

Project setup

Clone the project repository and move to the branch `second-edition`. The full solution is contained in the branch `second-edition-top`, and the tags I will mention are there. I strongly advise to code along and to resort to my tags only to spot errors.

```
$ git clone https://github.com/pycabook/rentomatic
$ cd rentomatic
$ git checkout --track origin/second-edition
```

Create a virtual environment following your preferred process and install the requirements

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

You should at this point be able to run

```
$ pytest -svv
```

and get an output like

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

```
collected 0 items

============================ no tests ran in 0.02s ============
```

Later in the project you might want to see the output of the coverage check, so you can activate it with

```
$ pytest -svv --cov=rentomatic --cov-report=term-missing
```

In this chapter, I will not explicitly state when 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, obviously.

You may notice that I configured the project to use black with an unorthodox line length of 75. I chose that number trying to find a visually pleasant way to present code in the book, avoiding wrapped lines that can make the code difficult to read.

Source code
https://github.com/pycabook/rentomatic/tree/second-edition

Domain models

Let us start with a simple definition of the model `Room`. 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. This test ensures that the model can be initialised with the correct values

tests/domain/test_room.py

```
import uuid
from rentomatic.domain.room import Room


def test_room_model_init():
    code = uuid.uuid4()
    room = 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 empty file `__init__.py` in every subdirectory of `tests/` that you create, in this case `tests/domain/__init__.py`.

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

rentomatic/domain/room.py

```
import uuid
import dataclasses


@dataclasses.dataclass
```

```
class Room:
    code: uuid.UUID
    size: int
    price: int
    longitude: float
    latitude: float
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s01

The model is very simple and requires little explanation. I'm using dataclasses as they are a compact way to implement simple models like this, but you are free to use standard classes and to implement the method `__init__` explicitly.

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 allows us to initialise the model from this type of structure. The code can go into the same file we created before, and is

tests/domain/test_room.py

```
def test_room_model_from_dict():
    code = uuid.uuid4()
    init_dict = {
        "code": code,
        "size": 200,
        "price": 10,
        "longitude": -0.09998975,
        "latitude": 51.75436293,
    }

    room = Room.from_dict(init_dict)
```

```
    assert room.code == code
    assert room.size == 200
    assert room.price == 10
    assert room.longitude == -0.09998975
    assert room.latitude == 51.75436293
```

A simple implementation of it is then

rentomatic/domain/room.py

```
@dataclasses.dataclass
class Room:
    code: uuid.UUID
    size: int
    price: int
    longitude: float
    latitude: float

    @classmethod
    def from_dict(cls, d):
        return cls(**d)
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s02

For the same reason mentioned before, it is useful to be able to convert the model into a dictionary, so that we can easily serialise it into JSON or similar language-agnostic formats. The test for the method `to_dict` goes again in `tests/domain/test_room.py`

tests/domain/test_room.py

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

    room = Room.from_dict(init_dict)

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

and the implementation is trivial using dataclasses

rentomatic/domain/room.py

```
    def to_dict(self):
        return dataclasses.asdict(self)
```

If you are not using dataclasses you need to explicitly create the dictionary, but that doesn't pose any challenge either. Note that this is not yet a serialisation of the object, as the result is still a Python data structure and not a string.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s03

It is also very useful to be able to compare instances of a model. The test goes in the same file as the previous test

tests/domain/test_room.py

```
def test_room_model_comparison():
    init_dict = {
        "code": uuid.uuid4(),
        "size": 200,
        "price": 10,
        "longitude": -0.09998975,
        "latitude": 51.75436293,
    }

    room1 = Room.from_dict(init_dict)
    room2 = Room.from_dict(init_dict)

    assert room1 == room2
```

Again, dataclasses make this very simple, as they provide an implementation of `__eq__` out of the box. If you implement the class without using dataclasses you have to define this method to make it pass the test.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s04

Serializers

Outer layers can use the model `Room`, 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 APIs. 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.

This is the test for the JSON serialization of our class `Room`

tests/serializers/test_room.py

```
import json
import uuid

from rentomatic.serializers.room import RoomJsonEncoder
from rentomatic.domain.room import Room


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

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

    expected_json = f"""
        {{
            "code": "{code}",
            "size": 200,
            "price": 10,
            "longitude": -0.09998975,
            "latitude": 51.75436293
        }}
    """

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

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

Here we create the object `Room` and write the expected JSON output (please note that the double curly braces are used to avoid clashes with the f-string formatter). Then we dump the object `Room` 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 file `rentomatic/serializers/room.py` the code that makes the test pass

rentomatic/serializers/room.py

```
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:  # pragma: no cover
            return super().default(o)
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s05

Providing a class that inherits from `json.JSONEncoder` let us use the syntax `json_room = json.dumps(room, cls=RoomJsonEncoder)` to serialize the model. Note that we are not using the method `as_dict`, 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 method `as_dict` and then adjust the code field converting it with `str`.

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.

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 code 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 the latter. Put this code in the file `tests/use_cases/test_room_list.py`

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

tests/use_cases/test_room_list.py

```
import pytest
import uuid
from unittest import mock

from rentomatic.domain.room import Room
from rentomatic.use_cases.room_list import room_list_use_cas


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

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

    room_3 = Room(
        code=uuid.uuid4(),
        size=56,
        price=60,
        longitude=0.27891577,
        latitude=51.45994069,
```

```
    )

    room_4 = 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

    result = room_list_use_case(repo)

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

The test is straightforward. First, we mock the repository so that it provides a method `list` 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 method `list` 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 file `rentomatic/use_cases/room_list.py`

rentomatic/use_cases/room_list.py

```
def room_list_use_case(repo):
    return repo.list()
```

Such a solution might seem too simple, so let's discuss it. First of all, this use case is just a wrapper around a specific function of the repository, and it 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.

The next thing you might notice is that I used a simple function. In the first edition of this book I used a class for the use case, and thanks to the nudge of a couple of readers I started to question my choice, so I want to briefly discuss the options you have.

The use case represents the business logic, a process, which means that the simplest implementation you can have in a programming language is a function: some code that receives input arguments and returns output data. A class is however another option, as in essence it is a collection of variables and functions. So, as in many other cases, the question is if you should use a function or a class, and my answer is that it depends on the degree of complexity of the algorithm that you are implementing.

Your business logic might be complicated, and require the connection with several external systems, though, each one with a specific initialisation, while in this simple case I just pass in the repository. So, in principle, I don't see anything wrong in using classes for use cases, should you need more structure for your algorithms, but be careful not to use them when a simpler solution (functions) can perform the same job, which is the mistake I made in the previous version of this code. Remember that code has to be maintained, so the simpler it is, the better.

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s06

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 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 fourth 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 to 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 to 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 elements 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`.

tests/repository/test_memrepo.py

```
import pytest

from rentomatic.domain.room import Room
from rentomatic.repository.memrepo 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(room_dicts)

    rooms = [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 method `list`. The implementation that passes the test goes in the file `rentomatic/repository/memrepo.py`

```
rentomatic/repository/memrepo.py
```

```
from rentomatic.domain.room import Room


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

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

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s07

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, its basic structure would remain 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 in-

terface 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 through 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.cfg`. 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

cli.py

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

from rentomatic.repository.memrepo import MemRepo
from rentomatic.use_cases.room_list import room_list_use_cas

repo = MemRepo([])
result = room_list_use_case(repo)

print(result)
```

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s08

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

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

which is correct as the class `MemRepo` in the file `cli.py` 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 most important part of the script is

cli.py

```
repo = MemRepo([])
result = room_list_use_case(repo)
```

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

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

cli.py

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

from rentomatic.repository.memrepo import MemRepo
from rentomatic.use_cases.room_list import room_list_use_cas

rooms = [
    {
```

```
        "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,
    },
]

repo = MemRepo(rooms)
result = room_list_use_case(repo)
```

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

Source code
https://github.com/pycabook/rentomatic/tree/ed2-c03-s09

Again, remember that we need to hardcode data due to the trivial nature of our storage, and not to the architecture of the system. Note that I changed the instruction `print` as the repository returns domain models and 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,
    '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
  }
]
```

Please note that I formatted the output above to be more readable, but the actual output will be on a single line.

What we saw in this chapter is the core of the clean architecture in action.

We explored the standard layers of entities (the class `Room`), use cases (the function `room_list_use_case`), gateways and external systems (the class `MemRepo`) and we could start to appreciate the advantages of their separation into layers.

Arguably, what we designed is very limited, which is why I will dedicate the rest of the book to showing how to enhance what we have to deal with more complicated cases. We will discuss a **Web interface** in chapter 4, a **richer query language** and **error management** in chapter 5, and the **integration with real external systems** like databases in chapters 6, 7, and 8.

Previous
Chapter 2 - Components of a clean architecture

Next
Chapter 4 - Add a web application

Last update: 20/08/2021

Section 5 of Clean Architectures in Python

Previous

- Introduction
- About the book
- Chapter 1 - A day in the life of a clean system
- Chapter 2 - Components of a clean architecture

Next

- Chapter 4 - Add a web application
- Chapter 5 - Error management
- Chapter 6 - Integration with a real external system - Postgres
- Chapter 7 - Integration with a real external system - MongoDB
- Chapter 8 - Run a production-ready system
- Changelog
- Colophon