Clean Architectures in Python

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 to 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 cleanness in favour of execution speed, as you cannot afford to waste resources.

In these cases, break the rules.

With methodologies you are always free to keep the parts you consider useful and discard the rest, and if you have understood the reason behind the methodology, you will also be aware of the reasons that support your decisions. 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.

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 all these 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 inputs 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 go over another 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 (if they found what they were looking for). 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 it. 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, who are underemployed waiting for clients because we overestimated the size of the business. Or it might show that the time taken to serve clients 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 to 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 allocate (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 hardware 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"? ¶

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 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 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 "architectures"? ¶

While I was writing the first edition of the book it became clear to me that the goal of this book is to begin a journey and not to define the specific steps through which each software designer has to go through. The concepts explained here are rooted in some design principles that are much more important than the resulting physical structure of the system that you will create.

This is why I wanted to stress that what I show in this book can (and hopefully will) be an inspiration for many different architectures that you will create to solve the problems you will have to face.

Or maybe I just wanted to avoid looking like a clone of Robert Martin.

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 project I will discuss.

The clean architecture concepts are independent of 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, who proofread the introduction.
- Roberto Ciatti, who introduced me to clean architectures.
- Readers Eric Smith, Faust Gertz, Giovanni Natale, Grant Moore, Hans Chen, Max H. Gerlach, Michael O'Neill, Paul Schwendenman, Ramces Chirino, Rodrigo Monte, Simon Weiss, Thiago C. D'Ávila, robveijk, mathisheeren, 4myhw, Jakob Waibel, 1110sillabo, Maxim Ivanov who fixed bugs, typos and bad grammar submitting issues and pull requests.
- Łukasz Dziedzic, who developed the free "Lato" font (http://www.latofonts.com), used for the cover.

The cover photograph is by pxhere. 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.

1 From the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Vlissides, Johnson, and Helm.

Next
About the book

Section 1 of Clean Architectures in Python

Next