Clean Architectures in Python

# Chapter 2 - Components of a clean architecture

## 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

In this chapter I will analyse the set of software design principles collectively known as "clean architecture". While this specific name has been introduced by Robert Martin, the concepts it pushes are part of software engineering, and have been successfully used for decades.

Before we dive into a possible implementation of them, which is the core of this book, we need to analyse more in depth the structure of the clean architecture and the components you can find in the system designed following it.

### Divide et impera

One of the main goals of a well designed system is to achieve control. From this point of view, a software system is not different from a human working community, like an office or a factory. In such environments there are workers who exchange data or physical objects to create and deliver a final product, be it an object or a service. Workers need information and resources to perform their own job, but most of all they need to have a clear picture of their responsibilities.

While in a human society we value initiative and creativity, however, in a machine such as a software system, components shouldn't be able to do anything that is not clearly stated when the system is designed. Software is not alive, and despite the impressive achievements of artificial intelligence in the latter years, I still believe there is a spark in a human being that cannot be reproduced by code alone.

Whatever our position on AIs, I think we all agree that a system works better if responsibilities are clear. Whether we are dealing with software or human communities, it is always dangerous to be unclear about what a component can or should do, as areas of influence and control naturally overlap. This can lead to all sorts of issues, from simple inefficiencies to complete deadlocks.

A good way to increase order and control in a system is to split it into subsystems, establishing clear and rigid borders between them, to regulate the data exchange. This is an extension of a political concept (divide et impera) which states that it is simpler to rule a set of interconnected small systems than a single complex one.

In the system we designed in the previous chapter, it is always clear what a component expects to receive when called into play, and it is also impossible (or at least, forbidden) to exchange data in a way that breaks the structure of the system.

You have to remember that a software system is not exactly like a factory or an office. Whenever we discuss machines we have to consider both the way they work (run time) and the way they have been built or will be modified (development time). In principle, computers don't care where data comes from and where it goes. Humans, on the other hand, who have to build and maintain the system, need a clear picture of the data flow to avoid introducing bugs or killing performances.

## Data types

An important part in a system is played by data types, that is the way we encapsulate and transmit information. In particular, when we discuss software systems, we need to make sure that types that are shared by different systems are known to all of them. The knowledge of data types and formats is, indeed, a form of coupling. Think about human languages: if you have to talk to an audience, you have to use a language they understand, and this makes you coupled with your audience. This book is written (tentatively) in English, which means that I am coupled with English-speaking readers. If all English speakers in the world suddenly decided to forget the language and replace it with Italian I should write the book from scratch (but with definitely less effort).

When we consider a software system, thus, we need to understand which part defines the types and the data format (the "language"), and ensure that the resulting dependencies don't get in the way of the implementer. In the previous chapter we discovered that there are components in the system that should be considered of primary importance and represent the core of the system (use cases), and others which are less central, often considered implementation details. Again, mind that calling them "details" doesn't mean they are not important or that they are trivial to implement, but that replacing them with different implementations does not affect the core of the system (business logic).

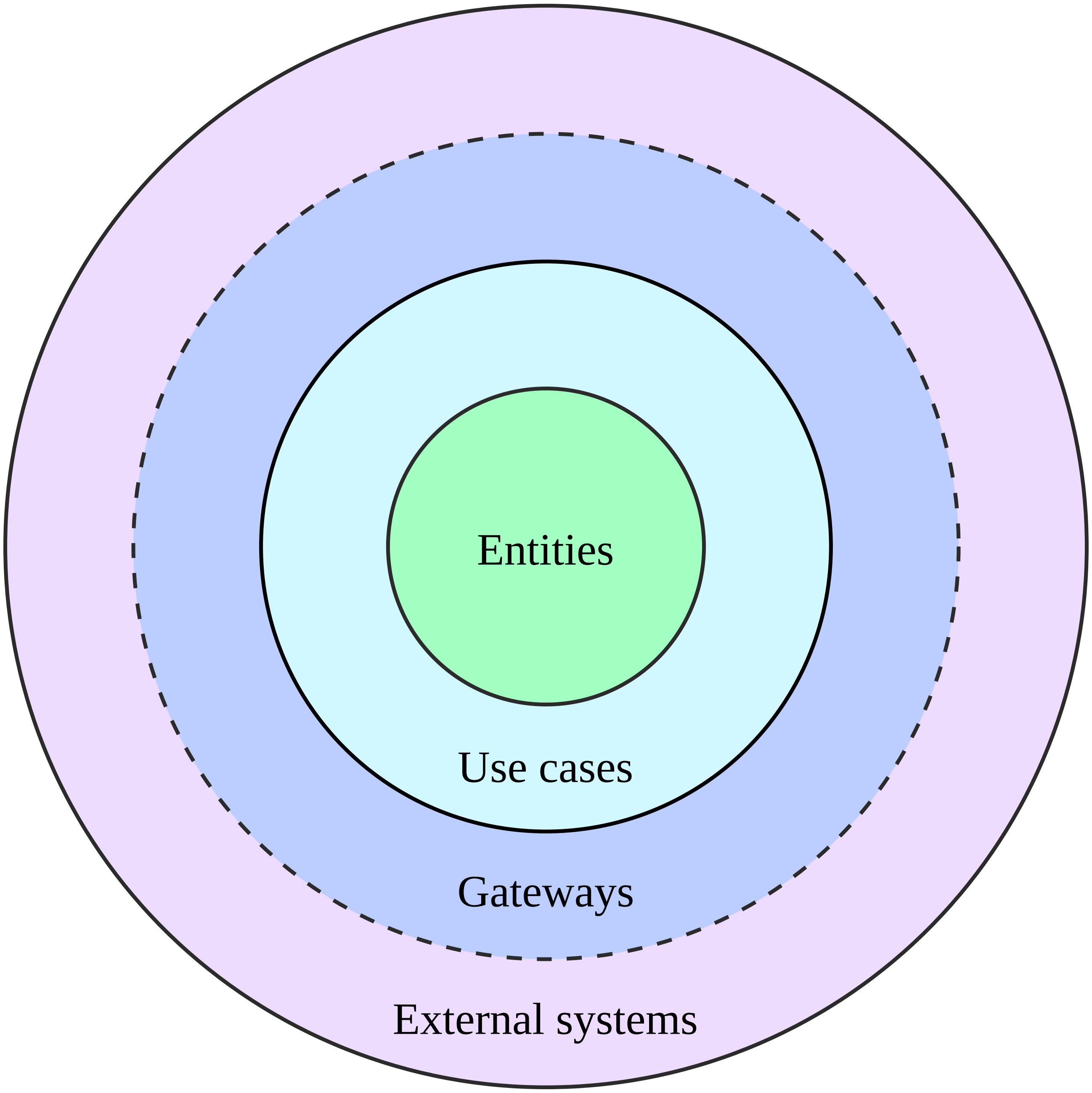
So, there is a hierarchy of components that spawns from the dependencies between them. Some components are defined at the very beginning of the design and do not depend on any other component, while others will come later and depend on them. When data types are involved, the resulting dependencies cannot break this hierarchy, as this would re-introduce a coupling between components that we want to avoid.

Let's go back to the initial example of a shop that buys items from a wholesale, displays them on shelves, and sells them to customers. There is a clear dependency between two components here: the component called "shop" depends on the component called "wholesale", as the data ("items") flow from the latter to the former. The size of the shelves in the shop, in turn, depends on the size of the items (types), which is defined by the wholesale, and this follows the dependency we already established.

If the size of the items was defined by the shop, suddenly there would be another dependency opposing the one we already established, making the wholesale depend on the shop. Please note that when it comes to software systems this is not a circular dependency, because the first one is a conceptual dependency while the second one happens at the language level at compile time. At any rate, having two opposite dependencies is definitely confusing, and makes it hard to replace "peripheral" components such as the shop.

## The main four layers

The clean architecture tries to capture both the *conceptual hierarchy* of components and the *type hierarchy* through a layered approach. In a clean architecture the components of the system are categorised and belong to a specific layer, with rules relative to the communication between components belonging to the same or to different layers. In particular, a clean architecture is a spherical structure, with inner (lower) layers completely encompassed by outer (higher) ones, and the former being oblivious of the existence of the latter.



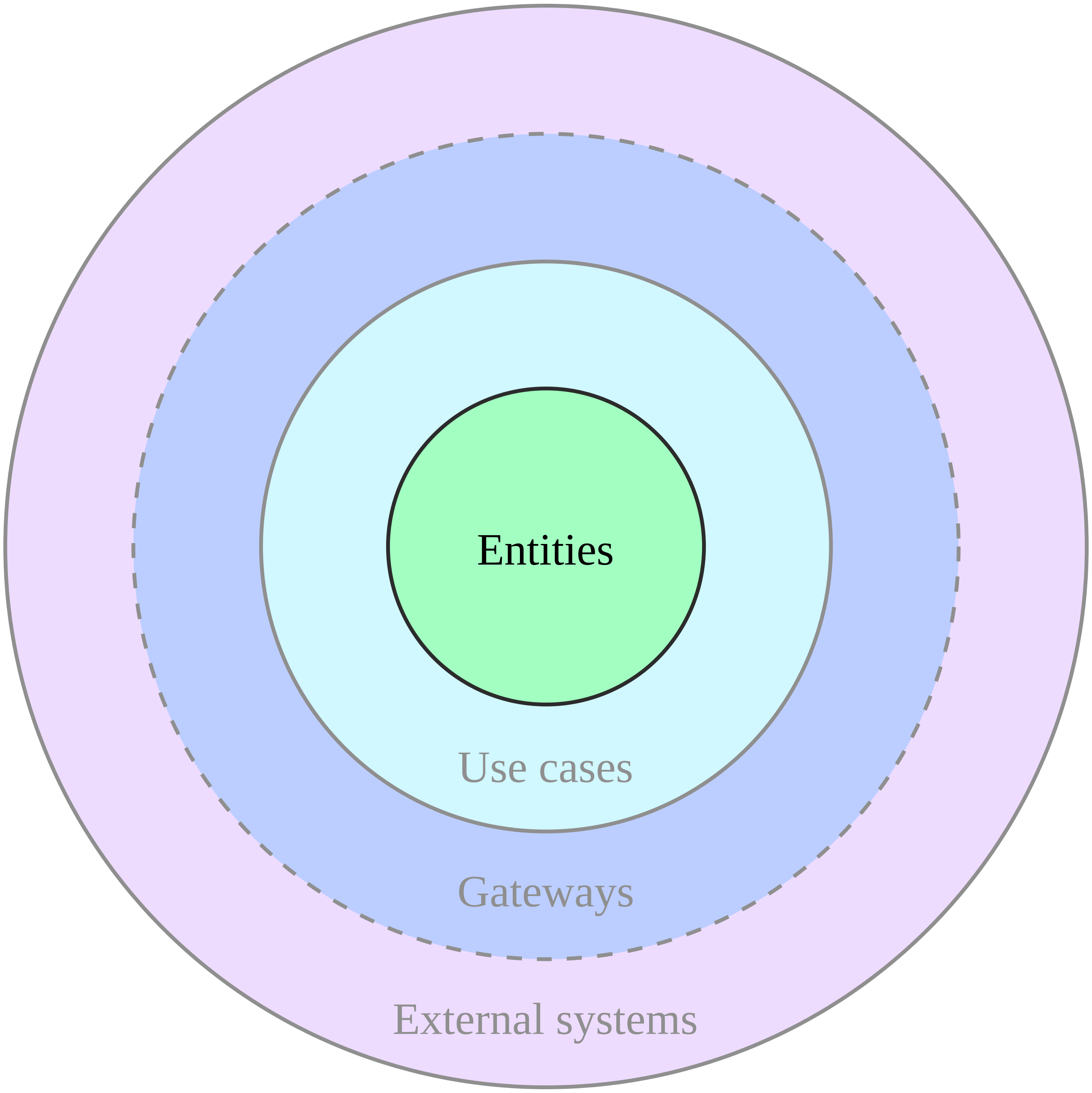
**The basic layers of the clean architecture**

Remember that in computer science, the words "lower" and "higher" almost always refer to the level of abstraction, and not to the importance of a component for the system. Each part of a system is important, otherwise it would not be there.

Let's have a look at the main layers depicted in the figure, keeping in mind that a specific 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 system needs 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 was a tool to analyse medieval manuscripts, however, you might need to isolate sentences and their features, and at this point it might be reasonable to define a specific entity.



### Entities

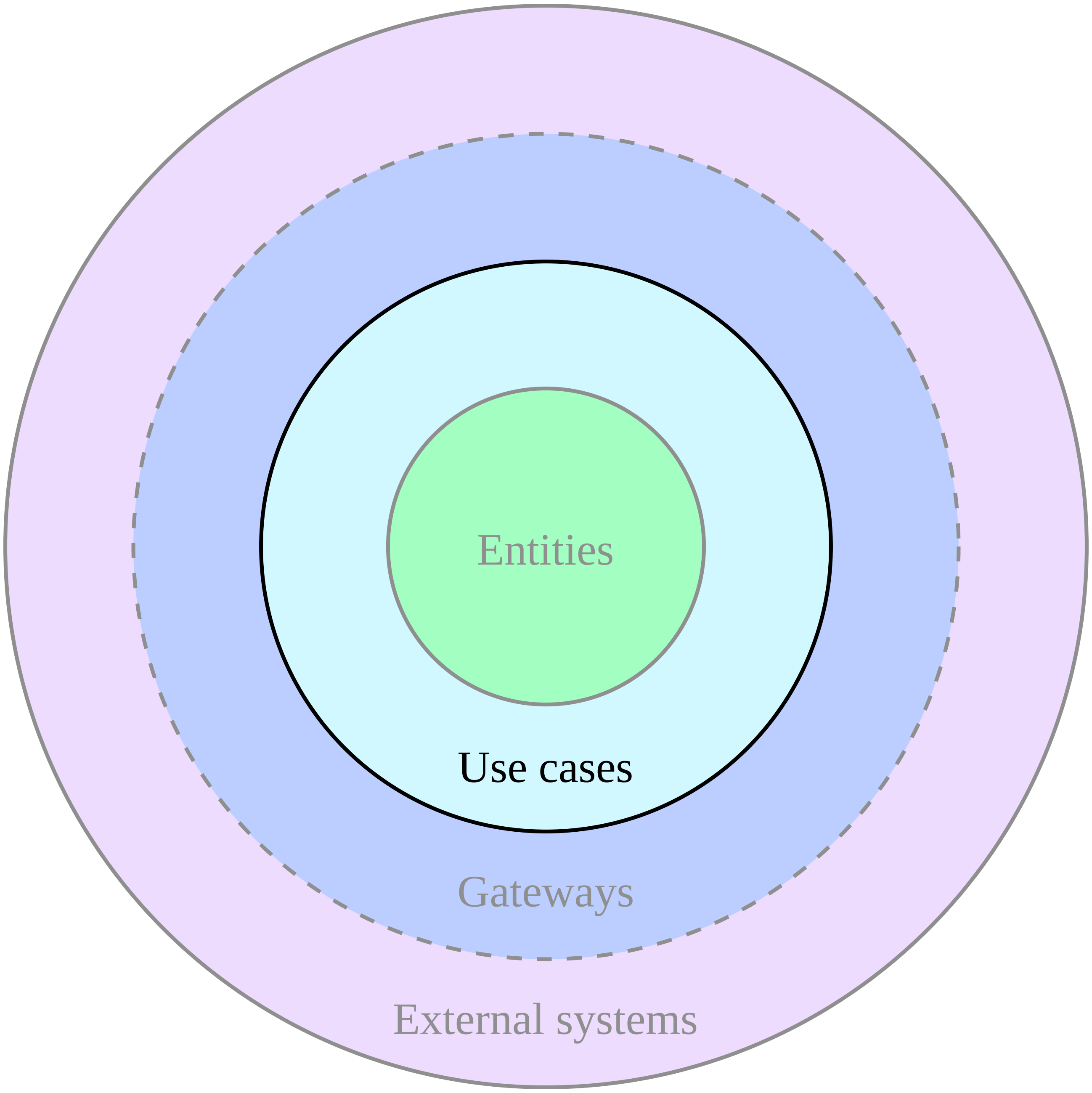
Since we work in Python, this layer will likely contain classes, with methods that simplify the interaction with them. It is very important, however, to understand that the models in this layer are different from the usual models of frameworks like Django. These models are not connected with a storage system, so they cannot be directly saved or queried using their own methods, 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 mutual knowledge since they live in the same layer, so the architecture allows them to interact directly. This means that one of the Python classes that represent an entity can use another one directly, instantiating it and calling its methods. Entities don't know anything that lives in outer layers, though. They cannot call the database, access methods provided by the presentation framework, or instantiate use cases.

The entities layer provides a solid foundation of types that the outer layers can use to exchange data, and they can be considered the vocabulary of your business.

## Use cases

As we said before the most important part of a clean system are use cases, as they implement the business rules, which are the core reason of existence of the system itself. Use cases are the processes that happen in your application, where you use your 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.

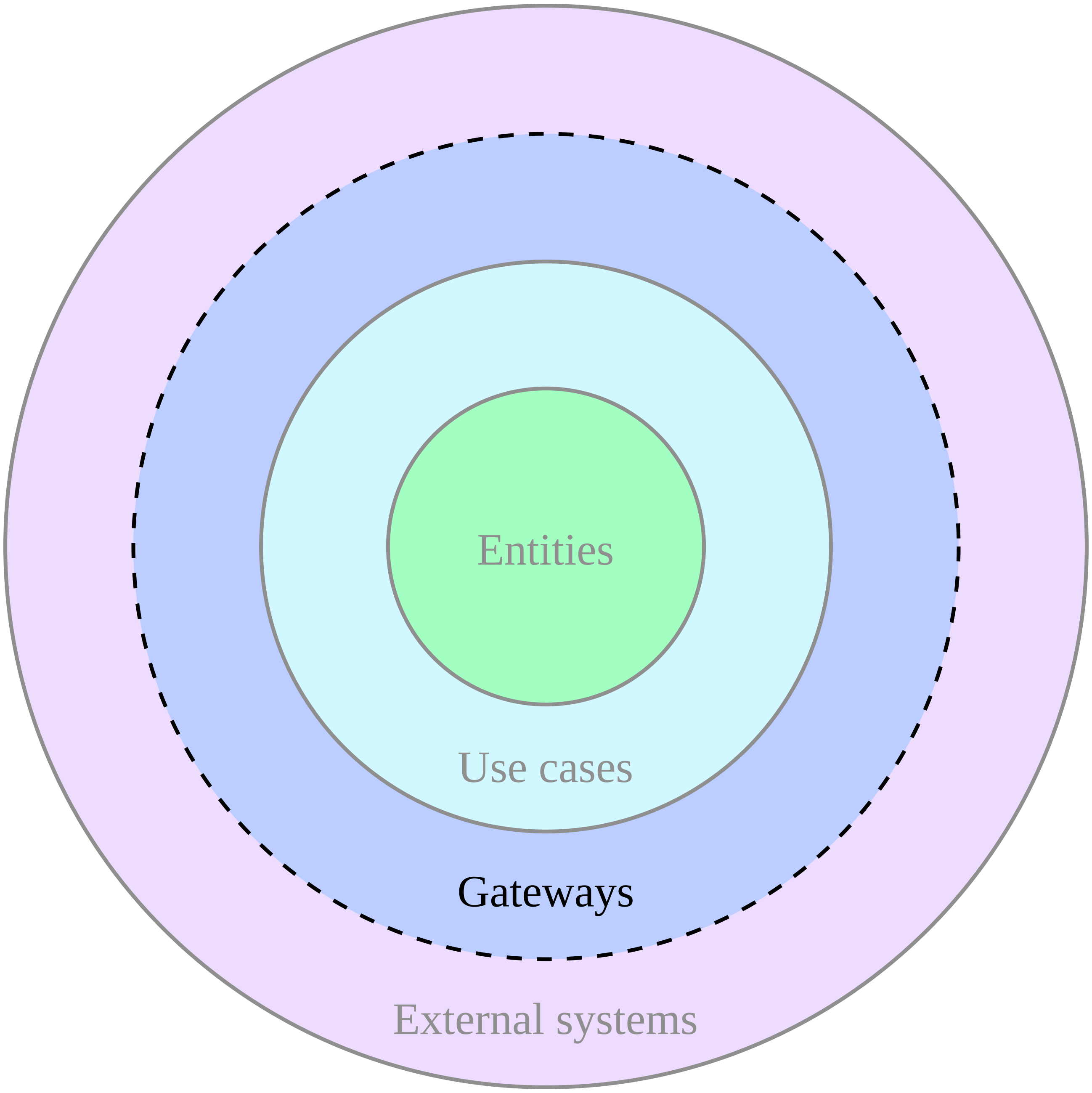


**Use cases**

Use cases should be as small as possible. It is very important to isolate small actions into separate use cases, as this makes the whole system easier to test, understand and maintain. Use cases have full access to the entities layer, so they can instantiate and use them directly. They can also call each other, and it is common to create complex use cases composing simple ones.

### Gateways

This layer contains components that define interfaces for external systems, that is a common access model to services that do not implement the business rules. The classic example is that of a data storage, which internal details can be very different across implementations. These implementations share a common interface, otherwise they would not be implementations of the same concept, and the gateway's task is to expose it.

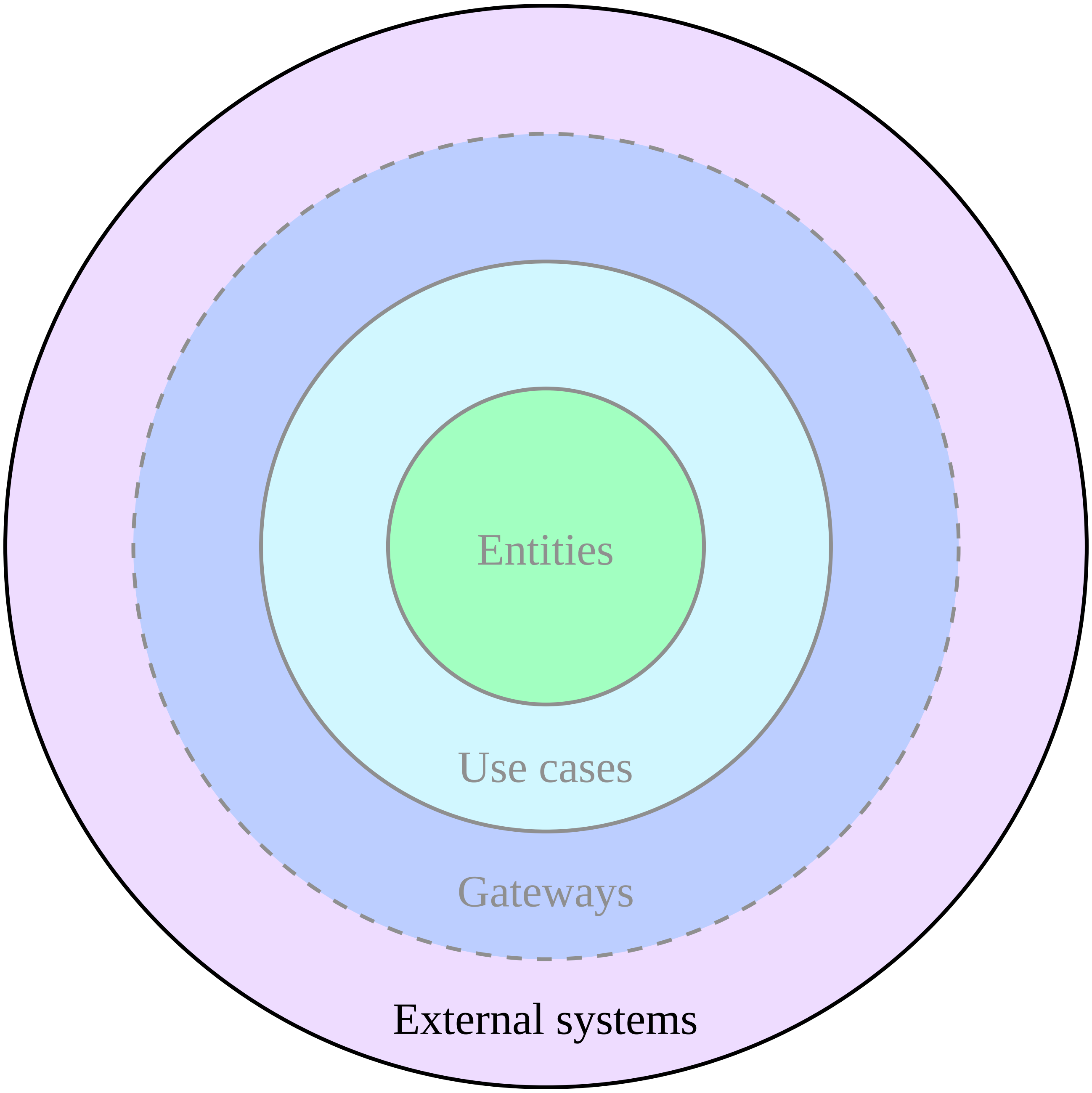


**Gateways**

If you recall the simple example I started with, this is where the database interface would live. Gateways have access to entities, so the interface can freely receive and return objects which type has been defined in that layer, as they can freely access use cases. Gateways are used to mask the implementation of external systems, however, so it is rare for a gateway to call a use case, as this can be done by the external system itself. The gateways layer is intimately connected with the external systems one, which is why the two are separated by a dashed line.

### External systems

This part of the architecture is populated by components that implement the interfaces defined in the previous layer. The same interface might be implemented by one or more concrete components, as your system might want to support multiple implementations of that interface at the same time. For example, you might want to expose some use cases both through an HTTP API and a command line interface, or you want to provide support for different types of storage according to some configuration value.



**External systems**

Please remember that the "external" adjective doesn't always mean that the system is developed by others, or that it is a complex system like a web framework or a database. The word has a topological meaning, which shows that the system we are talking about is peripheral to the core of the architecture, that is it doesn't implement business logic. So we might want to use a messaging system developed in-house to send notifications to the clients of a certain service, but this is again just a presentation layer, unless our business is specifically centred around creating notification systems.

External systems have full access to gateways, use cases, and entities. While it is easy to understand the relationship with gateways, which are created to wrap specific systems, it might be less clear what external systems should do with use cases and entities. As for use cases, external systems are usually the parts of the system that trigger them, being the way users run the business logic. A user clicking on a button, visiting a URL, or running a command, are typical examples of interactions with an external system that runs a use case directly. As for entities, an external system can directly process them, for example to return them in a JSON payload, or to map input data into a domain model.

I want to point out a difference between external systems that are used by use cases and external systems that want to call use cases. In the first case the direction of the communication is outwards, and we know that in the clean architecture we can't go outwards without interfaces. Thus, when we access an external system from a use case we always need an interface. When the external system wants to call use cases, instead, the direction of the communication is inwards, and this is allowed directly, as external layers have full access to the internal ones.

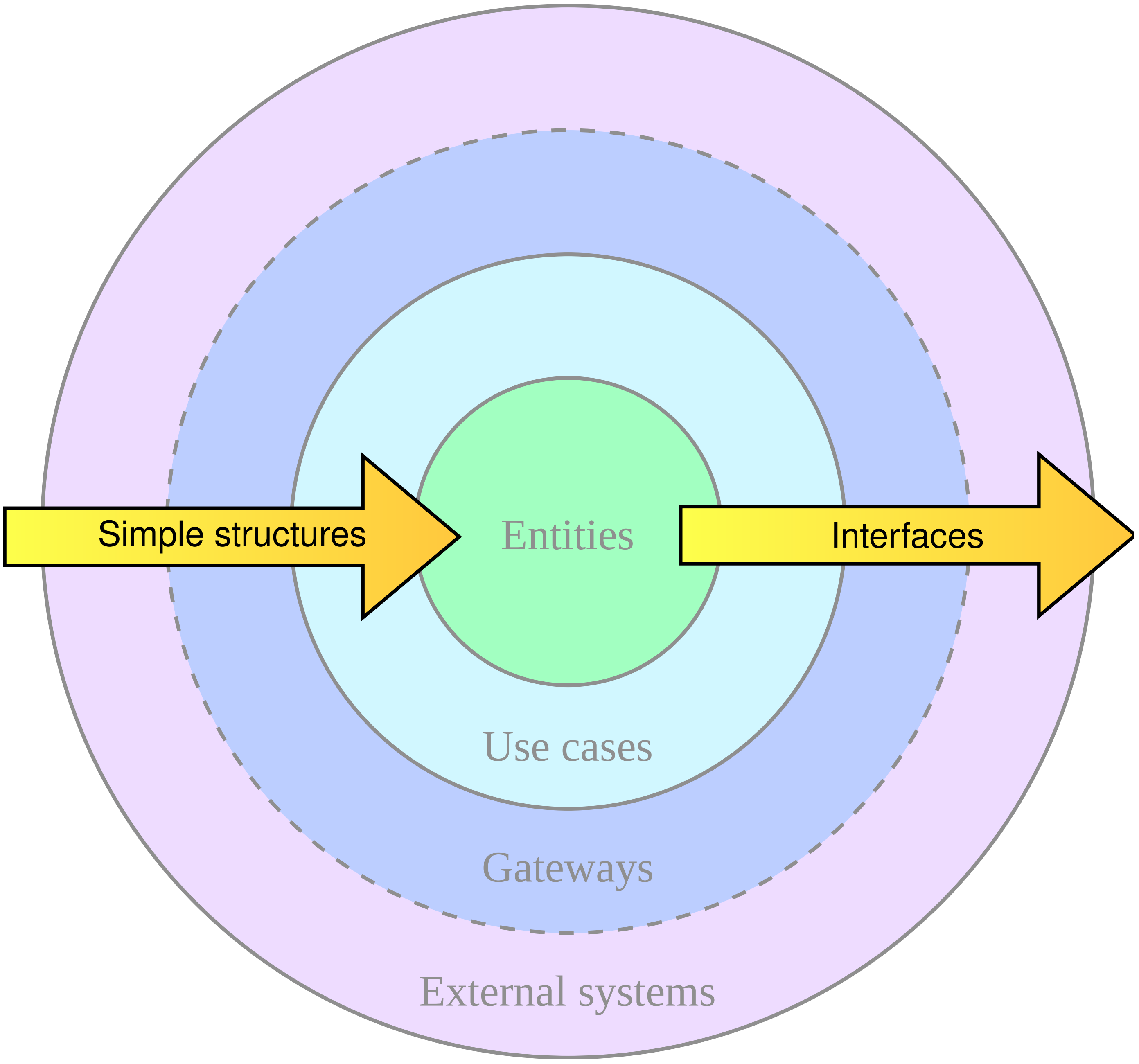
This, practically speaking, translates into two extreme cases, well represented by a database and a web framework. When a use case accesses a storage system there should be a loose coupling between the two, which is why we wrap the storage with an interface and assume that in the use case. When the web framework calls a use case, instead, the code of the endpoint doesn't need any interface to access it.

## Communication between layers

The deeper a layer is in this 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.

**The Golden Rule: 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, that is entities and everything provided by the programming language you are using.



**The golden rule of the clean architecture**

Your 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.

## 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[1] collection of entry points (methods or objects).

The separation between layers and the content of each layer is 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, and be particularly 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 your documentation explaining why you did it. If you access an outer layer 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 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 probably won't 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.

1 Here "fixed" means "the same among every implementation". An API may obviously change in time.

**Previous**
Chapter 1 - A day in the life of a clean system

**Next**
Chapter 3 - A basic example

Last update: 20/08/2021

Share on: Twitter LinkedIn HackerNews Email Reddit

## Section 4 of Clean Architectures in Python

### Previous

### Next