Clean Architecture is not only about business logic

Pablo Aguilar[1], **Lucas Baggio Figueira**[2]

[1,2]Faculdade de Tecnologia do Estado de São Paulo – FATEC Ribeirão Preto
14.030-250 – Ribeirão Preto – SP – Brazil

[1]pablo.aguilar@fatec.sp.gov.br, [1]lucas.figueira@fatec.sp.gov.br

Abstract *A quick introduction to Clean Architecture and its usage outside an application without strong business logic but has to communicate to multiple data providers.*

Resumo *Uma breve introdução ao "Clean Architecture" e seu uso fora do contexto de uma aplicação sem regras de negócios fortes, porém tem que se comunicar com vários provedores de dados.*

1. Introduction

When we are building something most of the time we want it to have a strong basis, a good architecture, a good maintainability, a good readability. In software development this is not different, everyone wants to achieve those goals and they have many ways to do it. That is how all software design patterns and good practices of software development were born.

One of the most important things to decide when we are building a system is its architecture, because it will define how we separate and organize our code (e.g. context, responsibility), and how it directly effects system maintainability.

2. Materials and Methods

Over the years the way software is developed has changed as computing and business problems grow. To deal with these problems there is an area called “software architectural patterns” which tries to achieve the perfect separation inside software project.

In the end, everything is about how we can organize our code to make it more understandable, maintainable and team friendly. These goals mean cost reduction for the companies because the software engineer will fix the problems and implement new features more easily with less time.

Nowadays, there are many software architectural patterns, each with its own specificities and history, but one thing is common among of them, they try to separate the business logic from the everything else. The “Clean Architecture” proposed by Robert C. Martin is one of them.

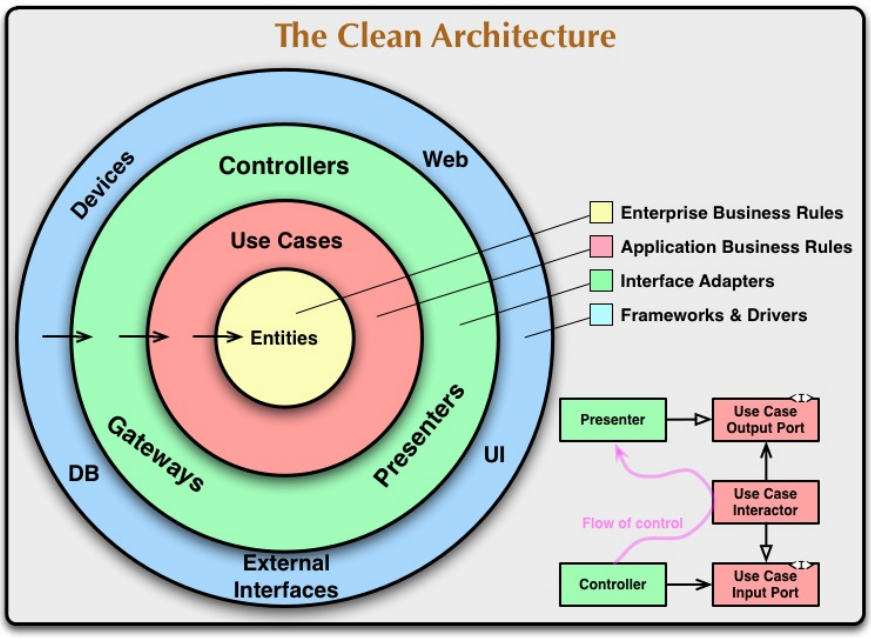

Figure 1. The "Clean Architecture" schema proposed by Robert C. Martin

Below an important quote concerning circles:

"...the circles are schematic. You may find that you need more than just these four. There's no rule that says you must always have just these four. However, The Dependency Rule always applies..." Martin (2012)

It reflects that we have to be pragmatic while applying the proposed architecture which is not the final one. We can adapt the architecture to our problems, we just must keep the principles of it.

Before Robert C. Martin introduces the concepts of his architecture, he presents other architectures as cited below:

> Though these architectures all vary somewhat in their details, they are very similar. They all have the same objective, which is the separation of concerns. They all achieve this separation by dividing the software into layers. Each has at least one layer for business rules, and another for interfaces. (MARTIN, 2012).

The business logic/rules have their own layer in all the architectures because all of them consider the business logic the most important element in a system, however, these architectures are not indicated for systems with poor/weak business logic (e.g. Create Read Update Delete (CRUD) systems) because it will just add unnecessary complexity to the system.

2.1. Dependency Rule

According to Martin (2012), "The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies."

The dependency rule informs how each layer will relate with each other, the arrows in Figure 1 show the relations, an inner circle cannot access an outer circle.

The outer circle must pass the most convenient data format to the inner circle, nothing from the outer circle can influence the inner circle.

2.2. Circles

Each circle in the schema expresses its responsibility, maybe it's one of the most important aspects to follow because it is defining the boundaries between the circles.

2.2.1. Entities

Martin, 2012 describes how entities function.

> Entities encapsulate Enterprise wide business rules. An entity can be an object with methods, or it can be a set of data structures and functions. It doesn't matter so long as the entities could be used by many different applications in the enterprise.
> If you don't have an enterprise, and are just writing a single application, then these entities are the business objects of the application. They encapsulate the most general and high-level rules. They are the least likely to change when something external changes. For example, you would not expect these objects to be affected by a change to page navigation, or security. No operational change to any particular application should affect the entity layer. (MARTIN, 2012)

2.2.2. Use Cases

Another relevant concept, according to Martin, 2012 relates to use cases:

> The software in this layer contains application specific business rules. It encapsulates and implements all of the use cases of the system. These use cases orchestrate the flow of data to and from the entities, and direct those entities to use their enterprise wide business rules to achieve the goals of the use case.
> We do not expect changes in this layer to affect the entities. We also do not expect this layer to be affected by changes to externalities such as the database, the UI, or any of the common frameworks. This layer is isolated from such concerns.
> We do, however, expect that changes to the operation of the application will affect the use-cases and therefore the software in this layer. If the details of a use-case change, then some code in this layer will certainly be affected. (MARTIN, 2012)

2.2.3. Interface Adapters

Martin , 2012 discusses adapters and informs:

> The software in this layer is a set of adapters that convert data from the format most convenient for the use cases and entities, to the format most convenient for some external agency such as the Database or the Web. It is this layer, for example, that will wholly contain the MVC architecture of a GUI. The Presenters, Views, and Controllers all belong in here. The models are likely just data structures that are passed from the controllers to the use cases, and then back from the use cases to the presenters and views.
> Similarly, data is converted, in this layer, from the form most convenient for entities and use cases, into the form most convenient for whatever persistence framework is being used. i.e. The Database. No code inward of this circle should know anything at all about the database. If the database is a SQL database, then all the SQL should be restricted to this layer, and in particular to the parts of this layer that have to do with the database.
> Also in this layer is any other adapter necessary to convert data from some external form, such as an external service, to the internal form used by the use cases and entities. (MARTIN, 2012)

2.2.4. Frameworks and Drivers

Martin, 2012 explains how frameworks and drivers influence the process:

> The outermost layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc. Generally you don't write much code in this layer other than glue code that communicates to the next circle inwards. This layer is where all the details go. The Web is a detail. The database is a detail. We keep these things on the outside where they can do little harm. (MARTIN, 2012)

2.3. Chosen Architecture

We have developed a system using the "Clean Architecture" principles; even though business logic is not strong, it has multiple data sources and one of them is a bit complex to work with. We used an alternative schema than proposed by Robert C. Martin.

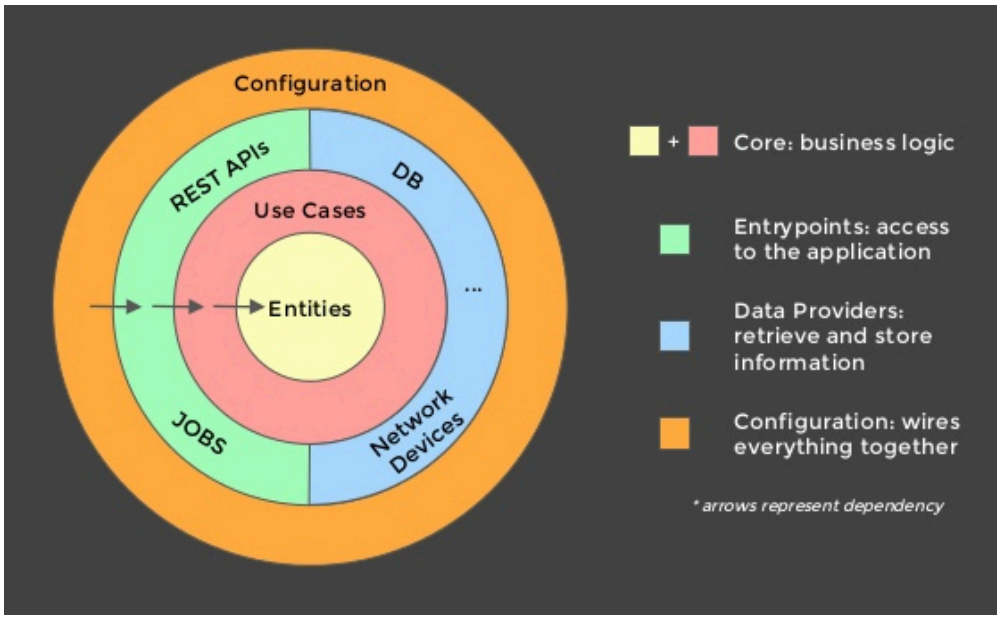

Figure 2. Clean Architecture schema used as a base to our service

2.3.1. Configuration Layer

The main difference between the original schema proposed by Robert C. Martin and the chosen architecture extracted from Battiston, 2016 presentation is the configuration layer.

It is responsible to tie everything together and set up some services, so, in this layer the frameworks configuration can be seen (e.g. Database, Web Framework, Dependency Injection Framework).

3. Results and Discussion

The system that we have made is called Kamui, it is basically a wrapper above KSQL, we used Python to build it which can be found at https://github.com/thepabloaguilar/kamui.

One important thought we have followed while building the system was the "Single Responsibility Principle" to make sure our classes will do anything outside their context (e.g. database interaction inside the use case). The classes' name informs what it should execute, such as GetStreamInformationFromKafkaUseCase, and using the special method, "__call__", to transform our instances in callable instances we just have this unique entrance to our classes.

To define the circles illustrated in the schema (Figure 2) we have chosen to use Python Modules structure, as below each circle is represented by a module:

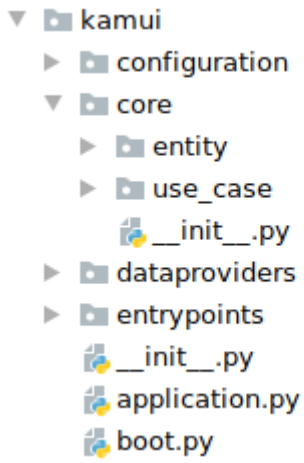

Figure 3. Kamui project folder structure

3.1. Code Examples

3.1.1. Entity

All the entity are quite simple classes.

```python
@dataclass
class Project:
    project_id: UUID
    title: str
    created_at: datetime
    status: ProjectStatus = field(default=ProjectStatus.ACTIVE)
```

Figure 4. Entity class example

3.1.2. Use case

We have a notable example below; it shows a use case coordinating the access to different interfaces.

```
class GetStreamDetailsUseCase:
    def __init__(
        self,
        find_stream_by_stream_id: FindStreamByStreamId,
        find_projects_by_stream: FindProjectsByStream,
        get_stream_by_name: GetStreamByNameUseCase,
    ):
        self.__find_stream_by_stream_id = find_stream_by_stream_id
        self.__find_projects_by_stream = find_projects_by_stream
        self.__get_stream_by_name = get_stream_by_name

    def __call__(self, stream_id: UUID) -> Result[StreamDetails, FailureDetails]:
        stream = self.__find_stream_by_stream_id(stream_id).bind(
            self.__verify_if_stream_exist
        )
        partial_projects = partial(stream.bind, self.__find_projects_by_stream)
        partial_ksql = partial(
            stream.bind, lambda stream_: self.__get_stream_by_name(stream_.name)
        )

        return flow(
            Result.from_value(StreamDetails.build),
            stream.apply,
            bind(self.__call_and_apply(partial_projects)),
            bind(self.__call_and_apply(partial_ksql)),
        )

    def __verify_if_stream_exist(
        self, stream: Maybe[Stream]
    ) -> Result[Stream, BusinessFailureDetails]:
        return maybe_to_result(stream).alt(
            lambda __: BusinessFailureDetails(
                reason="NOT_FOUND", failure_message="Stream not found"
            )
        )

    @curry
    def __call_and_apply(
        self, function: Callable[[], Result[Any, Any]], value_to_apply: Any
    ) -> Result[Any, Any]:
        return function().apply(Success(value_to_apply))
```

Figure 5. Use case example

3.1.3. Entry point

The example below represents a REST entry point to our application, it just calls the use case because the whole business logic is there.

```
class GetStreamDetailsResource(Resource):
    API_PATH = "/streams/<uuid:stream_id>"

    def __init__(self, *args: Any, **kwargs: Any) -> None:
        super().__init__(*args, **kwargs)
        self.__get_stream_details: GetStreamDetailsUseCase = di_container.resolve(
            GetStreamDetailsUseCase
        )

    @json_response
    @unwrap_result_response(success_status_code=200)
    def get(self, stream_id: UUID) -> Result[StreamDetails, FailureDetails]:
        return self.__get_stream_details(stream_id)
```

Figure 6. Entry point example

3.1.4. Data provider

3.1.4.1. Model

This is a simple database model; in this project we are using SQLAlchemy as the Object Relational Mapper (ORM).

```
class StreamModel(DatabaseBase):
    __tablename__ = "stream"

    stream_id = Column("stream_id", UUID(as_uuid=True), default=uuid4, primary_key=True)
    name = Column("name", String(50), nullable=False)
    source_type = Column("source_type", Enum(SourceType))
    source_name = Column("source_name", String(50))

    projects = relationship(StreamProjectModel, back_populates="stream")

    def to_entity(self) -> Stream:
        return Stream(
            stream_id=self.stream_id,
            name=self.name,
            source_type=self.source_type,
            source_name=self.source_name,
        )
```

Figure 7. Model example

3.1.4.2. Repository

A repository is responsible to interact with the database using the database models, typically it implements an interface from the use case layer.

```python
class FindStreamByStreamIdRepository(FindStreamByStreamId):
    def __call__(self, stream_id: UUID) -> Result[Maybe[Stream], FailureDetails]:
        stream = StreamModel.query.filter(StreamModel.stream_id == stream_id).first()
        maybe_stream: Maybe[Stream] = Maybe.from_value(stream).map(
            lambda _stream: _stream.to_entity()
        )
        return Success(maybe_stream)
```

Figure 8. Repository example

4. Conclusion

The “Clean Architecture” showed itself a good architecture to work with. In an advanced form of the project we decided to switch from an HTML + CSS pages to a Simple Page Application (SPA), it was simple and everything in the use case layer stayed unmodified, we just had to add more entry points in our REST layer. This shows how this architecture is flexible.

How we have said earlier that our application does not have a strong business logic, but it has multiple data providers and some interactions are complex as shown below:

```python
class CreateNewStreamFromStreamRepository(CreateNewStreamFromStream):
    def __init__(self) -> None:
        self.__client: HttpClient = client
        self.__KSQL_SERVER_URL: str = "http://localhost:8088/"

    def __call__(
        self, create_new_stream_command: CreateNewStreamCommand
    ) -> Result[CreateNewStreamCommand, FailureDetails]:
        desired_stream_fields = ",".join(
            [field.name for field in create_new_stream_command.fields_]
        )
        filters = " AND ".join(
            filter_.to_statement() for filter_ in create_new_stream_command.filters
        )

        response = self.__client.post(
            url=f"{self.__KSQL_SERVER_URL}ksql",
            payload={
                "ksql": f"""
                            CREATE STREAM {create_new_stream_command.stream_name} AS
                            SELECT {desired_stream_fields}
                            FROM {create_new_stream_command.source_name}
                            {f'WHERE {filters}' if filters else ''}
                            EMIT CHANGES;
                        """
            },
            headers={
                "Accept": "application/vnd.ksql.v1+json",
                "Content-Type": "application/vnd.ksql.v1+json",
            },
        ).map(lambda result: create_new_stream_command)

        return response
```

Figure 9. Complex data provider example

If you have to communicate with multiple data providers the "Clean Architecture" is recommended, as well. Your use case can be simple, it just coordinates the access to the different data providers and aggregates the data, while the complexity is on the data provider layer. This illustrates that applications with non-strong business rules can be beneficiated by this architecture.

References

Martin, R. C. (2012) "Clean Architecture", https://vimeo.com/43612849, June.

Martin, R. C. (2012) "The Clean Architecture", https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html, December.

Martin, R. C. (2014) "The Single Responsibility Principle", http://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html, May.

Martin, R. C. (2017), Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st edition.

Battiston, M. (2016) "Real Life Clean Architecture", https://slideshare.net/mattiabattiston/real-life-clean-architecture-61242830, April.

Giordani, L. (2016) "Clean architectures in Python: a step-by-step example", https://www.thedigitalcatonline.com/blog/2016/11/14/clean-architectures-in-python-a-step-by-step-example, December.

Sobolev, N. (2019) "Enforcing Single Responsibility Principle in Python", https://medium.com/@sobolevn/enforcing-single-responsibility-principle-in-python-cc1ee00de9fb, March.