SEBASTIAN BUCZYŃSKI

IMPLEMENTING THE CLEAN ARCHITECTURE

FOREWORD

WHY I WROTE THIS BOOK?

This book is meant to be a supplement to Robert C. Martin's *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. It is focused on practical aspects of applying the Clean Architecture in IT projects. I find a scarcity of good - quality implementation examples unsatisfactory. Since I used this approach successfully in a few projects, I believe I have plenty of illuminating insights to share with the community.

At the same time, I came across some limitations which I had to overcome. Sometimes the cure was to use other technique (such as CQRS - *Command Query Responsibility Segregation*), sometimes it would be better not to use the Clean Architecture at all.

In short, this book was conceived to share all experience me and my colleagues got during the implementation of the Clean Architecture.

TOOLS-DRIVEN ERA

World of Python is a magical, enchanting place. Imagine you are to write some boilerplate code needed to implement an actual feature. Virtually every time you are about to fall under such an evil spell, you can break it by casting a counter-spell:

```
pip install <name of a 3rd party library solving your problem>
```

Ease of using this command combined with libraries profusion enables everyone, including apprentices of sorcery, to solve seemingly complex problems with a little mana expense. Nowadays, wizardry called software development can be picked up and practised almost effortlessly without knowing its arcana, though nature of the magic itself has not changed at all. This creates an illusion that knowledge about principles and patterns is no longer needed. Although entry point is lowered, deluded sorcery apprentices are far from being enlightened.

Literally every tool python developers use daily is an implementation of long-know (more than decades) and an extensively described pattern of some sort. Django ORM? It is an example of *Active Record* pattern implementation, widely known thanks to Ruby On Rails which follows the same pattern. For example, it was described in Martin Fowler's *Patterns of Enterprise Application Architecture* using these words:

"It’s easy to build Active Records, and they are easy to understand. Their primary problem is that they work well only if the Active Record objects correspond directly to the database tables: an isomorphic schema. (...) Another argument against Active Record is the fact that it couples the object design to the database design. This makes it more difficult to refactor either design as a project goes forward.”

What about something more sophisticated, like SQLAlchemy's session? It turns out the pattern behind it is called *Unit of Work* and is described in the same book. Suddenly an impression of magic powering PyPi packages fades away to eventually vanish. Such knowledge is an invaluable help to choose the right tools for the job. At the same time, a tool which solves your most acute problem will cause several lesser ones, yet those ones you can live with. For example, SQLAlchemy's session forces a developer to register any newly created model using *add* method. Without it, no data will be persisted upon commit. Is the necessity for manual models management worth the trouble, or maybe Django ORM is just ok for this particular project?

The most effective cure for indecisiveness is to stay pragmatic and flexible. In fact, this is what this book is truly about. Even though it explains an exciting approach which highlights the importance of business concerns, it does not conveniently omit drawbacks.

Whenever a new project is started, developers should ask themselves: which approach/framework/library should they use? I may ask in return: What problems would you prefer to have? What issues you have to avoid?

WHO IS THIS BOOK FOR?

This book is aimed at intermediate-/senior-level software developers who wish to broaden their knowledge with various software engineering techniques that emerged over the last several years.

Almost all code examples are written in Python, so the reader's acquaintance with its syntax will be helpful. Luckily, software engineering is mostly technology-agnostic discipline, so even if readers do not write code in Python for a living, they still might use this book to learn something new. All code snippets were written using Python 3.7 and later modernized to Python 3.8.

WHAT WILL YOU FIND IN THIS BOOK?

“In theory, theory and practice are the same. In practice, they are not."

This book is mainly to provide tons of practical advice on implementing the Clean Architecture. Everything is based on my experiences, learnt the hard way. Sometimes it was immediately apparent that a certain solution is a bad idea. Sometimes we needed a laborious refactoring weeks later to undo bad design. Few times we have not had an opportunity to improve something that desperately needed it.

With trial and error, we found out how we can evolve our software using more and more sophisticated techniques, like CQRS, Event Sourcing or Domain-Driven Design. The greatest thing was the ability to pick one them up whenever we actually needed them, without investing a lot of time and effort in a big design upfront or having to rewrite everything from scratch.

Implementing the Clean Architecture is a bit like a buffet - a reader is encouraged to get out of it whatever seems to suit best their need and mood. It makes no sense to follow every rule & recommendation rigorously if a simpler approach would suffice.

THE CLEAN ARCHITECTURE BASICS

WHAT IS IT ALL FOR?

IT is an industry which changes rapidly all the time. New languages and frameworks emerge daily, just to be forgotten several years later. Solutions that once were popular become an enormous technical debt soon after the last contributor abandoned the project. On the other hand, there are few successful, long-living projects which are continuously maintained and developed. Although we get new features and security updates regularly, it still requires some effort to keep up with the newest versions of your favourite web framework.

"The only thing that is constant is change" - Heraclitus

This task becomes cumbersome if the business logic of a project is tightly coupled to a framework. Every backwards-incompatible update in the framework's codebase breaks something in the actual application. Such a situation is inconvenient for both maintainers and users of the framework. The former group is under constant pressure not to break anything with a new release. Just imagine how discouraging the situation is.

Some applications are pretty straightforward. All they need to do is just fetch some data from a database, modify it and save back. A common name for a *database browser* is *CRUD* (*Create Read Update Delete)*. Adding REST API increases complexity only a bit. Using Django for such a project is one of the best choices one may make in the Python world.

The situation becomes a lot trickier when we deal with more complex domains. They are actually pretty easy to recognize. One of the symptoms might be a vast number of checks to conduct. Invariants spanning multiple objects are even more interesting. Say, we are to build a new project where people can bid on auctions. An auction can have 0, 1 or multiple winners at the same time. An auction has an end time after which no one can bid.

If we were to use CRUD approach a'la Django/RoR, then most likely we would end up with separate models for *Auction* and *Bid*:

```python
class Auction(models.Model):
    title = models.CharField(max_length=255)
    starting_price = MoneyField(
        max_digits=19,
        decimal_places=4,
        default_currency="USD",
    )
    current_price = MoneyField(
        max_digits=19,
        decimal_places=4,
        default_currency="USD",
    )


class Bid(models.Model):
    price = MoneyField(
        max_digits=19,
        decimal_places=4,
        default_currency="USD",
    )
    bidder = models.ForeignKey(
        get_user_model(), on_delete=models.PROTECT
    )

    auction = models.ForeignKey(
        Auction,
        related_name="bids",
        on_delete=models.CASCADE,
    )
```

The problem is that in terms of a bidding process, these two are strongly connected. We can not just save a new *Bid* to a database whenever someone clicks a *Bid*! button. New *Bid* has to be checked against *Auction*. Has not the latter just ended? If the new *Bid* is the highest one, then we have to set it as a winning one for the *Auction*. At the same time, we have to change the current price of *Auction*. Previously winning *Bid* is now considered a losing one. As you can see, these two, seemingly distinct entities, can not be treated independently. In other words, there are invariants in the domain that span both *Auction* and *Bid*. It does not make any sense to reason about them separately, at least not in the bidding process.

This example was not too complicated. Yet code that enforces these business rules does not fit into any of building blocks included in Django (or any other web framework, to be fair). Invariants span beyond a single model. At the same time it is hard to imagine putting them in a view (a function or class handling single HTTP request). Although there are no physical obstacles, it just does not *feel* right.

Another issue with code coupled to a framework is visible when one tries to test it - one is not able to test business logic without involving heavy machinery. Initializing the whole thing, inserting rows to the database, executing web framework code (e.g. for URL routing), cleaning DB afterwards - it all takes time. As a matter of fact, time is the only cost of running tests. If executing test suite takes ages, then it will not be run too often. As it happens, complex domains have multiple cases to be checked. If one wants to cover them all, they are stuck with long execution time of a test suite.

The last category of things that can give a headache - integrations with 3rd party services. Using as many external services as possible is a trendy approach these days. 3rd party services can not be avoided for many seemingly simple projects. The first example that comes to mind - e-commerce. Customers have to pay for their shoppings, so making friends with some payments platform is a worthwhile idea. Such platforms income comes from charged fees. Now imagine you are to replace one integration with another because different payments platform is slightly cheaper. How many orders must be placed to compensate for development time? Reckless, naive integration will tightly couple payment processes within the application in the same way as web frameworks do. Therefore it will be hard to change.

So far only problems were described, without proposing any solution. All of them can be addressed with elegance and style. This is where the Clean Architecture comes in. Simply saying, it is an approach to software architecture that gives special treatment to business rules. It is unacceptable for a framework, database or 3rd party service to leak to and poison business logic. Correctly applied, the Clean Architecture gives us the following:

- independence of frameworks - upgrading framework or even switching it should be much less of a headache than before
- testability - all business rules can be tested using unit tests, without inserting anything into a database
- independence of UI/API - delivery mechanism must not shape logic
- independence of database - way of storing data should not limit a developer
- independence of any 3rd party - business rules do not need to know which payments platform you are using

- flexibility - certain architectural decisions can be delayed without stopping development
- extensibility - projects can be easily extended with more sophisticated techniques like CQRS, Event Sourcing or Domain Driven Design if needed

These are benefits of a strict separation of concerns, arranging codebase into clearly separated layers and applying the Dependency Rule between them.

CODE ORGANIZATION - HORIZONTAL SLICING

In a basic form of the Clean Architecture, there are four layers. Naturally, one can use more if it is justified.

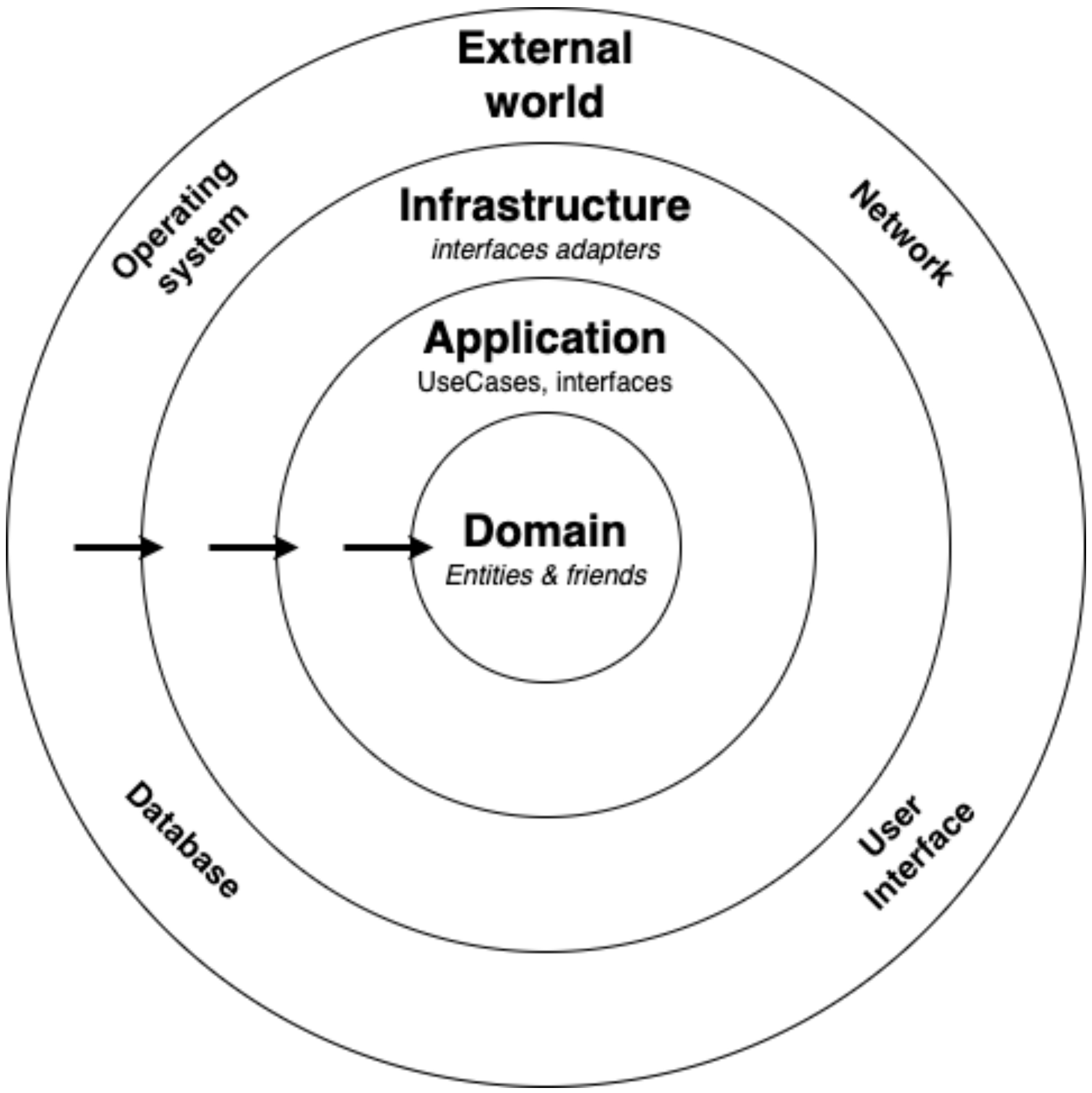

Figure 1.1 Layers of the Clean Architecture

EXTERNAL WORLD

The outermost one, *External world*, represents all services and code that project uses, but it does not belong to the same code base. Simply saying, this layer encompasses everything that was implemented outside the project.

INFRASTRUCTURE

The second layer is called *Infrastructure*. It contains all the code needed for the project to use goodies from *External World*. For example, if we use MariaDB for our primary data store, classes and functions responsible for communication with MariaDB will be sitting in the *Infrastructure* layer. The same is true for any 3rd party service we have to integrate with. For example, if we are building an e-commerce solution, we are going to place here classes implementing integration with payment providers. Kinds of integrations depend on the type of the project.

APPLICATION

The third layer is for application-specific business rules. Therein lies code that specifies what a project actually does. *Application* layer is a home for *Use Cases* (also known as *Interactors*). *Use Case* is a single operation within the project that leads to changing the state of the system, assuming everything goes right. Using auctioning example, we could have a *Use Case* for placing a bid and another one for withdrawing a bid. If we were building an e-commerce solution, we could have one *Use Case* for adding an item to a cart and another for removing an item from the cart. *Use Case* represents a single action of a user (or another actor) that is significant from the business point of view. If you are familiar with Scrum, these can be more or less translated into user stories.

The second kind of building blocks which will always reside in this layer is an *Interface* (also known as *Port*). These are abstractions over anything that sits in the layer above - *Infrastructure* and is required by at least one *Use Case*. In Python, this can be implemented using abstract base classes (`abc`) module.

```python
# application/interfaces/email_sender.py
import abc


class EmailSender(abc.ABC):
    @abc.abstractmethod
    def send(self, message: EmailMessage) -> None:
        pass


# infrastructure/adapters/email_sender.py
import smtplib

from application.interfaces.email_sender import (
    EmailSender,
)


class LocalhostEmailSender(EmailSender):
    def send(self, message: EmailMessage) -> None:
        server = smtplib.SMTP("localhost", 1025)
        # etc.
```

These code snippets show a relation between *Interfaces* from *Application* layer and their adapters from *Infrastructure*. What is important - a *Use Case* MUST NOT be aware whether we are using `LocalhostEmailSender` or any other class inheriting from `EmailSender`. More on this later. To sum up, *Application* layers contains code for all actions and defines interfaces for the external world to execute actions' logic.

DOMAIN

This layer is a place for all business rules that have to be enforced regardless of a context in which they were used. Basic building block to use here is called *Entity*. Using auctioning example once again - we could have an *Entity* for Auction with methods for placing a bid and withdrawing one:

```python
class Auction:
    def place_bid(
        self, user_id: int, amount: Decimal
    ) -> None:
        pass

    def withdraw_a_bid(self, bid_id: int) -> None:
        pass
```

Why would anyone place such logic here instead of implementing it inside *PlacingBid*UseCase? One could just change an instance of `Auction` directly:

```
class PlacingBidUseCase:
    def execute(self, _args):
        ...
        auction.winners = [new_bid.bidder_id]
        auction.current_price = new_bid.amount
```

Such an approach would effectively make our *Entities* anemic. Such creatures are also called *Data Classes*[1] (not to confuse with data classes from standard library*!)* or Plain Old Python Objects. They are just dummy bags for data and have no methods (behavior). Whole logic would be implemented outside such classes. This pattern is known as *Transaction Script*[2].

This can work in certain circumstances, but certainly not in this case, because auctioning domain has invariants to protect. For example, every change of the winner affects the current price. We already know at least two situations when this happens - when someone offers more than the previous winner and when we are to withdraw currently winning bid. We are going to have separate *Use Cases* for *PlacingBid* and *WithdrawingBid*, so naive approach with methodless classes and *Transaction Scripts* implies that we would have to duplicate logic of calculating current price, which is unacceptable. When *Transaction Script* is misused and implements the logic that should be encapsulated by *Entity*, we are talking about anti-pattern called *Anemic Entities*. Yet another principle that warns against changing object data from the outside is **Tell, Don't ask**[3].

```
class PlacingBidUseCase:
    def execute(self, _args) -> None:
        # Tell, don't ask violated
        # if auction.current_price < new_bid.amount:
        #     auction.winners = [new_bid.bidder_id]
        #     auction.current_price = new_bid.amount

        # let Auction handle it! It knows best
        auction.place_bid( ... )
```

[1] Martin Fowler, *Refactoring: Improving the Design of Existing Code 2nd edition*, Chapter 3, Data Class

[2] Martin Fowler, *Patterns of Enterprise Application Architecture*, Chapter 9, Transaction Script

[3] Martin Fowler, TellDontAsk https://martinfowler.com/bliki/TellDontAsk.html

THE DEPENDENCY RULE

Grouping classes and functions into layers is not enough to get clear, maintainable codebase. Obviously, control flow has to cross at least few (if not all) layers to actually do something in projects that use the Clean Architecture. Having benefits and goals of this approach in mind, interactions between layers cannot be left to chance. One possibly could import and call some framework-specific code in the domain layer if it not had been for the Dependency Rule. It says that no lower layer is allowed to know and use anything from any upper layer. For example, one is not permitted to use any class, function or a module from the *Infrastructure* if we are in the *Domain* layer. The Dependency Rule not only forbids developer from explicitly importing symbols from the outer layer but also discourages accepting these as functions arguments. The Dependency Rule is illustrated with arrows in the architecture diagram. The direction of arrows is the same as dependencies: *Infrastructure* uses *Application*, *Application* uses *Domain*, but it is not allowed for *Domain* to use *Infrastructure* or *Application* etc.

Naturally, in Python, the whole thing has to be treated as a gentlemen's agreement. Language does not provide any native method to enforce layering rules. They are certain half-measures described later in the book. In languages such as Java or C#, a developer can rely on packages and access modifiers to achieve a nice, clean separation.

BOUNDARIES

Last, but definitely not least thing layers need to have are sharp boundaries. The boundary defines a communication protocol with the layer. A layer groups code. It contains classes and functions. Most of them will not be meant to be used from the outside. They are *private*, in a manner of speaking. This implies that no one from the outside should be even bothered by their existence. To point lost developers in the right direction, one should expose the layer's API and make it look like an obvious path to take whenever someone needs layer's functionality. Effectively, a boundary is a set of interfaces. Their methods are like doors and arguments of these methods are like locks. They expect a specific argument which will open them, like a key.

Arbitrary types should not be passed between layers. Following the Dependency Rule, one is strictly forbidden from passing data structure from the upper layer down to the lower layer. For example, passing an ORM model to *Domain* violates the rule, because it implies that *Domain* knows something about the outer world. Languages without static typing or type annotations (like Python before 3.4) can have that easily overlooked. Fortunately, importing something from an upper layer only to annotate an argument already gives bad feelings.

Input arguments are part of a boundary, and they should belong to a layer that accepts them. In a real-world API of a layer will consist of many methods accepting a varying number of arguments. This complexity cannot be taken lightly. Thus, it makes perfect sense to group boundary parameters for individual entry points - methods - in data structures. This pattern is called *Data Transfer Objects* (DTOs).

```python
@dataclass(frozen=True)
class EmailDto:
    src: EmailAddress
    reply_to: EmailAddress
    contents: str
```

The most crucial boundary is placed on the edge of *Application* layer. *Application*'s boundary that is to be used from the outside world is formed by *Use Cases*. To avoid exposing concrete classes (and hence, coupling) with *Application's* clients, another interface can be introduced that will abstract a *Use Cases - Input Boundary*. From *External World*'s perspective *Use Case/Input Boundary* is just an interface communicating business intent of an application. To call it, one has to prepare a *DTO* and pass it as the only argument. Analogously, another *DTO* is a result of actions taken by a *Use Case* (though it is not directly returned - more on this later). These three (*Input DTO, Output DTO, Input Boundary*) together form a rock-solid boundary that hides all details of *Application* layer. Other names that may be used to refer to *Input-* and *Output DTOs* are respectively called *Request* and *Response*. However, to avoid confusion with HTTP protocol, I will refer to them using *Input-* and *Output DTOs* throughout the book. *Data Transfer Objects* are immutable (`frozen=True`). There is no reason why would anyone want to mutate data inside. They are like messages - one coming in and another coming out.

```python
@dataclass(frozen=True)
class PlacingBidInputDto:
    bidder_id: int
    auction_id: int
    amount: Decimal


@dataclass(frozen=True)
class PlacingBidOutputDto:
    is_winning: bool
    current_price: Decimal
```

```
class PlacingBidInputBoundary:
    @abc.abstractmethod
    def execute(
        self, request: PlacingBidInputDto
    ) → None:
        ...
```

MVC ANYONE?

If you are a Pythonista who wrote some code in Django, Flask or Pyramid, you might be confused a bit with naming. *Controller* in the diagram corresponds to a concept you know as *view*, whereas *View* resembles *template*. This fuss roots in different patterns adoption between Python and other programming communities. The diagram assumes readers acquaintance with *Model-View-Controller*, while Django embraces something known as *Model-Template-View*. More information can be found on djangobook.com - *Django's Structure – A Heretic's Eye View* https://djangobook.com/mdj2-django-structure/.

Nevertheless, all confusion will disappear once we analyse referential implementation.

CHAPTER SUMMARY

Actual value of IT projects lies right next to the most significant complexity they have. Provided that a project is something more than just a browser for a relational database, there will be plenty of business rules that have to be enforced. The Clean Architecture treats the latter as first-class citizens. Instead of hiding this most-valued logic in a soup of frameworks and ORMs it exposes business rules and processes on separate layers - *Domain* and *Application*. Distilled business logic can be easily tested as it is completely unaware of the external world. Code responsible for communicating with it lies in the *Infrastructure* layer. The latter can use *Application*, but *Application* must not know anything about *Infrastructure*. This is enforced by the Dependency Rule:

```
External World → Infrastructure → Application → Domain
```

Obviously, during the execution of a business scenario, one will have to insert rows to a database or call an external service at some point. The Clean Architecture forbids coupling business logic with the external world, so *Application* defines set of *Interfaces* (also known as *Ports*) which are a form of abstract plugins. Concrete implementations are to be eventually provided by *Infrastructure*.

Keeping everything in order requires drawing sharp, distinctive boundaries. Layers expose some functionality via *Interfaces* that accept *Input DTO* (sometimes called *Request*) as arguments. All details are hidden behind the boundary. From the outer world, one can only see method signatures and data structures required to call method lying on the boundary.

REFERENTIAL IMPLEMENTATION

DISCLAIMER

This chapter is to present example implementation according to the original idea presented by *Robert C. Martin* in *The Clean Architecture* article[4], few talks given on conferences[5] and described in his book[6].

I must admit I have never tried implementing the Clean Architecture in a commercial project rigorously following original Uncle Bob's vision. I felt that a few parts could be removed or done differently without losing too much. Although my implementations look a bit different, I decided to illustrate the original concept with code for the sake of completeness of this book. In the next chapter, I describe possible simplifications one may make without compromising much of the quality and benefits.

CONTROL FLOW IN THE CLEAN ARCHITECTURE

This example is a standard web application that uses a database for storing data. Control flow begins in *Controller*, which is invoked by a web framework upon dispatching request. Role of the *Controller* is to repack HTTP request data into *Input DTO* and pass it to *Input Boundary*, implemented by *Use Case* (also known as *Interactor*). The latter uses data from *Input DTO* to fetch required *Entities* from *Database* using *Data Access Interface*. Then *Use Case* orchestrates *Entities* to perform business logic and optionally saves them using *Data Access Interface*. *Use Case* finishes its task by building *Output DTO* and passing it into *Output Boundary* implementation - *Presenter*. Its role is to reformat data to be convenient for displaying in the final *View*. *View* receives data in another DTO, called *View Model*. *Use Case* that implements *Input Boundary*, does not return anything. *Presenter* that implements *Output Boundary* is to actually *present* the result using *Output DTO*.

This is a complete description in a nutshell. Before we delve into actual implementation, please take note about boundaries and layers. Starting from the right-upper corner, *Entities*

[4] Robert C. Martin, *The Clean Architecture* https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

[5] Robert C. Martin, *Architecture the Lost Years* https://www.youtube.com/watch?v=WpkDN78P884

[6] Robert C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design

belong to *Domain* layer. Majority of elements in the diagram (*Input DTO, Input Boundary, Output Boundary, Output DTO, Use Case, Data Access Interface*) belongs to *Application* layer. The rest is less important. *Data Access* implementation lies in *Infrastructure* layer, while *Database* belongs to *External World*.

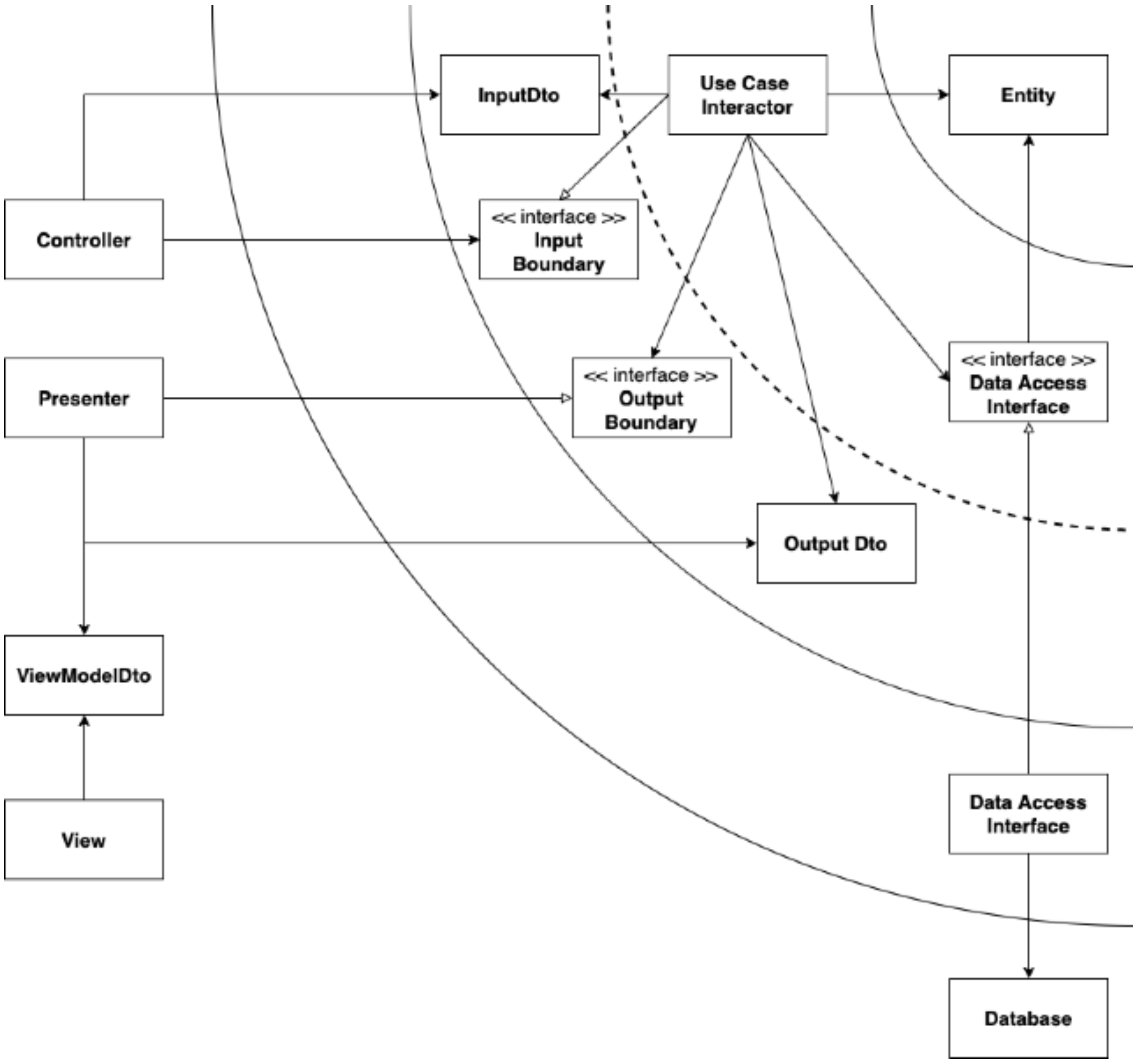

Figure 2.1 The Clean Architecture referential implementation diagram

BUSINESS REQUIREMENTS

The code is to be derived from a set of business rules. Therefore I present them before the implementation is shown:

- Bidders can place bids on auctions to win them
- An auction has a current price that is visible for all bidders
    - current price is determined by the amount of the lowest winning bid
    - to become a winner, one has to offer a price higher than the current price
- Auction has a starting price. New bids with an amount lower than the starting price must not be accepted

IMPLEMENTATION

SEQUENCE DIAGRAM

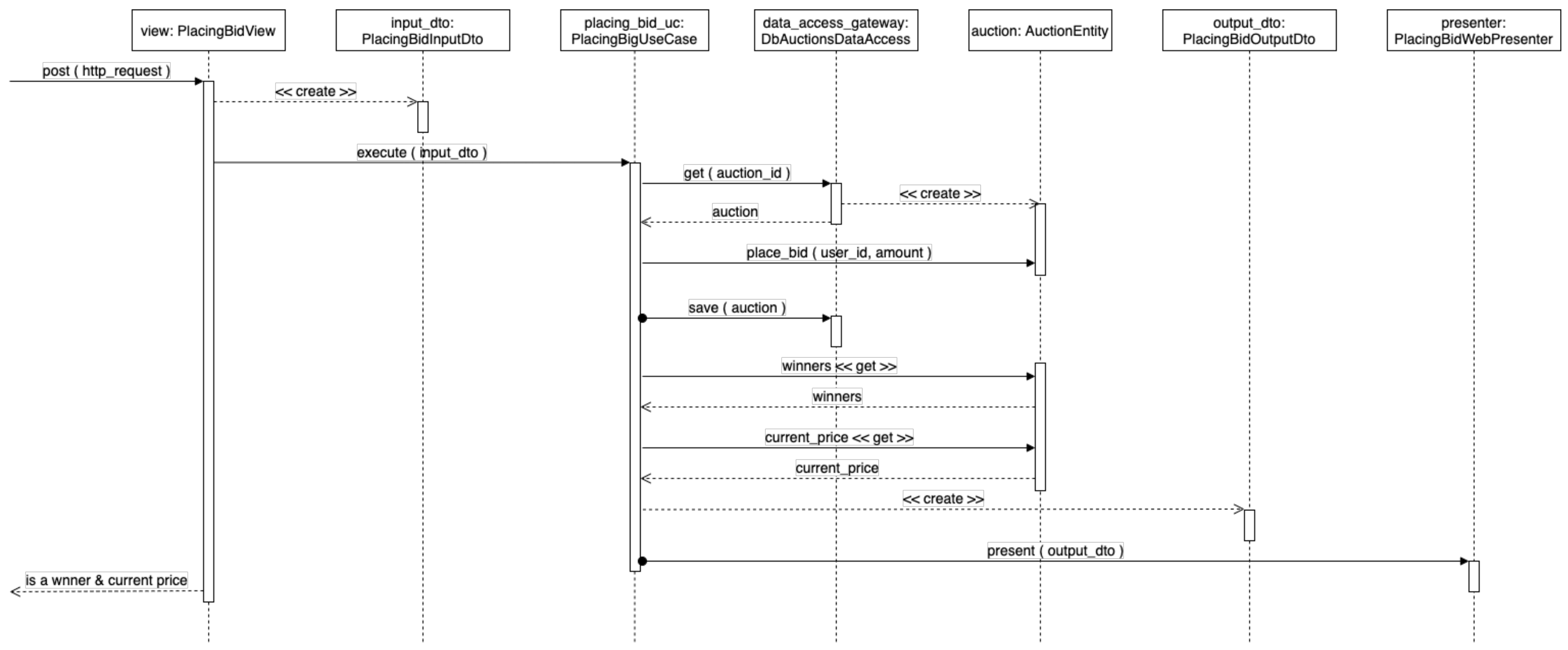

Figure 2.2 A sequence diagram of the Clean Architecture referential implementation

It may look confusing that there is no arrow from *Presenter* to *View* just before the end. There is a reason for that described below.

INPUT BOUNDARY

Our application's functionality is visible on a framework level as an *Input Boundary* - an interface accepting an *Input DTO*. The latter is a relatively simple data structure with typed fields, understandable by lower layer - in this case, we accept only standard Python's data types:

```python
@dataclass(frozen=True)
class PlacingBidInputDto:
    bidder_id: int
    auction_id: int
    amount: Decimal
```

We assume that the authentication aspect is to be dealt with on a web framework level - we just accept bare id that belongs to a person that places a bid and trust it. *Input DTO* is to be passed into *Use Case* abstracted by an *Input Boundary*:

```python
class PlacingBidInputBoundary(abc.ABC):
    @abc.abstractmethod
    def execute(
        self,
        input_dto: PlacingBidInputDto,
        presenter: PlacingBidOutputBoundary,
    ) -> None:
        pass
```

OUTPUT BOUNDARY

At the same time, we expect our operation to produce some data in the form of *Output DTO*:

```python
@dataclass(frozen=True)
class PlacingBidOutputDto:
    is_winning: bool
    current_price: Decimal
```

This data structure is to be accepted by the only method of `PlacingBidOutputBoundary` (an interface for presenters):

```python
class PlacingBidOutputBoundary(abc.ABC):
    @abc.abstractmethod
    def present(
        self, output_dto: PlacingBidOutputDto
    ) -> None:
        pass
```

PRESENTER

A class that implements **`PlacingBidOutputBoundary`** is called *Presenter*. Its role is to convert output data to the format most convenient for a presentation layer. In our example, we return the boolean flag to indicate whether bidder became a winner and a piece of information about the current price of the auction. *Presenter* is to format decimal number to the desired number of decimal fields, add currency symbol - in short, to convert data into a string, appropriate for showing it to a bidder.

```
class PlacingBidWebPresenter(
    PlacingBidOutputBoundary
):
    def present(
        self, output_dto: PlacingBidOutputDto
    ) → None:
        formatted_data = {
            "current_price": f'${output_dto.current_price.quantize(".01")}',
            "is_winning": "Congratulations!"
            if output_dto.is_winning
            else ":(",
        }
        ...
```

As you might have deduced from the sequence diagram, the flow of control ends in a *Presenter* implementation, namely `present` method. We do not return anything to *Controller* (or view in MVT). A bidder should see new data immediately after the **`present`** call ends. This is hard to imagine in the most popular Python web frameworks when *Controller* is expected to return something that the framework is going to send to the client later. However, approach with flow ending in the **`present`** method works perfectly fine for mobile applications which can build the next screen depending on contents of **`PlacingBidOutputDto`** and show it to the user. One could also get to such behavior in frameworks that create a response object beforehand and lets you manipulate it. Examples for this particular case will be shown later in the book. For the sake of simplicity, one would rather extend **`PlacingBidOutputBoundary`** interface with another method that can be used for retrieving data in *Controller*:

```
class PlacingBidOutputBoundary(abc.ABC):
    @abc.abstractmethod
    def present(
        self, output_dto: PlacingBidOutputDto
    ) → None:
        pass

    @abc.abstractmethod
    def get_presented_data(self) → dict:
        pass
```

Any concrete implementation would essentially be just giving back formatted data:

```python
class PlacingBidWebPresenter(
    PlacingBidOutputBoundary
):
    def present(
        self, output_dto: PlacingBidOutputDto
    ) -> None:
        self._formatted_data = {
            "current_price": f'${output_dto.current_price.quantize(".01")}',
            "is_winning": "Congratulations!"
            if output_dto.is_winning
            else ":(",
        }

    def get_presented_data(self) -> dict:
        return self._formatted_data
```

Finding an appropriate output data type for *Presenters* which returns through `get_presented_data` may be tricky. In Python returning `dict` is the best bet as it could be passed down to template rendering function. Popular templating engines accept template object and a `dict` instance with data for prepared placeholders. However, this diminishes *Presenter's* responsibility. This problem does not exist when a *Presenter* does not return data but is able to actually *present* result of the process. This topic will be discussed further in the next chapter.

VIEW MODEL

This is nothing more but another *Data Transfer Object* that is obtained from a *Presenter* to be passed down to the *View*. In this case, a simple `dict` does the job, because most templating engines used in Python web frameworks accept such a format. However, if there is a need for more control over the structure of a View model, then introducing a class would do the trick.

USE CASE

Use Case is the most interesting part of the Clean Architecture, where something is finally happening. *Use Case* implements *Input Boundary* and orchestrates an entire business process:

```python
class PlacingBidUseCase(PlacingBidInputBoundary):
    def __init__(
        self,
        data_access: AuctionsDataAccess,
        output_boundary: PlacingBidOutputBoundary,
    ) -> None:
        self._data_access = data_access
        self._output_boundary = output_boundary

    def execute(
        self, input_dto: PlacingBidInputDto
    ) -> None:
        auction = self._data_access.get(
            input_dto.auction_id
        )
        auction.place_bid(
            input_dto.bidder_id, input_dto.amount
        )
        self._data_access.save(auction)

        output_dto = PlacingBidOutputDto(
            input_dto.bidder_id
            in auction.winners,
            auction.current_price,
        )
        self._output_boundary.present(output_dto)
```

This example is intentionally kept simple. It does not take into consideration any edge cases or error handling - it is just to reflect what was shown in the sequence diagram. Firstly, we retrieve **`Auction`** *Entity* using an implementation of **`AuctionsDataAccess`**. Having an *Entity* instance, we call **`place_bid`** method. The latter is a command - it is to change the state of an *Entity* but does not return any value. In the next step, we persist changes using an implementation of **`AuctionsDataAccess`**. Finally, we assemble an instance of **`PlacingBidOutputDto`**, by feeding it with data got from query methods on **`Auction`** *Entity* - `winners` and `current_price` property. In the last step, we pass `output_dto` into *Output Boundary* **`present`** method call.

One interesting thing here is how `data_access` and `output_boundary` are created. They are not explicitly instantiated by **`PlacingBidUseCase`** - rather, they are passed into **`__init__`** (a Python's rough equivalent of constructor). We know for sure that objects cannot be instances of **`AuctionsDataAccess`** or **`PlacingBidOutputBoundary`** because they are abstract. Actually, we have concrete implementations of these interfaces, namely **`PlacingBidWebPresenter`** and **`DbAuctionsDataAccess`** respectively. It is crucial for **`PlacingBidUseCase`** to not know what exact implementation is it using. Why? Because they belong to higher layer and it would be against the Dependency Rule for *Use Case* to know anything about upper layers. On the other hand, **`AuctionsDataAccess`** and **`PlacingBidOutputBoundary`** both belong to *Application* layer, so they can safely be referred in the *Use Case*.

A technique of passing dependencies into an object's constructor is called Dependency Injection. Normally, one would not do that manually and automate that by using a dependency injection container. In short, the latter behaves a bit like a dictionary that keeps concrete implementations under interfaces they implement. There must be a configuration somewhere in the codebase that instructs dependency injection container what it should do whenever someone requests an instance of a given type. For the snippet above, if we would use `inject` library, it might look as follows:

```python
import inject


def di_config(binder: inject.Binder) → None:
    binder.bind(
        AuctionsDataAccess, DbAuctionsDataAccess()
    )
    binder.bind_to_provider(
        PlacingBidOutputBoundary,
        PlacingBidWebPresenter,
    )


inject.configure(di_config)
```

Once configured, `inject` stores a mapping between types (usually abstract classes) and their implementations. More information on that subject will be presented later. For now, it is sufficient to know that **`PlacingBidUseCase`** does not create its dependencies nor knows which implementations of abstract classes are used.

DATA ACCESS INTERFACE

This specifies an interface for retrieving/storing *Entities*. The simplest interface will consist of two methods - get by primary key and save.

```
class AuctionsDataAccess(abc.ABC):
    @abc.abstractmethod
    def get(self, auction_id: int) → Auction:
        pass

    @abc.abstractmethod
    def save(self, auction: Auction) → None:
        pass
```

DATA ACCESS

`DbAuctionsDataAccess` is a concrete implementation of `AuctionsDataAccess` abstract class that is to use whatever data store we use for storing our dear auctions. We could be keeping data in files, an RDBMS, NoSQL database or some external service hidden behind REST API.

ENTITIES - BID

Finally, we land in a domain layer to model a bid as a separate *Entity*.

```
@dataclass
class Bid:
    id: Optional[int]
    bidder_id: int
    amount: Decimal
```

This is a simple class that has three fields: `id`, `bidder_id` and `amount`. First one is optional as newly created bids (before writing them down somewhere) will not have IDs. There is another approach - to use UUID and always give new bids an ID[7]. For the sake of simplicity, I am not adding fields for creation time etc.

ENTITIES - AUCTION

An auction in the simplest form will need a public method for placing a new bid, getting winners list and current price.

[7] Vaughn Vernon, *Implementing Domain-Driven Design*, Chapter 5. Entities, Unique Identity, Application Generates Identity

```python
class Auction:
    def __init__(
        self,
        id: int,
        starting_price: Decimal,
        bids: List[Bid],
    ) -> None:
        self.id = id
        self.starting_price = starting_price
        self.bids = bids

    def place_bid(
        self, user_id: int, amount: Decimal
    ) -> None:
        pass

    @property
    def current_price(self) -> Decimal:
        pass

    @property
    def winners(self) -> List[int]:
        pass
```

Please note that `place_bid` changes an auction (mutates its state), while `current_price` and `winners` do not. Each methods of `Auction` belong to one of two distinct categories:

- *commands* that change the state and do not return any value,
- *queries* that return value but cannot change anything

This approach is known as *Command Query Separation* (*CQS*) and was originally described by Bertrand Meyer in his Object Oriented Software Construction[8] back in 1988. Bear in mind that queries are considered to be *safe* - they can be rearranged, used anywhere and will not affect the state of the system, while one has to be more careful with commands. Usually, order of invoking commands is meaningful, whereas queries can be invoked in any sequence. The reason why this pattern was applied here is that it simplifies and orders `Auction` class interface. It will also make it a bit easier to test the class.

CHAPTER SUMMARY

This chapter described a flow of control in applications implementing the Clean Architecture in its original vision.

[8] Bertrand Meyer, *Object-Oriented Software Construction*

Controller repacks request data into *Input Dto*, passes it to *Use Case* abstracted by *Input Boundary*. *Use Case* leverages concrete implementation of *Data Access* to fetch *Entities* from persistent storage (e.g. relational database), gives them commands to change their internal state. Finally, *Entities* are saved. The last step for *Use Case* is to gather data required for *Output Dto* that is to be passed into a *Presenter*, abstracted by *Output Boundary*.

All these layers and abstractions are here to distill code driven by business requirements from non-functional stuff.