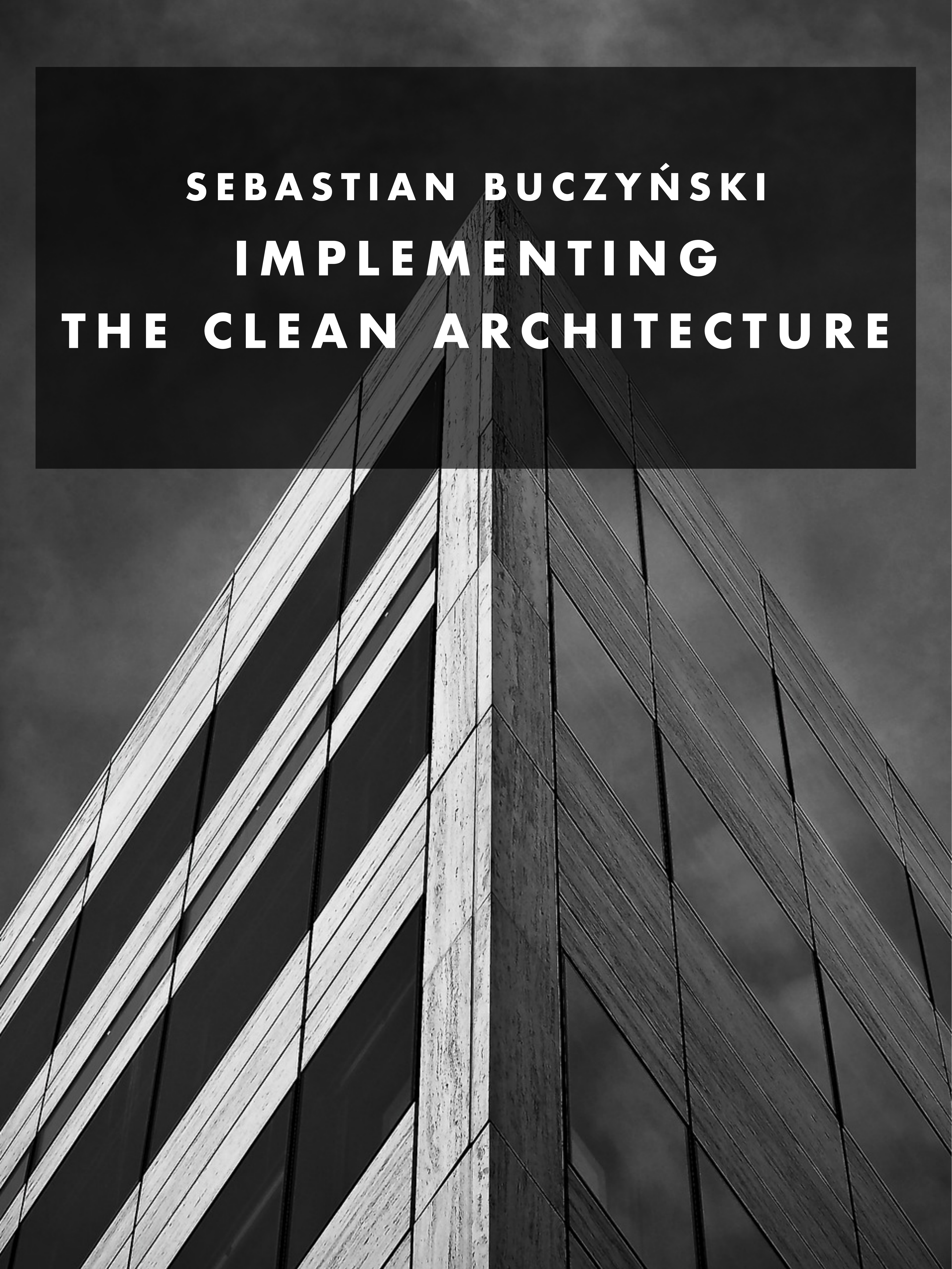
SEBASTIAN BUCZYŃSKI

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.

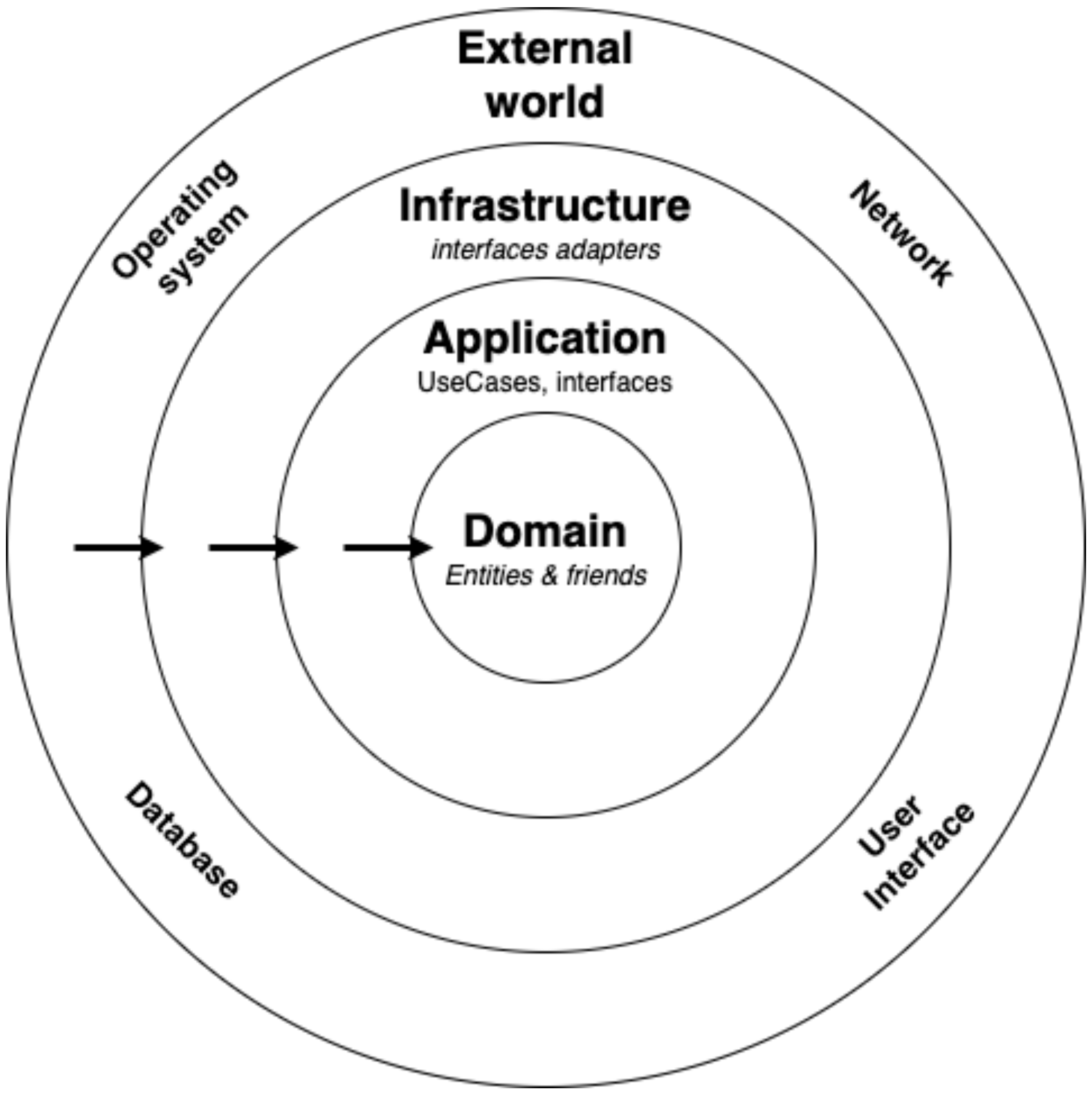


Figure 1.1 Layers of the Clean Architecture

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:

```python
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].

```python
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*.

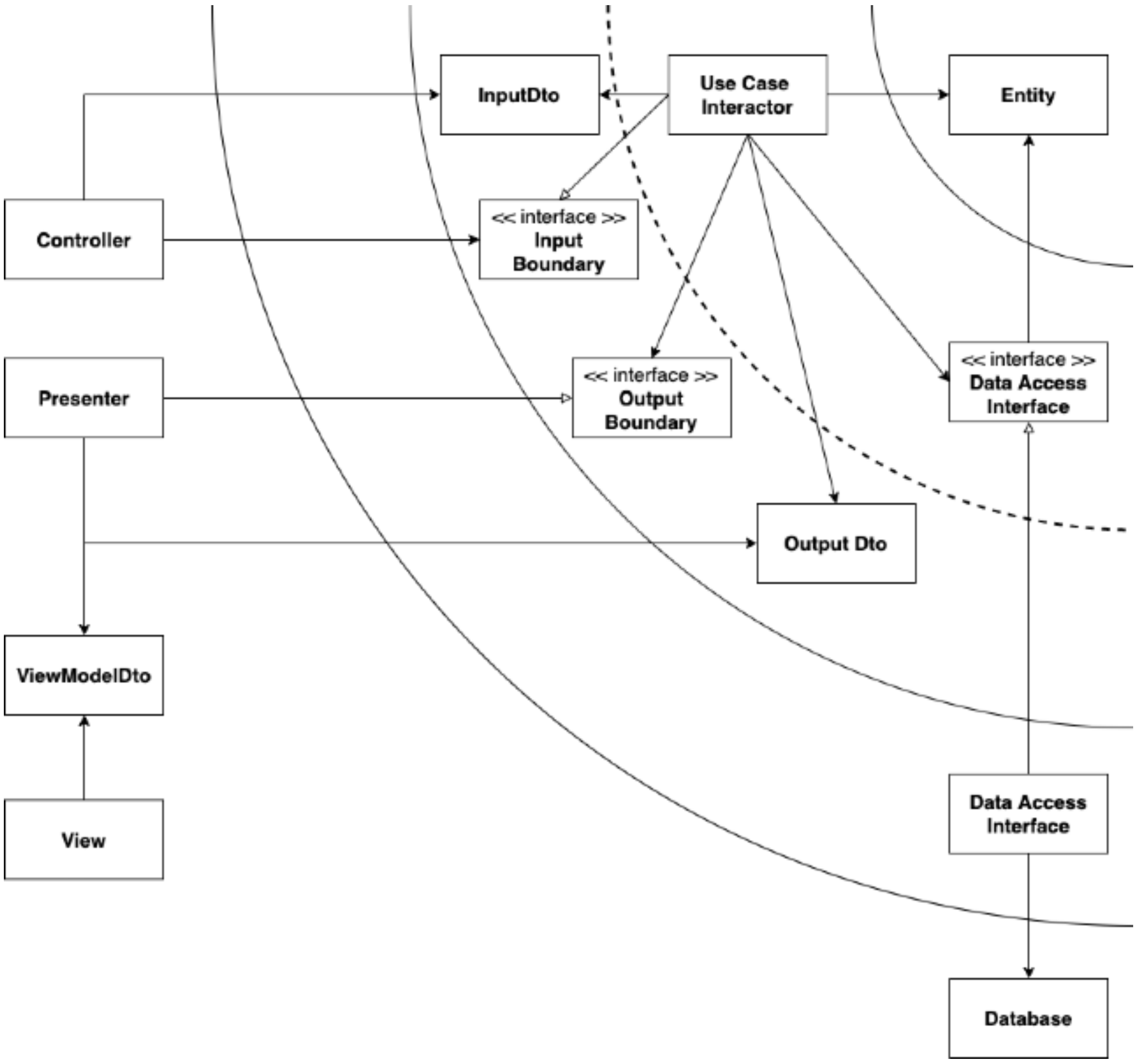


Figure 2.1 The Clean Architecture referential implementation diagram

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

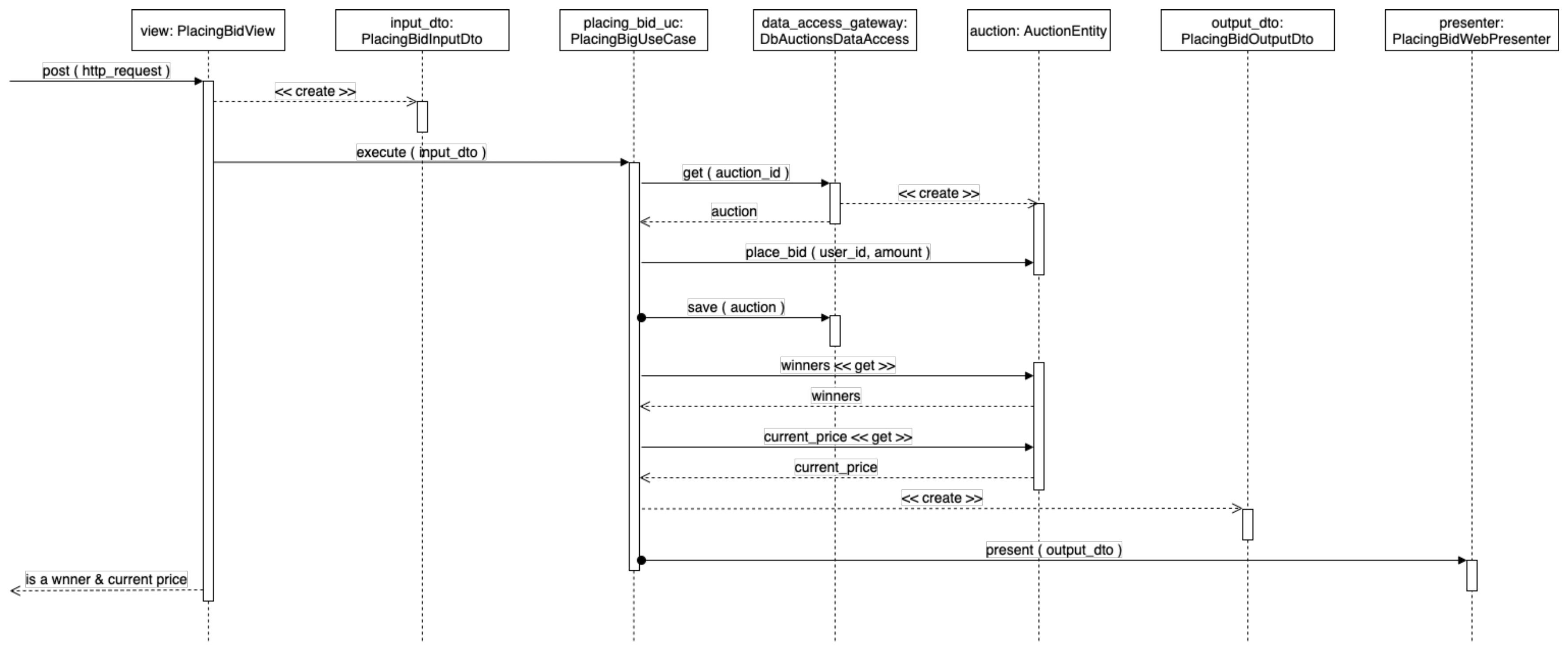


Figure 2.2 A sequence diagram of the Clean Architecture referential implementation

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*:

```
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*:

```
@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):

```
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

```
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.

THE CLEAN ARCHITECTURE MODIFICATIONS

“Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away” - Antoine de Saint-Exupery

PRESENTER DILEMMA

The original recipe for the Clean Architecture contains a lot of moving parts. Certain assumptions, like control flow ending in *Presenter*, are not suitable for flows we are accustomed to in web applications programming, e.g. in mainstream web frameworks written in Python (and many other programming languages, to be fair). It is expected from a developer to explicitly return some value from a *Controller* (or *View*, using MVT terminology):

```python
def index(request: HttpRequest) → HttpResponse:
    # Django will not forgive you if you do not return HttpResponse from a view
    return HttpResponse(f"Hello, world!")
```

One cannot simply hack their way around this behavior, so the only option left is to make *Presenter* return a value to *view* and push it forward from there:

```python
def index(request: HttpRequest) → HttpResponse:
    ...
    return presenter.get_html_response()
```

Not every framework will force you to use such tricks. For instance, Falcon[9] passes a response object to every *Controller*. Therefore, *Presenter* is able to freely manipulate a response without the need to return anything to *Controller*:

[9] The Falcon Web Framework https://falcon.readthedocs.io/en/stable/

```
class PlacingBidResource:
    def on_post(
        self, req: Request, resp: Response
    ) -> None:
        ...  # validation, assembling input_data
        presenter = PlacingBidJsonPresenter(resp)
        use_case_cls = inject.instance(
            PlacingBidInputBoundary
        )
        use_case_cls(presenter).execute(
            input_data
        )


class PlacingBidJsonPresenter(
    IndexOutputBoundary
):
    PRECISION = Decimal("0.01")

    def __init__(self, resp: Response) -> None:
        self.resp = resp

    def present(
        self, output_dto: PlacingBidOutputDto
    ) -> None:
        if output_dto.is_winning:
            message = (
                "Congrats! You are a winner! :)"
            )
        else:
            message = "Sorry, your bid was not high enough!"

        price_formatted = output_dto.current_price.quantize(
            self.PRECISION
        )
        self.resp.media = {
            "current_price": f"${price_formatted}",
            "message": message,
        }
```

More frameworks share this approach, especially asynchronous ones. express.js, which is node.js-based solution, also would allow for presenting response without explicit returns:

```
app.get('/', function(req, res) {
   // we manipulate response object, no returns!
   res.send('Hello, world!');
});
```

Yet another approach is to not use *Presenter* at as well as *Output DTOs* in our *Use Cases*. We assume that if the Use Case has not thrown an exception during execution, then it succeeded. If returning something to an app client is required, we issue separate queries to build a response. Separating reads from writes is known as CQRS. There is a dedicated chapter about this approach in the book.

The original control flow would change to:

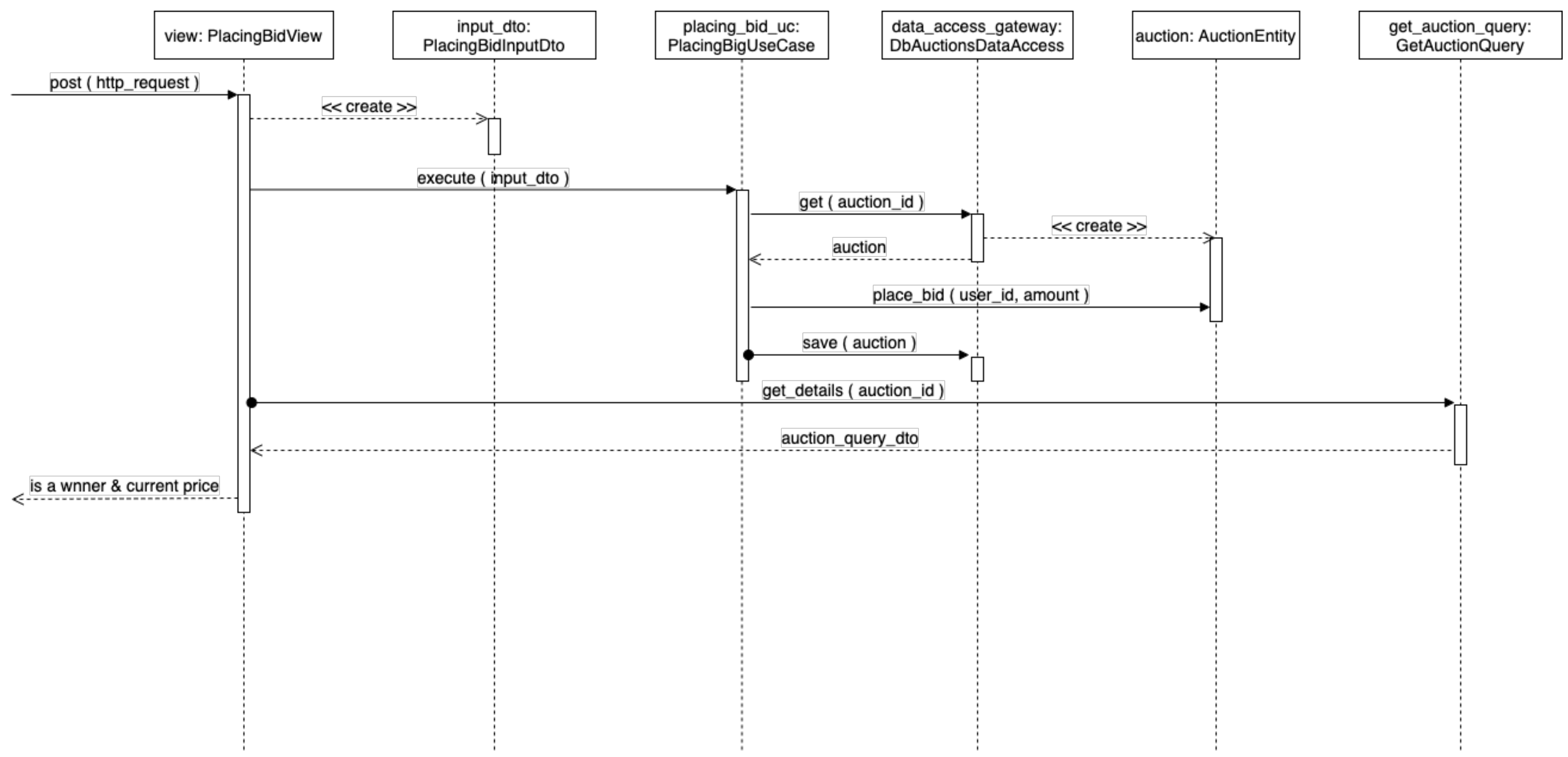

view: PlacingBidView
input_dto: PlacingBidInputDto
placing_bid_uc: PlacingBigUseCase
data_access_gateway: DbAuctionsDataAccess
auction: AuctionEntity
get_auction_query: GetAuctionQuery
post (http_request)
<< create >>
execute (input_dto)
get (auction_id)
<< create >>
auction
place_bid (user_id, amount)
save (auction)
get_details (auction_id)
auction_query_dto
is a wnner & current price

Figure 3.1 A sequence diagram of the Clean Architecture with queries instead of Output Boundary

Such an approach considerably simplifies *Use Case*, because it makes assembling *Output DTOs* unnecessary. As a result, *Output Boundary* and *Presenter* are also thrown out of the picture. On the other hand, using *Query* may involve additional round-trips to the database (or other data source), which looks like a waste. That is true, but only when we have all data required to produce the response in the *Use Case*. There is a trade-off to be made between simplifying *Use Case* and slightly lowering performance in certain cases. In case *Use Case* does not have enough data to produce response despite taking care about executing business logic, additional calls to the database are inevitable anyway.

In such a case, sticking to *Output Dto -> Output Boundary -> Presenter* forces a developer to put fetching additional data somewhere in the flow. Maybe *Use Case* should put everything that is needed for presentation to *Output DTO*? Doubtful, since it would mean that data presentation drives implementation of a business flow. It is rather *Presenter's* role to take care of having all the required data that cannot be provided by a *Use Case* whose *Output Boundary* is implemented by the *Presenter*. Leaving this to *Presenter* allows for some flexibility.

If we have to support multiple *Presenters* for every possible delivery mechanism of the system (e.g. web or CLI), then each of them can have different requirements about additional data and fetch only what it needs.

To sum up, we have two options - use *Output DTO -> Output Boundary -> Presenter* chain or abandon this part of the Clean Architecture completely in favour of queries inspired by CQRS.

GETTING RID OF INPUT BOUNDARY

Many interfaces along the way are there to provide loose coupling. One argument for keeping abstractions wherever possible is that they foster testability. The truth is that in dynamic languages that allow for monkey-patching, we can unit-test classes even if they are explicitly instantiating and using other concrete classes. In other words, tight coupling is not really an obstacle, though it lowers the quality of the design. If you can't help rubbing your eyes, I want to calm you down: I do not use neither approve monkey-patching. It is one of these dirty tricks that are going to bite a developer sooner or later. The point I am trying to make is that loose coupling itself is not a goal. A goal is to deliver features in time while having a maintainable, easily extendable codebase. **It might not be such a big deal to have *Controller* coupled to a *Use Case*.** We do not expect *Controller* to do anything but repacking request data into *Input Dto*. This integration actually can (and should!) be checked with higher-level tests.

Abstractions are a must-have in *Application* layer to establish a sharp boundary between business logic and the external world. Abstracting *Use Case* by hiding them behind *Input Boundary* in *View/Controller* is not of the same importance. The almighty Dependency Rule is not violated in this case, because *View/Controller* resides on a higher layer compared to *Use Case*.

Getting rid of *Input Boundary* interfaces has one consequence - *View/Controllers* will be tightly coupled to concrete *Use Cases/Interactors*. If that is something you can afford, there is no reason for keeping *Input Boundaries*. It should not lower the testability that much - *View/Controllers* contain little if any logic and are rather tested in end-to-end tests.

ALTERNATIVE DESIGN OF USE CASES

APPLICATION FACADE

If applications were hotels, each *Use Case / Interactor* would be an employee dedicated to looking after a single service the hotel offers. In such a design, a hotel guest willing to use spa treatments would contact Spa Treatments Keeper. If they were keen to use on-site golf field, they would talk to Golf Field Keeper etc. This approach is not a practical one - in the hotel industry, a hotel guest would be rather contacting reception - either personally or via phone call. The hotel reception is guests-facing interface of various services offered. The resemblance to *Facade* design pattern is visible to the naked eye. With regard to *Use Cases / Interactors*, having each of them as a separate class is roughly equivalent to hotel design with dedicated keepers. The alternative design assumes we have a single class (*Facade*) with separate methods, each responsible for one business flow. Effectively, all *Use Cases* are turned into methods of a *Facade*. Naturally, *Facade's* methods can use auxiliary classes to do the job (just like receptionist, passing guests' requests further), but these collaborators should not be visible neither accessed from the outside. You want something - go to the application *Facade*. This approach works nice provided that our *Use Cases* are hardly sophisticated, e.g. follow the same schema: get an *Entity* using *Data Access Interface*, call an *Entity*'s method, then save it back.

Such a pattern is dangerous when an application is not modular because such a *Facade* would quickly become enormous. Modularity will be discussed later, in probably the most important chapter of the book. Also, have in mind potentially complicated injecting dependencies.

```python
class AuctionsFacade:
    def place_bid(
        dto: PlacingBidInputDto,
    ) -> None:
        ...

    def withdraw_bid(
        dto: WithdrawingBidInputDto,
    ) -> None:
        ...
```

COMMAND HANDLERS + MEDIATOR (COMMAND BUS)

A few paragraphs before a possibility of resigning from *Input Boundary* was discussed. Even earlier, a design with *Query* replacing *Output DTOs*, *Output Boundary* and *Presenter* was shown.

Assuming that our *Use Cases* do not build *Output Dtos* and we still see decoupling from *Controllers* beneficial, we can turn our *Input DTOs* into CQRS's *Commands* and introduce a *Mediator*[10], called *Command Bus*. From now on, there is no *PlacingBidInputDto* - it becomes a *PlaceBid*, a *Command*. *Controller* still has to assemble this *Data Transfer Object*, but now it passes it into universal `dispatch` method of *Command Bus*:

```
def placing_bid_view(request) → None:
    command = PlaceBid( ... )
    # command_bus passes command to appropriate handler
    command_bus.dispatch(command)
```

During application start-up, *Command Bus* is configured to route *Commands* to *Use Cases*. To be strict, our *Use Cases* become *Command Handlers*.

The main benefit here is that we are completely decoupled from the handler. This also simplifies our *Controllers*, because they only have to know *Command* classes (e.g. *PlaceBid*) and *Command Bus*. With such an approach, *Input Boundary* becomes completely redundant.

Command Bus makes possible one more, not-so-obvious design. Our *Command* classes are *Data Transfer Objects*, which makes them easy to serialize and send over the wire. With such a design, it is easier to evolve towards a distributed, message-driven architecture.

MODELING ENTITIES USING ORM

If one uses an ORM library backed by an RDBMS, it might be tempting to reuse ORM models as *Entities*. Knowing the Dependency Rule, placing storage-coupled classes in the centre of *Domain* layer is unforgettable. First, model classes may cause leaking details of the underlying persistence mechanism into *Domain*. In turn, the latter gets unnecessarily complicated due to the extra coupling. Secondly, *Domain* built around things that rely on the database may no longer be easily tested. *Application*'s testability will also suffer because it is built around *Domain*. Thirdly, by reusing ORM models, one will not be able to create useful abstractions for things more complicated than tables in relational databases which are characterized by a flat, normalized structure. Rows from RDBMS tables are a very weak metaphor for business *Entities*, especially when graphs of objects are involved. Thinking through the prism of database rows incapacitates our innate ability to model more complex things. Last but not least - unrestricted lazy loading can be seen as an invitation for unwanted side effects in *Domain* & *Application*.

[10] Erich Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Chapter 5: Behavioral Patterns, Mediator

Database schemas are built around data, while *Entities* are there to protect business invariants. The latter are rules that have to be enforced regardless of the application context. An illustrative example of an *Entity* that has some invariants to protect is a cart in an e-commerce project. Examples:

- Whenever someone adds a new product to a cart, the number of items increases
- If someone removes an item/decreases its quantity, then we have to recalculate the total value
- One cannot add the same product twice - if they do, we increase the quantity
- A customer must not order more than ten pieces of a single product at once

If a developer cannot find invariants to protect, then it means one of two things: there is not enough information about business requirements or the problem to solve is too trivial. In the latter case, the Clean Architecture is not needed for the project.

On the other hand, if a developer clearly sees that there are non-trivial business invariants but is still keen on using models as *Entities*, let's consider a few benefits it may bring. Initially, all *Entities* will have their corresponding models with exactly the same fields. It is completely natural to perceive this phenomenon as some kind of undesired duplication. The same feeling may occur to someone writing a *Use Case / Interactor* responsible for creating an *Entity*.

"Fields of Input DTO are exactly the same as my Auction Model and Auction Entity! These three are almost identical. Something is very wrong with this approach because it causes duplication."

The cure is to look into the future - initially, when a project starts, these three may indeed have identical fields. At some stage, they will stop looking identically because they all have different reasons to change.

For example, creating an auction (as a flow in the application) is unlikely to change dynamically. We set title, choose a product and set initial price. After some time we notice it would be super comfortable to have a current price kept on the *Entity* instead of dynamically calculating it every time with a list of auction bids, so we add a field to both `AuctionModel` and `Auction` *Entity*. It does not affect *Input DTO* for creating auctions at all.

After some time a new feature request comes - show number of winners for each auction in a dashboard of an administrator. The leanest and most efficient way to achieve this would be to just add a column to `AuctionModel` and recalculate its value when we save `Auction` *Entity*. Please note the latter does not have to know anything about the new column. Data structures that appear to be identical in the beginning of the project may evolve in very different directions if only their reasons to change differ. However, there is one regularity - when Auction *Entity* changes, it is almost certain that `AuctionModel` will follow. That is fine - it comes from *the Dependency Rule*. The model depends on the *Entity* after all.

That being said, in an ideal world, we should be able to write an arbitrary pure class in a chosen language and use it as an *Entity*. Then code responsible for persisting the *Entity* should be generated automatically for us. On the other hand, given how JPA (Java Persistence API), Entity Framework (C#) or SQLAlchemy models look like, how far are we from this ideal vision?

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "auctions")
public class Auction {
  @Id
  @GeneratedValue
  private Integer id;

  private String title;
  private BigDecimal startingPrice;
  private BigDecimal currentPrice;
  private LocalDateTime endsAt;

  /* Methods omitted for readability */
}
```

Here is how `Auction` would look like in JPA - it is a pure Java class with annotations providing extra metadata for a persistence mechanism. Tentatively, it is acceptable for an *Entity* to be such a hybrid. **However, a mental model of our *Entities* will still be partially constrained because it has to fit into tables in an RDBMS.** Such a class alone does not allow for any direct interactions with the database without JPA's `EntityManager` except lazy loads. It would be really nice to block them.

Python's SQLAlchemy gives such a possibility:

```
auction_with_bids = (
    session.query(Auction)
    .options(
        joinedload(Auction.bids), raiseload("*")
    )
    .filter(Auction.id == 1)
    .one()
)
```

`auction_with_bids` from the above example will not allow for lazy loading anything that was not fetched during the original query.

One note: *Active Record* ORMs as in Django or Ruby on Rails, are utterly inappropriate for building *Domain* around them. They expose too much to *Domain* which should not be able to touch the persistence layer in any way. None of JPA, Entity Framework or SQLAlchemy implements *Active Record*. On the other hand, ORM of Django or Ruby on Rails are flagship examples of this pattern.

In conclusion, reusing ORM models as *Entities* is tempting, but it has numerous negative consequences. On the bright side, it would relieve from tedious writing code responsible for persisting. On the downside, it limits freedom in modeling *Domain* and can seriously damage testability.

CHAPTER SUMMARY

Control flow in the Clean Architecture ends in a *Presenter*, which may not always be possible to implement in specific frameworks. There are two alternatives:

1. add another method to *Presenter* interface for getting formatted data to be called inside *Controller*
2. give up entirely on *Presenters* and use *Queries* from CQRS instead

One can get rid of *Input Boundaries* abstractions if tightly coupling of *Controllers* to concrete *Use Cases* is not a problem in their project. There is also an alternative approach to decoupling with CQRS' *Command Bus*, which involves turning *Input DTOs* into *Commands* and *Use Cases* into *Command Handlers*.

It is advised against using ORM models as *Entities*. A developer is then confined to seeing through prism of database rows. Thinking in terms of such a flat data structure is limiting. *Entities* should be built to protect business invariants. In order to do so, a developer often has to reach beyond a single object and resort to objects graph instead. If there are no invariants to protect in the project, then probably the Clean Architecture is not a suitable approach.

DEPENDENCY INJECTION

ABSTRACTIONS & CLASSES EVERYWHERE!

A profusion of interfaces in diagrams of the Clean Architecture may be controversial a bit for some. Out of many programming structures, abstract classes and interfaces are the most hated and misused ones. It is not that bad if a developer follows a framework's convention that specifies what should be abstract and what should not. Using abstract classes is also commonly seen with implementations of Template Method[11] design pattern. However, proper (if any!) usage of abstractions is less common off the beaten track.

Reason to blame for such a state of affairs is a lack of an intentional design. It is said that object-oriented code of decent quality has loose coupling. To start with, what is coupling?

```python
class CreditCardPaymentGateway(PaymentGateway):
    pass


# tight coupling
class Order:
    def finalise(self) → None:
        payment_gateway = CreditCardPaymentGateway(
            settings.payment["url"],
            settings.payment["credentials"],
        )
        payment_gateway.pay(self.total)


# loose(r) coupling
class Order:
    def __init__(
        self, payment_gateway: PaymentGateway
    ) → None:
        self._payment_gateway = PaymentGateway

    def finalise(self) → None:
        self._payment_gateway.pay(self.total)
```

Consider the above example. In the first part, `Order` instantiates a concrete class `CreditCardPaymentGateway` to use one of its methods. `Order` has to know exactly what is required to build such an instance (note passing settings). This is an example of tight

[11] Erich Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Chapter 5: Behavioral Patterns, Template Method

coupling because it is highly probable that changing `CreditCardPaymentGateway` will entail adjustments in `Order`. The second part of the listing presents loosened coupling. `Order` accepts an instance of a class that inherits from abstract `PaymentGateway`. `Order` is not only no longer burdened with instantiation, but also remains ignorant of the type of `payment_gateway`. Simply saying, coupling is the measure of how hard it is to change one part of code while keeping the rest working.

A reader might have noticed that this refactoring does not eliminate a need for instantiation, only shifts responsibility to anyone who will use `Order` class. Does it mean that if `Order` is instantiated in 10 places, one will have to instantiate `CreditCardPaymentGateway` 10 times? Luckily, that is not true. Dependency injection containers is a solution to this problem.

Before you start introducing abstractions in your code, bear in mind that loose coupling is not a goal in itself. What we would ideally want is to be able to do swiping changes in code without breaking half of the functionalities at the same time. At the same time, extra abstractions do not come for free and can do more harm than good if they are misused.

ABSTRACTIONS IN THE CLEAN ARCHITECTURE

The trickiest aspect of using abstract classes/interfaces is knowing where to put them. The Clean Architecture mainly thanks to a layered structure, exactly steers developers through these intricacies. In the example from the previous chapter, there is *Input Boundary* interface that abstracts *Use Case*, *Output Boundary* abstracting *Presenter* and finally, *Data Access* which is an abstraction for `DbAuctionsDataAccess`. In the last case, we are dealing with a plugin (*Data Access*) to a business process flow (*Use Case*). To explain why having an abstraction here is so important, we will play in *five whys*:

1. Why do we need to introduce an interface for `DbAuctionsDataAccess` and we are not allowing `PlacingBidUseCase` for direct access to it?

We do not want them to be tightly coupled together. `PlacingBidUseCase` does not care where *Entities* are stored or where they are retrieved from. It just needs something to perform such operations. Something that will fulfill *Data Access* contract in a form of interface, imposed by the application layer. Having an interface in between allows for loose coupling, which is vital in this case.

2. Why would I want to have `PlacingBidUseCase` and `DbAuctionsDataAccess` to be loosely coupled?

They represent two different worlds you do not want to mix. `PlacingBidUseCase` changes whenever business requirements change, which in most cases is something like *quite often*. `DbAuctionsDataAccess` can change on its own only if currently used database no longer suffices and has to be replaced. In other words, the system is no longer able to meet non-functional requirements. I imagine this is really an extremely rare situation, but may happen if for example, the project succeeds and a number of users exceeds all expectations. To sum up, these two classes change for a different reason and in a different ratio. Having them coupled together forces a developer to reason about them both whenever only one of them has to be changed. A more immediate benefit one gains is an ability to test both classes in separation.

3. Why would I allow for testing these classes separately? In a production environment they will always be used together, so what guarantee do I have everything will be fine once we deploy the project?

A lot of time will be saved. Code inside `PlacingBidUseCase` runs entirely in memory; there is no need for calling external services, doing disk operations, etc. Therefore, it is very fast. At the same time, it is a crucial part where business requirements are materialized. There are at least a few possible scenarios with different outcomes. In contrast, `DbAuctionsDataAccess` is responsible for non-functional requirements. In order to fulfill them, `DbAuctionsDataAccess` leverages an RDBMS, so it needs to perform I/O operations. This is a few orders of magnitude slower than executing code residing in memory. Finally, in `DbAuctionsDataAccess` there are usually no alternative scenarios. No if-statements, no branches. If there is no possibility to test these two strikingly different classes independently, a developer would have to test them together. This can mean a huge time waste just because `DbAuctionsDataAccess` cannot work without interacting with a database.

Concerning guarantee of correctness, even the most exhaustive test suites testing these classes in separation does not assure that the entire project will work fine. To be certain, we need higher-level tests that will check if `PlacingBidUseCase` and `DbAuctionsDataAccess` co-operate smoothly. Of course, such a test would somehow duplicate checks we are doing separately for both classes, but that is a trade-off worth making. An end-to-end test executed via a REST API (or user interface) can be checking a happy path not only to assure that `PlacingBidUseCase` works together with `DbAuctionsDataAccess`, but also ensure there is no friction between all other components involved. The testing strategy is a huge topic. A separate chapter later in the book has been devoted to it.

4. What else can I get from loose coupling? What are other problems I avoid by getting rid of tight coupling between `PlacingBidUseCase` and `DbAuctionsDataAccess`?

A reduced cognitive load since a developer is able to reason about one class at a time without bothering with the second one. Only its interface is to be taken into account.

If `PlacingBidUseCase` was responsible for managing `DbAuctionsDataAccess` instance (for example, by creating it), it would also have to manage all of its dependencies. Can we easily get database connection object into `PlacingBidUseCase` and pass it further into `DbAuctionsDataAccess`? Or maybe `DbAuctionsDataAccess` is responsible for establishing a connection, but it still needs a username, password, hostname and port number. Where would `DbAuctionsDataAccess` get these configuration options from? Should `PlacingBidUseCase` provide them?

5. How to manage loosely coupled classes then? Are there any patterns, libraries?

The answer is Inversion of Control (IoC) and Dependency Injection (DI).

INVERSION OF CONTROL

IoC relieves classes from creating their dependencies, letting them only specify what interfaces they need. What concrete implementations will be really used is no longer their business. In the Clean Architecture, it basically means that layers above the application layer are making decisions what implementations will be used by *Use Case*.

Dependency injection with *Inversion of Control* is used even in Django. The best example is `cache` module. Its basic usage is very simple:

```
# put string 'world' under key 'hello' for 30 seconds
cache.set("hello", "world", 30)

# retrieve whatever is stored under 'hello' key or None
cache.get("hello")
```

A developer gets `cache` object by simply importing it - there is no need for instantiating anything. It is ready to use right away. Looking so simple on the surface, a lot of things happen behind the scenes. There are many available backends for cache. One of them keeps data in memory. Another is based on memcached[12]. There is even a dummy implementation that never saves anything anywhere and always returns `None`, simulating cache miss.

[12] Memcached https://memcached.org/

However, a developer just sees an object `get` with and `set` methods. `cache` is, in fact, a proxy[13] to concrete implementation.

Yet, there must a way to define which real implementation will be proxied by `cache`. Or speaking more generally - to map interfaces into concrete classes. *Dependency Injection* technique exists exactly for this purpose. So-called *Inversion of Control Container* is to orchestrate the process of instantiation and configuring objects. Going back to the example from the beginning of the chapter - **`Order`** requiring an instance of a subclass of **`PaymentGateway`** - developer would only configure IoC Container to instantiate **`CreditCardPaymentGateway`** when **`PaymentGateway`** is requested.

Naturally, there is no reason to write your own *Inversion of Control Container*. There are many excellent libraries available to use. C# programmers have outstanding Autofac[14] at their disposal. Java people can use Guice[15] (Java) while Pythonistas should check out Injector[16] (Python). Even if you are not going to use any of them, their manuals alone provide lots of useful knowledge about dependency injection and inversion of control.

Returning to Django and its `cache`, the framework configures cache backend (does the job of *IoC Container*). A developer is just to configure what they would like to have under the hood and lets Django do the rest.

```
CACHES = {
    "default": {
        # choose concrete implementation
        "BACKEND": "django.core.cache.backends.memcached.MemcachedCache",
        # provide extra details, e.g. location of memcached server
        "LOCATION": "127.0.0.1:11211",
    }
}
```

Initializing concrete backend class and wiring it together with `cache` is just a stage in booting up an application. This is the right time for *Inversion of Control Container* to be prepared for any application/language etc. The mapping between interfaces and concrete

[13] Erich Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Chapter 4: Structural Patterns, Proxy

[14] Autofac https://autofac.org/

[15] Guice https://github.com/google/guice

[16] Injector https://injector.readthedocs.io/en/latest/

classes is a part of configuration after all. It should be done before any real job is taken care of by an application.

IOC CONTAINER VS SERVICE LOCATOR

In Django whole injection ceremony is hidden, and that is a good thing. However, how a real *IoC Container* should be used? Let's say it has been configured. Should it be freely used by arbitrary code in the project? For example, whenever we need `Cache`, we could just call `container.get(Cache)`, and we would receive an instance of a concrete class. Such kind of a global registry is called *Service Locator* and is largely considered an antipattern[17].

Let's take a referential implementation of the Clean Architecture presented earlier. We want to call `PlacingBidUseCase` from the outer layer. It is abstracted by `PlacingBidInputBoundary`, so we ask the container to provide it. We expect to get an instance of a concrete class - `PlacingBidUseCase`. The latter requires `AuctionsDataAccess`, so inside `PlacingBidUseCase` we use container once again. We get an instance of `DbAuctionsDataAccess`. If `DbAuctionsDataAccess` requires a database connection or settings object we would also have to use container inside it to fetch required stuff. I guess it is obvious where does this thing go. This is how *Service Locator* looks in the wild.

The right way to do it would be to ask the container for an instance `PlacingBidInputBoundary` (1st step from the previous example). A decent *IoC Container* should be able to resolve the whole graph of dependencies to create everything for us. No need to pollute code with calls to container. A minimal example using Injector:

```python
class AuctionsRepo:
    def get(self, auction_id: int) -> None:
        pass


class PlacingBidUseCase:
    @inject  # instruct Injector to perform injection
    def __init__(
        self, repo: AuctionsRepo
    ) -> None:
        self._repo = repo
```

[17] Mark Seemann, *Service Locator is an Anti-Pattern* https://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/

```
injector = Injector()
# injector automatically assembles dependencies
use_case = injector.get(PlacingBidUseCase)
# AuctionsRepo has been created and injected
assert isinstance(use_case._repo, AuctionsRepo)
```

It is also recommended to use some glue code with a web framework (or any other delivery mechanism) that will hide calls to the container. An example of flask-injector code should shed some light:

```
# with framework integration
@app.route("/auction/bids", methods=["POST"])
def auction_bids(
    input_boundary: PlacingBidInputBoundary,
) -> Response:
    input_boundary.execute( ... )
    ...


# without framework integration
@app.route("/auction/bids", methods=["POST"])
def auction_bids():
    # manual call to container required :(
    input_boundary = container.get(
        PlacingBidInputBoundary
    )
    ...
```

DEPENDENCY INJECTION VS CONFIGURATION

The application can be configured differently for different environments. A developer will surely prefer to use a local database instead of the production one on their computer. Perhaps it would also help to disable some external vendors and use stubs instead. *Dependency Injection* shines here - if only we have interfaces and *IoC Container* in place. This opens another door - techniques such as Branch By Abstraction[18] for gradual migrating from one solution to another. For example, we might be developing `MongoDbAuctionsDataAccess` for days/weeks, but we must not use it in production until the implementation is finished. A programmer who develops `MongoDbAuctionsDataAccess` can use it locally, while the rest of the team uses stable `DbAuctionsDataAccess` at the same time. Making *Dependency Injection*

[18] Martin Fowler, *Branch By Abstraction* https://martinfowler.com/bliki/BranchByAbstraction.html

dependent on configuration makes it easier to implement feature toggles[19] without cluttering the codebase with awkward `if` statements. Instead, there will be polymorphic calls which are much cleaner and easier to read.

The rest depends on dependency injection library we decide to use.

CHAPTER SUMMARY

Loose coupling achieved by reasonable use of abstract classes and interfaces allows for increased testability and maintainability of code. Produced design is more elegant and flexible. This is especially visible in the Clean Architecture, which explicitly states the need for interfaces between business logic and infrastructure-specific code.

Management of concrete classes implementing interfaces required by the application layer is dealt with by a mechanism called *Dependency Injection*. To fully leverage the technique, a mature *Inversion of Control Container* is needed. The latter keeps configuration and knowledge about instantiation of all classes injected in the project. It should be capable of creating whole graphs of objects when asked for one.

In an ideal world, Application and Domain layers remain thoroughly ignorant of the existence of *Inversion of Control Container*. To achieve it, one has to avoid *Service Locator* anti-pattern.

[19] Pete Hodgson, *Feature Toggles* (aka Feature Flags) https://martinfowler.com/articles/feature-toggles.html

CQRS

INTRODUCTION

The mysterious acronym CQRS stands for *Command Query Responsibility Segregation*. It is a pattern for separating the code that alters the state of the system from the code that does not change anything but returns data. Code that changes state of the system is referred to as *Commands*, while the latter constructs for retrieving data are called *Queries*.

Think of a typical query operation. In an e-commerce web application, it might be displaying a list of available products or getting user's delivery addresses. These *Queries* have few things in common:

- they do not have any side effects
- state of the application remains intact, no matter how many times a day we request data
- it does not matter if we request products first or delivery addresses - we will get exactly the same results
- usually, they boil down to just fetching data from a database

In other words, *Queries* are simple and safe (they don't alter system state). What particular data should they return is imposed by a user interface which will eventually present it to the user. So the only reason for *Queries* to change is to conform with the user interface.

Commands, in contrast, are very different creatures. Their sole purpose is to change the system state. In an e-commerce application examples might be adding an item to a cart, adding or removing delivery address. There are few special things about *Commands*:

- they always change the state of the system provided their execution succeeded
- a sequence of *commands* execution matter - it simply does not make any sense to remove delivery address before we have added one
- all business rules have to be enforced during their execution to guarantee that the system is not in an incorrect state afterwards

In conclusion, *Commands* are unsafe and a few orders of magnitude more complex than just reading data from a SQL database or another data store. *Commands* are affected only by changes in business requirements. New shiny user interface after weeks of redesign should not change the rules of the game.

Commands and *Queries* are very different from each other. *CQRS* focuses on this dichotomy. The division is not about different classes naming schema (i.e. *GettingAuction, PlaceBid*). It reaches far beyond. In its most extreme form, CQRS proposes separate stacks for *Commands* and *Queries*:

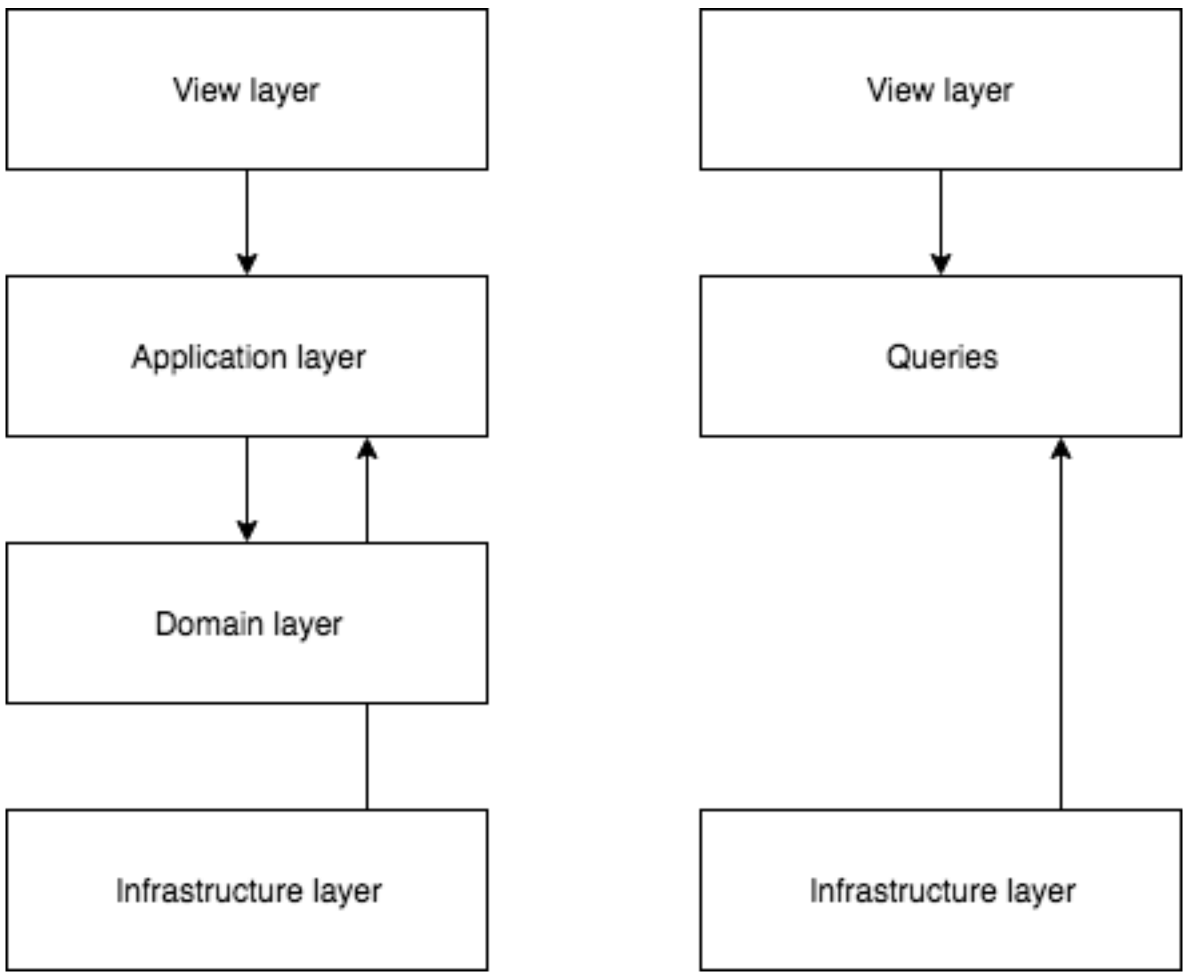


Figure 4.1 Separate stacks for commands and queries

The left side of **Figure 4.1** shows the write stack (for *Commands*). We see here all layers from the Clean Architecture. The right side of the picture is occupied by the read stack (for *Queries*). One can immediately tell there are less moving parts in the query stack. Enigmatic *Queries* layer plays the same role as *Application* layer - it will contain classes (*Queries*) that can be directly used by upper layers, for example, web interface. If we were to find an analogy for any building block from the Clean Architecture, it would be *Use Cases*. Both *Query* and *Use Case* are deliberately exposed to be used by the outer world. The difference is that *Query* must guarantee it will not alter the state of the system in any way, while *Use Cases* do not make such promises.

WHAT DOES IT HAVE TO DO WITH THE CLEAN ARCHITECTURE?

What was discussed so far in the book perfectly fits the write stack definition. We are dealing there with the essential complexity of the project, carefully modeling business rules in code. In CQRS building blocks look a bit differently, but mostly they do the same thing. In write stack of CQRS, we would be using *Commands*, which are *Data Transfer Objects*. In an implementation, they can be indistinguishable from *Input DTOs*. The latter is passed to *Use Cases*, while *Commands* are executed by *Command Handlers*. Both *Use Cases* and *Command Handlers* represent a business scenario in code. As a result, one does not have to know about the existence of *Command Handlers* to use services of the application. Since we replace calling a *Use Case* with sending a *Command*, it is much easier to distribute an application written in such a way. *Commands* become messages which can be then sent over the wire.

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

The Clean Architecture does not say anything specific about query operations. Hence, it is assumed that all scenarios like loading list of delivery addresses, getting details of the user etc. are to be implemented using *Use Cases*, *Entities*, *Data Access Interfaces* and *Presenters*. It is a lot of work for simple and safe operation. This is an excellent opportunity to supplement the Clean Architecture with a pattern stolen from CQRS.

SEPARATE READ STACK - WHY?

The main argument in favor of leveraging *Queries* is a much simpler implementation. Simply put, considerably less code is needed to achieve the same result. One does not sacrifice any advantage of the Clean Architecture because read operations are safe. Business rules are not applicable in the context of queries since they do not affect the state of the system.

Secondly, using *Queries* is bigger freedom in modeling. A developer does not have to reflect every requirement about viewing data in the write stack and vice-versa. In other words, even if models have to be enriched with another field because it has to be available on the API or user interface, they do not necessarily have to be added to entities. The read stack should be trivial to use and as handy as possible.

The final advantage is leaving room for optimizations and enabling scalability. Read stack works best when it can use a denormalized data store. Since *Queries* are implemented separately, we are free to use different tables or even another database that can be asynchronously fed with data.

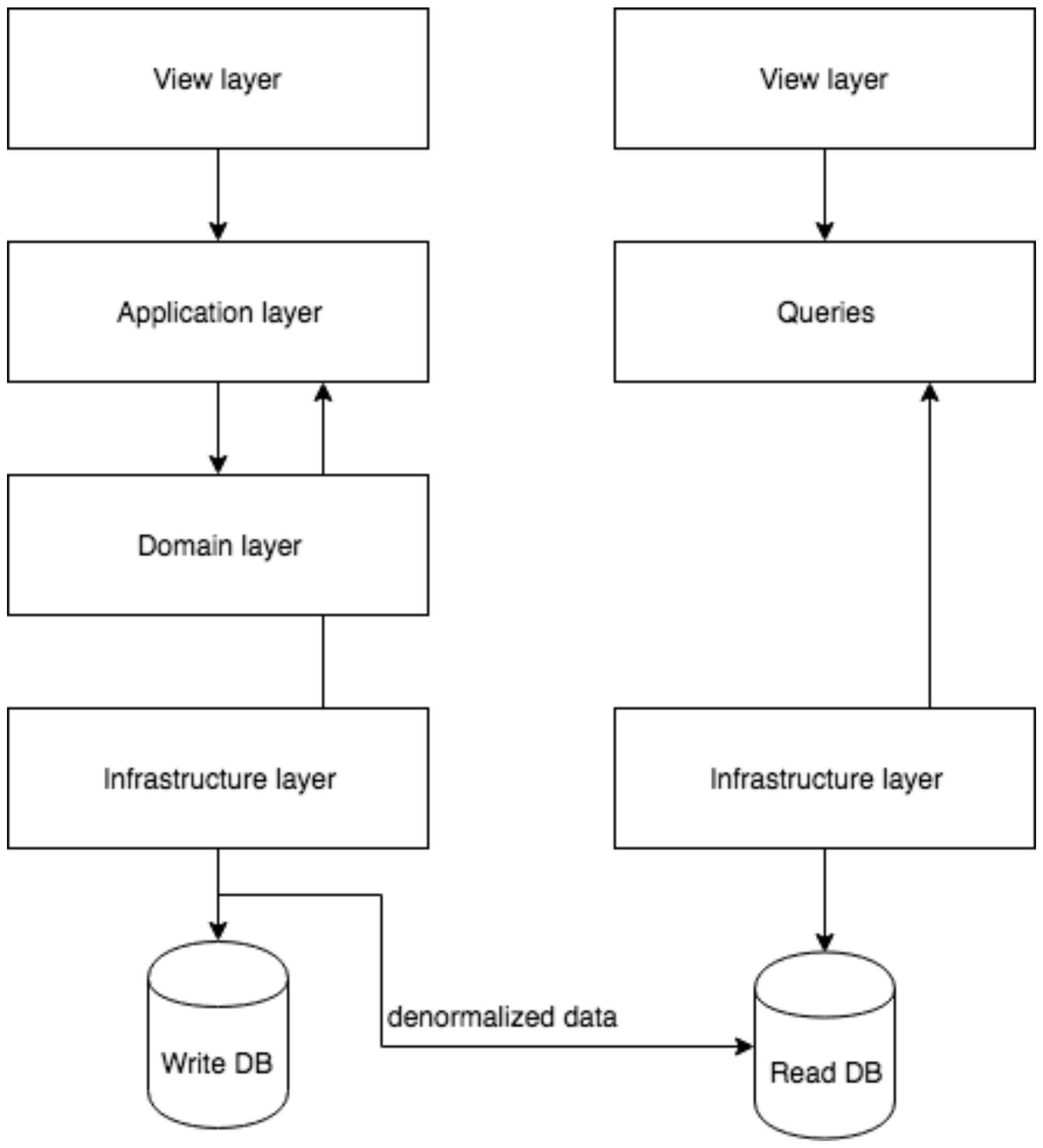


Figure 4.2 Specialized read database

SEPARATE READ STACK - HOW?

Keeping data for Queries in a separate database is an extreme form of CQRS. There can be several different approaches:

- The same database, the same tables, just different code for accessing it
- The same database, different tables filled synchronously
- The same database, different tables, filled asynchronously

- Different databases, filled asynchronously

As you can see, there are many possible approaches. In the book, I will be sticking to first, the simplest way of doing CQRS. The main difference will be implemented in code accessing data. However, patterns should not differ a lot even if we denormalize our data to a specialized, read the database.

QUERY AS DTO

In this approach, every Query is represented by a single class being a *Data Transfer Object*. The class is to represent the intention of getting some data along with the required parameters. It does not contain any logic of execution. The latter should be placed in a *Query Handler*, another class or function. Executing *Query* involves constructing a query class and passing it into *Query Handler*. There is an option of adding another level of indirection between - one can create a class responsible for dispatching queries to concrete handlers. This pattern is called *Query Bus*. With the latter, we do not have to know anything about concrete handlers.

With regard to the Clean Architecture and its dependency principle, *Query* class belongs to *Application* layer. Concrete *Query Handler* obviously should be placed in *Infrastructure* layer.

This approach resembles *Command - Command Bus - Command Handler* combination with the difference that in dispatching a *Command* **never** returns any result, while dispatching a *Query* has to.

```python
@dataclass(frozen=True)
class GetListOfDeliveryAddresses(Query):
    user_id: int

    Dto = List[DeliveryAddress]


def query_handler(
    query: GetListOfDeliveryAddresses,
) -> GetListOfDeliveryAddresses.Dto:
     ...


# using via QueryBus
query = GetListOfDeliveryAddresses(user_id=1)
result = query_bus.dispatch(query)
```

QUERIES AS SEPARATE CLASSES

In the second approach, we still create new classes for every query, but this time there is no *Query Bus* or *Query Handler* involved. Each query is an abstract class placed in *Application* layer and has its concrete implementation lying in *Infrastructure*.

```python
# somewhere in Application layer
class GettingListOfDeliveryAddresses(Query):

    Dto = List[DeliveryAddress]

    def __init__(self, user_id: int) -> None:
        self.user_id = user_id

    @abc.abstractmethod
    def execute(self) -> Dto:
        pass


# in Infrastructure layer
class SqlGettingListOfDeliveryAddresses(
    GettingListOfDeliveryAddresses
):
    def execute(
        self,
    ) -> GettingListOfDeliveryAddresses.Dto:
        models = self.session.query(
            Address
        ).filter(
            (Address.type == Address.DELIVERY)
            & (Address.user_id == self.user_id)
        )

        return [
            self._to_dto(model)
            for model in models
        ]
```

This approach looks more familiar in the context of the Clean Architecture, especially when we think about relations between abstract *Input Boundary* and its implementation - *Use Case*. Whenever one wants to invoke logic starting from web view, they request *Input Boundary*. The dependency injection mechanism is then responsible for instantiating concrete *Use Case* corresponding to requested *Input Boundary*. With *Queries* this can look the same. One requests an abstraction (*Query* living in *Application* layer) and machinery under the hood returns a concrete implementation from *Infrastructure*.

```python
# dependency injection configuration
@inject(config=Config)
def configure(binder, config):
    binder.bind(
        GettingListOfDeliveryAddresses,
        to=SqlGettingListOfDeliveryAddresses,
    )


# in web view
@app.route(
    "/delivery_addresses", methods=["POST"]
)
def auction_bids(
    query: GettingListOfDeliveryAddresses,
) -> Response:
    result = query.execute(
        user_id=current_user.id
    )
    ...
```

READ MODEL FACADE

The third approach is the most flexible, yet a little controversial. It comes down to exposing query interface from underlying infrastructure directly to view layer, completely bypassing *Application* or *Domain* layers.

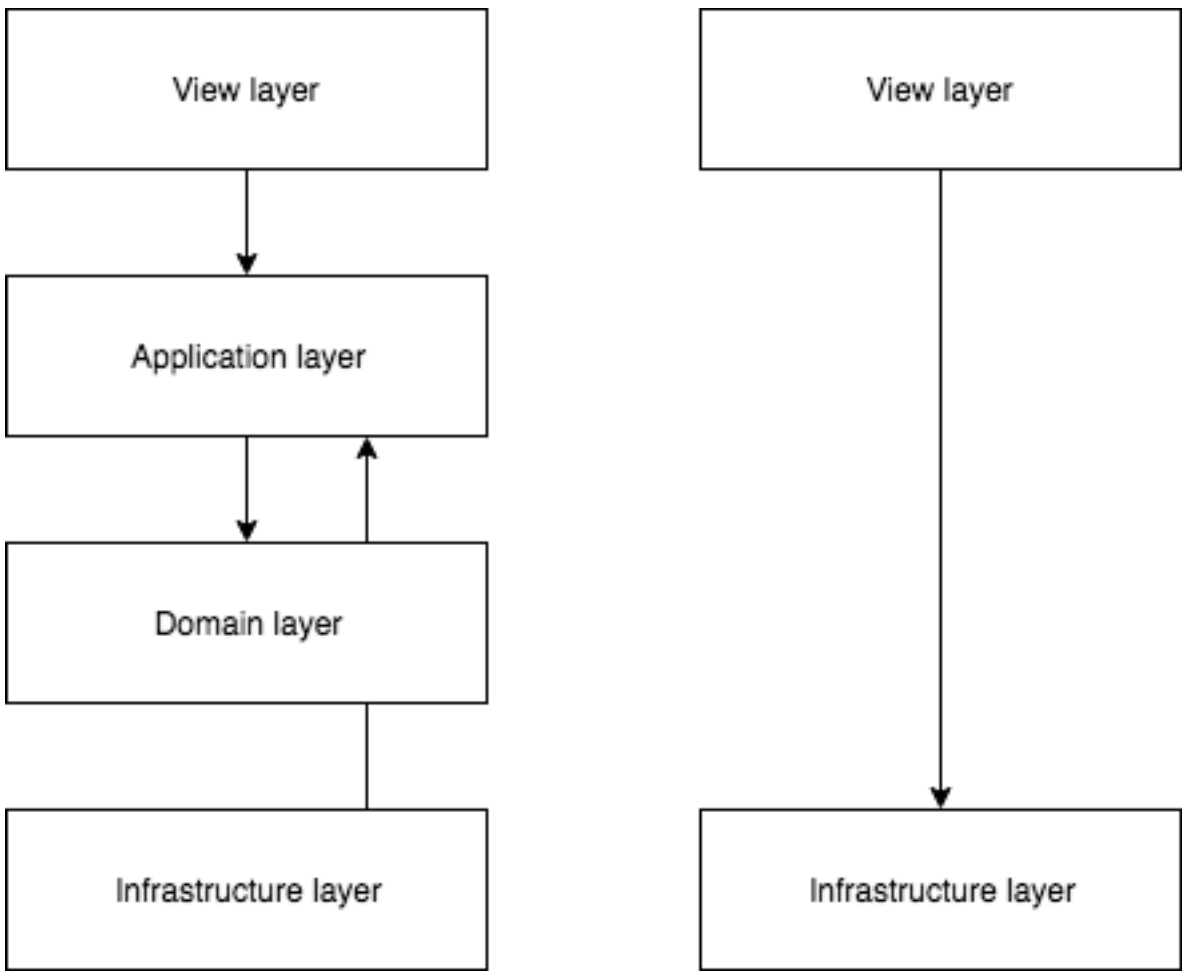


Figure 4.3 Bypassing Application and Domain layers with Read model facade

Flexibility comes from removing the need for writing specialized queries for each view and constraining these details to *Infrastructure* layer.

Given what we know about the read stack, even if *Read Model Facade* looks controversial, it is still 100% safe. At least as long as we can guarantee that no one can use exposed read model facade to mutate any data in the datastore.

Achieving this is a bit tricky with common Python tools. Django ORM requires a dedicated Manager that will raise exceptions for insert/update/delete operations. With SQLAlchemy, the only safe way is to execute Query using separate, read-only connection to the database. .NET developers have a more convenient method for achieving the same result[20].

CQRS VS REST API

CQRS looks very good on paper, but if a developer tries to fit it into REST API, they may get few unpleasant surprises. They all origin in differences between a lifetime of an HTTP request and read/write stacks dichotomy.

The common expectation for a REST APIs is that after mutating data, we get changed entity in response. If our commands are synchronous and we know immediately if they succeeded, then in view we issue query immediately after executing a command.

```python
# commands never return any result!
command_bus.dispatch(
    UpdateEmailCommand(
        user_id=1,
        email="sebastian@cleanarchitecture.io",
    )
)
result = query_bus.dispatch(GetUser(user_id=1))
# use result to build view model
```

If we execute commands asynchronously, then we would rather respond with *HTTP 202 Accepted* and then use other transport mechanisms, e.g. WebSockets, for notifying the client about finishing operation.

However, if it is REALLY necessary to return a response, one can resort to polling for command results. Naturally, this is affordable only when we use asyncio, node.js or any other solution with coroutines when waiting for I/O is not a problem. In a classical prefork

[20] Nick Chamberlain, *The Read Model Facade* https://buildplease.com/pages/fpc-22/

model (which is for example common in PHP) we would block one of the workers during polling and dramatically limited throughput of the application.

CQRS VS GRAPHQL

A newer kid in town - GraphQL - plays really nicely with CQRS. This is because it also splits commands from queries, uses different names - mutations and …queries. It is completely up to API's client to specify what data they need after executing a mutation or not.

CHAPTER SUMMARY

CQRS is a compelling approach. This chapter described it briefly without delving too much into details. A crucial part from the point of view of the Clean Architecture apprentices is that using read stack concept will considerably benefit and simplify projects. Just reading data is safe operation (as opposed to changing it), so there is no real benefit from imposing a discipline of always having *Use Case* between *Controller* and *Infrastructure* layer.

SHARP BOUNDARY

A WORD ON COMPLEXITY

Complexity is the main antagonist of software developers. It is a measure of how difficult it is to read and understand code. Not to mention changing it. High complexity means it takes a lot of mental effort to fully comprehend control flow. It is better not to be disturbed during such a desperate struggle. Every interruption or context switch costs a lot - it is both irritating and fatiguing trying to get back on the track. Complexity manifests itself in code with if-statements, loops and many other structures that lead to nesting code. Cyclomatic complexity[21] is a commonly used measure of how complex a code fragment is. There are many static code analysis tools that keep an eye for cyclomatic complexity. For example, Pythonistas have flake8 at their disposal.

Software engineering is fighting a fierce battle against complexity. To gain an advantage over any opponent, one must get to know them first. Fred Brooks wrote in his famous paper *No Silver Bullet* (included in a book The Mythical Man-Month[22]) that there are two types of complexity: accidental and essential.

We deal with accidental complexity when the code we read is written in a way we can improve it with a finite effort. You have not heard about modulo operator (%), so you have written your own code that calculates remainder. Someone wrote a function that has over 200 lines of code, but with tools provided by IDE you can split it into four smaller ones and eliminate duplicated logic, eventually getting 80 lines of code. In short, this is how accidental complexity looks like - it is effectively combatted with clean code, keeping functions short - in other words, by being an educated engineer who leverages modern tools.

The second type of complexity has strikingly different nature. Essential complexity reflects how complicated is the problem one tries to model in code. If an application has to provide 50 features for various types of users, then cleanly written code is not sufficient for the system to be easy to understand. To be brutally frank, there is no way of getting rid of this kind of complexity. Our last resort as software developers is to learn how to manage it. On the other hand, if one can negotiate for reducing the scope and throwing out a few minor features, then they will reduce essential complexity. When I participated in an Event

[21] Thomas J. McCabe, *A Complexity Measure* http://www.literateprogramming.com/mccabe.pdf

[22] Fred Brooks, *The Mythical Man-Month*

Storming[23] session in 2018, we have spent more than hour on discussing the flow of a particular process in the application. When we covered walls with sticky notes, we had some time to delve into particular processes. We were trying to formulate an algorithm for charging fees to investors leaving and joining an investment. In both cases, money has to be transferred. Our initial idea was to make leaving investors bear all transfers fees imposed by a payment mechanism. After one hour of fruitless discussion, we called for a domain expert that put our debate to an end in less than 3 minutes. We decided that every investor in our scenario will contribute to fees, regardless if they are joining or leaving. As a result, we have drastically simplified the algorithm. Please note that we were far from the implementation phase. It was still when we used only whiteboard, markers and sticky notes to define the flow. A moral from this story is that simplifying business requirements spares developers days if not weeks of work. Codebase remains simpler.

“If something has to be compromised — cost, schedule, or scope —the default choice should routinely be scope”[24]

TWO WORLDS

It was mentioned that **essential complexity is not something that can be simply eliminated, but has to be managed**. A goal of the Clean Architecture is to have all possible complexity that root in business requirements contained in two inner layers - domain and application. That is why they should have no knowledge about the outer world - they are complex enough without such details. *Domain* and *Application* layers together form a *core* - a place where, ideally, all decisions justified by business requirements are made. Both core layers are easily testable with unit tests because they either do not have any dependencies or all of them are abstracted away. Bear in mind that unit tests are the fastest and easiest to write among all kinds of tests. Hence, it is cheap to get high coverage in this part of the project where every *if*-statement is meaningful and was written due to business requirements.

Layers above the core, namely *Infrastructure* and higher, are completely different creatures. It is almost impossible to unit test code placed there because it relies heavily on a scary world outside - networks, disks, databases. Therefore, we aim for having no decision making there at all. The control flow should be a straight line - no branches, no *if* statements. No

[23] https://en.wikipedia.org/wiki/Event_storming

[24] Tom Poppendieck, Mary Poppendieck, *Leading Lean Software Development: Results Are Not the Point*

alternative scenarios whenever possible. To reliably test code residing in these layers one has to resort to integration or higher-level tests. Of course, such tests will be few orders of magnitude slower and more complicated than unit tests which cover two core layers.

BOUNDARY BETWEEN APPLICATION AND EXTERNAL WORLD

Between two inner layers (*Domain* encompassed by *Application*) and all remaining ones there is an abyss. Control flow must not cross it carelessly. The Clean Architecture makes it hard to do something reckless there by introducing *Input Boundary* and *Input DTO*. While the former can be omitted under certain conditions, one still must pay attention to *Input DTOs*.

WRITING INPUT DTOS

It is crucial to always have the Dependency Rule at the back of the head when designing *Input DTOs*. This is the only thing that stands between core layers and scary outer world.

A developer must not pass anything that cannot be understood by *Domain* and *Application* layers. This means that *Input DTOs* can be created only using built-in data types or classes defined in one of the core layers.

However, it is unclear which building block of the Clean Architecture is ultimately responsible for data validation. Checking data correctness may not be a role of *Input DTOs*. On the other hand, it is absolutely certain that when a field from DTO is accessed by *Use Case,* we expect it to be correct, at least with regard to types.

```python
@dataclass
class PlacingBidInputDto:
    bidder_id: int
    auction_id: int
    amount: Decimal
```

For example, we should be able to trust `PlacingBidInputDto` that `bidder_id` field is a valid integer, but of course, we are not obliged to know if a bidder with such an id even exists before control flow reaches *Use Case*.

VALUE OBJECTS

Being sure about types is a really nice thing to have (now developers who work with statically typed languages smile), but it is insufficient (now they are not smiling anymore).

In code examples that were shown so far whenever I needed to represent some amount of money, I used Python built-in class - `Decimal`.

The problem is that not every valid `Decimal` makes sense as a money amount.

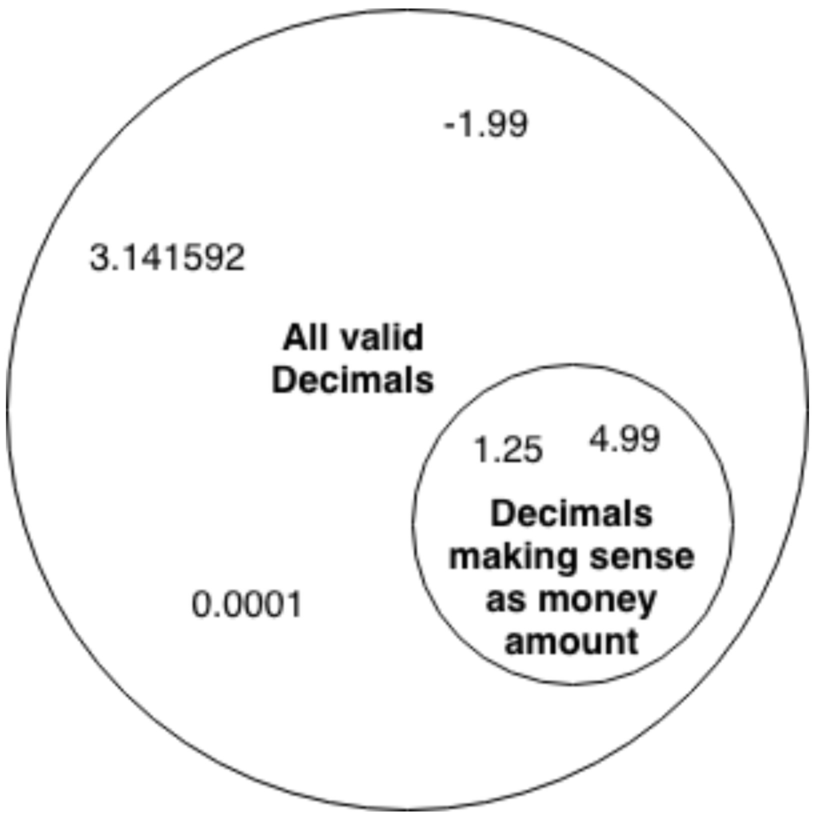


Figure 5.1 Only subset of possible `Decimal` objects can be considered a valid amount of money

Examples of `Decimal` objects that can be used as money amount:

- `Decimal('0.01')`
- `Decimal('10.99')`
- `Decimal('5.49')`

Respectively, `Decimal` that cannot be treated as money amount:

- `Decimal('-1.99')`
- `Decimal('3.1415')`
- `Decimal('-1.0E3')`

The point of these examples is to illustrate that the built-in `Decimal` type is not enough to express the concept of money. We cannot enforce the desired precision to two decimal places. Also, there is no notion of currency which is a vital property when we talk about money. It is clear we need another dedicated type. How about `Decimal`?

Before delving into implementation details, let's think about characteristics our `Money` type should have:

- it should be immutable - once created, cannot be changed
- it must not be possible to create such an object with an invalid state. `Decimal('python')` raises an exception
- it represents value, not a long-living object - it has no identity
- instances with the same value are always considered equal

+ `Money` specific:

- it should support arithmetic operations, just like `Decimal` does

Such types are called *Value Objects*. We are going to use them extensively in the end-to-end example presented later in this book.

The implementation can be driven by tests to better illustrate our expectations. We start off by specifying a base class for currency, so we can easily extend `Money` whenever new (crypto)currency emerges. `Currency` will ensure that our `Money` is opened for extensions, but closed for modifications. It means that it is enough to subclass `Currency` to parametrize behavior of `Money` class without having to modify its source code. By the way, this is called Open-Closed Principle and stands for "O" in a famous acronym SOLID[25].

```
class Currency:
    decimal_precision = 2
    symbol = None


class USD(Currency):
    symbol = "$"
```

`Currency` is required for creating `Money` instance. Since classes are objects in Python, we could just pass desired `Currency` subclass:

```
class Money:
    def __init__(
        self,
        currency: Type[Currency],
        amount: str,
    ) -> None:
        ...
```

[25] SOLID https://en.wikipedia.org/wiki/SOLID

The first property of *Value Objects* is immutability. One cannot really guarantee that in Python due to its dynamic nature. It is usually sufficient to prepend instance variables names with a single underscore, so linter and IDE can warn us. Exposing these fields as read-only properties may still be a good idea, though:

```
class Money:
    def __init__(
        self,
        currency: Type[Currency],
        amount: str,
    ) -> None:
        self._currency = currency
        self._amount = Decimal(amount)

    @property
    def currency(self) -> Type[Currency]:
        return self._currency

    @property
    def amount(self) -> Decimal:
        return self._amount
```

The second feature of *Value Objects* is that it is impossible to instantiate one with an invalid value. Hence, *Value Object* carries out validation during initialization:

```python
class Money:
    def __init__(
        self,
        currency: Type[Currency],
        amount: str,
    ) -> None:
        if not inspect.isclass(
            currency
        ) or not issubclass(currency, Currency):
            raise ValueError(
                f"{currency} is not a subclass of Currency!"
            )
        try:
            decimal_amount = Decimal(amount)
        except decimal.DecimalException:
            raise ValueError(
                f'"{amount}" is not a valid amount!'
            )

        d_tuple = decimal_amount.as_tuple()
        if d_tuple.sign:
            raise ValueError(
                f"amount {amount} must not be negative!"
            )
        elif (
            -d_tuple.exponent
            > currency.decimal_precision
        ):
            raise ValueError(
                f"given amount has invalid precision! It should have "
                f"no more than {currency.decimal_precision} decimal places!"
            )

        self._currency = currency
        self._amount = decimal_amount
```

Value Objects have no concept of identity - they are meant to be indistinguishable if only were created using the same `Currency` and equal `amount`:

```python
class Money:
    ...

    def __eq__(self, other: Money) -> bool:
        if not isinstance(other, Money):
            return False
        return (
            self.currency == other.currency
            and self.amount == other.amount
        )


Money(USD, 1) == Money(USD, "1.00")  # True
Money(USD, 1) == Money(USD, "5.00")  # False
```

Value Objects are great for expressing the intricacies of reality. For instance, let's assume that someone has to support multiple currencies. It makes little sense to compare bare amounts when one is in USD and another in EUR. Just like Python does not like when one tries to add `int` to `str`. Why not write code that will guard that?

CHAPTER SUMMARY

This chapter discussed the meaning and motivation behind establishing a boundary between two inner, core layers - *Domain* encompassed by *Application* and the rest of the world. All code that makes decisions should be placed inside *Domain* or *Application* where it can be unit tested almost effortlessly.

The Dependency Rule remains in power, so nothing from outer layers can cross the boundary. One has to translate things from the outside to concepts known in the core layers. *Value Object* is a handy pattern that helps to achieve that. It relieves inner layers of validation. *Value Objects* thanks to their immutability, can be safely passed around.

END-TO-END EXAMPLE

WHERE TO START?

It is high time to use knowledge of the Clean Architecture to build something. This chapter will guide you step by step through the development process of an exemplary project.

In an ideal world, developers would be given extensive documentation describing all requirements. That document would not change over time. The whole dev team would be working with constant velocity, live in happy relationships and never get sick. The world described does not exist. It is common to deal with change in requirements since software development is a learning process - companies discover what they need and what is expected by the market all the way through the lifetime of a project. During conference talk, I once heard that before attaching any user interface whole domain part and its complexities can be coded. I imagine this approach makes perfect sense as it deals with the most significant uncertainty in the first place. However, in an Agile environment, teams are expected to deliver business value after every iteration from the very beginning. One needs some user interface; otherwise, functionality cannot be considered delivered because there is no way to use it.

Agile methodologies do not rely on detailed documentation describing every possible scenario. Instead, they use a tool called *User Story*. These are short sentences that are much more like an invitation to talk than actual specification. Few examples:

- As a bidder I want to place a bid so that I can win the auction.
- As a bidder I want to receive e-mail notification when my bid is a winning one.
- As an administrator I want to withdraw bids so that a malicious bidder does not win an auction.
- As an administrator I want to create auctions so that bidders can place bids on them.

Majority of *User Stories* makes perfect candidates for *Use Cases*. Since the former is usually all we have, it makes perfect sense to start coding from a *Use Case* when one sits at a new, shining *User Story*. It is not the only thing to worry about in a new project, though. Since one has to ensure that the results of their work will be presentable from the very beginning, it is equally important to bootstrap project as usual. In other words, one can consider the first iteration finished when at least a basic scenario of *User Story* is coded, AND it can be shown to stakeholders or other interested parties.

That is exactly how I started my journey with the Clean Architecture. Right after creating a basic structure for the project from a template, we sat with my colleague Dominika for a pair programming session. For two hours, we were crafting the flow of the basic scenario that was part of the most complicated process in the application. After the session that resulted in *Use Case*, *Entities* and tests for them, Dominika continued to work on these on her own. She stayed in touch with the client, resolving ambiguities and discovering edge cases. At the same time, I was working on connecting *Use Case* to REST API layer and providing a simple, in-memory implementation of *Data Access*. After less than a week, we demonstrated results to the client using Postman. Frontend part was still in progress at the time, but this did not stop us from delivering business value. Oh, and of course database was not yet connected at this stage.

WALKING SKELETON

The certain amount of groundwork has to be done during the first iteration. This step is inseparably connected with the technology one is using, so I will not be delving into too many details here.

For instance, in Python, one would use Django's *startproject*, then *startapp* commands to create basic structure and configuration. For different frameworks (and technologies as well) one can resort to cookiecutter. The goal of this step is simple - to get an ability to quickly connect *Use Case* to the desired delivery mechanism. In this case, I mean REST API.

PLACINGBID USE CASE / INTERACTOR

Let's deal with the first *User Story*:

- As a bidder I want to place a bid so that I can win the auction.

This one is a perfect fit for a *Use Case*.

NAMING

Name of the *Use Case* has to reflect business scenario one is modeling. In this case, I propose to name it **`PlacingBid`**. As a class, it will have only one public method. It can either be something generic, like `execute` or more specific - `place_bid`.

```
class PlacingBid:
    def execute(self) → None:
        pass
```

ARGUMENTS

If a *Use Case* requires arguments (it is not always the case), one defines an *Input DTO*:

```
@dataclass(frozen=True)
class PlacingBidInputDto:
    bidder_id: BidderId
    auction_id: AuctionId
    amount: Money
```

`@dataclass` decorator generates `__init__` method, so one can create an instance of `PlacingBidInputDto` by:

```
input_dto = PlacingBidInputDto(
    1, 2, Money(USD, "10.00")
)
```

Note that all fields of the *Input DTO* are *Value Objects*. Here we have dedicated types for `BidderId` and `AuctionId`. A few words about that will be provided in the following section about *Entities*.

Input DTO itself is also a *Value Object*. An *Input DTO* does not have to copy the name of the *Use Case*. It makes sense to model it as an inner class of *Use Case*, hence allowing for shortening name of the *Input DTO*:

```
class PlacingBid:
    @dataclass(frozen=True)
    class InputDto:
        bidder_id: BidderId
        auction_id: AuctionId
        amount: Money

    def execute(
        self, input_dto: InputDto
    ) → None:
        pass
```

OUTPUT

If a *Use Case* outputs some data (not always a case!), one defines an *Output DTO*:

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

The rule that applies to both *Input*- and *Output DTO* is that they have to be very strict about types of fields and check if data passed in is of an expected type.

If one needs to output some data from a *Use Case*, an interface (*Output Boundary*) that will eventually present `PlacingBidOutputDto` is missing:

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

Output Boundary is to be injected during `PlacingBid` construction, so we extend its *__init__*:

```
class PlacingBid:
    def __init__(
        self,
        output_boundary: PlacingBidOutputBoundary,
    ) → None:
        self._output_boundary = output_boundary

    def execute(
        self, input_dto: PlacingBidInputDto
    ) → None:
        pass
```

UNIT TESTING

Now when we have laid groundwork, it is possible to start writing actual code. Since we defined input and output, we can write the most straightforward test for the *Use Case*. Let's try Test-Driven Development:

```
class PlacingBidTests(unittest.TestCase):
    def setUp(self) → None:
        self.output_boundary_mock = Mock(
            spec_set=PlacingBidOutputBoundary
        )
        self.use_case = PlacingBid(
            self.output_boundary_mock
        )
    def test_presents_data_for_winning(self):
        price = Money(USD, "10.00")
        input_dto = PlacingBidInputDto(
            bidder_id=1,
            auction_id=2,
            amount=price,
        )

        self.use_case.execute(input_dto)

        expected_output_dto = PlacingBidOutputDto(
            is_winning=True, current_price=price
        )
        self.output_boundary_mock.present.assert_called_once_with(
            expected_output_dto
        )
```

This is a pretty simple and naive test. Not only it is based on many assumptions (happy path - winning, auction exists, bidder exists, etc.), but will also fail because **`PlacingBid`**`.execute` has no code inside. TDD is about making tiny steps. We could make the test green with this code:

```
class PlacingBid:
    def __init__(
        self,
        output_boundary: PlacingBidOutputBoundary,
    ) -> None:
        self._output_boundary = output_boundary

    def execute(
        self, input_dto: PlacingBidInputDto
    ) -> None:
        self._output_boundary.present(
            PlacingBidOutputDto(
                is_winning=True,
                current_price=input_dto.amount,
            )
        )
```

Making such small steps may be beneficial in the beginning, but when one feels more confident, they can leave this test failing for now and start crafting missing parts. Apart from that, there is really not much we can do about **`PlacingBid`** *Use Case* for now.

AUCTION AND BID ENTITIES

As soon as we identify a name for a concept in the domain that can protect Enterprise business rules, we create an *Entity*.

NAMING

Entities are usually named in a singular form. They should be given an unambiguous name that clearly indicates their role. If you struggle to find it, talk to other people, especially domain experts and project managers.

VALUE OBJECTS FOR IDENTITY TYPES

You might have noticed enigmatic `AuctionId` or `BidderId` in the previous section about *Use Cases*. Under the hood, these are just aliases for `int`, but thanks to their naming they are much more meaningful than integers. It is also a form of encapsulation. Only very few places should really care about the type of identity, so it makes sense to hide this information. Definitions of `AuctionId` and `BidderId` are straightforward:

```
BidderId = int
AuctionId = int
```

IMPLEMENTATION

Ideally, *Entities* are classes, written in pure Python. They do not inherit from ORM' base classes etc. When we write them, we strive for as little dependencies as possible, though helpers such as Java's Lombok[26] or Python's dataclasses/attrs that can generate repetitive code for us, are welcome. This library can, for example, give us a default constructor for setting all fields defined in a class.

Going back to Python, that's how a class outline can look like:

[26] Project Lombok https://projectlombok.org/

```
class Auction:
    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> None:
        pass

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

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

This piece of code defines only a method and two properties (read-only field with an implementation for no-Pythonistas) that will be required for a `PlacingBid` *Use Case*. Obviously one needs to tell `Auction` to place a bid and finally ask for a list of winners and current price to ultimately build an instance of `PlacingBidOutputDto`.

Pause for a second and look again at `winners` method definition and return value that was annotated - `List[BidderId]`. It bears much more information than `List[int]`, doesn't it?

UNIT TESTING

Since *Entities* are mostly pure Python (or any other language) classes without external dependencies, they are trivial to unit-test. There is a plethora of test cases one can think of in terms of the `Auction` *Entity*. For starters, think about the current price. What should it be when Auction has just been created, and no one ever touched it? It is a common knowledge that auctions have some starting price to prevent items from being sold way below their value. This is one of the auctioning domain intricacies we have just discovered by wondering what should be the current price of an auction provided no one placed a bid yet.

```
class AuctionTests(unittest.TestCase):
    def test_untouched_auction_has_current_price_equal_to_starting(
        self,
    ) -> None:
        starting_price = Money(USD, "12.99")
        auction = Auction(
            starting_price=starting_price
        )

        assert (
            starting_price
            == auction.current_price
        )
```

This will fail for two reasons: `Auction` currently does not accept any parameters during construction, and current_price property always returns `None`. Such an implementation will make the test pass:

```
class Auction:
    def __init__(
        self, starting_price: Money
    ) → None:
        self._starting_price = starting_price

    @property
    def current_price(self) → Money:
        return self._starting_price
```

With ease, one can produce many unit tests that will check numerous scenarios. Such a test suite is pretty extensive and takes very little time to execute. That is exactly what we want to achieve by pulling decision-making down to the *Domain* layer where it is ridiculously cheap to code and test. **Although it is tempting to extensively unit-test our *Entities* separately, avoid that.** Much greater flexibility can be achieved if one decides to do it via tests of *Use Cases*. There is much more information on testing strategies in **Testing** chapter.

Previously, also during public speeches, I advised on unit-testing *Entities* heavily. While it still may be useful to nail certain edge cases on this testing level, I no longer think this should be the default strategy. The way I currently recommend, is to test implementation of *Entities* via tests calling *Use Cases*. At least in the beginning.

IMPLEMENTATION CONTINUED

Another *Entity* should be introduced in the process - `Bid`. It represents a single offer made by a bidder. Note we do not introduce an *Entity* for a bidder. There is no need to - we really need just their id to identify winners. If the existence of a separate *Entity* is justified by business requirements, then we would create `Bidder` as well. `Bid` *Entity* is very simple. Newly created `Bids` have no ids, but once we persist them, they all will get an identity.

```
@dataclass
class Bid:
    id: Optional[BidId]
    bidder_id: BidderId
    amount: Money
```

Going back to `Auction`, here is an implementation that actually *does something*:

```python
class Auction:
    def __init__(
        self,
        id: AuctionId,
        title: str,
        starting_price: Money,
        bids: List[Bid],
    ) -> None:
        self.id = id
        self.title = title
        self.starting_price = starting_price
        self.bids = sorted(
            bids, key=lambda bid: bid.amount
        )

    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> None:
        if amount > self.current_price:
            new_bid = Bid(
                id=None,
                bidder_id=bidder_id,
                amount=amount,
            )
            self.bids.append(new_bid)

    @property
    def current_price(self) -> Money:
        if not self.bids:
            return self.starting_price
        else:
            return self._highest_bid.amount

    @property
    def winners(self) -> List[BidderId]:
        if not self.bids:
            return []
        return [self._highest_bid.bidder_id]

    @property
    def _highest_bid(self) -> Bid:
        return self.bids[-1]
```

This is how a textbook example of an *Entity* should look like. There are no dependencies, all code is written just to enforce business rules. No messing with databases or any other external stuff.

DATA ACCESS INTERFACE (ABSTRACT REPOSITORY)

Code responsible for logic that places bids resides in `Auction` *Entity*, but the *Use Case* still does not have a possibility to fetch the *Entity* and persist it afterwards. It is time to write an interface for this. *Data Access Interface* will reside on the same layer as *Use Cases* - the *Application*.

NAMING

Since I will use a pattern called persistence-oriented repository[27], my repository will be called like [EntityName]Repository or [EntityName]Repo. Please bear in mind that there is also a collection-oriented repository[28]. Its description remains beyond the scope of this book, though.

IMPLEMENTATION

Persistence-oriented repository - in its basic (and sufficient for the example) form it will look as follows:

```
class AuctionsRepository(abc.ABC):
    @abc.abstractmethod
    def get(
        self, auction_id: AuctionId
    ) -> Auction:
        pass

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

`get` method exists for retrieving `Auction` *Entity* using its `AuctionId`, while `save` method is responsible for persisting `Auction`.

DATA ACCESS (REPOSITORY)

In this example, an implementation of *Data Access Interface* will finally be a concrete class that relies on a relational database, e.g. PostgreSQL.

[27] Vaughn Vernon, *Implementing Domain-Driven Design*, Chapter 12. Repositories, Persistence-Oriented Repositories

[28] Vaughn Vernon, *Implementing Domain-Driven Design*, Chapter 12. Repositories, Collection-Oriented Repositories

IN-MEMORY IMPLEMENTATION

Although keeping data in persistent storage is an absolute must in production, it is beneficial for many reasons to start with an in-memory *Data Access*. Firstly, it is much easier to write so that one can deliver such an implementation in no time. Secondly, it can be used in tests of *Use Cases* instead of lousy mocks. Lastly, one can iterate faster with the entire flow - it should be acceptable to present a working feature with an in-memory datastore to get some feedback from stakeholders.

EVOLVING IN-MEMORY IMPLEMENTATION WITH TDD

There are only two methods in `AuctionsRepository`. A developer could cover them with a single test provided they can assume that all `Auctions` will be created (saved for the first time, to be precise) using it. Testing two methods at once may sound like an anti-pattern, but for now it is exactly what we need to fully check the behavior of a class under test. This is TDD, so we devote only minimal effort to push things forward:

```python
class PlacingBidTests(unittest.TestCase):
    def test_should_get_back_saved_auction(
        self,
    ) -> None:
        bids = [
            Bid(
                id=1,
                bidder_id=1,
                amount=Money(USD, "15.99"),
            )
        ]
        auction = Auction(
            id=1,
            title="Awesome book",
            starting_price=Money(USD, "9.99"),
            bids=bids,
        )
        repo = InMemoryAuctionsRepository()

        repo.save(auction)

        assert repo.get(auction.id) == auction
```

For this to work, `Auction` has to support a comparison operator (==). In languages that do not support operators overloading we would use whatever convention is present there (e.g. *equals*() method in Java). In Python, an implementation that makes it possible to compare *Entities* can look like this:

```
class Auction:
    def __eq__(self, other: "Auction") → bool:
        # we check type and fields being identical
        return isinstance(
            other, Auction
        ) and vars(self) == vars(other)
```

Repository implementation that passes the test looks as follows:

```
class InMemoryAuctionsRepository(
    AuctionsRepository
):
    def __init__(self) → None:
        self._storage: Dict[
            AuctionId, Auction
        ] = {}

    def get(
        self, auction_id: AuctionId
    ) → Auction:
        return copy.deepcopy(
            self._storage[auction_id]
        )

    def save(self, auction: Auction) → None:
        self._storage[auction.id] = copy.deepcopy(
            auction
        )
```

You may wonder why the class keeps and returns copies of objects? In Python, objects are passed using references. If one uses such reference-based repository to get `Auction`, they would see dirty changes even if they were not explicitly saved with the repository. That is not acceptable.

With an in-memory implementation, we are now good to go back to `PlacingBid` *Use Case*.

FINISHING OUR FIRST USE CASE

DEPENDENCY INJECTION

Eventually, `PlacingBid` needs a concrete implementation of `AuctionsRepository` to do the job. Although it is just fine to use `InMemoryAuctionsRepository` in tests, in-memory implementation is not something we can run in production. It is clear *Use Case* should not instantiate `AuctionsRepository` on its own to avoid coupling. Luckily, there is dependency injection we can leverage.

To start with, one needs to enable *Use Case* for dependency injection, namely accepting `AuctionsRepository` via its `__init__` / constructor. An implementation of `PlacingBidOutputBoundary` is already passed that way, so adding `AuctionsRepository` feels like a natural next step.

```
class PlacingBid:
    def __init__(
        self,
        output_boundary: PlacingBidOutputBoundary,
        auctions_repo: AuctionsRepository,
    ) -> None:
        self._output_boundary = output_boundary
        self._auctions_repo = auctions_repo
```

MAKING FIRST REASONABLE TEST PASS

As a consequence, we need to alter our test a bit to accommodate this change. We have to create an instance of `InMemoryAuctionsRepository`, save an auction and pass it to the *Use Case*:

```
class PlacingBidTests(unittest.TestCase):
    FRESH_AUCTION_ID = 2

    def setUp(self) -> None:
        self.output_boundary_mock = Mock(
            spec_set=PlacingBidOutputBoundary
        )
        repo = self._create_repo_with_auction()
        self.use_case = PlacingBid(
            self.output_boundary_mock, repo
        )

    def _create_repo_with_auction(
        self,
    ) -> AuctionsRepository:
        repo = InMemoryAuctionsRepository()
        fresh_auction = Auction(
            self.FRESH_AUCTION_ID,
            "socks",
            Money(USD, "1.99"),
            [],
        )
        repo.save(fresh_auction)
        return repo
```

Finally, there goes an implementation of a *Use Case* that passes the test:

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

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

Finally, the original test for the *Use Case* passes. Now, one could write another failing test and evolve the implementation of `PlacingBid` *Use Case* with underlying *Entities*.

REMOVE BOILERPLATE CODE WITH REFACTORING

For a TDD cycle to be complete, we should polish code a bit with refactoring. Currently, our code is pretty simple and there are not many opportunities for improving it. However, we can get rid of initializer methods __init__ if we leverage excellent `attrs`[29] library which is a 3rd party replacement with more features for built-in `dataclasses`.

```
import attr


@attr.s(auto_attribs=True)
class PlacingBid:
    _output_boundary: PlacingBidOutputBoundary
    _auctions_repo: AuctionsRepository

    def execute(
        self, input_dto: PlacingBidInputDto
    ) -> None:
        ...
```

Same refactoring could be applied to `Auction` *Entity*.

[29] https://www.attrs.org/en/stable/

PACKAGING CODE

PACKAGING APPLICATION & DOMAIN CODE

Code shown so far have not been yet arranged into layers. Although there is no notion of web, real databases etc., it is already possible to treat Application with Domain as a separate code artefact. In Python, it means this code can become a standalone package - like one of those we get using `pip install`. This trick allows for enforcing The Dependency Rule, even in such a liberal language like Python. In different programming languages, such as Java, there are more appropriate building blocks for enforcing layers separation.

For starters, there will be four separate packages:

1. Application & Domain
2. Infrastructure
3. Main
4. Web

Application & Domain will be stripped of external dependencies whenever possible, while Infrastructure will be dependant on the former. Main package is introduced to decouple assembling objects from any delivery mechanism. Obviously, it must be aware of two former packages. Web will know Main, since it will not be assembling stuff on their own and needs IoC container to get its hands on *Use Cases*. Suggested directories and files structure looks as follows:

```
/
└── auctions
    ├── auctions  # 1
    │   ├── application  # 2
    │   │   ├── __init__.py
    │   │   ├── repositories
    │   │   │   ├── auctions.py
    │   │   │   └── __init__.py
    │   │   └── use_cases
    │   │       ├── __init__.py
    │   │       └── placing_bid.py
    │   ├── domain   # 3
    │   │   ├── entities
    │   │   │   ├── auction.py
    │   │   │   ├── bid.py
    │   │   │   └── __init__.py
    │   │   ├── exceptions.py
    │   │   ├── __init__.py
    │   │   └── value_objects.py
    │   └── __init__.py
    ├── tests  # 4
    │   ├── application  # 5
    │   │   ├── __init__.py
    │   │   └── test_placing_bid.py
    │   ├── factories.py
    │   └── __init__.py
    ├── requirements-dev.txt  # 6
    ├── requirements.txt  # 7
    └── setup.py  # 8
```

A packaged code for Application & Domain is kept in a separate directory named as the auctions. Inside, there is another directory with the same name (1) and `__init__.py` inside. In Python, this is required to to be able to import code. Application (2) and Domain (3) are inside auctions (1) contain two subdirectories with actual production code.

The test suite (4) is kept inside the package. Note it does not mirror the structure of the application and domain directories. There is just a (5) application package to show where to look for tests of *Use Cases* that are package's API after all. One should avoid mirroring the structure in tests to avoid unnecessary coupling. Such an approach also encourages practices like unit-testing every class/function separately, which may make it much harder to refactor in the future.

Remaining files - 6, 7 and 8 - are another Python-package specific files. Respectively, they keep information about package dependencies and package metadata, like version or name.

Note for Pythonistas: it may look a bit odd that we create separate files for every entity, repository etc. and do not keep them together in one file - `entities.py` or `repositories.py`. With so little code that was shown so far it may look unnecessary, but in the long run, keeping things separated is a much cleaner approach. To shorten import paths in other modules, one may import all *Entities* from submodules to `domain/entities/__init__.py`

This is a standalone, database- and framework-agnostic package that has its own test suite. It will be mentioned in other packages' requirements files - in other words, other packages will depend on it.

PACKAGING INFRASTRUCTURE CODE

Eventually, **`InMemoryAuctionsRepository`** is going to be used just in tests, so we would leave it with Application & Domain. Currently, it is our only implementation, so for the sake of the example, let's pretend it is a production-grade solution and put it in a separate package:

```
/root
├── auctions
│   ├── …  # see previous directory structure
└── auctions_infrastructure
    ├── auctions_infrastructure
    │   ├── __init__.py
    │   ├── repositories
    │   │   ├── __init__.py
    │   │   └── in_memory_auctions_repository.py
    │   └── settings.py
    ├── tests
    │   └── repositories
    │       └── in_memory_auctions_repository.py
    ├── requirements-dev.txt
    ├── requirements.txt
    └── setup.py
```

There it is, a cohesive, self-contained package that depends on the previously presented `auctions` package.

MAIN - PUTTING EVERYTHING TOGETHER

There is a concrete, although keeping data only in memory, implementation in one package (`auctions_infrastructure`). In another package (`auctions`) lies its abstraction. It is high time we wire those together - whenever abstraction is requested; a concrete class should be returned. In other words, we are about to configure IoC container.

It is going to be placed in yet another packaged called just `main`. Every delivery mechanism (web, CLI, background task queue, etc.) will have to use it in order to assemble the project.

```
/
├── auctions
│   ├── ...
├── auctions_infrastructure
│   ├── ...
└── main
    ├── main
    │   └── __init__.py
    ├── requirements-dev.txt
    ├── requirements.txt
    └── setup.py
```

Structure of main is straightforward and initially, will consist of just one file. With the project growth, more files would appear, for example, to handle various aspects of configuration. For starters, main will be responsible just for building IoC container.

It is going to be placed in yet another packaged called just *main*. Every delivery mechanism (web, CLI, background task queue, etc.) will have to use it in order to assemble the project.

There are many different implementations of IoC containers out there. Some are configured with XML, others with YAML. Yet another simply use code. For the sake of the example, assume injector has been chosen. It is designed after Java's Guice, so it should look familiar even for no-Pythonistas. Before we inject everything everywhere, stop for a moment and let me remind, that the whole goal of packaging is to organize code, by hiding information that is irrelevant outside the package and expose only classes that are absolutely necessary. For this to work with Injector, we make each of our packages to define a class inheriting from `main.Module`:

```
from auctions import AuctionsRepository  # 3


class AuctionsInfrastructure(injector.Module):
    @injector.provider
    def auctions_repo(
        self,
    ) -> AuctionsRepository:  # 2
        example_auction = Auction(
            id=1,
            title="Exemplary auction",
            starting_price=Money(USD, "12.99"),
            bids=[],
        )
        repo = InMemoryAuctionsRepository()  # 1
        repo.save(example_auction)
        return repo
```

Infrastructure will provide a concrete implementation (1) of the abstract repository (2), so it has to import the abstract class from auctions package (3). In the snippet above, we build an instance of `InMemoryAuctionsRepository` with a predefined, example `Auction` *Entity*. That helps to verify if everything works fine after attaching web interface. In production, one would rather not be adding fake data like this, but imagine how powerful it might be to have dedicated *main* files for different environments. For example, automated tests with in-memory repositories or another *main* customized for frontend developers that mocks out problematic dependencies. Looking from the perspective of our first package with code of Application & Domain, it has to expose `AuctionsRepository`.

```
class Auctions(injector.Module):
    @injector.provider
    def placing_bid_uc(
        self,
        boundary: PlacingBidOutputBoundary,
        repo: AuctionsRepository,
    ) -> PlacingBidInputBoundary:
        return PlacingBid(boundary, repo)
```

The module defined in `auctions` package is providing *Use Cases* using their abstractions. If we use this extra layer of indirection, auctions should also expose all *Input Boundaries*. If not, we expose all *Use Cases* directly:

```python
class Auctions(injector.Module):
    @injector.provider
    def placing_bid_uc(
        ... ,
    ) -> PlacingBid:
        return PlacingBid(boundary, repo)
```

A word “expose” has been used several times, but what does it actually mean? It was mentioned that a package should hide all information that is not relevant to the outside world. For example, there is no reason to expose our *Use Cases* if we use *Input Boundaries*. We only expose the latter. To visualize this, let’s use Java. By default, classes within the package are not visible outside. To *publish* them, we use public access modifier. It should not be added recklessly to every class, though. We only expose as little as possible. It is a little harder to achieve a similar effect in Python, but one can hint clients of a package what can and what should not be imported from it. Each of Python packages we shown before has top-level `__init__.py`. Inside, we import everything we want to expose and append names to a special list inside - `__all__`:

```python
__all__ = [
    # module
    "Auctions",
    # repositories
    "AuctionsRepository",
    # types
    "AuctionId",
    # use cases
    "PlacingBid",  # no input boundaries
    "PlacingBidInputDto",
    "PlacingBidOutputBoundary",
    "PlacingBidOutputDto",
]
```

If one tries to import from auctions something that is not inside `__all__`, they’ll be warned by linter and IDE.

Now that Injector modules are defined, they can be assembled inside `main`:

```python
def setup_dependency_injection() -> injector.Injector:
    return injector.Injector(
        [Auctions(), AuctionsInfrastructure()],
        auto_bind=False,
    )
```

Returned `Injector` instance can be used to build any configured object. For example, if one requests `PlacingBid`, IoC container would use `Auctions.placing_bid_uc` method to see it also needs `AuctionsRepository`, so it would call `AuctionsInfrastructure.auctions_repo` to build it first. IoC container makes it effortless to create objects graphs.

ATTACHING WEB INTERFACE

At this point, we are ready to expose our functionality to the outer world via HTTP API. My weapon of choice is Flask microframework because it is lightweight and integrates smoothly with the injector. To start with, we need a separate package for the web:

```
/
├── auctions
│   ├── ...
├── auctions_infrastructure
│   ├── ...
├── main
│   ├── ...
└── web_app
    ├── requirements-dev.txt
    ├── requirements-dev.txt
    ├── setup.py
    └── ...
```

The basic structure of `web_app` package is not standing out, although most of it has been omitted. A tree of directories would be typical for the chosen web framework. For Flask, we would see blueprints, app factory file etc. The goal of this book is not to teach you, dear reader, how to use web frameworks, so please kindly turn a blind eye on this. This example is to demonstrate how to call *Use Case* from any delivery mechanism, not only web.

`web_app` also defines its injector module in order to provide all *Output Boundaries* specific for the web. First things first, implementation of `PlacingBidOutputBoundary` - a *Presenter*:

```python
class PlacingBidWebPresenter(
    PlacingBidOutputBoundary
):
    response: Response

    def present(
        self, output_dto: PlacingBidOutputDto
    ) -> None:
        message = (
            f"Hooray! You are a winner"
            if output_dto.is_winning
            else f"Your bid is too low. Current price is {output_dto.current_price}"
        )
        self.response = make_response(
            jsonify({"message": message})
        )
```

Injector module:

```python
class AuctionsWeb(injector.Module):
    @injector.provider
    @flask_injector.request  # request scope
    def placing_bid_output_boundary(
        self,
    ) -> PlacingBidOutputBoundary:
        return PlacingBidWebPresenter()
```

Note there is a new decorator applied - `@flask_injector.request`. It instructs injector to use the same instance during one request. You will see the example below. It will inject the same instance to *Use Case* as well as *Controller*, so we can share data without breaking encapsulation of a *Use Case*.

Here comes Flask app factory, leveraging `main`:

```python
def create_app() -> Flask:
    app = Flask(__name__)

    FlaskInjector(
        app,
        modules=[AuctionsWeb()],
        injector=main.setup_dependency_injection(),
    )

    return app
```

and a Flask-powered view:

```
@auctions_blueprint.route(
    "/<int:auction_id>/bids", methods=["POST"]
)
def place_bid(
    auction_id: AuctionId,
    placing_bid_uc: PlacingBid,
    presenter: PlacingBidOutputBoundary,
) → Response:
    if not current_user.is_authenticated:
        abort(403)

    placing_bid_uc.execute(
        get_input_dto(  # 1
            PlacingBidSchema,
            context={
                "auction_id": auction_id,  # 2
                "bidder_id": current_user.id,  # 3
            },
        )
    )
    return presenter.response  # 4
```

Interesting lines of code:

1. there is a helper function `get_input_dto` in use that returns **`PlacingBidInputDto`** after parsing request data
2. id of the auction is taken from URL
3. bidder_id is an id of the authenticated user
4. we have to return a response from a view due to certain design assumptions

Note the very same instance of *Presenter* has been injected to *Controller* and to *Use Case*. That happened because we defined a request scope for the **`AuctionsWeb`**`.placing_bid_output_boundary` provider method. Otherwise, injector would create a new instance upon every injection, and that would not work as expected.

For more details concerning web layer, especially those regarding JSON deserialization and nuances of constructing **`PlacingBidInputDto`** please refer to code on GitHub[30]. Most of it is heavily Python-specific anyway, so there would be little value added in explaining it all here.

30 https://github.com/Enforcer/clean-architecture

On the other hand, there is no magic there - it is just a glue code that leverages popular serialization/deserialization Pythonic library - `marshmallow`.

IMPLEMENTING ENDINGAUCTION USE CASE

Auctions are typical *Entities* - long-living creatures that have an identity. A lifetime of an auction starts when it is created. Various settings can be adjusted, for example, starting price. An auction may be published immediately or become a draft to be published later. When an auction becomes publicly available, bidders are encouraged to place their bids. Such a state only lasts to a certain moment in time when we consider the auction to be ended. Ending an auction occurs at a time specified when the auction was created.

What is so special about **`EndingAuction`** *Use Case* that I decided to devote next pages to it? There are several reasons:

1. ending an auction involves finalizing a transaction, including payment,
2. it will not be invoked manually from any User Interface, it should rather happen automatically once the auction ends or shortly after ending time has come,
3. since there is no UI involved, there also will not be anyone waiting for a result of **`EndingAuction`** *Use Case*. Hence, there is no need for an *Output DTO* or *Output Boundary*,
4. an auction can be ended exactly once.

Regarding point 1, in existing platforms I know, a winner pays in a separate step. Real-life equivalent to our **`EndingAuction`** is limited to changing status of the auction and notifying all bidders whether they won or lost. After that, winner has to go back to platform, select how they want the item delivered etc. However, to boost educational value of the example I combined payment and ending auction into one *Use Case*.

In a previous *Use Case* we implemented, the database was our only dependency. This time we will require a payment provider. That is a bit different kind of dependency since such integration will involve communication over the internet to a 3rd party service not hosted within our infrastructure (as it was the case with the database). This part of an end-to-end example is meant to teach you what is the role of *Interface/Port* and *Interface Adapter/Adapter* in the Clean Architecture.

USE CASE OUTLINE WITH *INPUT DTO*

One can quickly start implementing the *Use Case* with the almost empty body:

```python
@dataclass(frozen=True)
class EndingAuctionInputDto:
    auction_id: AuctionId


class EndingAuction:
    def __init__(
        self, auctions_repo: AuctionsRepository
    ) -> None:
        self._auctions_repo = auctions_repo

    def execute(
        self, input_dto: EndingAuctionInputDto
    ) -> None:
        auction = self._auctions_repo.get(
            input_dto.auction_id
        )
        auction.end()
        # ?? payment ??
        self._auctions_repo.save(auction)
```

Note that in this case there is absolutely no need for an *Output DTO* and an *Output Boundary*. An administrator will be able to see the results of the *Use Case* using different means.

EXTENDING AUCTION *ENTITY* TO FULFILL NEW REQUIREMENTS

From a paragraph introducing `EndingAuction` *Use Case* you learned that bids cannot be placed on an auction that has ended and an auction cannot be ended twice. Auction *Entity* should make sure of it. There are few steps needed: `Auction` should get a new field for ending date and creation time of new bids should be checked against it. Additionally, a boolean field will be helpful to make sure an auction was ended exactly once.

Changes can be introduced gradually with TDD. First, a failing test for the existing *Use Case*:

```python
class PlacingBidTests(unittest.TestCase):
    def test_bid_on_ended_auction_raises_exception(
        self,
    ) -> None:
        yesterday = datetime.now() - timedelta(
            days=1
        )
        self.create_auction(ends_at=yesterday)
        price = Money(USD, "10.00")
        input_dto = PlacingBidInputDto(
            bidder_id=1,
            auction_id=self.AUCTION_ID,
            amount=price,
        )

        with self.assertRaises(BidOnEndedAuction):
            self.use_case.execute(input_dto)
```

This test is doomed to fail for several reasons. Firstly, `Auction` does not accept `ends_at` argument. Secondly, `BidOnEndedAuction` does not exist yet. Before fixing that, you might have noticed a little helper method `create_auction` for creating `Auction`:

```python
class PlacingBidTests(unittest.TestCase):
    def create_auction(
        self,
        ends_at: Optional[datetime] = None,
        ended: Optional[bool] = None,
    ) -> None:
        if ends_at is None:
            ends_at = datetime.now() + timedelta(
                days=7
            )

        auction = Auction(
            self.AUCTION_ID,
            "socks",
            Money(USD, "1.99"),
            [],
            ends_at,
        )
        self.repo.save(auction)
```

Now, let’s create a domain exception with base class:

```
class DomainException(Exception):
    pass


class BidOnEndedAuction(DomainException):
    pass
```

I don’t use Error or Exception suffixes in class names because they are redundant. Name of the exception (especially one coming from the domain) itself should convey enough information to identify what was exactly the problem.

To finally make this test pass, we alter `Auction` code:

```
class Auction:
    def __init__(
        ...
        ends_at: datetime,
    ) -> None:
        ...
        self.ends_at = ends_at

    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> None:
        if datetime.now() > self.ends_at:
            raise BidOnEndedAuction
        ...
```

Please note I only show new code that was added to the *Entity*.

IF *ENTITES* SHOULD NOT HAVE ANY DEPENDENCIES, CAN THEY USE TIME FUNCTIONS?

Generally speaking, *Entities* should be free from dependencies, including system clock. Purists would surely scream in a fury before tearing this book apart. However, let’s be pragmatic. It’s not a big deal in Python where we can control clock easier than Gaunter O’Dim in Witcher 3. All we need is a freezegun[31] library.

If for whatever reason we don’t want/can’t use it, one can always pass current date and time to `place_bid` method, so `Auction` only has to compare timestamps. A current date time can be obtained in *Use Case* using *Port* / *Adapter* for system clock. Speaking of *Port* & *Adapter*…

[31] freezegun https://github.com/spulec/freezegun

INTRODUCING *PORT* FOR PAYMENTS

Port is for external services, the same thing as *Data Access Interface* is for *Entities* and their persistence. It abstracts away details, for example, communication protocol. Is the payment provider talking JSON and REST? Or maybe it understands only XML sent over SOAP? It does not matter from a perspective of `EndingAuction` *Use Case*. It only needs an interface to make payment.

First, a bunch of assumptions and important notes about popular payment providers - especially for those readers who do not work with such services on a daily basis. Let's assume the payment is made using a credit or debit card. A bidder has to enter their card details before they can bid. We do not want to store this data because a) it is risky b) you would become an attack target c) it involves serious legal obligations. What is important is that details of payment card are not even flowing through our backend. This data is sent from frontend to a selected payment provider. In return, we get a token that we store.

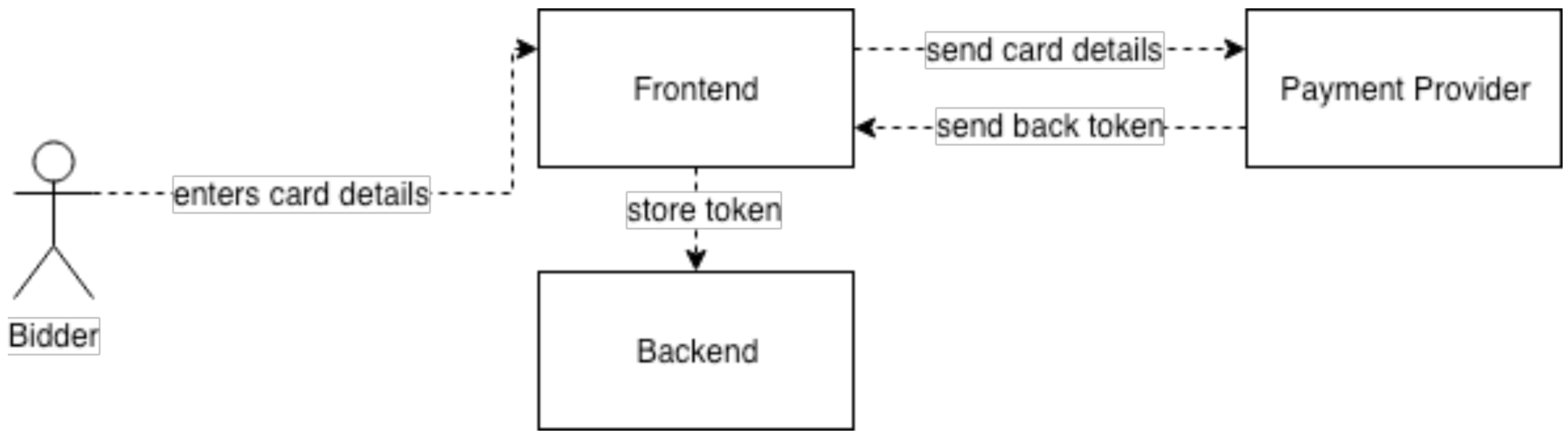


Figure 6.1 Data flow during saving payment card details

Whenever we want to charge the payment card, we send an authorized request to the payment provider with that token.

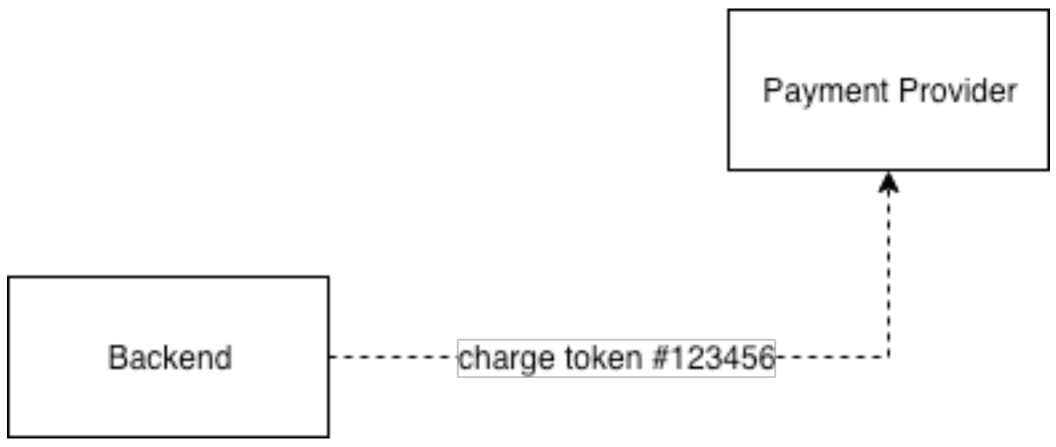


Figure 6.2 Backend uses a token whenever it needs to charge a bidder

The token is associated with a bidder that won the auction. Perhaps we could store the token alongside the bidder. In order to do so, we have to write **`Bidder`** *Entity*, then **`BiddersRepository`**. Then use both inside **`EndingAuction`** *Use Case* and pass the token to the *Port* abstracting away a payment provider. This is definitely an option, but the token is something very specific to a concrete payment provider. We cannot reuse the same token across different payment providers. If we ever switch payment providers (this happens, trust me) or add an alternative payment procedure, such coupling will hurt.

These deliberations are for one thing - to discover where a boundary between *Use Case* and the *Port* should be. What I propose is to pass a bidder's id and **`Money`** to be paid to *Port*'s method. Then *Adapter* (implementation of the *Port*) is responsible for finding the associated token. In the end, **`EndingAuction`** *Use Case* will be simpler, and all things specific to the given payment provider nicely separated. Moreover, **`Bidder`** will not be polluted with any stuff that is not specific to auctions.

Finally, there goes a **`PaymentProvider`** *Port*:

```python
class PaymentProvider(abc.ABC):
    @abc.abstractmethod
    def pay_for_won_auction(
        self,
        auction_id: AuctionId,
        bidder_id: BidderId,
        charge: Money,
    ) -> None:
        pass
```

Notice a comfort we have here, thanks to using **`Money`** *Value Object*. We can be sure that the `amount` argument is valid and after a little love (I meant, conversion) will be accepted by virtually any API.

ERROR HANDLING VS THE DEPENDENCY RULE

Failures will happen. It would be naive to believe that our *Adapters* would always succeed. An actual exception class that will be thrown depends on the *Adapter*. There are cases when we would like to catch an exception inside *Use Case* and do something with it. Naturally, we can not refer to adapter-specific exceptions classes inside a *Use Case* because it would violate The Dependency Rule!

What we can do instead, is to define a class (or a hierarchy) of exceptions alongside the *Port*:

```
class PaymentFailed(Exception):
    pass


class NotEnoughFunds(PaymentFailed):
    pass
```

Use Cases can use them since they will reside on the same layer (*Application*). Now *Adapter* can raise exceptions that either inherits from this class or raise it directly. I recommend having at least one generic exception class for every *Port*.

Beware of excessive exception handling inside *Use Case*! If you are not able to recover or take action justified by business requirements in a *Use Case* once exception was thrown, do not even bother with catching it. Let higher layers deal with it or just let the request fail.

IMPLEMENTING *ADAPTER*

The *Adapter* for `PaymentProvider` is going to be something very, very concrete. Let's assume that we have a token stored in the database somewhere and we are able to get it using `bidder_id` passed to `pay_for_won_auction` method. Actual charging involves an HTTP request to payment provider's API - */api/v1/charge* endpoint with basic auth and *application/json Content-Type* and *Accept* headers. Data to be passed in the request's body is supposed to be a JSON.

Example:

```
{
     "card_token": "123456",
     "currency": "USD",
     "amount": "14.99"
}
```

If everything goes fine, we expect to get a 200 OK along with following JSON:

```
{
     "charge_uuid": "67f0e9db-015d-407f-a58f-9c76551dc771",
     "success": true
}
```

Without further ado, let’s see a code that does what we need:

```python
class CaPaymentsPaymentProvider(PaymentProvider):

    BASE_URL = "http://ca-payments.com/api/v1/"
    CHARGE_SUFFIX = "charge"

    def __init__(
        self, login: str, password: str
    ) -> None:  # 1
        self.auth = (login, password)

    def pay_for_won_auction(
        self,
        auction_id: AuctionId,
        bidder_id: BidderId,
        charge: Money,
    ) -> None:
        card_details_model = BidderCardDetails.objects.filter(  # 2
            bidder_id=bidder_id
        ).first()

        response = requests.post(  # 3
            self.BASE_URL + self.CHARGE_SUFFIX,
            auth=self.auth,
            json={
                "card_token": card_details_model.card_token,
                "currency": charge.currency.iso_code,
                "amount": str(charge.amount),
            },
        )

        if not response.ok:
            raise PaymentFailedError  # 4
        else:
            # record payment charge_uuid in DB, etc
            ...
```

Interesting lines:

1. `__init__` is where the authentication object is prepared for a format convenient for Python’s *requests* library. Both login and password are some kind of secrets that should not be hardcoded and will vary depending on the environment. `main.py` should take care of passing proper values there

2. To retrieve `card_token` associated with a given `bidder_id`, one needs to reach the database. You may wonder why there is no entity nor repository for

`BidderCardDetails`? Because there is no need to. The *Adapter* is meant to be concrete. We cannot test it anyway without mocking HTTP calls. Such tests would have hardly any value. Storage options, as well as structure of HTTP requests and responses, are tightly coupled to this class anyway.

3. An HTTP call is made using the excellent *requests* library.

In case anything goes wrong, we raise an exception defined alongside the `PaymentProvider` *Port* (provided there it makes sense to handle it instead of just letting the request fail).

HOW TO LIVE WHEN *ADAPTER* GROWS?

You probably noticed that `pay_for_won_auction` is not the best function in the world. It is quite long and does several things. Depending on what is actually abstracted away by a *Port*, a corresponding *Adapter* may grow over time to eventually become one of these monster classes that have over one thousand lines of code, and no one wants to touch them. In fact, there is no rule telling to keep everything inside *Adapter* class! The latter is meant to be thin. Initially, when an *Adapter* implements one or two methods, a developer might be reluctant about refactoring it to not fall into the trap of premature optimization. Once an *Adapter* grows, it will be wise to resort to *Facade* design pattern.

See this refactoring in practice:

```python
class CaPaymentsPaymentProvider(PaymentProvider):
    ...
    def pay_for_won_auction(
        self,
        auction_id: AuctionId,
        bidder_id: BidderId,
        charge: Money,
    ) → None:
        request = ChargeRequest(  # 1
            card_token=dao.get_bidders_card_token(
                self._session,
                bidder_id
            ),  # 2
            currency=charge.currency.iso_code,
            amount=str(charge.amount),
        )
        response = self._execute_request(
            request, ChargeResponse
        )  # 3

        dao.record_successful_payment(
            self._session,
            auction_id,
            bidder_id,
            charge,
            charge_uuid=response.charge_uuid,
        )

    def _execute_request(
        self,
        request: Request,
        response_cls: Type[ResponseCls],
    ) → ResponseCls:
        response = requests.post(
            request.url,
            auth=self.auth,
            json=asdict(request),
        )
        if not response.ok:
            raise PaymentFailedError
        else:
            return response_cls(**response.json())
```

We start from the final look of the *Adapter*. Now it looks much more generic.

Interesting lines:

1. We introduced data classes for HTTP requests and responses. They enforce the presence of parameters. We will see the implementation in just a moment
   Second refactoring amounted to pulling all DB-interacting code to a separate module - `dao`.
   The thing that was left here is the logic responsible for making HTTP calls. It is generic, easily extendable with custom Requests and Responses dataclasses. We could have also pulled it to separate function/module/class.

Now let's see how clear these requests and responses look like:

```python
@dataclass(frozen=True)
class Request:
    url = "http://ca-payments.com/api/v1/"
    method = "GET"


@dataclass(frozen=True)
class ChargeRequest(Request):
    card_token: str
    currency: str
    amount: str
    url = Request.url + "charge"
    method = "POST"
```

Such a structure allows for a declarative extending list of handled endpoints with a little effort. Responses look very similar, except there is no base class.

Finally, `dao` is not very interesting, but I show it for the sake of completeness:

```python
def get_bidders_card_token(
    session: Session, bidder_id: BidderId,
) → str:
    card_details_model = (
        session.query(BidderCardDetails)
        .filter_by(bidder_id=bidder_id)
        .one()
    )
    return card_details_model.card_token
```

```python
def record_successful_payment(
    session: Session,
    auction_id: AuctionId,
    bidder_id: BidderId,
    charge: Money,
    charge_uuid: str,
) -> None:
    entry = PaymentHistoryEntry(
        auction_id=auction_id,
        bidder_id=bidder_id,
        amount=charge.amount,
        currency=charge.currency.iso_code,
        charge_uuid=charge_uuid,
    )
    session.add(entry)
```

After such a refactoring session, *Adapter* becomes merely a thin *Facade* over quite a complex subsystem.

READ ONLY OPERATIONS

USE CASE - BASED APPROACH

It is high time we covered a touchy subject - implementing read-only operations with the Clean Architecture. It is something we could definitely implement right now using building blocks that we have seen so far - *Use Case, Input-, Output DTO* and *Repository*. Let's assume we are to allow potential bidders look at auction details - its title, starting and current prices. To make the whole example more interesting, we can add a list of 3 top bids with anonymized bidders username. Anonymization, in this case, means that we will display the first letter of each username followed by an ellipsis.

We shall not be prejudiced. Let's see how the first idea works out. `GettingAuctionDetails` *Use Case* should, for sure, accept an `auction_id` in its *Input DTO*. As for *Output DTO*, we already know what is expected.

```python
@dataclass(frozen=True)
class GettingAuctionDetailsInputDto:
    auction_id: AuctionId


@dataclass(frozen=True)
class TopBidder:
    anonymized_name: str
    bid_amount: Money
```

```python
@dataclass(frozen=True)
class GettingAuctionDetailsOutputDto:
    auction_id: AuctionId
    title: str
    current_price: Money
    starting_price: Money
    top_bidders: List[TopBidder]
```

Job of `GettingAuctionDetails` is to pull required data from repositories and repack them to *Output DTO*:

```python
class GettingAuctionDetails:
    def __init__(
        self,
        output_boundary: PlacingBidOutputBoundary,
        auctions_repo: AuctionsRepository,
        bidders_repo: BiddersRepository,
    ) -> None:
        self._output_boundary = output_boundary
        self._auctions_repo = auctions_repo
        self._bidders_repo = bidders_repo
    def execute(
        self,
        input_dto: GettingAuctionDetailsInputDto,
    ) -> None:
        auction = self._auctions_repo.get(
            input_dto.auction_id
        )
        top_bids = auction.bids[-3:]

        top_bidders = []
        for bid in top_bids:
            bidder = self._bidders_repo.get(
                bid.bidder_id
            )
            anonymized_name = (
                f"{bidder.username[0]}..."
            )
            top_bidders.append(
                TopBidder(
                    anonymized_name, bid.amount
                )
            )

        output_dto = GettingAuctionDetailsOutputDto(
            auction_id=auction.id,
            title=auction.title,
            current_price=auction.current_price,
            starting_price=auction.starting_price,
            top_bidders=top_bidders,
        )
        self._output_boundary.present(output_dto)
```

I believe you did NOT like this solution. I also hope you do not want to give up on the Clean Architecture yet. My point here is that although *Use Case* approach works great for scenarios that involve mutating data, it becomes a burden when all that one wants to is to retrieve some information in a possibly efficient way. The code above is not only verbose but also inefficient. It suffers from a classical flaw of ORM n + 1 queries. Luckily, we still have some rabbits left in our hat.

CQRS TO THE RESCUE

CQRS pattern introduced in one of the previous chapters demonstrated how beneficial separating code responsible for writes from reads might be. It is the right moment we leveraged that knowledge and talked a bit about how to implement read side in the project.

Quick reminder: three variants of how one can implement read side of CQRS were described in this book. For this example, the last one (Read Model Facade) will be used. To make things simpler, assume there is a single database for both writes and reads.

LIBERAL APPROACH WITH *READ MODEL FACADE*

Read Model Facade interface will be a bunch of methods for each model/entity. Every one of them will return prepared query object that client can customize by adding filters or (depending on implementation) joining more models. Actual implementation will naturally be something very specific to the underlying database. Here is how it could for Django and the aforementioned example with a list of 3 top bidders:

```python
class AuctionsReadFacade:
    def auctions(self) -> models.Manager:
        return Auction.objects

    def bids(self) -> models.Manager:
        return Bid.objects
```

Usage example:

```python
def details(
    request: HttpRequest, auction_id: int
) -> HttpResponse:
    try:
        auction = (
            AuctionsReadFacade()
            .auctions()
            .get(pk=auction_id)
        )  # 1
    except ObjectDoesNotExist:
        raise Http404(
            f"Auction #{auction_id} does not exist!"
        )
    bids = (
        AuctionsReadFacade()
        .bids()
        .filter(auction_id=auction_id)  # 2
        .select_related("bidder")
        .order_by("-amount")[:3]
    )
    ctx = Context(
        {"auction": auction, "bids": bids}  # 3
    )
    tpl = Template(  # 4
        """{% load app_filters %}"""
        """Auction: {{ auction.title }}<br>"""
        """Price changed from {{ auction.starting_price|dollars }}"""
        """to {{ auction.current_price|dollars }}<br>"""
        """Top bids:<br>"""
        """{% for bid in bids %}"""
        """{{ bid.amount|dollars }} by {{ bid.bidder.username|anonymize }}<br>"""
        """{% endfor %}"""
    )
    return HttpResponse(tpl.render(ctx))
```

`AuctionsReadFacade` is very thin in this case. It may make little sense with Django (just like the Clean Architecture doesn't play well with this framework), but we are only to grasp the idea. Interesting lines:

1. We fetch auction first to be able to quickly discover if it is not there and react with HTTP 404

2. Notice how heavily customized this Queryset is. We filter it, join Bidder model, order and limit the result set

3. Unaltered data obtained from *Read Model Facade* is passed for further processing

4. Template in Django is responsible for displaying and formatting output. It is shown here to emphasize that *Read Model Facade* is not responsible for converting data to the desired format.

A LITTLE BIT MORE STRUCTURED APPROACH WITH QUERY CLASSES

An alternative approach that would relieve *view* (or *controller* in other languages) from having to customize raw query objects of the underlying persistence mechanism is to use Query classes. The latter is to contain details and hide it behind some nice, descriptive name.

In the above example, it could look like this:

```python
class GetAuctionDetails:
    @dataclass(frozen=True)
    class Dto:  # 1
        auction: Auction
        bids: List[Bid]

    def query(
        self, auction_id: AuctionId
    ) -> "Dto":
        auction = Auction.objects.get(
            pk=auction_id
        )  # 2
        bids = (
            Bid.objects.filter(
                auction_id=auction_id
            )
            .select_related("bidder")
            .order_by("-amount")[:3]
        )
        return self.Dto(auction, bids)
```

Interesting lines:

1. Data is returned in the form of a *DTO* to enforces structure. An alternative is to return a plain `dict` that will be happily accepted by Django's Context. It might be beneficial for performance reasons - it means less data repacking.

2. Underlying persistence library is used directly by the class.

Usage compared to *Read Model Facade* is significantly simpler:

```
def details(
    request: HttpRequest, auction_id: int
) → HttpResponse:
    try:
        dto = GetAuctionDetails().query(
            auction_id
        )
    except ObjectDoesNotExist:
        raise Http404(
            f"Auction #{auction_id} does not exist!"
        )

    ctx = Context(asdict(dto))
    tpl = Template( ... )
    return HttpResponse(tpl.render(ctx))
```

Bear in mind that *Query* and *Read Model Facade* classes are part of our application's interface next to *Use Cases*. It means that if one finds it valuable to have an abstract class (or interface) for every *Use Case - Input Boundary*, it may also help to abstract away *Query*. Doing this with *Read Model Facade* would be actually much, much harder - one would have to abstract whole underlying ORM which makes absolutely no sense.

Example for `GetAuctionDetails`:

```
class GetAuctionDetails(abc.ABC):
    @dataclass(frozen=True)
    class Dto:  # 1
        auction: Auction
        bids: List[Bid]

    @abstractmethod
    def query(
        self, auction_id: AuctionId
    ) → "Dto":  # 2
        pass
```

Interesting lines:

1. `Dto` is a return value, so it has to be a part of the abstracting class

2. `query` is left as abstract. Concrete implementation can now be wired using Dependency Injection. In this way, one decouples persistence from the delivery mechanism (if that's desirable, of course).

Read Model Facade and *Query* classes are two example approaches to implementing read side. Choose whichever suits your needs best. If unsure, start with Query classes without abstract superclasses.

INVERTING CONTROL WITH EVENTS

Use Cases shown so far controlled the flow only directly. Simply saying, a *Use Case* always behaved as if it was the most strict and observant conductor - nothing could happen without its knowledge.

EXAMPLE - SENDING E-MAILS

This approach is a desirable one if you need to coordinate a dance of *Repositories*, *Entities* and *Ports* in which everything has to happen in the specific order. However, in real-life *Use Cases* will have side effects that may be crucial from the perspective of stakeholders, yet they simply do not fit into our current set of building blocks. For example, sending e-mails to bidders that have just been overbid. To actually send an e-mail, there must be some network communication with an SMTP server. If you are itching to write a *Port/Adapter* pair, hold your horses for a moment. There are a few tricky questions to answer. Where does the content of the e-mail, its looks and feel, template etc. belong? To *Application*? No, too many details are concrete. If not, should *Port* be something more generic, like *CommunicationGateway* that could as well send push notifications or SMS messages to mobile devices? It becomes obvious that *Port/Adapter* approach would be clumsy. Another idea is needed.

INVERTING CONTROL TECHNIQUES

Sending e-mails has to be decoupled from *Use Cases*. Neither network communication nor contents of e-mails belong to the *Application* layer. Primarily used inverting control technique (*Port/Adapter + Dependency Injection*) does not fit here. If only there would be a way to let know all interested parties about some significant event that happened within auction… Well, there is. And it is called just *Event*. Before we delve into implementation, let's consider what *Event* actually is. To put it simply, *Events* represent facts - their sole occurrence means that *something happened*. They cannot be denied or rejected. *Events* are to decouple a sender from any other object that is interested in state changes of the sender. The latter is not even aware of whether anyone is listening for events. This is the main difference between *Port/Adapter* and *Event/Listener*. In the picture below, you can see, there is an additional party involved - *Event Bus*, serving as a *Mediator*. Its goal is to decouple *Event* sender from *Listener*.

EVENT IMPLEMENTATION

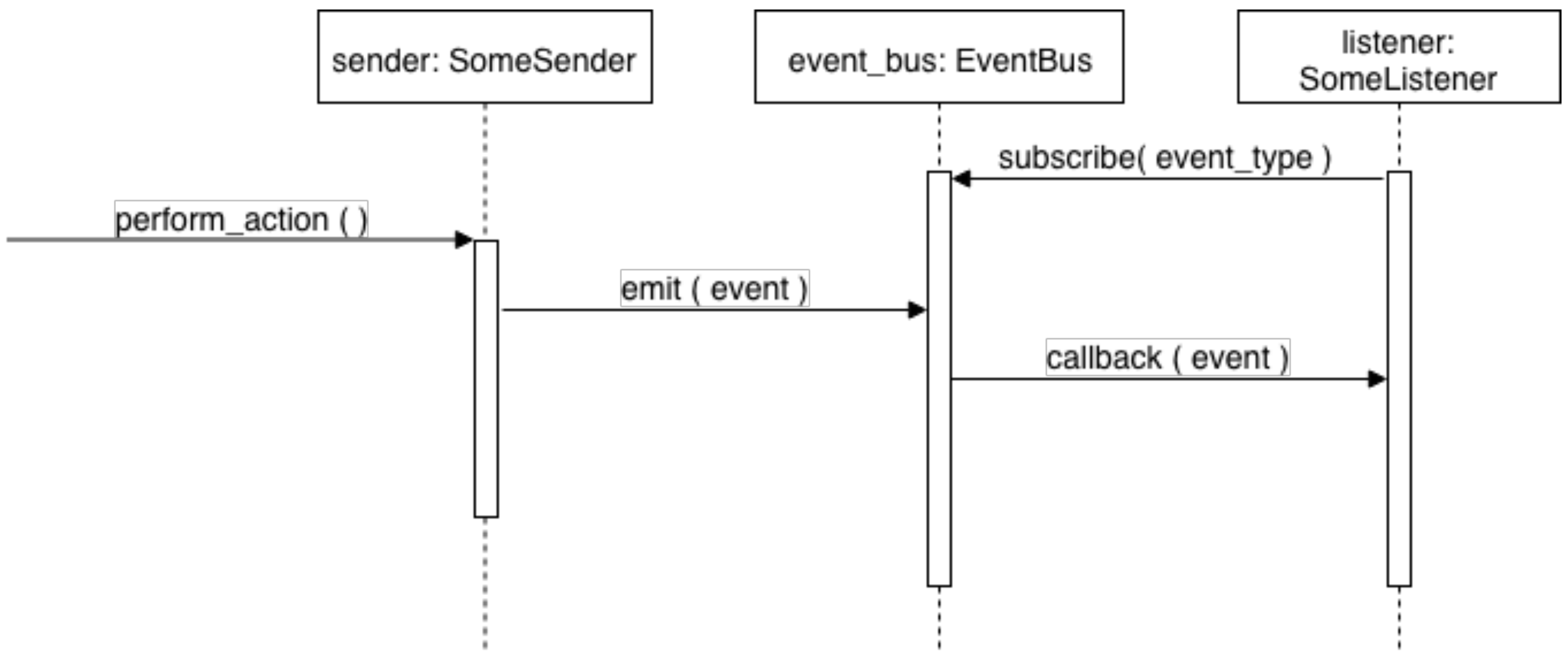


Figure 6.3 Emitting Events via EventBus decouples sender from a listener. The sender is not aware of how many listeners are subscribed or even if there is one

In the context of the example, with sending e-mails after a bidder has been overbid, we could express that situation with `BidderHasBeenOverbid` *Event*. Notice the past tense used - it emphasizes what *Event* is - a piece of information, that something happened. More examples related to auctions we could think of are `BidPlaced`, `AuctionStarted` or `AuctionEnded`. In terms of implementation, an *Event* is simply a *Data Transfer Object*:

```
@dataclass(frozen=True)
class BidderHasBeenOverbid:
    auction_id: AuctionId
    bidder_id: BidderId
    new_price: Money
```

We now know example *Events*; now the question is which building block is actually responsible for sending them? In the *Port/Adapter* approach, it was the *Use Case* which was responsible for coordination. *Use Cases* may simply not know enough when it comes to emitting *Events*, though. The knowledge needed for creating *Events* is available in *Entities*. *Use Cases* would need extra *Entity*'s method calls and potentially complex ways of rediscovering what just happened to build an *Event*. By doing so, we would risk lowering the quality of *Entities* encapsulation. Hence *Events* belong to the most inner circle in the Clean Architecture - they are part of the *Domain*. Hence, they can be (and often are) just called *Domain Events*.

Events as a decoupling technique have enormous potential. Since they are simple data structures, they can be serialized and then sent over the network to a different application. Let's not jump into this swamp at this moment. For now, assume that both sending and listening objects live within one operating system process. We will get back to more complex scenarios, I promise.

WHERE DO I GET ONE OF THESE *EVENT BUSES* FROM?

How about *Event Bus*? At a minimum, we expect it to give us an ability to subscribe for given *Event* type (single *Event* may have 0 or more subscribers) and to emit *Events*:

```
class EventBus:
    def emit(self, event: Event) → None:
        ...

    def subscribe(
        self,
        event_cls: Type[Event],
        listener: Callable,
    ) → None:
        ...
```

Normally, *Event Bus* should be a dependency of the project. 3rd party solutions are more than enough. If we wrote one ourselves, we would have to resort to a certain workaround. *EventBus* and base *Event* class would be placed in another package, called for example, *foundation*. Beware, the *foundation* is not a place for so-called utility classes (an incoherent bunch of functions where one is responsible for striping off non-ascii characters and another one for checking if a given date belongs to a given range)! This "trick" is to make *Event Bus* part of our standard library, so to speak.

```
/root
├── auctions
│   ├── auctions
│   │   ├── application
│   │   │   └── ...
│   │   └── domain
│   │       ├── events
│   │       │   ├── bidder_has_been_overbid.py
│   │       │   └── ...
│   │       └── ...
│   └── tests
│       └── ...
└── foundation
    ├── foundation
    │   ├── __init__.py
    │   ├── event.py
    │   └── event_bus.py
    ├── requirements.txt
    └── setup.py
```

It should not be placed within the directories structure of the Clean Architecture, because this will be, spoiler alert, reused among different project modules.

HOW TO GET EVENTS OUT OF ENTITIES?

Events itself are a part of the *Domain*, no doubt. *Event Bus* resembles a *Port*. Hence we would expect it to be a part of the *Application*. However, that would lead to a paradoxical situation. The almighty The Dependency Rule states clearly - *Entity* MUST NOT use *Event Bus* since it resides above the *Domain* layer. Despite the fact that *Entity* has to emit *Event* and the latter should be passed to `EventBus.emit`.

Entity keeps *Events* to be emitted by *Repository*

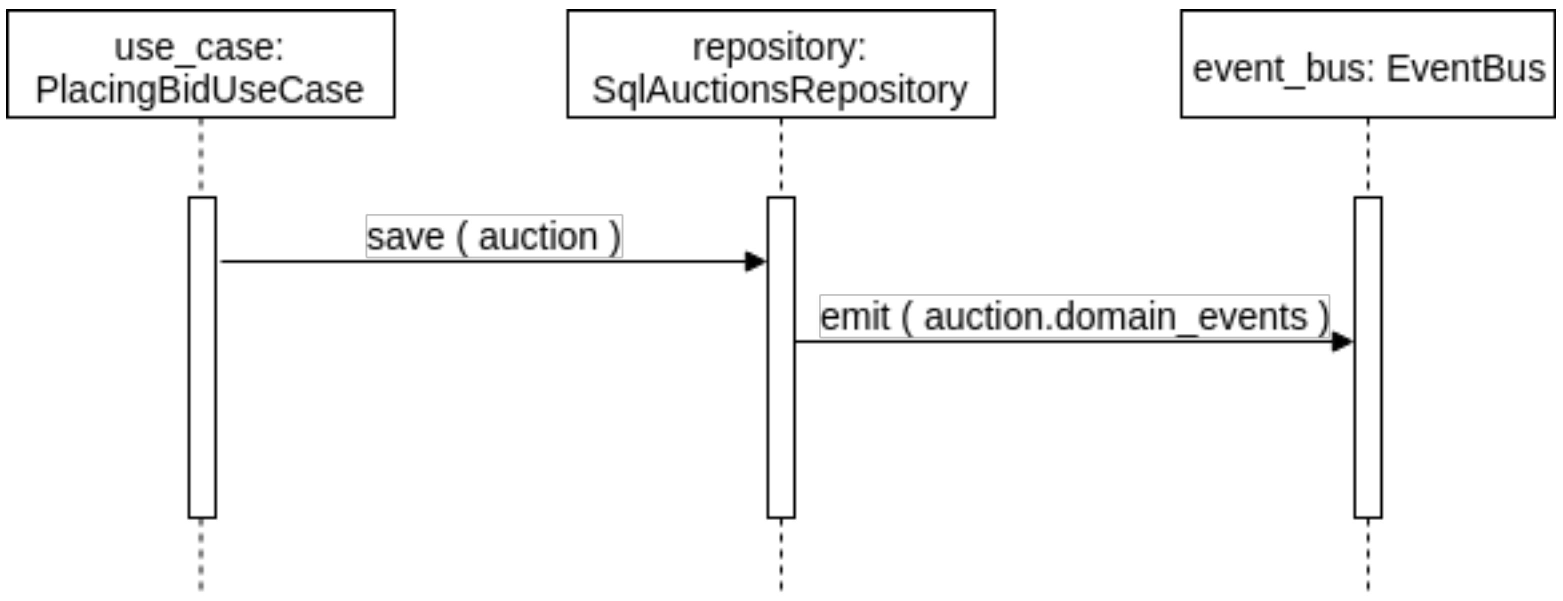

use_case: PlacingBidUseCase
repository: SqlAuctionsRepository
event_bus: EventBus
save (auction)
emit (auction.domain_events)

Figure 6.4 Repository emits events obtained from Entity via EventBus

In this approach *Entity* creates events instances, but since it is not permitted to use *Event Bus* (or anything else outside *Domain*) it hoards them in a private field. Later, when *the Repository* saves an *Entity*, it has to collect all pending events, and then pass them to *Event Bus*.

```python
class Auction:
    def __init__( ... ) → None:
        ...
        self._pending_domain_events: List[
            Event
        ] = []  # 1

    def _record_event(
        self, event: Event
    ) → None:  # 2
        self._pending_domain_events.append(event)

    @property
    def domain_events(self) → List[Event]:  # 3
        return self._pending_domain_events[:]

    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) → None:
        ...
        self._record_event(  # 4
            BidderHasBeenOverbid(
                self.id,
                old_winner,
                amount,
                self.title,
            )
        )
```

Interesting places:

1. During object creation, we initialize a private list of pending events
2. A private method, handy for appending events to the list
3. The repository will use this method to get pending events. It returns a copy of a list
4. Whenever a *Domain Event* occurs, we record it.

There goes an example of using *EventBus* in a concrete repository:

```python
class SqlAlchemyAuctionsRepo(AuctionsRepository):
    def __init__(
        self,
        connection: Connection,
        event_bus: EventBus,
    ) -> None:  # 1
        self._conn = connection
        self._event_bus = event_bus

    def save(self, auction: Auction) -> None:
        ...

        for event in auction.domain_events:  # 2
            self._event_bus.emit(event)
```

Interesting places:

1. *Event Bus* becomes a dependency of a *Repository*
2. At the end of saving, we emit all pending *Domain Events*

This design is pretty clean, though it requires to put emitting logic in a concrete repository. **It is expected the same logic will appear in all repositories and can be considered a major downside of this approach**. Naturally, we could refactor it and polish the code even more. However, the goal of these snippets is to present an idea - pending *Domain Events* are passed to *Event Bus* during saving of an *Entity*.

Entity returns *Events* from methods that change state

Let's remind what a CQS (Command-Query Separation) class design is. Simply saying, methods can be categorized into one of two categories - commands or queries. Commands change state of the object and return nothing while queries return anything, but are forbidden to mutate state of the instance. This is a design that was deliberately chosen for our `Auction` *Entity*. The second approach comes down to returning *Events* from methods that are commands. Although it is a trade-off, it shifts responsibility of collaborating with *Event Bus* from *Repository* to *Use Case*. I bet you admit it actually suits *Use Case* pretty well.

Implementation-wise, our *Entity* has to return *Events* at the end of command execution:

```
class Auction:
    ...

    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> List[Event]:
        events = []  # a list for events created
        if self._should_end:
            raise BidOnEndedAuction

        old_winner = self.winners[0] if self.bids else None
        if amount > self.current_price:
            self.bids.append(
                Bid(
                    id=None,
                    bidder_id=bidder_id,
                    amount=amount,
                )
            )
            #  ... a first event is appended
            events.append(
                WinningBidPlaced(
                    self.id, bidder_id, amount, self.title,
                )
            )
            if old_winner and old_winner != bidder_id:
                # second event appended
                events.append(
                    BidderHasBeenOverbid(
                        self.id,
                        old_winner,
                        amount,
                        self.title,
                    )
                )

        # finally, list of events is returned
        return events
```

Then, *Use Case* calling *Entity* has to collect *Events* and emits them using *Event Bus:*

```
class PlacingBid:
    ...

    def execute( ... ) -> None:
        ...
        events = auction.place_bid( ... )
        for event in events:
            self._event_bus.emit(event)
```

Since this is at least partially a Python book, we can also make the whole thing a bit easier by getting rid of `events` list in a method. How? First, we turn our command method into a generator:

```
class Auction:
    ...

    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> Generator[Event, None, None]:
        ...
        if amount > self.current_price:
            ...
            yield WinningBidPlaced(
                self.id, bidder_id, amount, self.title
            )
            if old_winner and old_winner != bidder_id:
                yield BidderHasBeenOverbid(
                    self.id, old_winner, amount, self.title
                )
```

We can not yet stop here - a `yield` keyword stops the execution of a method. With a combination of a snippet above from *Use Case* or *Repository*, it means that first *Event* can be dispatched to listeners even though we have not finished processing of command yet! What is more, if we find ourselves in an undesired situation and want to raise an exception between *Events*, we may end up with part of the events emitted already. This is probably not what we want to happen.

To compensate, we could always flatten generated *Events* using `list` or write a decorator that will be doing it for us:

```
def command_returning_events(
    method: Callable[..., Generator[Event, None, None]]
) -> Callable[..., List[Event]]:
    @functools.wraps(method)
    def wrapped(*args: Any, **kwargs: Any) -> List[Event]:
        return list(method(*args, **kwargs))

    return wrapped


class Auction:
    ...

    @command_returning_events
    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> Generator[Event, None, None]:
        ...
```

SUBSCRIBING TO EVENTS

Without subscribers, the *Event* would simply get lost in the void. *For now*, the best place for subscribers to sign up for future *Events* is already known to you - *main*. The next chapter shows other method, but for now let's reuse *main*:

```
def setup() -> None:  # 1
    event_bus = EventBus()
    setup_dependency_injection()
    setup_event_subscribing()


def setup_dependency_injection(
    event_bus: EventBus,
) -> None:
    def di_config(binder: inject.Binder) -> None:
        binder.bind(EventBus, event_bus)  # 2
        ...

    inject.configure(di_config)
```

```python
def setup_event_subscriptions(
    event_bus: EventBus,
) → None:
    event_bus.subscribe(  # 3
        BidderHasBeenOverbid,
        lambda event: send_email.delay(
            event.auction_id,
            event.bidder_id,
            event.money.amount,
        ),
    )
```

Interesting lines:

1. `setup` procedure has been extended with setting up event subscriptions
2. *EventBus* has to be configured for dependency injection container
3. In this example, we wire together `BidderHasBeenOverbid` with a task implemented using Celery[32] (distributed task queue) task that will send an informative e-mail

Events are enormously powerful decoupling technique. When direct control with *Ports/Adapters* does not *feel* right or leads to creating awkwardly looking *Ports'* interfaces, *Events* are your best bet.

TESTING *ENTITIES* THAT EMIT *EVENTS*

Regardless of the implementation we choose, the lovely thing about making *Entities* producing *Events* is that the former stay easily testable. By nature, *Events* are *Data Transfer Objects* and can be seen as *Value Objects* - indistinguishable provided their fields have equal values.

If we were to test *Entity* that keeps *Events* inside, then all that one needs to check is compare value for `domain_events` property with a list of expected *Events*:

[32] Celery: Distributed Task Queue http://www.celeryproject.org/

A bit rough example that gives an idea:

```python
def test_auction_upon_overbid_emits_bidder_has_been_overbid_event() -> None:
    auction = create_auction()
    winning_amount = (
        auction.current_price
        + get_dollars("1.00")
    )

    auction.place_bid(
        bidder_id=1, amount=winning_amount
    )

    assert auction.domain_events == [
        WinningBidPlaced(
            auction.id,
            1,
            winning_amount,
            auction.title,
        )
    ]
```

EVENTS VS TRANSACTIONS VS SIDE EFFECTS

As long as everything happens within one database transaction and communication with external services over the network is not involved, one can sleep well. Regrettably, this is almost never the case. Even aforementioned, simplified examples can fail in a number of interesting ways. Though “interesting” is not probably a word you would use being woken up in the middle of the night during your on-call shift. Reliability is not something one can take lightly.

Let’s consider a scenario when someone overbids another bidder on a given auction:

1. HTTP request is received
2. A database transaction is started
3. The Auction is pulled from a storage using *AuctionsRepository*
4. A new bid is placed
5. The auction emits an event
   Subscriber invokes background job to send an e-mail
6. The *Auction* is saved back to database

7. HTTP response is built
8. The transaction is committed
9. HTTP response is sent to the client.

What could possibly go wrong here…?

1. If we react to the event right after the *Entity/Repository* emits it, we will be sending e-mail before a transaction is committed, meaning that overbid may still fail. The Auction would look like as if an e-mail receiver has not been overbid at all
2. if the background job is triggered before commit of the original transaction AND it reaches to the database for data it may happen it will not find simply because the first transaction is still in progress and changes it made are invisible to other connections. A classic race condition.

Although these problems seem to be very hard to solve at first glance, their cause is trivial - misuse of *Events*! In the context of transactional RDBMS, *nothing* actually happened as long as the transaction is still in progress. It means that any side effects caused by events cannot be triggered until the DB transaction is committed. Before that, emitted *Events* are unfounded. The situation quickly gets tricky if there are more databases or message brokers involved. Welcome to the world of distributed systems.

Let's not fall into paranoia, though. Without considerable scale, problems like the first one are unlikely to happen. This is not the case with 2. situation though. Such a race condition can bite us unexpectedly, even if there is only one user.

Is there an easy way out for both problems? There is. Many database access libraries implement callback functionality to call user-defined logic after the transaction is committed. If we combine it with task queues (every event subscriber schedules a task to be executed after transaction commit), we will get the desired behavior. A more formal approach is to use *Unit of Work* pattern. However, the solution is not 100% reliable, although it is *good enough* in certain cases and especially at a lower scale.

INTRODUCING *UNIT OF WORK*

Unit of Work is an abstraction over a so-called *business transaction*. Please do not confuse it with a database transaction which has a narrower scope. There is a lot of similarities,

RELIABLE MESSAGE-SENDING: THE OUTBOX PATTERN

Distributed systems that communicate using messages sent over the network are tricky beasts. Even if we disregard the fallible nature of networks, things still may not work as we want.

Let's consider a scenario when microservice A sends a message that is supposed to reach microservice B. Between them there is a broker of some kind. Now, in the perfect world, the message will reach microservice B and will reach it exactly once. Unfortunately, that is not something we can take for granted. It largely depends on our broker configuration and how we use it. We can easily get one of two delivery guarantees - *at-most-once* or *at-least-once*. In the first case, we accept the fact we may sometimes not get the message at all while in the second scenario, we may expect that the same message will sporadically arrive several times. A holy grail is *exactly-once* delivery, but that is not trivial. The world of distributed systems is a fascinating place, but at the same time, is beyond the scope of this book. The author recommends reading about Kafka and RabbitMQ guarantees and their approaches to messaging reliability. Kafka Streams are an especially interesting piece of functionality.

Now, in simpler cases that involve just sending a message out or scheduling a background job AFTER transaction is committed The Outbox Pattern comes to the rescue. The idea is very simple - we save messages to the same database within a transaction that caused them. We end up with persisted changes and temporarily saved messages. Then, another designated thread, coroutine or process opens another transaction, picks up pending messages and sends them. Next, messages are removed from the database, and the second

though - *Unit of Work* also groups a few operations to either succeed or fail as one. It is just not restricted to the database queries.

Originally, *Unit Of Work*[33] was only to track changes made on models to minimize the number of issued queries. In fact, advanced ORMs (like SQLAlchemy) implement such a mechanism. Our *Unit of Work* will be a bit more specialized, and it will expose four handy methods - `begin`, `commit`, `rollback` and `register_callback_after_commit`. Under the hood, it

[33] Martin Fowler, *Patterns of Enterprise Application Architecture*, Chapter 11, Unit Of Work

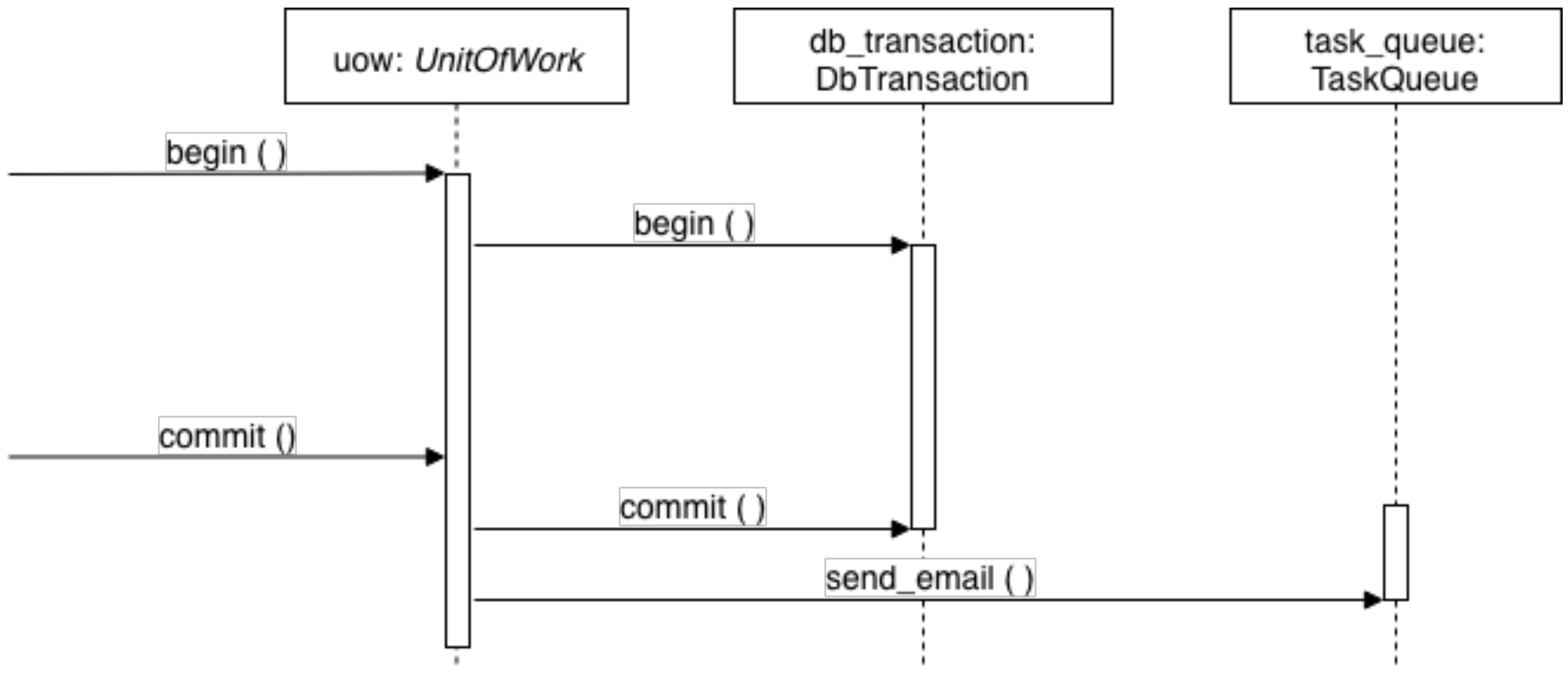


Figure 6.5 Simplified diagram showing the flow of control. send_email is a callback that has been scheduled due to an event edited during the lifetime of a current Unit Of Work

will call appropriate methods of a database transaction and maintain a list of scheduled callbacks:

```
class UnitOfWork(abc.ABC):
    @abc.abstractmethod
    def begin(self) → None:
        pass

    @abc.abstractmethod
    def rollback(self) → None:
        pass

    @abc.abstractmethod
    def commit(self) → None:
        pass

    @abc.abstractmethod
    def register_callback_after_commit(
        self, callback: typing.Callable[[], None]
    ) → None:
        pass
```

An example usage could look like follows:

```python
def setup_event_subscriptions(
    event_bus: EventBus,
) → None:
    event_bus.subscribe(
        BidderHasBeenOverbid,
        lambda event: (  # 1
            inject.instance(  # 2
                UnitOfWork
            ).register_callback_after_commit(  # 3
                lambda: send_email.delay(  # 4
                    event.auction_id,
                    event.bidder_id,
                    event.money.amount,
                )
            )
        ),
    )
```

Interesting places:

1. Upon `BidderHasBeenOverbid` event an anonymous function is called (lambda)
2. During execution (still in the transaction) an instance of current UnitOfWork is obtained…
3. …and told to run another anonymous function right after the current transaction is committed
4. Callback does not accept any arguments, but we can create a closure using another lambda, so we still have data we need.

Depending on the underlying database access library, a *Unit Of Work* can be merely a thin wrapper around it or provide more advanced features, like increased reliability and guarantees about executing after-commit callbacks.

UNIT OF WORK LIFETIME

Putting it all together - the lifetime of *Unit Of Work* is insignificantly longer than the lifetime of a database transaction in the scenario explained under **Events vs transactions vs side effects**. In the context of web, *Unit Of Work* is created once we receive a new request and committed upon the request handling completion. If there is an unhandled exception thrown in the process, *Unit Of Work*'s rollback method will be called. Its role is to discard any changes made. In most cases, it will be just rollbacking database transaction and get rid of pending *Events*.

RELATION BETWEEN *UNIT OF WORK* AND *EVENT BUS*

Event Bus has to be aware of *the Unit Of Work* existence. Most notably, it must operate within the context of a certain instance of *Unit Of Work*. While mapping between *Events* and their subscribers is to be considered a part of the configuration and typically is not changed once the application is started, *Event Bus* scheduling callbacks after the transaction is committed indicates there is a need for statefulness. *Unit Of Work* provides it.

It is up to the software developer to decide whether given subscriber should execute synchronously (within the same *Unit Of Work*) or should it be run asynchronously, for example in a background task queue (outside current *Unit Of Work*, after a commit). A synchronous execution is simpler for *Event Bus* - it just calls subscriber, and it executes right after *Event* has been emitted. For handling asynchronous behavior, a co-operation is required. *Event Bus* will use *Unit of Work*'s `register_callback_after_commit` to ensure subscriber gets *Event*. If introducing *Unit Of Work* is undesired for any reason, it will be enough if we enable *Event Bus* to schedule callbacks after transaction commit in another way.

The tricky part is to make sure that *Event Bus* will always get the right *Unit Of Work* instance. *IoC Container* should be able to provide it - a capability we look for is called *scopes*. If we are talking about handling HTTP requests, we want so-called *request scope*, that exists only as long as handling a single HTTP request. Of course, each request gets its own scope to be fully isolated from others. When we are processing tasks in a worker, we usually want to have scope per task.

Usually, there should be no problem in using mature IoC Containers in the aforementioned way. There should be either plugin to popular frameworks or a possibility for writing a minimal amount of glue code. Documentation of an exemplary DI container, Autofac, is very elaborate about integrating it.

DEALING WITH OTHER CROSS-CUTTING CONCERNS

A lot of space was devoted to discussing transactions handling. You might have thought this had little to do with the Clean Architecture, but it had to be to mentioned since transactions are one of the so-called cross-cutting concerns. Simply saying, it is something that is present everywhere in the application and affects a lot of things by neglecting transactions handling one risks data inconsistency. In certain projects, this may not be a significant problem, and it would be more feasible just to call the customer and apologize. In other businesses, data inconsistency is a serious problem. It might require halting entire system, fixing inconsistency and reverse all side-effects since it occurred. Of course, a bug that caused the disaster still needs to be fixed. It is like stopping a scorching train that is pulled by a steam locomotive, filled up with expensive goods that have a short expiration date and having to fix tracks that train is already on. Oh, I forgot about upset passengers and stakeholders asking every few minutes how long will the delay last. You get the picture.

Not all cross-cutting concerns are able to bite you that much, but putting a slight on them may result in bad code that will be later difficult to maintain. Since this is the Clean Architecture book, let's pay them some attention.

CONFIGURATION

Various settings, like access tokens to external services or credentials to the database, has to be loaded once an application starts. There is also a second class of settings - that are used to parametrize various parts of the application, like frequency of periodic tasks or pool sizes. The third kind of settings are so-called *feature toggles*. Basically, these are boolean flags responsible for turning on/off features they concern. For example, they may be used for disabling features that are not yet complete or make sense only in certain environments, like development or production. One example is using dummy, test payment gateway instead of a real one.

As we can see, there are plenty of occasions for reaching to configuration throughout all application. A *quick and dirty* solution would be to create a class that will represent the configuration like:

```python
class Config(dict):
    pass
```

Then we plug it into dependency injection by initializing the instance with settings. Whenever we need some configuration, we inject `Config` and use it as if it was a raw dictionary:

```python
def setup_dependency_injection(
    settings: dict,
) -> None:
    def di_config(binder: inject.Binder) -> None:
        binder.bind(Config, Config(settings))

    inject.configure(di_config)
```

```python
# somewhere in code
class CaPaymentsPaymentProvider(PaymentProvider):
    @inject.autoparams()
    def __init__(self, config: Config) → None:
        self.auth = (
            config["payments.login"],
            config["payments.password"],
        )
```

On the bright side, this solution is very fast to implement. On the downside, it is not especially clean. Suddenly, dozens of classes & functions start to depend on `Config`. It makes `Config` to be practically immutable code, since changing it would break too much. Naturally, there is little we can do about the implementation shown above. Making a lot of classes depend on `Config` is not a good idea. Besides that, there are no restrictions on what data is available to what class. To sum up - such a design is suboptimal.

Ideally, it would be to contain awareness of configuration in one place - *main*, where everything is assembled:

```python
def setup_dependency_injection(
    settings: dict,
) → None:
    def di_config(binder: inject.Binder) → None:
        binder.bind(
            PaymentProvider,
            CaPaymentsPaymentProvider(
                settings["payments.login"],
                settings["payments.password"],
            ),
        )

    inject.configure(di_config)
```

In the next chapter you will also see a more object-oriented approach to configuration passing when we dive deeper into modularity.

VALIDATION

Validation is one of the trickiest parts of the Clean Architecture. On the one hand, there are plenty of libraries and web frameworks components that do just that - validation. They play smoothly with CRUD applications. On the other hand, there we have *Input DTOs* being passed to *Use Cases*. *Input DTOs may not* be the right place to carry on validation. However, once it gets to *Use Case* we expect it to be correct and safe to use.

To sum up, *Input DTO* provides:

- In *Use Case*, I should be able to rely on semantic correctness (e.g. type) of attributes
    - I am given a `bidder_id` equal to 3. That is fine; I know that IDs of users are positive integers. *Input DTO* instances must not exist with semantically incorrect attributes, e.g. `bidder_id` equal to "incorrect".
- In *Use Case* I usually do not know yet if a semantically correct *Input DTO* will suffice for performing a business operation
    - Typically one will not try to check if a Bidder with given `bidder_id` even exists before calling *Use Case*

To guarantee semantic correctness, the ultimate solution would be to leverage *Value Objects* and existing validation solutions. In Python, there are plethora of excellent libraries that allows for serializing dictionaries into objects, for example marshmallow[34] or Pydantic[35] just to name a few. For a complete example, please refer to `web_app` package[36] of the exemplary project.

In an ideal world, inner layers (*Domain* & *Application*) would be relying on *Value Objects* that by design are immutable and guard their correctness.

SYNCHRONIZATION

If you have ever taken part in any online auction, you probably know how does it feel like when someone overbids you in the last second. Perhaps the winner was not the only one who tried their luck hoping to offer a better price in the very last moment. This task is, however, better suited for bots. I imagine that countless times bots were competing against each other in the last second of an auction. Such a competition provokes the most ordinary race conditions. Unless one secures themselves.

[34] marshmallow - simplified object serialization https://marshmallow.readthedocs.io/en/stable/

[35] https://pydantic-docs.helpmanual.io/

[36] https://github.com/Enforcer/clean-architecture/blob/master/auctioning_platform/web_app/web_app/blueprints/auctions.py#L82

I cut my teeth on tracking down sophisticated race conditions, so I could not resist putting a paragraph about dealing with them in this book. This is a broad topic, so I decided to show one technique that proved to be extremely helpful when applying the Clean Architecture.

It is called optimistic locking. In short, we assign a version to every *Entity*. When we fetch one from a datastore, we also remember what version it had. When the time comes for persisting the *Entity*, we do it conditionally. The condition checks if the version in the datastore equals to what we got when we were fetching the *Entity* in the beginning. If it does, we update the *Entity* and bump up the version by one. In case the version changed (has been bumped up already by someone else in the meantime) then we either fail or retry the whole operation. The problem is that the datastore we use has to support such a conditional update.

Implementation details will vary on the database one uses. I will present how to do it using plain SQL:

```
BEGIN;  # 1

SELECT id, current_price, version FROM auction;  # 2

# do some stuff outside SQL

UPDATE  # 3
    auction
SET
   current_price = {new current price},
   version = {original version} + 1
WHERE
   id = {id}
   AND version = {original version};

COMMIT;  # 4
```

Interesting lines:

1. Everything is wrapped in a transaction, though it does not protect us completely
2. Auction is fetched with a current version number
3. This statement persists our changes provided that no one bumped up the version in the meantime
4. It is up to us to detect whether UPDATE statement has updated any rows (#3). Otherwise, the transaction would be committed as usual regardless of UPDATE result.

Many mature ORMs (including SQLAlchemy[37] for Python, Hibernate[38] for Java or Doctrine[39] for PHP) has support for this feature, so one does not have to think that low-level.

Synchronization is mentioned as a cross-cutting concern because in many cases *Application* cannot be ignorant about it, especially when we coordinate a complex dance of *Repositories* and *Ports*. It is the implementation of *Data Access Interface* (e.g. *Repository*) that will have to raise `ConcurrentSave` exception after all.

In many cases, though, locking is undesirable since it impacts performance. If two bidders try their luck at the same time, one of them will have to retry and will block server resources for at least twice as much. An alternative is to rework a design, so it does not need locking at all. If we think about auctions, we could insert every new bid to the table in the database. However, it would then no longer be feasible to keep the current price on an auction. It would become a virtual property that could be calculated only by getting the highest bid. It is also quite hard to detect who was overbid and when. Choosing a synchronization strategy is a game of trade-offs. Moreover, it is heavily dependant on underlying data storage.

CHAPTER SUMMARY

This chapter has been a wild ride with lots of new information. Implementation of two *Use Cases* (`PlacingBid` and `EndingAuction`) has been presented. `Auction` *Entity* has undergone a gradual evolution to meet more and more requirements. All written business code was covered with tests. Concept of *Data Access* in the form of abstract *Persistence-Oriented Repository* was presented. *Port* for making payments and its concrete counterpart, *Adapter* has been crafted. Various approaches to implementing Read Side were discussed. Finally, inverting control with events was demonstrated. In the very end, few tips on dealing with cross-cutting concerns were given.

It takes time and writing some code to fully wrap one's head around all of these building blocks. It also helps to look at them from a broader perspective of an entire application:

The crucial thing to discover is where the boundaries of the application lie. State of the system is mutated using *Use Cases* which accept *Input DTOs*. During mutation, *Events* may be

[37] SQLAlchemy https://www.sqlalchemy.org/

[38] Hibernate https://hibernate.org/

[39] Doctrine https://www.doctrine-project.org/

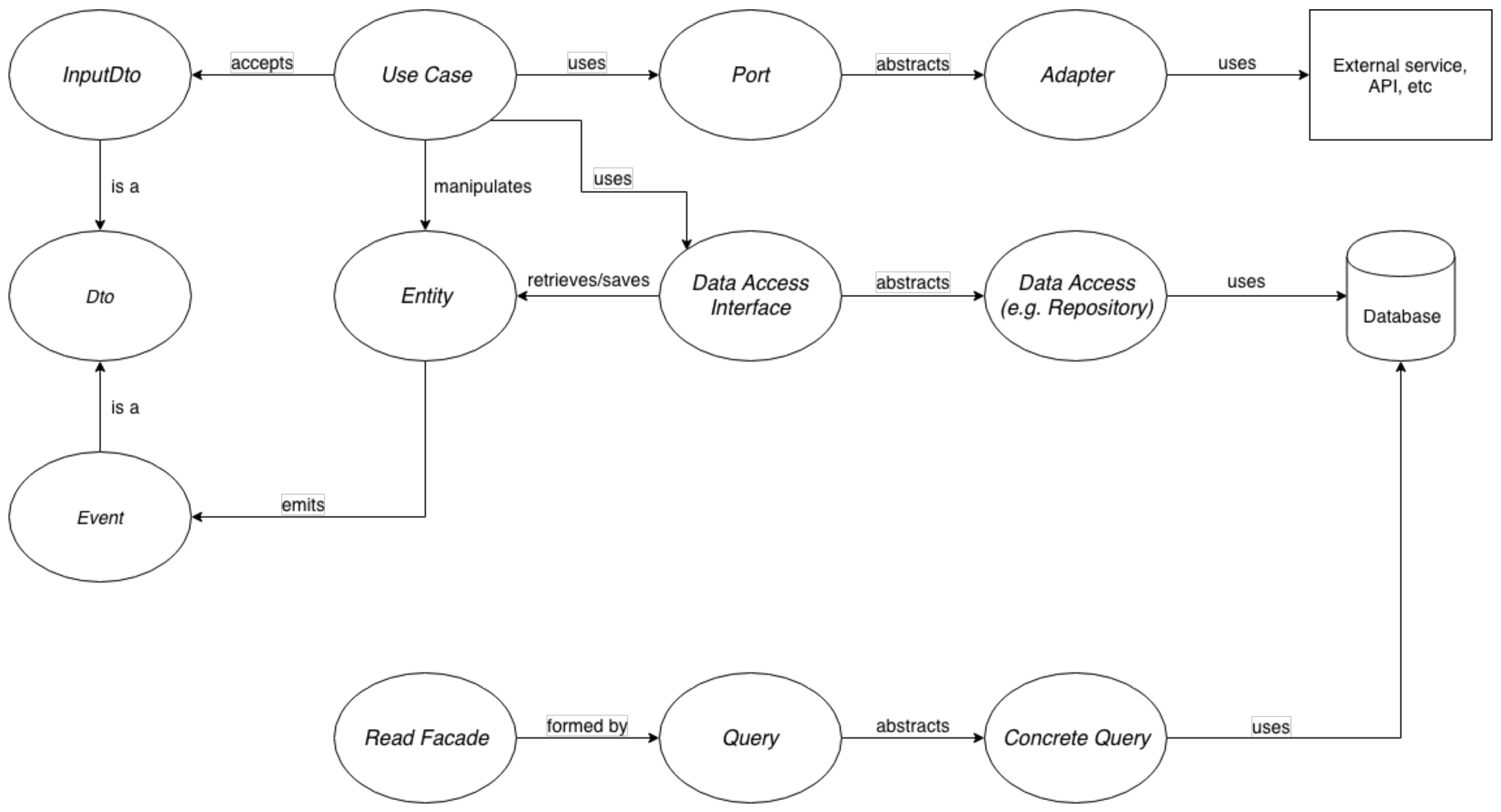


Figure 6.6 Dependencies between majority of building blocks demonstrated in this chapter

emitted to let know outer world what happened. There is also a *Read Facade* that allows querying the system about its state.

MODULARITY

"Software development is a learning process; working code is a side effect." - Alberto Brandolini

THE BURDEN OF SUCCESS - GROWTH & CONTINUOUS CHANGES

Let's say we successfully released our application to production. Even though the dust has not settled yet, stakeholders are already full of new ideas. Also, first customers gave feedback. It is inevitable - new features requests are coming. Brace yourself because this is going to be the ultimate trial for the design you produced. It is interesting to observe how much of a carefully crafted code will still be there once we start serious development under pressure.

COHESION AND MODULES

Where do business requirements come from? The answer varies depending on the team structure. If you are developing a pet project, you are your own boss and the only source of vision of the project. A single Scrum team with proxy product owner is in a slightly more complicated situation - the latter takes care of distilling requirements and present a clear vision to the developers. Even then, a product owner often has to deal with *a not-always-compatible* feature or change requests coming from different people, or in a greater scale - departments. Not to mention users demanding new features that are crucial from their point of view. This should already ring a bell - this is a situation that *Single Responsibility Principle* warns about - the same code might have more than one reason to change. Luckily, we already know the remedy - refactor code, so individual parts do not have to change for different reasons.

Humans (in particular software developers) are bad when it comes to naming things. Even within one organization, the same term may mean a completely different thing to various people. Consider *item* in the context of auctioning platform that also manages stock. When an auction is configured, an item is nothing more than a starting price, a bunch of images, name, maybe some attributes. From the standpoint of placing bids, the item is irrelevant. However, the item is everything for the bidder - they participate in the auction just to get it. When they do win an auction, and the item is about to be shipped, what matters most are its size, weight or location in the warehouse. It is next to impossible to reconcile all these different meanings and sets of features if we tried to create a single class. It would be bloated with many attributes or methods which are relevant only in specific contexts.

If it is so hard and impractical to satisfy everyone's needs at once, what else can we do? Recall good ol' *divide et impera* rule - meet needs separately when it makes sense, so instead of one giant problem we have several much smaller ones.

PACKAGING CODE BY FEATURE

Imagine stakeholders requested a new functionality: each user can opt-in for some areas of interest, e.g. technology, fantasy books, healthy food or parenting. Users can opt-in and opt-out anytime.

Here is how the current project structure looks like:

```
/root
├── auctions
│   ├── …
├── auctions_infrastructure
│   ├── …
└── web_app
```

Where should we put new code? Areas of interest have nothing in common with auctions after all. We want our core module to stay cohesive, meaning no one wonders "what the heck is this thing doing here? The package is called auctions, but that *areas of interest thing* are definitely not about auctioning. We should avoid sticking everything in one package unless we want to end up with a scary *Big Ball of Mud*[40].

One of the Clean Architecture chapters, *The missing chapter* is precisely about this issue - packaging code. Keeping our modules, cohesive quickly pays off as they are much easier to understand and maintain. Hence, it is logical to create a new package for this feature - let's call it preferences. Such packing approach would be a materialization of the idea from The missing chapter - *package by feature*.

MODULES AND FLEXIBILITY OF INTERIOR DESIGN

How should the preferences package look like? Writing *Use Cases, Entities, Data Access Interface* and its implementation for such a simple thing is a bit of an overkill. The actual feature is not too complicated - it is nothing more than saving a bunch of boolean values coming from a single form with several checkboxes.

[40] Brian Foote, Joseph Yoder, Big Ball of Mud http://www.laputan.org/mud/mud.html#BigBallOfMud

Although the Clean Architecture is a perfect match for projects that have a rich domain, it IS NOT a one-size-fits-all solution. Let's face it - it does not make sense to develop an entire project using only the Clean Architecture. Sometimes good ol' CRUD would do. Using the Clean Architecture, as well as other sophisticated techniques, is an investment. It makes sense for the most complex, valuable parts.

Hence, nothing prevents us from implementing certain parts of the application using different, simpler approaches. Even without any abstractions along the way to the database.

MODULES VERSUS MICROSERVICES

All these features of the modular approach mentioned above may resemble microservices. They have, in fact, a lot in common. Many qualities of a good microservice can also be used to describe a well-written module. For example, a microservice is considered to be properly decoupled from the rest of the system if introducing changes there does not require rebuilding half of the application. Being able to develop a microservice without worrying about breaking other moving parts in the system indicates a fine dose of decoupling and encapsulation.

Forming boundaries of microservices is a big thing. It is virtually impossible if the team does not have a strong understanding of the project and its domain. A potential mistake can cost a lot. In case of sub-packages, living in a single codebase, fixing a wrong design is more affordable - it usually amounts to moving code from one package to another. It is way harder with microservices, especially if they were developed in total separation for several months or longer.

The most obvious difference is that there is no network communication overhead when one module talks to another - unlike microservices. Microservices also require significantly more effort from the (Dev)Ops to keep them up and to run than a single, monolithic application. Not to mention increased cost of deployment and maintenance.

Good modular design does not obstruct splitting into microservices. What is more, it allows for deferring this decision with a far-reaching consequence until we find it is necessary. For example, a given module which is responsible for communication with Payment Provider might be moved to a microservice in order to implement extra security measures. Putting it on a dedicated machine, limiting the number of people with access to literally few, keeping credentials to Payment Provider only there would definitely reduce the number of possible attack vectors. Another example might be team growth and management challenges - why not organize a team around features and allow each one of them to own codebases separately?

To sum up, there is absolutely no need to migrate to microservices if all that one needs is to have nicely organized, maintainable codebase. It is perfectly fine to stay with a monolith provided it is modular. Also, there is a fancy word for a modular monolith - it is *modulith*. Modular monolith is a reasonable middle ground between one disorganized code base and microservices.

MODULES VERSUS USER

You might have noticed that before this chapter, there was no single occurrence of a term *user*. I omitted it on purpose since this is one of the vaguest and abused names in software projects. It definitely does not help to organize code into modules. In the context of auctions, Bidder is the right name to use, at least when we talk about a person that places bids. It is quite easy to come up with names for each module. There will be Subject (authentication), Recipient (mailing) or Reporter (support). However, in the end, one will have to associate Subject with corresponding *Entity* from another module. The easiest way is to share common ID, while the rest of *Entity*'s data depends on the module it belongs to.

Recall how **`PlacingBidInputDto`** was created in the web package. `bidder_id` is a required argument which can be obtained from the authentication mechanism that is in use in the web framework.

MODULES VS BOUNDED CONTEXTS

All this modules talk might (and I hope it does!) ring a bell for readers familiar with strategic Domain-Driven Design. For those unfamiliar with DDD, there is a modeling building block called Bounded Context. It delimits an area of applicability of a given model, guaranteeing its cohesion. It means that we should instead produce more specialized models. There is no need either benefit from putting everything in one User class. We also do not have to worry if our cohesive, specialized Bidder is not usable in another Bounded Context. It is better when it is not. The relation between Bounded Context and module is that Bounded Context can span one or more modules, but there should never be multiple Bounded Contexts within a single module. Strategic DDD is outside the scope of this book. I highly recommend having at least a read about it.

STRATEGIC DOMAIN-DRIVEN DESIGN

If the strategies as mentioned earlier sound like second-guessing to you and you would rather use something more formalized, try to apply strategic patterns of Domain-Driven Design. In particular, read about a technique called Context Mapping. The resultant artefact is called Context Map. The latter shows discovered Bounded Contexts and relations between them. The practical approach to get it is to conduct Event Storming session(s). One requirement, though - stakeholders has to be included in the workshop. Software developers will not be able to come up with an accurate vision on their own, period. The last piece of advice - do not treat the Context Map as something set in stone. Requirements and knowledge of all involved people will evolve over time and as every written document, it may simply become obsolete at some point.

MODULES IMPLEMENTATION

To sum up, a module is a cohesive package of code with its own vocabulary (in particular having a specific name for a user). It groups related functionalities and exposes them using a module API. A well-designed module resembles a microservice, though there is no network communication needed to use the module.

Modules that follow the Clean Architecture expose a subset of standard building blocks for others to use:

- *Use Cases*
- *Queries* (or another form of a read model)
- Events (other modules can subscribe to them)

`auctions` package already can be treated as a module.

Modules that do not implement the Clean Architecture are free to use the structure that is the most convenient. Since a module is expected to have an API, one can leverage *Facade* design pattern. A *Facade* is to be used in the same manner as *Use Cases* or Queries - synchronous call from the outside. Module’s *Facade* exposes methods for mutating and querying the state of the system within the module. Let’s take a module which contains code responsible for customer relationship management. More precisely, it sends informative e-mails.

The facade may look like this:

```python
class CustomerRelationshipFacade:
    def create_customer(
        self, customer_id: int, email: str
    ) -> None:
        ...

    def update_customer(
        self, customer_id: int, email: str
    ) -> None:
        ...

    def send_email_about_overbid(
        self,
        customer_id: int,
        new_price: Money,
        auction_title: str,
    ) -> None:
        ...

    def send_email_about_winning(
        self,
        customer_id: int,
        bid_amount: Money,
        auction_title: str,
    ) -> None:
        ...
```

MODULES DEPENDING ON EACH OTHER

The trickiest part about modules is that they need to cooperate. It is a rare situation that module can exist in complete isolation without being aware of any other modules. However, dividing codebase into modules and figuring out how they depend on each other is orders of magnitude more maintainable solution than adding more and more code to a single *Big Ball of Mud*. Grouping code quickly pays off.

Provided there are two modules (respectively A and B), there are three possible relations between them:

1. A and B do not know about each other

2. A depends on B
3. B depends on A

SEPARATE WAYS

Situation #1 does not require any additional explanations. A and B exist unaware of each other.

Situations #2 and #3 are much more interesting.

DIRECT DEPENDENCY - BOTH MODULES IMPLEMENT THE CLEAN ARCHITECTURE

Assuming both A and B implement the Clean Architecture, how A can call B's code? When it comes to a synchronous, direct control, we would usually want to call *Use Case* of one module from another. However, we do not want to call B's *Use Case* directly - it would couple A with B and made it next to impossible to test one without another. Luckily, in the Clean Architecture, there is a building block that solves the problem - *Input Boundary*, being just an abstraction over *Use Case*.

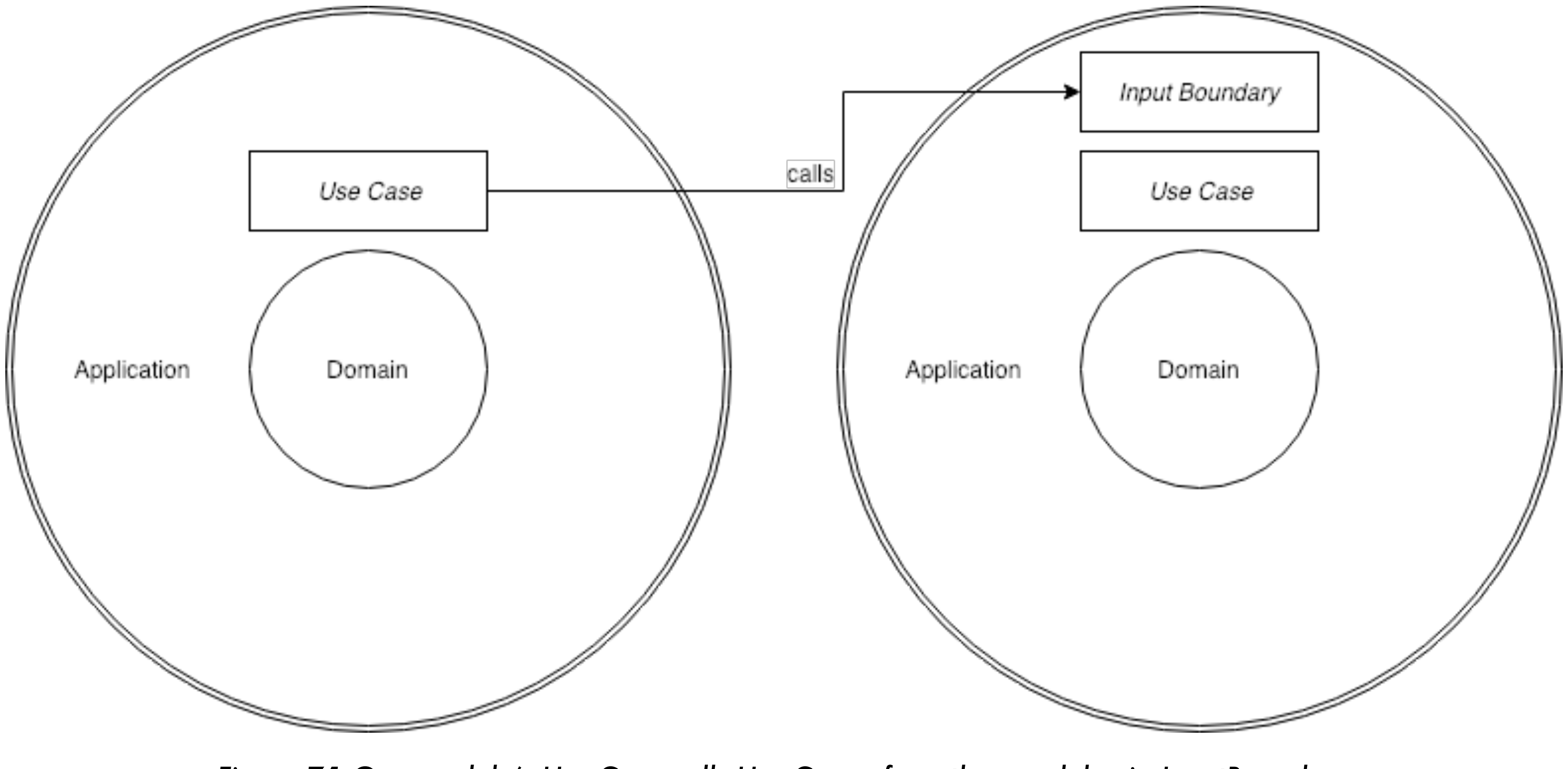


Figure 7.1 One module's Use Case calls Use Case of another module via InputBoundary

This solution is *good enough* if it is acceptable for a caller module to embrace naming of the called module. An *Input Dto* belonging to B has to be assembled in A after all. The decisive factor here is which module is more significant from the business perspective. If the called module (B) is, then one can only reconcile with the fact that A has to know something about B's language. In the opposite case, when a leak of B's (inferior) vocabulary into A (superior) is undesirable, there is still something one can do. What we can do instead, is to write a *Port* in A while B has to provide an *Adapter*. The former belongs to A where it will be used, so there is no risk of leaking knowledge from B to A. Method names of the *Port/ Adapter* will belong to A. Still, the implementation will repack data appropriately to call B's Use Case eventually. Therefore, the *Port/Adapter* pair will not be complex - it just separates two worlds.

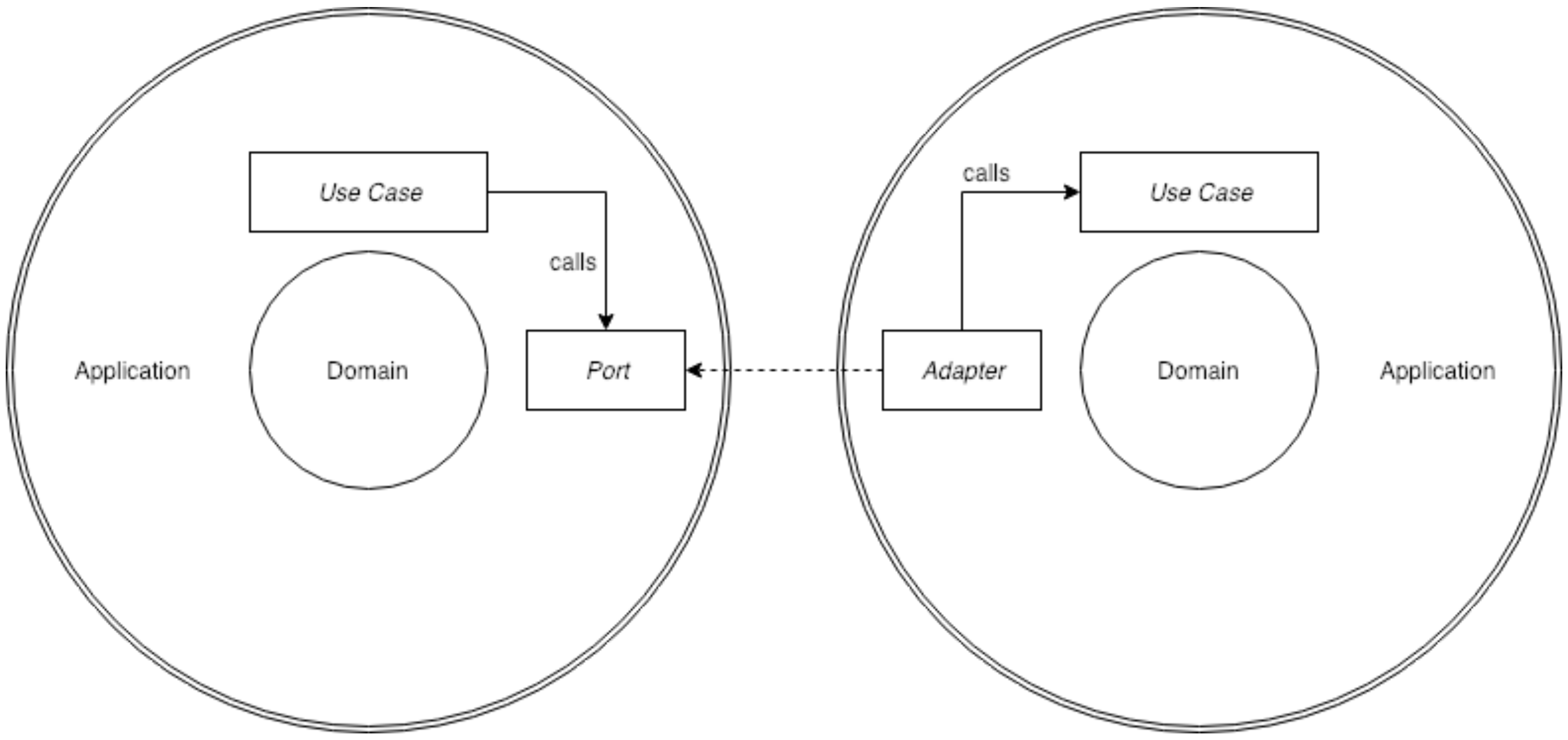


Figure 7.2 Use Case calls Port within the same module. The Port abstracts away Adapter from another module

However, bear in mind that **direct dependency is a very tight form of coupling and should be used sparingly**. It makes perfect sense when both modules are part of the same Bounded Context, and they share a vocabulary.

INDIRECT DEPENDENCY - TWO INSTANCES OF THE CLEAN ARCHITECTURE

In case of an indirect dependency, we rely on events. We introduce a new building block - *Handler* - that will react to the event from one module and call appropriate *Use Case* from the module it belongs to.

An indirect dependency decouples subscribing module from the one that emits events. The subscribing module may still be coupled to another one if it imports any code (e.g. event class to configure subscription). To fully decouple two modules, we need to configure subscriptions in the `main` module, just like it was shown in the previous example. For most of the time we won't be needing that, though.

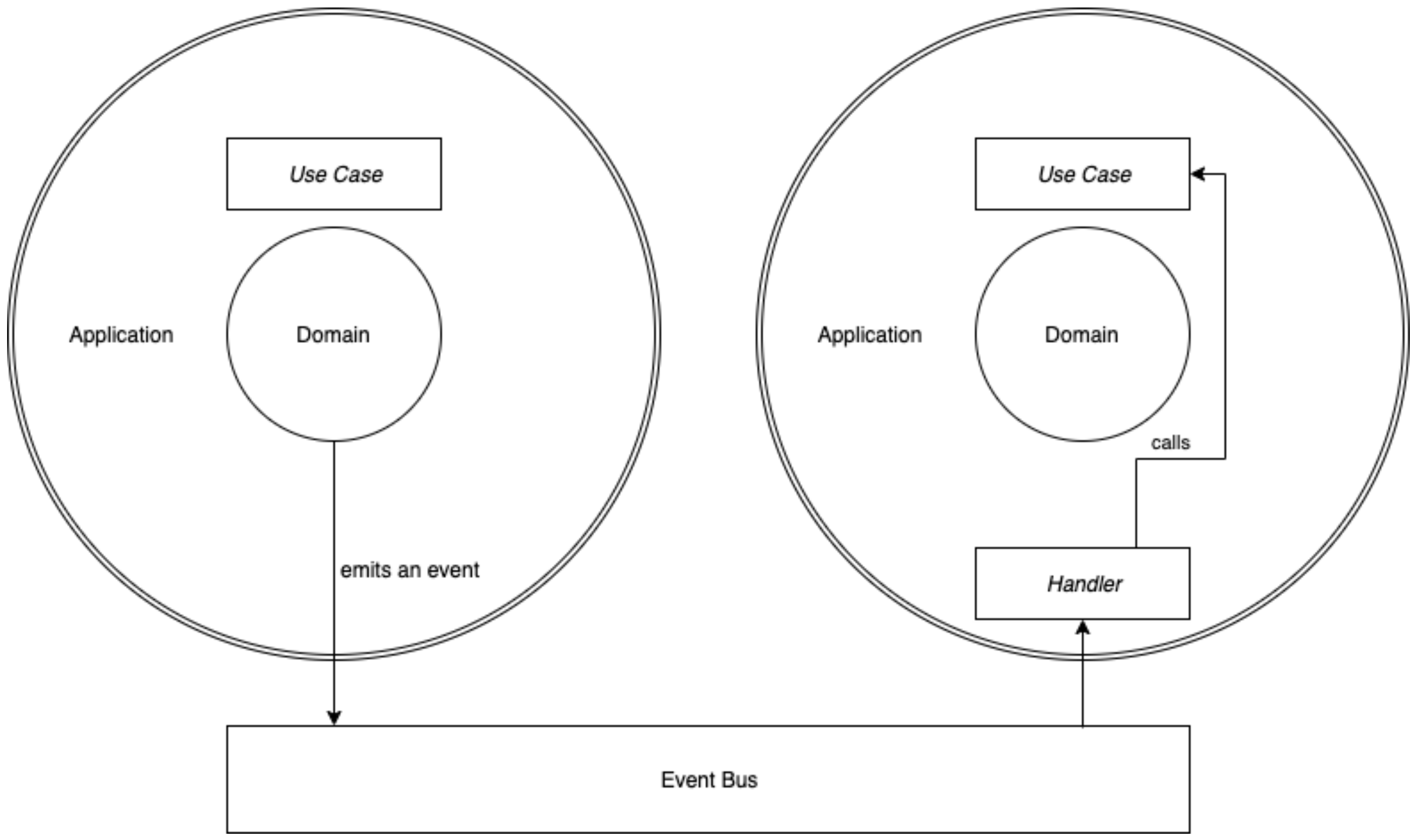


Figure 7.3 One module's Handler subscribes to events from another module. When it gets notified, it calls an appropriate Use Case

DEPENDENCY WHEN ONE OF THE MODULES IS NOT IMPLEMENTING THE CLEAN ARCHITECTURE

In case of a direct dependency, when CRUD module has to use another one that implements the Clean Architecture, we call the *Use Case* (optionally via corresponding *Input Boundary*).

When we deal with an opposite situation, we add a *Port* to the Clean Architecture module and implement Adapter in the CRUD module.

Again, direct use introduces strong coupling between modules. Use it with caution. In case of an indirect dependency, one resorts to events.

FLAVORS OF EVENTS-BASED INTEGRATION

One of the paragraphs above mention about one module subscribing to events emitted by another one. It suggests that the subscribing module still needs to rely on code from the second module directly. Naturally, this is a form of coupling. It is not always a problem, but when it is, one has a way out. In a yet another module, one puts handlers that subscribe to events and call appropriate logic from other modules. These handlers do all the translation on their own, effectively making modules unaware of each other. For a multi-stage process with cascades of events, a patterns called *Saga* or a *Process Manager* can help. *Process Manager* will be discussed and presented later in the book.

DEPENDENCIES BETWEEN MODULES - REASSURANCE

Don't worry if the above examples seem a bit abstract right now. Provided we have quite a few options we may hesitate when choosing the one to go with. There is no single good approach - each one of them has unique features which may or may not suit our modules. We will learn it soon. Everything will be illustrated with examples on a few next pages.

CASE STUDY - AUCTIONING PLATFORM

DISCOVER DIFFERENT MODULES

So far, code examples concentrated solely on auctioning itself. However, it is not enough to successfully run an auctioning platform. After all, we need to receive payments, notify bidders about various things or ship the won item to them. Usually, stakeholders are the best source for this kind of information. Domain knowledge may be acquired during meetings or workshops, like Event Storming.

If direct co-operation is limited or not possible, one can always try a mental exercise of mapping different features onto different departments of the company working. One employee conducts auctions, while another one accepts payment and makes financial reports. Yet another employee cares about making sure that won item gets to the winner. This way of modeling works best when we know how the business works.

It may also occur that a new module will emerge organically - developers will discover them themselves. In our case, we discovered a new module twice, when our *Ports* and *Adapters* were growing and growing. In both cases, at first Port and Adapter were pretty simple, with no more than three methods. When we added the 8th method to a *Port* and *Adapter* by then was a *Facade* over quite complicated subsystem, it became clear it is time to refactor and find a new home for this code.

AUCTIONING PLATFORM MODULES

Foundation - place for classes shared between different modules. Foundation serves as a standard lib extension. Also known as *Shared Kernel* in DDD. For example, *Money* class belongs here.

Auctions - everything related to auctions itself. In particular, all nuances of auctioning are modelled here.

Customer Relationship - hosts details about communication with the customer, like e-mails contents.

Payments - here lies code responsible for processing payments and integration with an external provider.

Shipping - models shipment process - addresses, statuses etc.

Processes - home of *Sagas* and *Process Managers*. It contains complex business processes that span multiple modules.

Main - special module that is solely responsible for assembling everything else.

Web Application - an HTTP interface to the platform. Heavily dependant on the chosen web framework.

Module	Architecture	Term for user
Auctions	the Clean Architecture	Bidder
Customer Relationship	Facade-based	Customer
Payments	Facade-based	Customer
Shipping	the Clean Architecture	Consignee
Web Application	framework-dependant	Imposed by a framework

Tab 1 Overview of chosen modules for auctioning platform

These are not all, of course. However, these are main, language-agnostic modules. We are going to need several auxiliary ones which are specific to the programming language and 3rd party libraries we use. We could also add more, for example, inventory management, but let's keep things a bit simpler to maximize educational value.

You surely noticed that Customer is used in Customer Relationship and Payments module. Needless to say, Customer means an entirely different thing in each of these modules.

ANATOMY OF A MODULE - COMMON PART

Each module of the project exposes public classes to the outer world.

The Clean Architecture-based ones will expose in particular:

- *Domain Events*
- *Types (e.g.* `AuctionId`*)*
- *Repositories* (abstract base class)
- *Input Boundaries* (or *Use Cases* if the module does not use the former)
- *Input DTOs*
- *Output Boundaries* (if we use them)
- *Output DTOs* (if we use them)
- *Queries* (if we implement read stack in this way)
- *Ports*
- *Dependency Injection module* (ouch, word “module” is so overloaded)

For **Auctions** module in Python, this will look like this:

```python
__all__ = [
    # module
    "Auctions",
    # events
    "AuctionEnded",
    "WinningBidPlaced",
    "BidderHasBeenOverbid",
    # repositories
    “AuctionsRepository",
    # types
    "AuctionId",
    # use cases
    "PlacingBid",
    "PlacingBidOutputBoundary",
    "WithdrawingBids",
    # input dtos
    "BeginningAuctionInputDto",
    "EndingAuctionInputDto",
    "PlacingBidInputDto",
    "WithdrawingBidsInputDto",
    # queries
    "GetActiveAuctions",
    "GetSingleAuction",
]


class Auctions(injector.Module):
    @injector.provider
    def placing_bid_uc(
        self,
        boundary: PlacingBidOutputBoundary,
        repo: AuctionsRepository,
    ) -> PlacingBid:
        return PlacingBid(boundary, repo)

    @injector.provider
    def withdrawing_bids_uc(
        self, repo: AuctionsRepository
    ) -> WithdrawingBids:
        return WithdrawingBids(repo)
```

A counterpart module with infrastructure-specific implementations, called **Auctions Infrastructure**, looks as follows:

```python
__all__ = [
    # module
    "AuctionsInfrastructure",
    # models, needed for SQLAlchemy ORM to discover them
    "auctions",
    "bids",
]


class AuctionsInfrastructure(injector.Module):
    @injector.provider
    def get_active_auctions(
        self, conn: Connection
    ) -> GetActiveAuctions:
        return SqlGetActiveAuctions(conn)

    @injector.provider
    def get_single_auction(
        self, conn: Connection
    ) -> GetSingleAuction:
        return SqlGetSingleAuction(conn)

    @injector.provider
    def auctions_repo(
        self,
        conn: Connection,
        event_bus: EventBus,
    ) -> AuctionsRepository:
        return SqlAlchemyAuctionsRepo(
            conn, event_bus
        )
```

Notice that we are exposing a lot of stuff outside. If we implemented a CQRS write stack, we could expose much less. Instead of showing off *Use Cases* (or *Input Boundaries*) and *Input DTOs*, we would be exposing only *Command* classes (roughly equivalent to *Input DTOs*).

A *Facade*-based module, **Payments**, exposes:

- *Events*
- *Facade*
- *Config DTO*
- *Dependency Injection module*

```python
__all__ = [
    # module
    "Payments",
    "PaymentsConfig",
    # facade
    "PaymentsFacade",
    # events
    "PaymentStarted",
    "PaymentCharged",
    "PaymentCaptured",
    "PaymentFailed",
]


class Payments(injector.Module):
    @injector.provider
    def facade(
        self,
        config: PaymentsConfig,
        connection: Connection,
        event_bus: EventBus,
    ) -> PaymentsFacade:
        return PaymentsFacade(
            config, connection, event_bus
        )
```

Second Facade-based module, **Customer Relationship**, should not surprise us. It also exposes just a *Facade*:

```python
__all__ = [
    # module
    "CustomerRelationship",
    "CustomerRelationshipConfig",
    # facade
    "CustomerRelationshipFacade",
]


class CustomerRelationship(injector.Module):
    @injector.provider
    def facade(
        self,
        config: CustomerRelationshipConfig,
        connection: Connection,
    ) -> CustomerRelationshipFacade:
        return CustomerRelationshipFacade(
            config, connection
        )
```

Processes is one of the most interesting modules, though the majority of its code will be skipped for now. It will be discussed in detail later in this chapter. It will also expose the Dependency Injection module:

```
__all__ = [
    # module
    "Processes"
]


class Processes(injector.Module):
    ...
```

Main and **Foundation** are a bit different creatures - the role of the former is to assemble all other modules into application while the latter is simply a place for commonly used classes or functions. Hence, neither of them expose a Dependency Injection module named **Main** or **Foundation**. On the other hand, in the example project written for the book, **Main** defines several helper module classes. Most of them will be technology-specific, but there is an exception - *Configs* module class to provide all configurations required for other modules. Both **Payments** and **CustomerRelationship** require such a *DTO* to work:

```
class Configs(injector.Module):
    def __init__(self, settings: dict) → None:
        self._settings = settings

    @injector.singleton
    @injector.provider
    def customer_relationship_config(
        self,
    ) → CustomerRelationshipConfig:
        return CustomerRelationshipConfig(
            self._settings["email.host"],
            int(self._settings["email.port"]),
            self._settings["email.username"],
            self._settings["email.password"],
            (
                self._settings["email.from.name"],
                self._settings[
                    "email.from.address"
                ],
            ),
        )
    @injector.singleton
    @injector.provider
    def payments_config(self) -> PaymentsConfig:
        return PaymentsConfig(
            self._settings["payments.login"],
            self._settings["payments.password"],
        )
```

Apart from that, **Main** will take *Dependency Injection module* classes and will use them to build an *Injector* - our IoC Container:

```
# somehere in Main
def setup_dependency_injection(
    settings: dict,
    connection_provider: ConnectionProvider,
) -> injector.Injector:
    return injector.Injector(
        [
            Db(connection_provider),
            RedisMod(),
            Rq(),
            EventBusMod(),
            Configs(settings),
            Auctions(),
            AuctionsInfrastructure(),
            CustomerRelationship(),
            Payments(),
            Processes(),
        ],
        auto_bind=False,
    )
```

A resultant IoC Container is capable of building for us a tree of objects to access our applications functionality. For the time being, it remains ignorant of delivery mechanism…

Last but not least, there is a **Web Application**. Structure of this one is ultimately dependant on the web framework and chosen technology. The one important matter is to choose IoC Container and web framework that play nicely together, as it is the case with Flask and injector. The ultimate result we want to achieve is to get rid of the explicit use of IoC Container and leave it to tools. We use a Flask-Injector lib to provide integration:

```
# during app construction ...
app = Flask(__name__)
# ... we register plugin …
main_injector = main.setup_dependency_injection()
FlaskInjector(app, injector=main_injector)
```

```
# later in view code
@auctions_blueprint.route("/")
def auctions_list(
    query: GetActiveAuctions,
) → Response:
    # query argument is injected by Flask-Injector
    # We don't use injector directly, woo-hoo!
    return make_response(jsonify(query.query()))
```

This example is kept to a minimum because it is heavily dependant on the choice of libraries. A mature IoC Container should provide tips on integrating it with a range of web frameworks, as it is the case for Autofac for C# projects.

The rule should be visible with the naked eye. In essence, a module is not only a bunch of code put under a uniquely named directory but also a first-class citizen that defines a *Dependency Injection module*. Later, IoC Container (Injector class in code examples) will use modules classes to resolve requested classes and handlers. However, note that Injector module classes are something very specific to Python and Injector. A Holy Grail is to make code ignorant about DI as much as possible - especially in **Auctions** module.

MODULES' ARCHITECTURE DIFFERENCES - CAN THEY BE UNIFIED?

It may be worrying that different modules do not follow the same architectural pattern. Although uniformity is highly desired, leaving it behind is a trade-off between complex modules requiring the Clean Architecture and simpler ones. In other words, this decision means giving up on uniformity but gets rid of over-engineering in less-complex modules.

It is relatively easy to accept that modules will have different internal structure. First of all, they have to be specialized for a particular problem they model. Secondly, a module's internal structure is like private attributes of a class - they are not meant to be used directly. They are implementation detail hidden behind the public API of a class. That being said, if unifying internal structure is not a good idea, how about modules API? Is there any middle-ground between approach derived from the Clean Architecture with *Input Boundaries* + *Input DTOs* and simple *Facades*?

Actually, there is, and it was already mentioned in the book - it is a combination of *Command*, *Command Handler* and *Command Bus* patterns from CQRS. Refactoring from the Clean Architecture is fairly easy - *Input DTOs* become *Commands*, *Input Boundaries* are removed, and *Use Cases* are registered as *Command Handlers* so that *Command Bus* can dispatch

Commands. For *Facades* it is also not complicated - we create *Command* classes and wire them together with *Facade*'s methods or split the *Facade* into a bunch of *Command Handlers*.

```
# PlacingBidInputDto becomes standalone Command
class PlaceBidCommand:
    bidder_id: BidderId
    auction_id: AuctionId
    amount: Money


# PlacingBid Use Case becomes PlaceBidHandler
class PlaceBidHandler:
    def execute(
        self, command: PlaceBidCommand
    ) → None:
        ...


# module registers command handler
class Auctions(injector.Module):
    @injector.provider
    def place_bid_handler(
        self,
        boundary: PlacingBidOutputBoundary,
        repo: AuctionsRepository,
    ) → Handler[PlaceBidCommand]:
        return PlaceBidHandler(boundary, repo)
```

From now on, to use **Auctions** module services, one only has to know about `PlaceBidCommand`. After instantiating a Command, one uses a *Command Bus* - *Mediator*-style pattern that will dispatch `PlaceBidCommand` to appropriate *Handler*.

```
TCommand = TypeVar("TCommand")


# we create a dummy Generic class
# for nicely looking bindings
class Handler(Generic[TCommand]):
    def __call__(self, command: TCommand) → None:
        pass
```

```
class CommandBus:
    def __init__(
        self, injector: Injector
    ) -> None:
        self._injector = injector

    def dispatch(self, command: Any) -> None:
        # CommandBus uses injector to find a handler
        handler = self._injector.get(
            Handler[type(command)]
        )
        # just to call it with a command instance
        handler(command)


# Command will be a simple dataclass
@dataclass(frozen=True)
class WithdrawBid:
    bid_id: BidId


class WithdrawBidHandler:
    def __call__(
        self, command: WithdrawBid
    ) -> None:
        # dummy handler just prints what it gets
        print(f"Handling {command}!")


class Auctions(Module):
    def withdraw_bid_handler(
        self,
    ) -> Handler[WithdrawBid]:
        return WithdrawBidHandler()


# IoC is configured with Module
injector = Injector([Auctions()])
# CommandBus is initiated with injector instance
command_bus = CommandBus(injector)

# et voila!
command_bus.dispatch(WithdrawBid(bid_id="123"))
# prints "Handling WithdrawBid(bid_id='123')!"
```

Note a profound consequence - the module no longer has to expose *Use Cases* or their *Input Boundaries* as it was the case with **`PlaceBidHandler`**. In other words, we encapsulate more, minimizing coupling with other modules.

This refactoring is fully optional - it will not be presented further in the book. It was mentioned for the sake of addressing doubts about lack of architectural uniformity across modules.

DEPENDENCIES BETWEEN MODULES

Thanks to the explicit nature of *Dependency Injection*, it becomes obvious how modules of auctioning platform depend on each other.

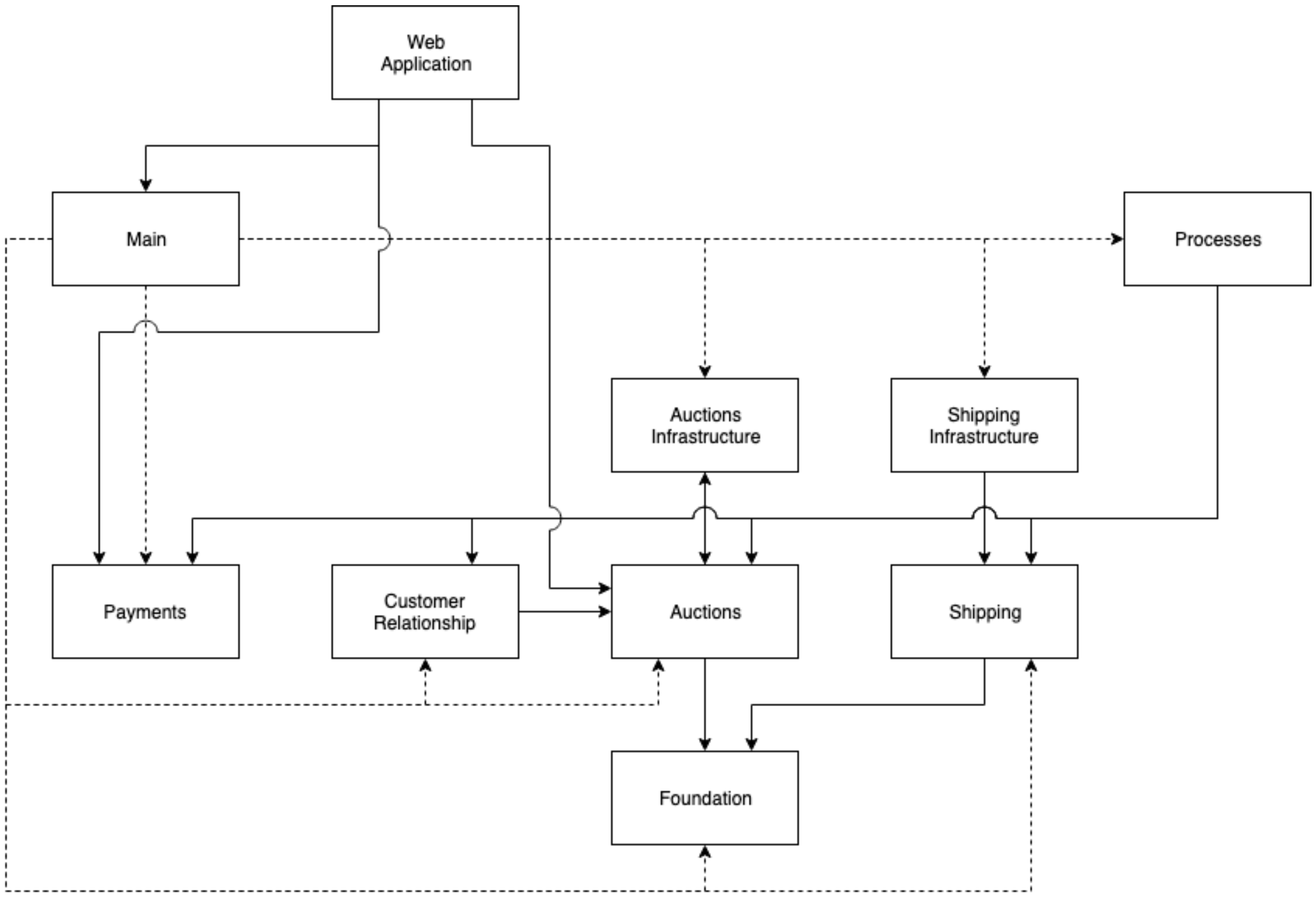


Figure 7.4 Dependencies between packages in the example project

Starting from the bottom, **Foundation** does not depend on any other module. Ideally, it would only use a standard library of the programming language and certain 3rd party libraries that do not tie it with a concrete database, framework etc.

Auctions, as well as **Shipping**, being both the Clean Architecture based modules, depend on **Foundation** only.

Infrastructure-specific implementations **Auctions Infrastructure** and **Shipping Infrastructure**, depend respectively on **Auctions** and **Shipping** modules. They may also depend on **Foundation**, mostly due to **Auctions** or **Shipping** depending on it. Moreover, Infrastructure modules are tied to 3rd party libraries providing access to database of choice etc.

Customer Relationship in our case rely on **Auctions** module because it will react to certain Auctions' domain events. For example, when `BidderHasBeenOverbid` or `WinningBidPlaced` occurs, the module sends appropriate e-mails. **Customer Relationship** also uses 3rd party libraries to access the database.

Payments does not depend on any other module, but it is tied to 3rd party libraries to access the database. It does not subscribe directly to any events. Therefore it does not rely on **Auctions** or **Shipping**. Its *Facade*'s methods will be called in a bit different way, which will be described later in this chapter.

Processes module glues all other modules that take part in more complex business scenarios. It will depend on **Auctions**, **Shipping**, **Customer Relationship** and **Payments** to either react to their events or call *Facade*'s methods/*Use Cases* accordingly. Implementation details will be explained thoroughly, just a few paragraphs further in this chapter.

Main will rely on every other module but **Web Application**. Although **Main** module knows how to assemble the application, it remains ignorant of a delivery mechanism, i.e. web, CLI, background task queue etc.

Web Application will depend on **Main** and any other module which will be connected to the Web API. In this case, these are **Auctions** and **Payments**.

Kind reminder - even though a few modules have access to the database, they do not share database tables! Models defined in one module stay private to this module.

EVENTS EMITTING AND HANDLING BASICS

Although events emitting were already discussed in the previous chapter, the most important facts have to reminded because *Events* are the cornerstone of integration between modules. Hence, we have to be much more serious about them.

To start with, in the Clean Architecture-based module, *Events* can be emitted from *Repository* (*Entity* keeps events in a private field until the moment it is being saved) or from *Entity* (then we consciously violate the Dependency Rule). It should also not be a big deal to emit events inside *Use Case* if it does not make sense inside *an Entity*.

In modules that are implemented using *Facade* based approach, we emit events inside *Facade* methods:

```python
class PaymentsFacade:
    def __init__(
        self,
        config: PaymentsConfig,
        connection: Connection,
        event_bus: EventBus,
    ) → None:
        ...

    def capture(
        self, payment_uuid: UUID, customer_id: int
    ) → None:
        ...
        self._event_bus.post(
            PaymentCaptured(
                payment_uuid, customer_id
            )
        )
```

From **Events vs transactions vs side effects** we already know that until the transaction is committed, *Events* are unfounded. A straightforward way to circumvent the problem is to run event handling logic in a background job that will be triggered only after the transaction is committed. This is a quite useful design, but a bit generalized version is going to be used - from now on, such behavior shall be known as asynchronous event handling.

Let's start with a few fundamental rules:

- A class that emits *Events* stays utterly ignorant about the way how they are handled
- Subscriber decides how it handles events
  - asynchronously - in the background, outside the current transaction/*Unit Of Work* scope
  - synchronously - within the current transaction/*Unit of Work*
- An *Event Listener* is also ignorant whether it is called synchronously or asynchronously
- *Events* subscription and dispatching is powered by dependency injection
- **Main** module provides helper classes/functions to ensure smooth integration with a background job engine and transaction management/*Unit of Work*.

Leveraging Dependency Injection to handle Events dispatching can be a huge help. A concrete approach to implementing this requires a thorough knowledge of a chosen IoC Container. We have to be able to bind multiple handlers with the same *Event*.

With Python Injector, it can be achieved by using a combination of multibind and generics. Firstly, we need a couple of generics for synchronous and asynchronous event handling:

```
class Handler(Generic[T]):
    """Simple generic used to associate handlers with events using DI.
    e.g. Handler[AuctionEnded].
    """

    pass


class AsyncHandler(Generic[T]):
    """An async counterpart of Handler[Event]."""

    pass
```

Whenever we want to bind a new handler for a given Event, we can do this in an elegant way inside Dependency Injection module class:

```
binder.multibind(
    # a synchronous handler for AuctionEnded
    Handler[AuctionEnded],
    to=SYNCHRONOUS_HANDLER,
)
binder.multibind(
    # an asynchronous handler for AuctionEnded
    AsyncHandler[AuctionEnded],
    to=ASYNCHRONOUS_HANDLER,
)
```

A handler itself is not aware whether it is called synchronously or asynchronously. It is up to binding to determine when the handler will be triggered. In the example project, handlers are very simple classes to easily manage dependencies, for example:

```python
class BidderHasBeenOverbidHandler:
    @injector.inject
    def __init__(
        self, facade: CustomerRelationshipFacade
    ) → None:
        self._facade = facade

    def __call__(
        self, event: BidderHasBeenOverbid
    ) → None:
        self._facade.do_something(…)
```

Due to certain implementation details of Python's Injector, handlers have to be wrapped with simple providers. In the simplest words, providers are Injector's nomenclature for factories of handlers. The reason we have to have our custom logic is that synchronous handlers has to be build using Injector and called while asynchronous do not - it will happen *later*, specifically in the background task queue. The providers are respectively **`EventHandlerProvider`** and **`AsyncEventHandlerProvider`**. Example binding:

```python
binder.multibind(
    Handler[AuctionEnded],
    to=EventHandlerProvider[SomeEventHandler],
)
binder.multibind(
    AsyncHandler[AuctionEnded],
    to=AsyncEventHandlerProvider[
        SomeEventHandler
    ],
)
```

The way binding is done is fully transparent to events and handlers. On the bright side, this design is very flexible and allows to model everything the way we need.

As always, implementation details are a very specific thing to how this particular Event Bus was written using Injector:

```python
class InjectorEventBus(EventBus):
    """A simple Event Bus that leverages injector."""

    def __init__(
        self,
        injector: Injector,
        run_async_handler: RunAsyncHandler,  # 1
    ) -> None:
        self._injector = injector
        self._run_async_handler = (
            run_async_handler
        )

    def post(self, event: Event) -> None:
        try:
            handlers = self._injector.get(
                Handler[type(event)]
            )  # 2
        except UnsatisfiedRequirement:  # 3
            pass
        else:
            assert isinstance(handlers, list)
            for handler in handlers:
                handler(event)  # 4

        try:
            async_handlers = self._injector.get(
                AsyncHandler[type(event)]
            )
        except UnsatisfiedRequirement:
            pass
        else:
            assert isinstance(
                async_handlers, list
            )
            for async_handler in async_handlers:
                self._run_async_handler(
                    async_handler, event
                )  # 5
```

Interesting lines:

1. `InjectorEventBus` depends on `run_async_handler` - a function that will schedule a background job that will call given handler with the passed event. This is provided by

Main module. Implementation depends on a background jobs engine and transaction management. `run_async_handler` triggers job only after the transaction is committed, so it is safe.

2. Thanks to multibind, one is able to retrieve a list of synchronous handlers
3. It may happen that no one is subscribed to a given event, so we silence exception here
4. Synchronous handlers are called in place as if they were functions. Notice that by now handler classes are instantiated with injected dependencies. Handler class has to define special method `__call__` so it can be called like a function.

A solution in a different programming language should work analogously. Always check your IoC Container docs to come up with the most optimal solution.

IN-MODULE EVENT HANDLING

In a limited range of cases, it might be helpful to handle events emitted within the same module. Take **Payments** module for example - once payment card of a customer is charged (funds are reserved), we need to capture it (confirm payment), which will eventually result in funds being transferred to our bank account.

There is a possibility to do both steps at once, but depending on payment gateway it may not always be reliable. Moreover, there are scenarios (not in auctioning platform, though) when we specifically want to split the payment into two phases. For example, we reserve funds, then we take care about merchandise using API of some external provider and only if the latter succeeds, we capture funds.

Nonetheless, for the sake of example, let's assume that once payment card is charged, one wants to capture the funds in the background. The charge should be online if possible because, in case of incorrect card details, a customer is able to see immediately if something is wrong. Capture is unlikely to fail, so it can be safely executed in the background.

The facade of **Payments** does charge, then emits event:

```
class PaymentsFacade:
    def charge(
        self,
        payment_uuid: UUID,
        customer_id: int,
        token: str,
    ) -> None:
        payment = self._dao.get_payment(
            payment_uuid, customer_id
        )

        try:
            charge_id = self._api_consumer.charge(
                payment.amount, token
            )
        except PaymentFailedError:
            self._event_bus.post(
                PaymentFailed(
                    payment_uuid, customer_id
                )
            )
        else:
            ...  # code skipped
            self._event_bus.post(
                PaymentCharged(
                    payment_uuid, customer_id
                )
            )
```

Handler is thin, it merely triggers another *Facade*'s method - `capture`:

```
class PaymentChargedHandler:
    @injector.inject
    def __init__(
        self, facade: PaymentsFacade
    ) -> None:
        self._facade = facade

    def __call__(
        self, event: PaymentCharged
    ) -> None:
        self._facade.capture(
            event.payment_uuid, event.customer_id
        )
```

Dependency Injection module class deals with subscribing for event:

```
class Payments(injector.Module):
    def configure(
        self, binder: injector.Binder
    ) -> None:
        binder.multibind(
            AsyncHandler[PaymentCharged],
            to=AsyncEventHandlerProvider(
                PaymentChargedHandler
            ),
        )
```

Et voilà! Note there is nothing unusual in both handler and *Facade* that would indicate there is any asynchronous process in place. This is important only when a developer writes the binding.

CROSS-MODULES EVENT HANDLING - SIMPLE CASES

Handling events coming from different modules is not really different from handling them within the same module. A simple example is sending an e-mail once a bidder has been overbid. Bidding logic is in **Auctions** module, especially inside `Auction` *Entity*. That's where an event is emitted from:

```
class Auction:
    def place_bid(
        self, bidder_id: BidderId, amount: Money
    ) -> None:
        old_winner = (
            self.winners[0] if self.bids else None
        )
        if amount > self.current_price:
            ...
            if old_winner:
                self._record_event(
                    BidderHasBeenOverbid(
                        self.id,
                        old_winner,
                        amount,
                        self.title,
                    )
                )
```

All communication with the customer belongs to **Customer Relationship** module where an appropriate handler is defined:

```
class BidderHasBeenOverbidHandler:
    @injector.inject
    def __init__(
        self, facade: CustomerRelationshipFacade
    ) -> None:
        self._facade = facade

    def __call__(
        self, event: BidderHasBeenOverbid
    ) -> None:
        self._facade.send_email_about_overbid(
            event.bidder_id,
            event.new_price,
            event.auction_title,
        )
```

All it does, it calls a *Facade*'s method. Handlers within modules are not going to be anything more complex than simple glue code.

As for the subscription, there is really nothing new:

```
class CustomerRelationship(injector.Module):
    def configure(
        self, binder: injector.Binder
    ) -> None:
        binder.multibind(
            AsyncHandler[BidderHasBeenOverbid],
            to=AsyncEventHandlerProvider(
                BidderHasBeenOverbidHandler
            ),
        )
```

Although this way of integrating modules is straightforward, it is not a go-to solution for every case. To subscribe to an event coming from **Auctions** module, one has to import it into **Customer Relationship**. This makes these two coupled together. It is not too problematic here, because this is merely an illustration of the nature of **Customer Relationship** - it is meant to inform customers about many different things that happened within the system. Inevitably, this module will have to know about many others.

Beware of modeling business flows spanning several modules this way, though. When module A subscribes for an event from module B, and the latter subscribes for an event from module C, imagine how pleasant it is to read such a scattered code.

It is extremely difficult to understand such a process when a developer has to jump through different modules and recreate the whole flow in their heads. Luckily, there is an appropriate pattern for this situation - *Process Manager*.

CROSS-MODULES EVENT HANDLING - COMPLEX CASES

Splitting codebase into modules increases the distance between code fragments responsible for complex business scenarios. In isolation, none of *Use Cases* or *Facades*' methods are difficult. The one missing piece here is some pattern that makes processes spanning multiple modules explicit and easy to comprehend.

There are two possible approaches there - *Saga* or *Process Manager*.

First of all, both *Saga* and *Process Manager* subscribe to *Events* coming from different modules. They take on coordination and removes the need for dependencies between modules.

The main difference between these patterns is that *Saga* is stateless whereas *Process Manager* is stateful. Thus, *Process Manager* maintains internal state to know how to react. This feature has much in common with *State*[41] design pattern. On the other hand, *Saga* does not keep any data with it. It may require modules to provide additional querying capabilities. In the book, we will see a case study of a *Process Manager*.

Let's consider the following business scenario:

- once an auction has ended:
    - an e-mail is sent to the winner
    - a new payment is demanded from the winner
- once the payment has been collected:
    - another e-mail is sent to the winner
    - the item is prepared for shipping
- once the item has been shipped:
    - yet another e-mail is sent to the winner

[41] Bert Bates et al, *Head First Design Patterns*, Chapter 10. The State Pattern: The State of Things

First, let's see *Events* involved in the process:

- `AuctionEnded`
- `PaymentCaptured`
- `PackageShipped`

Note that each one of them is emitted from a different module. A code snippet that gives an idea about implementation:

```python
class PayingForWonItem:
    @method_dispatch
    def handle(
        self,
        event: Any,
        data: PayingForWonItemData,
    ) -> None:
        raise Exception(
            f"Unhandled event {event}"
        )

    @handle.register(AuctionEnded)  # 1
    def handle_auction_ended(
        self,
        event: AuctionEnded,
        data: PayingForWonItemData,
    ) -> None:
        assert data.state is None
        self._payments.start_new_payment( ... )
        self._customer_relationship.send_email_about_winning(
            ...
        )

    @handle.register(PaymentCaptured)  # 2
    def handle_payment_captured(
        self,
        event: PaymentCaptured,
        data: PayingForWonItemData,
    ) -> None:
        assert (
            data.state
            == State.PAYMENT_STARTED
        )
        self._customer_relationship.send_email_after_successful_payment(
            ...
        )
        self._shipping.register_new_package( ... )
    @handle.register(PackageShipped)  # 3
    def handle_package_shipped(
        self,
        event: PackageShipped,
        data: PayingForWonItemData,
    ) -> None:
        assert (
            data.state
            == State.SHIPPING_STARTED
        )
        self._customer_relationship.send_email_after_shipping(
            ...
        )
```

Interesting lines:

1. Upon **`AuctionEnded`** we call *Facades* of Payments and Customer Relationship
2. Upon **`PaymentCaptured`** we call *Facades* of Customer Relationship and Shipping
3. Upon **`PackageShipped`** we call *Facade* of Customer Relationship

Each handler inspects the internal state of *Process Manager* before executing any logic. In this way, a *Process Manager* enforces correctness just like *State Machine* does. These checks also bring idempotency - we are safe from reacting to the same events more than once. This might not sound like a big deal in a modular, yet still monolithic application, but is a highly desired feature in distributed systems.

Even though **`PayingForWonItem`** is a trivial example, its state is not a single variable (e.g. `State.PAYMENT_STARTED` or `State.SHIPPING_STARTED`). A *Process Manager* has to keep just enough information to autonomously make decisions using limited data enclosed with new *Events*. In practice, *Process Manager* copies all data it needs from *Events*. In other words, handlers change *Process Manager's* state, which can be quite a complex data structure:

```
@dataclass
class PayingForWonItemData:
    process_uuid: uuid.UUID
    state: Optional[State] = None
    winning_bid: Optional[Money] = None
    auction_title: Optional[str] = None
    auction_id: Optional[int] = None
    winner_id: Optional[int] = None
```

For state mutation example, see the second half of the handler of **`AuctionEnded`**:

```
class PayingForWonItem:
    @handle.register(AuctionEnded)
    def handle_auction_ended(
        self,
        event: AuctionEnded,
        data: PayingForWonItemData,
    ) -> None:
        ...
        data.state = State.PAYMENT_STARTED
        data.auction_title = event.auction_title
        data.winning_bid = event.winning_bid
        data.auction_id = event.auction_id
        data.winner_id = event.winner_id
```

Later, we will pass this data to appropriate *Facades* or *Use Cases*, e.g:

```
class PayingForWonItem:
    @handle.register(PaymentCaptured)
    def handle_payment_captured(
        self,
        event: PaymentCaptured,
        data: PayingForWonItemData,
    ) -> None:
        ...
        self._customer_relationship.send_email_after_successful_payment(
            data.winner_id,
            data.winning_bid,
            data.auction_title,
        )
```

An alternative to keeping all data in the state is to use appropriate *Queries* to fetch extra information. Then, the state will be kept to minimum. Obviously, this is a no-go when our system is eventually consistent, because *Queries* not necessarily will return the newest information.

Process Managers's statefulness makes it possible to implement more complex scenarios, like retrying payments up to N times - one has to record the required information in the internal state. Second, very common *Process Manager*'s application is timeouts handling. Say a winner has a day to pay. Otherwise, we will have to send them a reminding e-mail. Possibilities are endless. From the implementation standpoint, one has to add a `timeout_at` field to the state structure:

```
@dataclass
class PayingForWonItemData:
    ...
    timeout_at: Optional[datetime] = None
```

Another step is to implement a `timeout` method in *Process Manager* itself:

```
class PayingForWonItem:
    def timeout(
        self, data: PayingForWonItemData
    ) -> None:
        self._customer_relationship.send_payment_reminder(
            self._state.winner_id,
            self._state.auction_title,
        )
        self._data.state = (
            State.PAYMENT_OVERDUE_A_DAY
        )
```

With this implementation one still has to invoke *Process Manager* somehow from the outside. One approach would be to periodically query database for *Process Managers* that run out of time and then call their `timeout` method. A more fancy solution involves using some external scheduling service that will invoke our code at a specific moment in the future.

PROCESS MANAGER VERSUS PERSISTENCE

Naturally, a *Process Manager* does not live in the memory of the program all the time. Just like an *Entity*, it is fetched from the database when needed and saved afterwards. Thus, requires a persistence mechanism. There are two problems to address:

- how to keep `PayingForWonItemData` in our database?
- how to find data of desired one if there are multiple *Process Managers* of the same type in progress?

A *good-enough* solution for the first dilemma is to keep state data structure as JSON. This approach is lean, but has a trade-off - it will work only if *Process Manager's* data structure is serializable and deserializable. This should be left up to 3rd party libraries if possible. For Java, there is excellent Jackson ObjectMapper that is able to handle POJOs. After all, it is possible to write a single, generic repository for all *Process Managers*. It will have to be able to translate DTOs into JSON to be later stored in the database. For example, the following object:

```
ExampleData(
    process_uuid=UUID(
        "9fc15305-2a0f-41ed-8c1c-eafa2416ee75"
    ),
    name="Example",
    counter=0,
    timeout_at=datetime.datetime(
        2019, 9, 29, 20, 20, 11, 426930
    ),
)
```

...could be serialized into following JSON:

```
{
    "name": "Example",
    "counter": 0,
    "timeout_at": "2019-09-29T20:20:11.426930"
}
```

...to be later inserted into PostgreSQL table created with:

```
CREATE TABLE process_manager_data (
    uuid UUID PRIMARY KEY,
    data jsonb NOT NULL
);
```

Now, there are two possible approaches to keeping `timeout_at`. One can keep it inside `data`, but it will require a partial JSON index for acceptable performance or create another column just for the timeout. Then, using it should be much simpler. By the way, *Process Manager's* data is clearly belongs to wide category of documents, which makes document databases (e.g. MongoDB) really handy here. If it is the one you already have in your project, then keeping *Process Managers'* data inside shouldn't give you much of a headache.

A second dilemma (how to find data of the desired one if there are multiple *Process Managers* of the same type in progress?) is a bit more complex one. What has not been mentioned yet is that between *Events* and a *Process Manager* there is a *Process Manager Handler*. The latter is a class that will be responsible for creating *Process Manager* instance if it just handles a starting *Event* or fetching it using the repository.

```
class PayingForWonItemHandler:
    @injector.inject
    def __init__(
        self,
        process_manager: PayingForWonItem,
        repo: ProcessManagerDataRepo,
    ) -> None:
        self._process_manager = process_manager
        self._repo = repo

    @method_dispatch
    def __call__(self, event: Event) -> None:  # 1
        raise NotImplementedError

    @__call__.register(AuctionEnded)
    def handle_beginning(
        self, event: AuctionEnded
    ) -> None:  # 2
        data = PayingForWonItemData(
            process_uuid=uuid.uuid4()
        )
        self._run_process_manager(data, event)

    @__call__.register(PaymentCaptured)
    def handle_payment_captured(
        self, event: PaymentCaptured
    ) -> None:  # 3
        data = self._repo.get(
            event.payment_uuid,
            PayingForWonItemData,
        )
        self._run_process_manager(data, event)

    def _run_process_manager(
        data: PayingForWonItemData,
        event: Event,
    ) -> None:  # 4
        self._process_manager.handle(event, data)
        self._repo.save(data.process_uuid, data)
```

Interesting lines:

1. catch-all handler method to be invoked in case of unhandled *Event*.
2. `AuctionEnded` is the first event in a lifetime of each `PayingForWonItem`. Hence, it always creates a new instance of `PayingForWonItemData`

3. `PaymentCaptured` always happens after creation of a *Process Manager*, so `PayingForWonItem` is loaded from the repository

4. `_run_process_manager` takes care of repeatable parts of the events handling

Now one can take care of the second dilemma.

There are (at least) two ways out:

- guarantee that all events will come back with the same UUID which one will just use for the primary key - `uuid`

- write handlers and repo in such a way one is able to use different UUIDs and query `process_manager_data` by various nested fields.

The first solution is straightforward to implement and results in simpler code but will affect involved modules. In practice:

```
# Process Manager Handler
@__call__.register(AuctionEnded)
def handle_beginning(
    self,
    event: AuctionEnded,
    data: PayingForWonItemData,
) → None:
    data = PayingForWonItemData(
        process_uuid=uuid.uuid4()
    )  # 1


# Process Manager
@handle.register(AuctionEnded)
def handle_auction_ended(
    self,
    event: AuctionEnded,
    data: PayingForWonItemData,
) → None:
    self._payments.start_new_payment(
        self._data.process_uuid, ...
    )  # 2
```

```
# Process Manager Handler
@__call__.register(
    PaymentCaptured
)
def handle_payment_captured(
    self,
    event: PaymentCaptured,
    data: PayingForWonItemData,
) → None:
    data = self._repo.get(
        event.payment_uuid,
        PayingForWonItemData,
    )  # 3
```

Interesting lines:

1. First event is handled - **`AuctionEnded`**. One just generates a new UUID
2. A *Process Manager* passes its UUID to be used as UUID of newly created payment
3. When **`PaymentCaptured`** is emitted one can be sure its the same UUID that *Process Manager* has

A side effect of this approach is that one will get a correlation ID - the same UUID will be passed and reused in different modules, making it easier to track what is actually going on. On the downside, this solution is not always possible to implement, especially when tracking multiple objects of the same type is required. It is not possible to reuse the same UUID then.

The second approach does not have specific requirements but is a bit more complex. However, this extra complexity is contained in *Process Manager Handler*:

```
class PayingForWonItemHandler:
    @__call__.register(AuctionEnded)
    def handle_beginning(
        self, event: AuctionEnded
    ) → None:  # 1
        data = PayingForWonItemData(
            process_uuid=uuid.uuid4()
        )
        self._run_process_manager(data, event)
    @__call__.register(PaymentCaptured)
    def handle_payment_captured(
        self, event: PaymentCaptured
    ) -> None:
        data = self._repo.get_by_field(  # 2
            PayingForWonItemData,
            payment_uuid=event.payment_uuid,
        )
        self._run_process_manager(data, event)
```

Interesting lines:

1. There are no changes in the first method

2. A method **handle_payment_captured** is where things get interesting. The repository has now a new method - `get_by_field` - that accepts an arbitrary keyword argument to use it in a query that is executed against a database

Note these solutions work only if there is exactly one event that can start *Process Manager*. If there are more, one has to find another way to guarantee continuity of a *Process Manager*.

PROCESS MANAGER VERSUS RACE CONDITIONS

Process Managers are stateful creatures. They have their state persisted between invocations. Events they handle can be emitted at the same time. Think of a person paying in the last moment, when a datetime specified in timeout_at passes. In other words, *Process Managers* are vulnerable to race conditions and one has to protect them.

In a previous chapter, a technique called optimistic locking was used. The method amounts to checking if someone else has changed Entity since the last time we fetched it. If so, then one has to retry the entire operation. It works great with a side-effect-free Entity, but may not be the best fit for *Process Managers*. Pessimistic locking would do better.

In its principles, it is even simpler than the former solution. Before one calls any handling logic of a *Process Manager*, a lock has to be explicitly acquired. When we finish with the processing, the lock is released. If we fail to acquire it in the first place - we abort. Sometimes we may also want to wait for some time until the lock is released or simply retry later. That is a theory, but for the sake of the example, let's assume if the lock cannot be acquired, we are ok with aborting. Consider the exaggerated example mentioned at the beginning of the section - a winner paying in the very last moment when a *Process Manager* times out. Let's assume that when the timeout occurs, we no longer want the money of a winner. They are late, end of the story. In other words, when two *Events* that can end *Process*

Manager are competing, it makes absolutely no sense to retry or wait. In other cases, like counting someone's achievements, it will make sense to retry or wait.

Implementation-wise, *Process Manager Handler* can be burdened with the logic of acquiring lock:

```python
class PayingForWonItemHandler:
    LOCK_TIMEOUT = 30

    @injector.inject
    def __init__(
        self,
        *omitted_args,
        lock_factory: LockFactory
    ) -> None:  # 1
        ...
        self._lock_factory = lock_factory

    def _run_process_manager(
        self,
        lock_name: str,
        data: PayingForWonItemData,
        event: Event,
    ) -> None:
        lock = self._lock_factory(
            lock_name, self.LOCK_TIMEOUT
        )  # 2

        with lock:  # 3
            self._process_manager.handle(event, data)
            self._repo.save(data.process_uuid, data)
```

Interesting lines:

1. A concrete `Lock` implementation is injected into *Process Manager Handler*
2. It is a good idea to always acquire locks with some timeout after which they will be automatically released if something goes wrong. This improves the resilience of the system
3. Pythonic context manager abstracts away from us acquiring and releasing the lock

For the sake of completeness of the example, Redis will be used for locking. An example implementation of a lock context manager may look as follows:

```
class RedisLock(Lock):
    LOCK_VALUE = "LOCKED"  # 1

    def __init__(
        self,
        redis: StrictRedis,
        name: str,
        timeout: int = 30,
    ) -> None:
        self._redis = redis
        self._lock_name = name
        self._timeout = timeout

    def __enter__(self) -> None:  # 2
        if not self._redis.set(
            self._lock_name,
            self.LOCK_VALUE,
            nx=True,
            ex=self._timeout,
        ):
            raise AlreadyLocked

    def __exit__(  # 3
        self,
        exc_type: Optional[Type[BaseException]],
        exc_val: Optional[BaseException],
        exc_tb: Optional[TracebackType],
    ) -> bool:
        if exc_type != AlreadyLocked:
            self._redis.delete(self._lock_name)
        return False
```

Interesting lines:

1. To do locking on Redis using String type, one has to put something inside. It is rather irrelevant. For some debugging purposes, one might want to put timestamp there

2. Method `__enter__` is executed before handling logic of a *Process Manager*. It is equivalent to running `SET {lock_name} “LOCKED” NX EX 30` in a Redis console

`__exit__` runs after block under with ends. Its goal is to release lock only if we acquired it.

CHAPTER SUMMARY

This chapter explained a vital part of architecting a system - packaging code by feature or vertical slicing. We learned that not every module has to follow the Clean Architecture if only one would benefit from it. In simpler cases, we may resort to *Facade*-based modules that interact with databases or external services in a more direct way, without additional layers of abstractions.

Then, different types of relations between modules were discussed to show ways of integrating. Among the presented techniques, there were:

- direct calls of one module from another,
- defining a *Port* in one module and letting another module provide an *Adapter*,

- integrations using *Domain Events*
    - simple, when one module subscribes to another module's *events*
    - complex, when *Process Manager* pattern was introduced to orchestrate processes spanning multiple modules

The preferred way to integrate modules should be using *Domain Events* unless a synchronous call is required.

TESTING

TESTING STRATEGY AND FEATURE FLAVORS

At the beginning of any project, delivering business value without causing regression is trivial. However, with adding more and more features, complexity grows. So does risk of breaking something up. Adherence to practices of good OOP design, like Open-Closed Principle or loose coupling lessens danger in case of adding new features, so it does not yet gives enough confidence. One needs more than a coincidence to deliver regularly. Luckily, the remedy is already known - automated tests. Here's what Adrian Sutton says about relying on test suite at LMAX[42]:

> *We've obviously invested a lot of time in building up all those tests but they're invaluable in the way they free us to try daring things and make sweeping changes, confident that if anything is broken it will be caught. We're happy to radically restructure components (…).*

If our test suite is to provide a complete safety-net, the architecture has to facilitate testing. Otherwise, we find ourselves hacking around the untestable monster. The Clean Architecture is definitely a powerful ally when it comes to writing testable code. Up to this moment, the focus was on testing smaller pieces, like *Entities* or *Use Cases*. It is high time we saw the bigger picture and wonder about testing whole modules as well as an entire application.

THE TEST PYRAMID - A MYTH OR THE ONLY RIGHT THING TO DO?

Chances are you heard about the concept of allegedly perfect tests distribution - The Test Automation Pyramid. This concept is attributed to Mike Cohn and was first published in a written form in 2009[43]:

The original *Test Automation Pyramid* strives to maximize return on investment by using different kinds of tests. Fast and cheap unit tests are not enough to be confident that

[42] Adrian Sutton, *Making End-To-End Tests Work* https://www.symphonious.net/2015/04/30/making-end-to-end-tests-work/

[43] Mike Cohn, *Succeeding with Agile: Software Development Using Scrum*

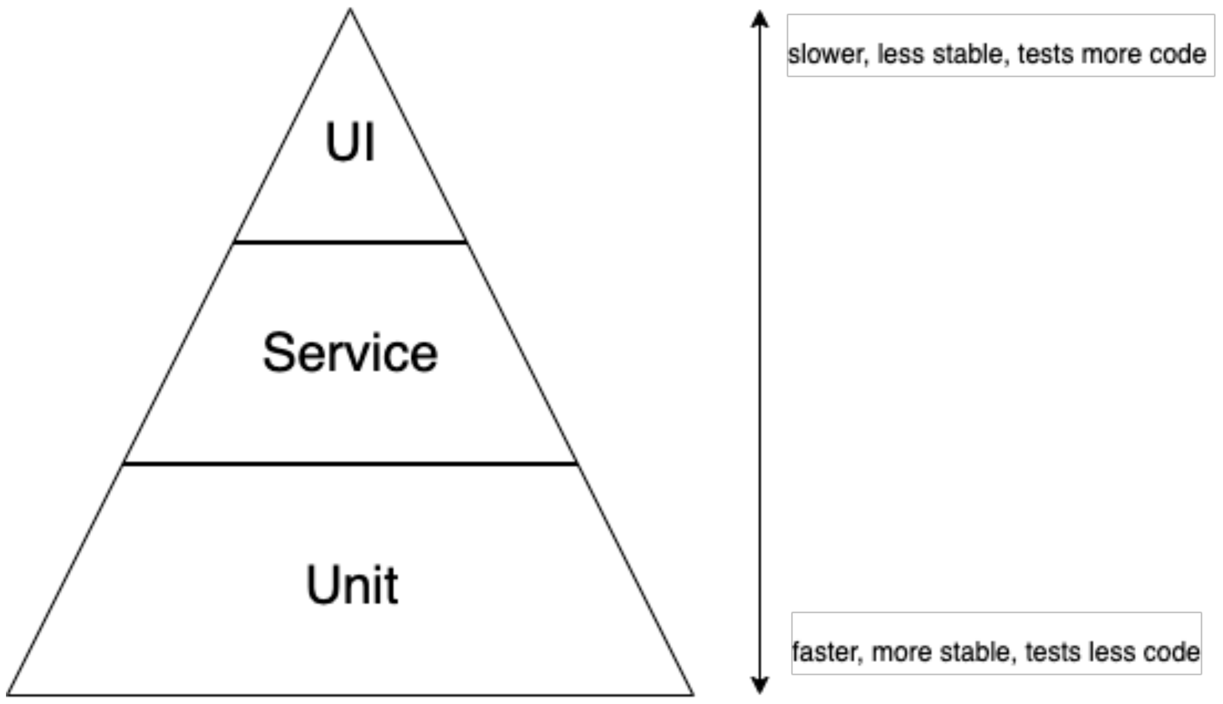

UI
Service
Unit
slower, less stable, tests more code
faster, more stable, tests less code

Figure 8.1 Test Automation Pyramid assumes that unit tests are the most numerous of all. They are supplemented with fewer Service tests and significantly fewer UI tests. The fundamental premise of Test Automation Pyramid is that unit tests are considerably faster and cheaper than others

program *works* when a user interacts with it, so they are supplemented with a lesser amount of higher-level service tests and even fewer tests that run atop user interface. Please note that this model does not take into account manual testing - it is Test **Automation** Pyramid after all. However, manual testing is still necessary - especially exploratory testing. A Test Pyramid model that takes it into account looks like this:

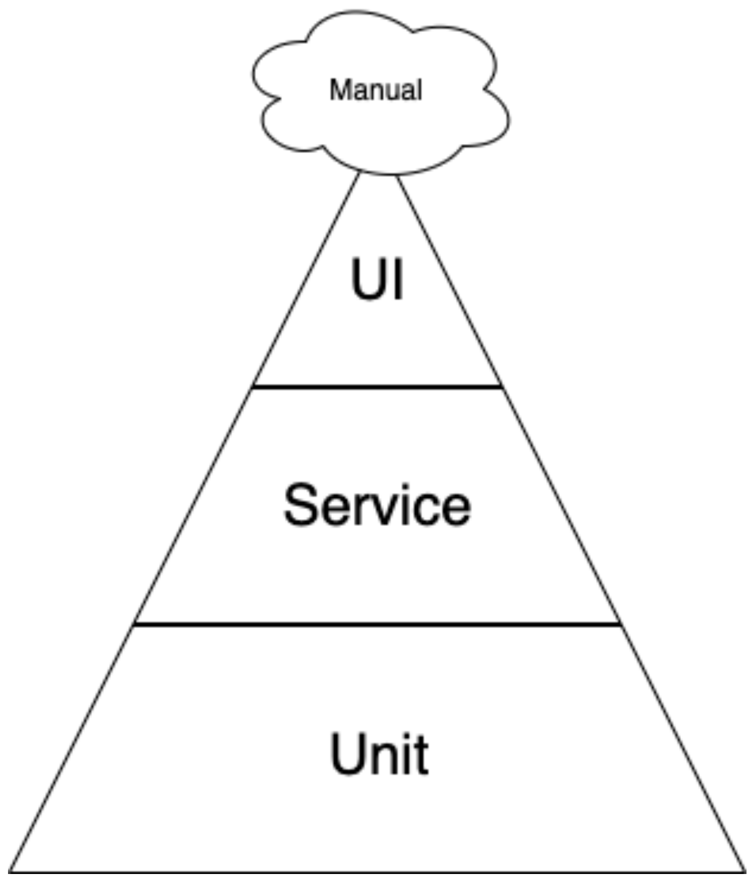


Figure 8.2 Test Pyramid with manual testing included

On the other hand, an anti-pattern when the majority of testing effort is done manually is called ice cream cone:

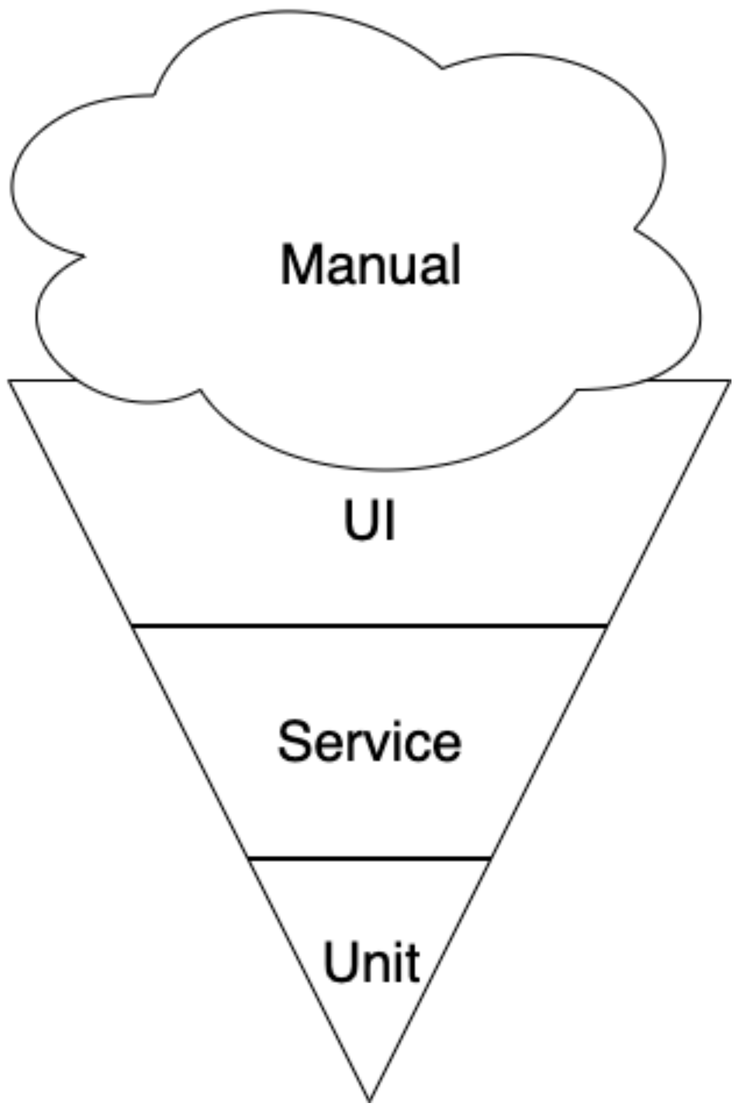


Figure 8.3 Ice Cream Cone anti pattern

TESTS TYPES TAXONOMY

Mike Cohn's Test Automation Pyramid consists of three types of tests - **unit**, **service** and **UI**.

Unit tests and **UI** should look familiar, but what are mysterious **service tests**? According to the author[44]:

In the way I'm using it, a service is something the application does in response to some input or set of inputs. Our example calculator involves two services: multiply and divide. Service-level testing is about testing the services of an application separately from its user interface. So instead of running a dozen or so multiplication test cases through the calculator's user interface, we instead perform those tests at the service level.

[44] Mike Cohn, *The Forgotten Layer of the Test Automation Pyramid*, https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

If we were to apply this to the Clean Architecture, our *Use Cases* (and *Queries*) would be our *Services*. Test Automation Pyramid can be definitely praised for its consistency - all types of tests involved refer to certain levels in a tested application, whether it is a user interface (**UI**), *Use Cases/Queries* (**Service**) or classes/functions (**Unit**).

Other commonly used names for tests types depending on their level are **system tests**, **end-to-end tests** or **integration tests**. Although it is generally agreed upon what **system tests** and **end-to-end tests** mean (just go through all the layers), **integration tests** are defined in different, often contradictory ways. For example, ISTQB glossary[45] says it is *A test level that focuses on interactions between components or systems*. Another definition assumes its a way of testing a few units together as a bigger whole - a component. Why not name it component testing, then? Oh, wait - there is such a term already. Anyway - the last known interpretation is to write tests with scope equal to unit tests, but do not stub database and still call them **integration tests**. Let's assume that in this book from now on **integration tests** stand for verifying the correctness of code responsible for communication with an external system or provider. In terms of the Clean Architecture think of a pair of *Port* and *Adapter* for a payment provider. Tests for the *Adapter* that require communication with the external system would be called **integration tests**. They are not meant to test the external provider itself, though - only *Adapter*.

I am going to make use of yet another type - **API tests**. They are placed slightly higher than Service tests and exercise web framework that is used. A characteristic feature of such tests would be calling a specific URL and make assertions about the response.

There is yet another attempt to classify tests that uses their role rather than how they are supposed to run. These types are **functional tests** and **acceptance tests**. Difference between **functional** and **acceptance tests** is subtle and some people openly treat as if they were identical. ISTQB Glossary states that **functional testing** is to evaluate if a system satisfies functional requirements while **acceptance testing** aims to determine whether to accept the system. From a certain perspective, these are indeed the same - both testing types check if the system works from a business perspective. The only difference in meaning I was able to find is that **acceptance tests** focus on real-life scenarios, while **functional testing** can also exercise the system in a more thorough fashion.

From now on, I am going to stick to **acceptance testing** name when referring to verification if the system does what it is supposed to do from a perspective of stakeholders and users.

[45] ISTQB Glossary https://glossary.istqb.org/

Test distributions as in the original Test Automation Pyramid is very desirable but rarely seen. The reason is twofold, one: the code is untestable, so it does not enable quality unit testing and two: benefits from unit testing the code solving a particular problem are negligible. As for the code testability, we know the answer already - refactor your code so it can leverage the Clean Architecture or at least some testability boosting techniques, like inversion of control. The second part is much more interesting. You read that right - in some situations, unit tests would not be worth the effort. A typical symptom is that all test cases can be covered by a very limited set of higher-level tests. At best, we would be able to duplicate only some checks with unit tests, but bring nothing new to the table, even with *hyper-testable* code.

To better illustrate the issue, let's see through the lens of a user interface and three differently flavoured features:

1. a database browser - doing direct changes to data stored in a database, repeatable results,
2. proxy to other systems - making lots of external calls and doing very little with results apart from storing or just presenting them,
3. a "deep" system - many preconditions involved, numerous alternative scenarios to handle.

Each one of these examples is an extremity, but that's all possible features flavours. A real-life application will be a combination of all. However, the prevalent flavour will determine the most appropriate testing strategy.

To conclude, the original Test Automation Pyramid is not a silver bullet. One may need to take a different approach, best suited to the project of theirs. Please note that with a modularized application, we can apply a test strategy tailored for each module. That remains true if eventually, we decide to split into microservices. There is no reason to impose *the only right* approach throughout the whole project, especially given that no one-size-fits-all solution exists.

Now we are going to walk through each feature flavour in detail.

HOW TO TEST A DATABASE BROWSER?

Such systems are sometimes referred to as CRUDs. One interprets all business requirements as database operations - **C**reate (INSERT), **R**ead (SELECT), **U**pdate (UPDATE) and **D**elete (DELETE). Let's consider an example.

A project manager says:

We are going to build a system for our Pizza Fridays. Each week, all participants will order a specific pizza by entering its name from the menu into a form. There is going to be a person called coordinator who will be responsible for making a call to a restaurant or ordering online. Once order is placed at the restaurant, the coordinator will clean the internal orders list.

Software developer interprets this as:

Participant adds order to the internal list - INSERT a row to the database with pizza name

Coordinator displays contents of the internal list - SELECT all rows from the database

Coordinator empties list after ordering - DELETE rows, alternatively use UPDATE for soft delete

Note that it is the simplest approach possible. For such an application to be used beyond a single office, one would have to consider many more scenarios, like updating participant's order, withdrawing from Pizza Friday etc. The problem is CRUD would then cease to work for this example.

"Pure" CRUD is an extreme case when there is an exactly one possible path for a given interaction. No alternative scenarios. Doing the same action repetitively has no (or limited) consequences. If one would apply the Clean Architecture for such a problem and it would be their first experience with it, they would think it is a complete waste of time and overengineered mumbo-jumbo.

Think of a standard flow:

1. repack request data into Input DTO
2. pass it to *Use Case*
    1. *Use Case* on the first line gets *Entity*
    2. on the second, *Use Case* calls *Entity's* method

3. *Entity* is saved on the third line.

And what does an *Entity*'s method does? Sets some fields. No *if*-statements, no logical branches, no special cases. Oh, and it is tempting to name *Use Case* like `Updating[Entity Name]` since one thinks through the lens of database rows. Of course, one can write a unit test for *Entity* or *Use Case* in such a situation. The singular form was used on purpose - there would be only one scenario to check after all. There is no added value by such tests - if we also test on a higher level (e.g. via API), unit tests would be checking exactly the same paths of code and would be giving us less confidence since they exercise less code at once. So when we face such a problem, we can safely skip unit testing for most of the time and our test automation "pyramid" can look like this:

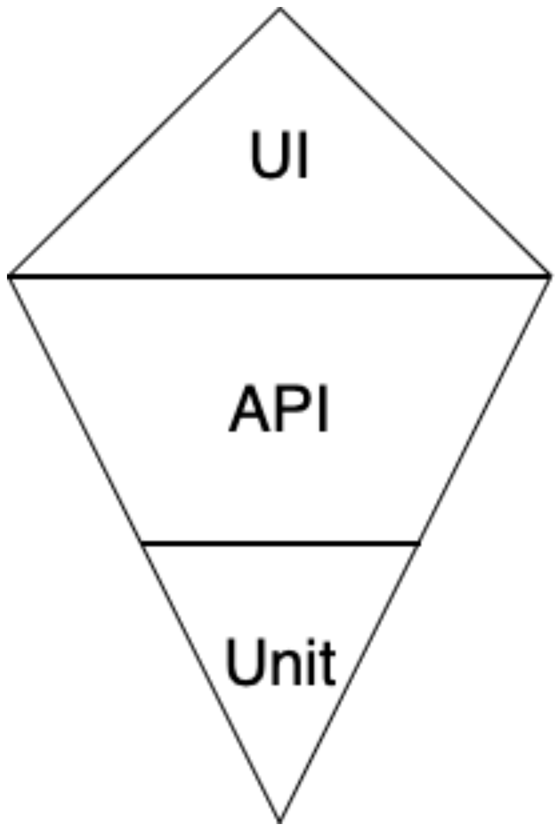


Figure 8.4 Test Automation Kite, tailored for CRUD application

In such a project, unit tests prove themself to be valuable only when we deal with some calculator or a validator. All functionalities are checked with API tests with an assumption that a minimal amount of tested scenarios suffices. One test for a positive scenario, one for negative scenario and potentially one for checking security (if we get authentication right) should give us enough confidence. Few extra UI tests should complement the strategy.

Beware of incorrectly discovered flavour. If an application would benefit from the Clean Architecture, but one implements it as if it was a CRUD then testing will be much less effective since more and more scenarios would have to be checked via API tests. Bear in mind the issue is not only about the speed of tests, but also their stability and maintainability. If your only dependency is a database, entire test suite finishes in a relatively short time (e.g. less than 10 minutes), and the whole setup comes down to putting up to few objects in the database, it is still fine.

HOW TO TEST A PROXY TO OTHER SYSTEMS?

Disclaimer: This part is written in mind with scarce integrations with 3rd party providers that have public APIs. It has little to no application to microservices. For the latter, there is a note in the end.

A project (or a part of it - *Adapter*) might be just a proxy over another service. Even if one provides another user interface, still the majority of business logic is being executed somewhere else. The role of our system is to translate requests from our GUI and handle responses. Another way to tell if we have this kind of problem is to ask what would happen if the 3rd party service disappeared? If the answer is “our system is not going to work at all” then you have a proxy. A typical example is a cheap flights search that checks a few providers. If it is not able to fetch available flights from elsewhere, it is useless and provides no service. Of course, it could be caching results for some time to provide minimal service for the most popular flight routes, but it is only a fraction of the original value.

In this case, once again, unit tests do not prove to be the most reliable source of truth whether our software works. This mostly depends on the external provider and not our isolated units. Eventually, we have to resort to integration tests. An absolute must is to test at least once code using each endpoint (or a method if we speak about RPC or SOAP). There are two more things we have to address - dependencies between methods and data validation.

One cannot use `Refund` endpoint if a `Charge` endpoint not been used beforehand. Obviously we could test how refund behaves if we give it a non-existing id, but that brings little to no value for an API client like us. In this case, one can just naively issue additional requests as needed to check each API endpoint individually. Second solution would be to write tests that will be checking real-life scenarios by calling several methods at once, e.g. creating charge (1. call), then checking its details (2. call) to finally refund (3. call). Everything in one test.

```python
@pytest.mark.super_cool_payment_provider
def test_charge_then_capture(
    api_consumer: ApiConsumer,
    source: str,
    api_key: str,
) -> None:
    # first, we call our code to charge & capture 15 dollars
    charge_id = api_consumer.charge(
        get_dollars("15.00"), source
    )
    api_consumer.capture(charge_id)

    # ...then we use Payment Provider's API to verify if charge was successful
    response = requests.get(
        f"{Request.url}/v1/charges/{charge_id}",
        auth=(api_key, ""),
    )
    capture_json = response.json()
    assert capture_json["amount"] == 1500  # cents
    assert capture_json["currency"] == "usd"
    assert capture_json["captured"]
```

Data validation can be mostly dealt with by using *Value Objects* as Facade's arguments and return values. Validation rules can be then easily enforced due to the nature of Value Objects. Number of integration tests should be very limited and they should be written with Robustness Principle (AKA Postel's Law) in mind[46]:

> *Be conservative in what you do, be liberal in what you accept from others (often reworded as "Be conservative in what you send, be liberal in what you accept")*

Simply saying, we write tests (and code!) checking only for things we rely on. For example, failing a test if there is an unexpected field in JSON violates Robustness Principle. Such a change in API is considered to be backwards-compatible and hence should not break our integration.

Another thing to consider is how stable these tests would be. If they will be failing randomly due to a fallible provider, one would neglect running them. There are little things more irritating to a software developer than a test suite failing locally without their fault. The same goes for a CI pipeline. Solutions may vary from excluding these tests from local runs to adding retries to tests or implementing them in the code under test itself.

[46] Robustness Principle https://en.wikipedia.org/wiki/Robustness_principle

Last, but not least - many external providers have test APIs we can leverage. That is extremely helpful when calling actual API has serious implications, like transferring money or booking a flight. It would be nice not to have to manually cancel bookings after few developers run tests on their machines. Naturally, we have to be certain that the test API is 100% compatible with the one we use in production.

What if I have tens or hundreds of integrations (e.g. in microservices?)

The advice given above is not really applicable in a system consisting of microservices. The very idea of integration testing is contradictory to the way how *done-right* microservices systems are developed and deployed. Testing services in isolation is not solving the problem either - it is like relying on unit tests only. Luckily, a proper solution has been discovered rediscovered - *Consumer-Driven Contract Testing*[47]. One of the most popular tools for facilitating this technique is Pact[48]. This topic is exciting, though it remains out of scope for this book. An interested reader will surely be able to study the *Consumer-Driven Contract Testing* further on their own.

HOW TO TEST A DEEP SYSTEM?

Once we realise we are dealing with an essentially complex problem, the Clean Architecture or more sophisticated techniques (like DDD) come in handy. Complexity means many potential scenarios, countless edge cases and lots of factors affecting the outcome of actions undertaken by users.

First and foremost, only appropriate code structure and architecture can enable us to use an effective testing strategy. Effective does not mean straightforward - it has to combine at least a few types of automated tests to earn its name. Enterprise-wide business logic (*Entities*) should be unit tested. Application business rules (*Use Cases*) fall under service testing. Then we have *Repositories* and *Adapters* which will benefit most from the integration testing just like it was described in the previous section - *How to test a proxy to other systems?*. Then there is a web framework and potentially a User Interface, so we should also have a handful of API & UI tests.

Then a testing automation pyramid will closely resemble the exemplary one from the beginning of this chapter.

[47] Ian Robinson, *Consumer-Driven Contract Testing* https://martinfowler.com/articles/consumerDrivenContracts.html

[48] Pact https://docs.pact.io/

BLACK- AND WHITE-BOX TESTING

Automated testing can be attributed another feature - how much a test knows about code that is being tested. Using the same terminology as for manual testing, we can think about black-box and white-box testing. Black-box testing means a tester has no idea about internals while white-box is exactly opposite - a tester knows and sees everything, including implementation. Black box testing validates the requirements and specifications, whereas white box testing validates the code.

Unit tests can be implemented in both ways. One extreme is an orthodox act of white-box, when the unit under test has no secrets from the test. In the context of unit-testing, this is rarely a good idea. Such approach duplicates implementation in testing, making it impossible to evolve our tests in such a way that *as the tests get more specific, the code gets more generic*. Moreover, tests are tightly coupled to the implementation, which makes them fail each time the latter changes. Unit-testing in such a way, resembles an act of pouring concrete - making sure nobody can move a thing. In white-box testing of a class, we call its public methods but then often verify the result by examining private fields or using mocks of types used internally.

Example:

```python
# piece of Auction class implementation
class Auction:
    def __init__(
        self,
        id: AuctionId,
        *omitted_args,
        ended: bool
    ) -> None:
        ...
        self._ended = ended
```

```
# Bad way of testing
def test_Auction_Ending_ChangesEndedFlag(
    yesterday: datetime,
) -> None:
    auction = AuctionFactory(ends_at=yesterday)

    auction.end_auction()

    assert auction._ended  # 1


def test_Auction_Ending_ChangesEndedFlag(
    yesterday: datetime,
) -> None:
    auction = AuctionFactory(ends_at=yesterday)
    auction._ended = True  # 2

    with pytest.raises(BidOnEndedAuction):
        auction.place_bid(
            bidder_id=1,
            amount=get_dollars("19.99"),
        )


def test_EndedAuction_Ending_RaisesException(
    yesterday: datetime,
) -> None:
    auction = AuctionFactory(ends_at=yesterday)
    auction._ended = True  # 2

    with pytest.raises(AuctionAlreadyEnded):
        auction.end_auction()
```

Note how much *inappropriate intimacy*[49] is there. At a glance, the first test looks pretty fine… Until you notice its title and assertion. Not to mention accessing *pseudo*-private field at (1). Second and third tests look better in terms of assert, but putting an auction in the expected state (2) by manipulating its private field is just plain wrong.

Now, imagine that for tracking purposes `_ended` flag is replaced with `_ended_at` that keeps `datetime` instance indicating moment of calling `end_auction` or `None` if the method has not been called. All these three tests would break, although most probably everything would work in Service or higher-level tests (and in production). Unit-tests are a useless burden, aren't they? Well, they are if written in a way that violates encapsulation.

49 https://refactoring.guru/smells/inappropriate-intimacy

A better approach is to respect class' privacy and test it in a black-box way. The rule is simple - we are not allowed to ask the class for its secrets and can touch only its public methods & properties.

```python
def test_EndedAuction_PlacingBid_RaisesException(
    yesterday: datetime,
) → None:
    auction = AuctionFactory(ends_at=yesterday)
    auction.end_auction()  # 1

    with pytest.raises(BidOnEndedAuction):
        auction.place_bid(
            bidder_id=1,
            amount=get_dollars("19.99"),
        )


def test_EndedAuction_Ending_RaisesException(
    yesterday: datetime,
) → None:
    auction = AuctionFactory(ends_at=yesterday)
    auction.end_auction()  # 1

    with pytest.raises(AuctionAlreadyEnded):
        auction.end_auction()
```

The most noticeable change is that we no longer manipulating a private field, but tell auction to end (1). Then we test class' behavior, by checking if an ended auction prevents us from placing new bids (test 1) and ending it again. Notice that we reduced the number of tests by one and still have the same coverage. Moreover, we kept an ability to rearrange implementation details as long as class' behavior remains unchanged.

I write about white- and black-box testing, but there is always a grey area in between.

"All problems in computer science can be solved by another level of indirection" - David Wheeler

In the above examples, a `AuctionFactory` class was used as a helper to create `Auction` instances. This is an example of a builder, a quite handy pattern to use in tests. That being said, in Python, there is an excellent library `factory_boy` that facilitates writing builders in a declarative way. To get a builder, we write a sort of specification with default values for an object being built.

```python
class AuctionFactory(factory.Factory):
    class Meta:
        model = Auction

    id = factory.Sequence(lambda n: n)
    bids = factory.List([])
    title = factory.Faker("name")
    starting_price = get_dollars("10.00")
    ends_at = factory.Faker(
        "future_datetime", end_date="+7d"
    )
    ended = False
```

In its basic form, all fields declared on `AuctionFactory` are used to populate `Auction.__init__` arguments. A rough equivalent, written in a more imperative way, looks as follows:

```python
def create_auction(
    id: Optional[AuctionId] = None,
    bids: Optional[List[Bid]] = None,
    title: Optional[str] = None,
    starting_price: Money = get_dollars("10.00"),
    ends_at: datetime = datetime.now()
    + timedelta(days=7),
    ended: bool = False,
) -> Auction:
    if id is None:
        # object we can iterate to get integers like 1, 2, 3...
        id = auction_sequence.next()
    if bids is None:
        bids = []
    if title is None:
        title = faker.name()

    return Auction(
        id,
        title,
        starting_price,
        bids,
        ends_at,
        ended,
    )
```

Now, if we were to change `_ended` field to `_ended_at` and still let our tests use it via `create_auction` or `AuctionFactory` with `ended` parameter, we simply have to implement logic in a builder:

```python
# factory_boy-based builder
class AuctionFactory(factory.Factory):
    class Meta:
        model = Auction

    class Params:
        ended = False

    id = factory.Sequence(lambda n: n)
    ...
    ended_at = factory.LazyAttribute(
        lambda o: datetime.now()
        - timedelta(days=1)
        if o.ended
        else None
    )


# function-based builder
def create_auction(
    *omitted_args, ended: bool = False,
) -> Auction:
    ...
    if not ended:
        ended_at = None
    else:
        ended_at = datetime.now() - timedelta(
            days=1
        )

    return Auction(
        id,
        title,
        starting_price,
        bids,
        ends_at,
        ended_at,
    )
```

Relying on builders in tests is a pretty good idea since they can abstract many implementation details and significantly limit the number of places where we need change. They also simplify objects creation, providing defaults values for all fields we decide not to specify ourselves. Of course, they will always have to follow changes in `Auction` implementation because they are tightly coupled to it.

All in all, relying on public methods during testing is generally a very good idea. Leaking knowledge about implementation will cost you in the longer term. Remember the analogy of pouring concrete. Implementation-coupled testing makes sure nobody can move a thing in the future.

STATE AND INTERACTION ORIENTED TESTING

INTRODUCTION

Another angle tests can be looked at is whether they are more state or interaction oriented. These two approaches look pretty similar during exercising system under test (Act or When step) but have a different approach to verification phase (Assert or Then step). State-based testing inspects the internal state of the system under test while interaction-based testing makes checks if the system under test called used mocked dependencies as expected.

```python
# interaction-based testing example
# Checks if a mocked method of EventBus was called with expected argument
@pytest.mark.usefixtures("transaction")
def test_AuctionsRepo_UponSavingAuction_PostsPendingEventsViaEventBus(
    connection: Connection,
    auction_with_pending_event: Auction,
    pending_event: Event,
    event_bus_mock: Mock,
) -> None:
    repo = SqlAlchemyAuctionsRepo(
        connection, event_bus_mock
    )

    repo.save(auction_with_pending_event)

    event_bus_mock.post.assert_called_once_with(
        pending_event
    )
```

```python
# state-based testing example
@pytest.mark.usefixtures("transaction")
def test_AuctionsRepo_UponSavingAuction_ClearsPendingEvents(
    connection: Connection,
    auction_with_pending_event: Auction,
    event_bus_mock: Mock,
) -> None:
    repo = SqlAlchemyAuctionsRepo(
        connection, event_bus_mock
    )

    repo.save(auction_with_pending_event)

    assert len(auction.domain_events) == 0
```

There has been a dispute about one approach being better than others, but personally, I find such an argument beyond the point. I think both state-based and interaction-based testing approaches are useful in appropriate situations. However, it does not mean they can always be used interchangeably. **Regardless which one you use, an overarching goal is to avoid violating encapsulation of a system under test.** Sadly, neither of the two approaches prevents that by design.

PERILS OF STATE ORIENTED TESTING

State-based testing seems to be a perfect choice when we implement Act (or When) step with a simple public method call, then verify its result using another public method. However, it is dangerous a test uses private fields or methods. When there is no obvious way to read a state, one may feel tempted to add a special method that will be used just in testing.

```python
def test_Auction_Ending_ChangesEndedFlag(
    yesterday: datetime,
) -> None:
    auction = AuctionFactory(ends_at=yesterday)

    auction.end_auction()

    assert (  # Linter complains about _ended? Let's add a getter!
        auction.is_ended()
    )
```

Avoid that temptation. There are several sound reasons not to use such tricks[50]:

1. it adds maintenance burden and will have to be maintained once internal implementation changes (consider swapping `_ended` with `_ended_at`)

2. Complexity of `Auction` class grows

3. The tests no longer describe how `Auction` should be used. `is_ended` is not meant to be used outside in production code but tests that are living documentation suggest otherwise

4. Checking internal state does not contribute to conveying domain knowledge. Okay, `Auction` is ended, but so what? What are the consequences? What I can / cannot do with an ended auction?

To conclude, state-based testing is a great choice as long as one can inspect the state without breaking encapsulation of a system under test.

PERILS OF INTERACTION ORIENTED TESTING

Interaction-based testing is convenient when we verify the usage of dependencies. This kind of testing is used in combination with mocks which are used to write assertions and potentially fail tests. Mocks are our salvation when we cannot / do not want to use external dependencies, like payment providers or other 3rd parties. In terms of the Clean Architecture, one could mock *Port* and then verify if *Use Case* called *Port* as expected:

```
def test_EndingAuction_WonEndedAuction_CallsPaymentProviderWithAuctionCurrentPrice(
    ending_auction_uc: EndingAuction,
    auction: Auction,
    payment_provider: Mock,
) -> None:
    ending_auction_uc.execute(
        EndingAuctionInputDto(auction.id)
    )

    payment_provider.begin_payment.assert_called_once_with(
        auction.current_price
    )
```

[50] Nat Pryce, *State vs. Interaction Based Testing Example* http://natpryce.com/articles/000356.html

We mock external dependency, which can be slow, unstable or should not be used for any other good reason. Mocking is good, as long as the interface we mock is stable and rarely changed. Putting this another way, having mocks for `PaymentProvider.begin_payment` introduces additional coupling. That should not be a problem unless one wrongly interprets *testing in isolation* rule. It does not mean that a class, a method or a function has to be tested alone. It merely implies that one test should not affect any other test. Code above uses a real instance of `Auction` class. If we were to replace it with a mock (and do so in all other *Use Case* tests) we would end up with making `Auction` immutable code - any slight change would make all mocks invalid, hence causing tests that use them to fail.

To sum up, behavior-based testing is dangerous when mocks are overused. They should be applied sparingly. Good examples are checking how the system under test uses Ports or external dependencies that we do not control.

STUBS VERSUS MOCKS

Mocking is not the only approach to replace objects for testing purposes. While mocks are flexible and enable us to replace virtually any object with dynamically specified behavior, they have their limits and ideal use case. Generally speaking, mocks are merely one category of pretend objects used for testing purposes. The most general name for this type of objects is **Test Doubles**. Another type is a stub - simple implementations that return canned responses. They are meant to be used in a different way than mocks. When stubs are used in a specific test, one would rather write assertions about an object that uses them than for stubs themselves.

Consider an example:

```
class PaymentProviderStub(PaymentProvider):
    def begin_payment(
        self, amount: Money
    ) → bool:
        return True
```

```
def test_EndingAuction_WonEndedAuctionStubbedPaymentProvider_EmitsDomainEvent(
    auction: Auction,
    payment_provider: PaymentProviderStub,
    event_bus_mock: Mock,
) → None:
    ending_auction_uc = EndingAuction(
        payment_provider, event_bus_mock
    )
    ending_auction_uc.execute(
        EndingAuctionInputDto(auction.id)
    )

    assert event_bus_mock.post.assert_called_once_with(
        ...
    )
```

Although the test is verifying `EndingAuction` and both `PaymentProvider` and `EventBus` collaborators were replaced, assertions are made only about the mocked `EventBus`. This test checks if `EndingAuction` posts a *Domain Event* via `EventBus` **provided** that `PaymentProvider.begin_payment` succeeds. Note that `PaymentProviderStub` is a custom implementation of a *Port*.

Usually, there are no shortcuts in writing stubs, though Python's mocks could potentially be abused to do so like:

```
payment_provider = Mock(
    spec_set=PaymentProvider,
    begin_payment=Mock(return_value=True),
)
```

There is another creative way of using stubs that allows turning virtually any behavior-oriented testing into state-based one. In order to do so, we have to break two aforementioned rules - first, a query method has to be added to stub just for testing purposes and the assertion in the test will be written against the stub (ouch).

```
class PaymentProviderStub(PaymentProvider):
    def __init__(self) → None:
        self._payments: List[Money] = []
    def begin_payment(
        self, amount: Money
    ) -> bool:
        # begin_payment not only returns a canned response, but also records the
call
        # this makes this class a hybrid of Stub and Spy - another test double type
        self._payments.append(amount)
        return True

    def get_requested_payments() -> List[Money]:
        # return a copy, so state cannot be manipulated from the outside
        return self._payments[:]


def test_EndingAuction_WonEndedAuction_CallsPaymentProviderWithAuctionCurrentPrice(
    ending_auction_uc: EndingAuction,
    auction: Auction,
    payment_provider: PaymentProviderStub,
) -> None:
    ending_auction_uc.execute(
        EndingAuctionInputDto(auction.id)
    )

    assert payment_provider.get_requested_payments() == [
        auction.current_price
    ]
```

Query method was exposed on a stub specifically to verify how its `begin_payment` method was called. A few paragraphs earlier, I warned you about adding such special methods, but that piece of advice is applicable for classes used in production, not those written exclusively for tests.

UNIT-TESTING OF AN ENTIRE MODULE

We know already we should avoid coupling tests with implementation because this will impede refactoring instead of speeding it up and make it next to impossible to make any changes in code without failing half of the test suite. A properly designed (or rather discovered by trial-and-error) encapsulation is what lets us write tests against stable API and maintain the freedom to rearrange private details. We also know that unit tests are cheap to write but are often underestimated due to the fact of how little code they exercise at once. Or maybe we are wrong about small scope…?

A common misconception is that a unit test is only about verifying a method of a single class or a standalone function - the smallest unit one can imagine. The confusion comes from the fact that neither *unit* nor *unit-testing* is strictly defined. There are only clues. We expect tests of a unit to be stable, isolated from the external world and significantly faster than other kinds of tests. Software developers should be able to write and run such tests on their machines. Hence, time for a shift in perspective - let's think how one can test an entire module as a single unit with a black-box approach. First, let's remind what a module is and where are its *contact points* with surroundings.

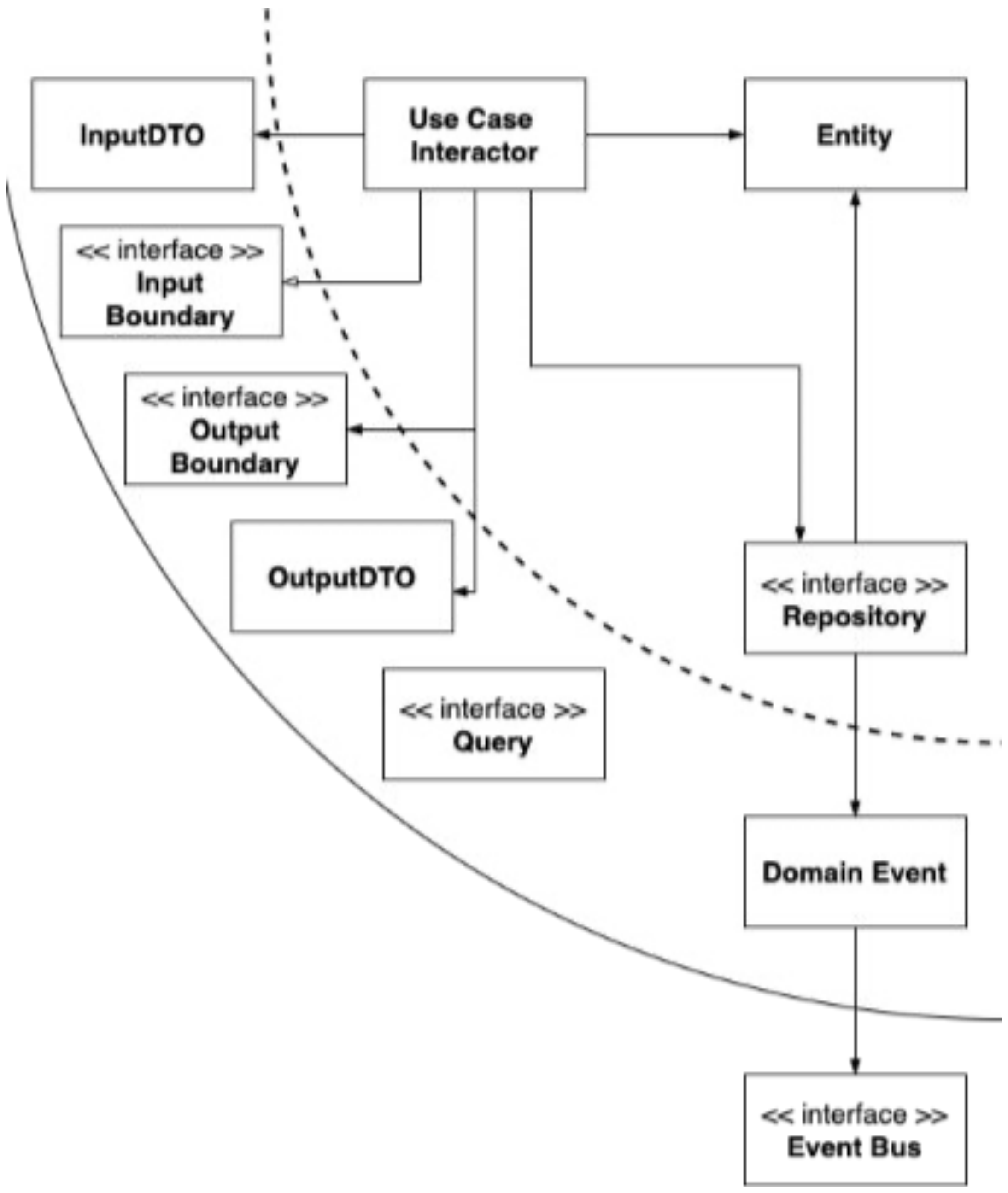

InputDTO
Use Case Interactor
Entity
<< interface >> Input Boundary
<< interface >> Output Boundary
OutputDTO
<< interface >> Repository
<< interface >> Query
Domain Event
<< interface >> Event Bus

Figure 8.5 Contact points of a module: Input- and Output Boundaries with respective DTOs, Queries and Domain Events

TESTING DOUBLES - QUICK RECAP

Stub - a test double implemented before the test that returns canned (hardcoded) responses. It should not fail the test. Its role is to replace the dependency of a system under test. Then, we check if it behaves as expected, provided with stubbed dependency.

Fake - simple yet functional implementation of a dependency that is not ready for production-use yet may prove valuable in tests

Dummy - usually a primitive value like None that is just passed around and not used. For example, we may need an argument to class' constructor which is not used in the particular test, yet still required by class

Mock - a test double that can be queried for how it was used. Used in assertions, especially in behavior-oriented testing

When we unit-test a class, we are doing so by calling its methods and checking results or fields of an instance. Hopefully, we are not using private methods or fields in tests, since we then break encapsulation and make it harder to refactor the class in the future. If we think about a module, it also has its own public API - these are *Use Cases* (or *Commands + Commands Handlers*) and *Queries*. A typical contact point with the outside world is a *Port* or *Repository*. Repository represents the private storage of a module that needs to be maintained between invocations of *Use Cases*. Things that come out of a module are *Domain Events* (via *Event Bus*) and *Output DTOs* (if we use *Output Boundaries / Presenters*).

To write any test against an entire module, we need to decide how to achieve four things:

1. Putting a system under test in the desired state (Arrange / Given)
2. Invoking an action of the system under test (Act / When)
3. Verifying output of action from point 2. (Assert / Then)
4. Dealing with dependencies (*Ports* & *Repositories*)

PUTTING A SYSTEM UNDER TEST IN A DESIRED STATE (ARRANGE / GIVEN)

The way *setting the stage* is implemented depends on whether we treat *Entities* as a private detail of a module or not. In a former case, one must not touch *Entities* in tests but is allowed to call *Use Cases* of a given module in such a way they should put the system under test into the state we desire. If our module is using a *Facade* instead of *Use Cases*, we are allowed only to call Facade's public methods:

```python
def test_Auction_OverbidFromOtherBidder_EmitsEvents(
    beginning_auction_uc: BeginningAuction,
    place_bid_uc: PlacingBid,
    *other_args
) -> None:
    auction_id = 1
    tomorrow = datetime.now(
        tz=pytz.UTC
    ) + timedelta(days=1)
    # First, auction begins
    beginning_auction_uc.execute(
        BeginningAuctionInputDto(
            auction_id,
            "Foo",
            get_dollars("1.00"),
            tomorrow,
        )
    )
    # Then, a new winning bid is placed
    place_bid_uc.execute(
        PlacingBidInputDto(
            1, auction_id, get_dollars("2.0")
        )
    )

    # Act / When
    ...
    # Assert / Then
    ...
```

On the bright side, that is very close to the way the module will be used in production code. On the other hand, introducing a fatal bug to `BeginningAuction` would create a cascade of failing tests since it is used in virtually all other tests. A situation like that can be easily contained by running tests often and working in a TDD way. Then we can exactly tell which change caused tests failures, then revert or amend it.

In general, keeping *Entities* hidden makes better encapsulation, hence allows for bolder code changes. However, when putting a module in the desired state using *Use Cases* only becomes more complex, one may want to sacrifice some encapsulation. Builders, like `AuctionFactory`, are then our way to go. Even if one goes for total encapsulation, builders still may come in handy, for example, to generate *Input DTOs* to our *Use Cases*.

INVOKING AN ACTION ON SYSTEM UNDER TEST (ACT / WHEN)

This step is the simplest of all. Implementation comes down to just calling a *Use Case*:

```python
def test_Auction_OverbidFromOtherBidder_EmitsEvents(
    place_bid_uc: PlacingBid, *other_args
) -> None:
    # Arrange / Given
    ...

    # Act / When
    place_bid_uc.execute(
        PlacingBidInputDto(
            2, auction_id, get_dollars("3.0")
        )
    )

    # Assert / Then
    ...
```

VERIFICATION (ASSERT / THEN)

While business logic indeed interacts with *Entities*, they are rarely the same what users of the system see (unless we are writing CRUD). In simple cases, we present a subset of fields. In more complex scenarios user sees a combination of data from different *Entities*, probably nested & twisted. That is why CQRS is so helpful because it relieves the burden on *Entities*. They no longer have to care for both executing business logic AND providing insights into the state of the system for users. That also implies *Queries* are not unit-testable since they are tightly coupled to the underlying infrastructure, hence have to be tested using higher-level tests. If we were to use a fake repository implemented in memory, obviously we would have to reimplement all queries just for tests. This makes the whole effort questionable.

Even if we take *Queries* out of the equation, there are still obvious ways to assert about during unit-testing of an entire module:

- Output DTO, if we use Output Boundary + Presenter
- exceptions thrown
- which *Domain Events* have been emitted from within the module
- how mocks for *Ports* were used

```python
# verifying Output DTO
def test_PlacingBid_FirstBidHigherThanIntialPrice_IsWinning(
    place_bid_uc: PlacingBid,
    output_boundary: PlacingBidOutputBoundaryFake,
    auction_id: AuctionId,
) -> None:
    place_bid_uc.execute(
        PlacingBidInputDto(
            1, auction_id, get_dollars("100")
        )
    )

    expected_dto = PlacingBidOutputDto(
        is_winner=True,
        current_price=get_dollars("100"),
    )
    assert output_boundary.dto == expected_dto
```

```python
# verifying thrown exceptions
def test_PlacingBid_BiddingOnEndedAuction_RaisesException(
    beginning_auction_uc: BeginningAuction,
    place_bid_uc: PlacingBid,
) -> None:
    yesterday = datetime.now(
        tz=pytz.UTC
    ) - timedelta(days=1)
    with freeze_time(yesterday):
        beginning_auction_uc.execute(
            BeginningAuctionInputDto(
                1,
                "Bar",
                get_dollars("1.00"),
                yesterday + timedelta(hours=1),
            )
        )

    with pytest.raises(BidOnEndedAuction):
        place_bid_uc.execute(
            PlacingBidInputDto(
                1, 1, get_dollars("2.00")
            )
        )
```

```
# verifying emitted Domain Events
def test_Auction_OverbidFromWinner_EmitsWinningBidEventOnly(
    place_bid_uc: PlacingBid,
    event_bus: Mock,
    auction_id: AuctionId,
    auction_title: str,
) -> None:
    place_bid_uc.execute(
        PlacingBidInputDto(
            3, auction_id, get_dollars("100")
        )
    )
    event_bus.post.reset_mock()

    place_bid_uc.execute(
        PlacingBidInputDto(
            3, auction_id, get_dollars("120")
        )
    )

    event_bus.post.assert_called_once_with(
        WinningBidPlaced(
            auction_id,
            3,
            get_dollars("120"),
            auction_title,
        )
    )
```

You may wonder if these approaches are sufficient since we are not validating *Queries* in any way. The truth is one should be able to check plenty of possible scenarios that way. Thanks to *Domain Events*, not being able to inspect the state directly should not be a problem. Especially that we could implement read models to be generated only using Domain Events[51].

Another quite obvious idea one may have is to make assertions about *Entities* being saved. This might sound like a great idea, but causes the same problems with encapsulation like those mentioned before in part about putting a module in the desired state before the test.

[51] Matthias Noback, *Object Design Style Guide*, Chapter 8. Build read models from domain events

DEALING WITH DEPENDENCIES (*PORTS* & *REPOSITORIES*)

Although a single module can deal with the most complex business logic we can imagine, it still requires *Ports* and *Repositories* to actually do something. What good would they be for if they could not communicate with the world?

Ports are often abstracting away external systems which we do not control. For example, we cannot rely on their stability or easily put them in the desired way. We definitely must not unit-test a module with a real Adapter. We either mock or stub this kind of dependency.

There is one seemingly special case when one module calls another - what do we do? The solution is the same - mock or stub such dependency.

In the case of *Repositories*, we want to replace them with another test double - fake. The latter is a simplified, yet functional implementation of a given interface (`AuctionsRepository` in this case). A simple in-memory implementation is what we are looking for. This should be aligned with our production-grade implementations. For example, if our repository uses event bus to emit events, an in-memory implementation should do the same.

CHAPTER SUMMARY

Although automated tests cannot guarantee there will be no bugs in code, they are still a worthy investment. There is no single *best practice* (I'm not too fond of that term by the way) for building automated test suites that will maximize your return on investment. It is just typical that lower-level tests, for example, unit tests, are faster, simpler and cheaper than high-level API tests. That being said, LMAX employees has been boasting their extensive test suite that is strongly focused on higher-level testing. However, they are building a blazing-fast trading facility, so unless one is building exact same product, they should not recklessly copy other testing strategies. Each project needs a bit of experimentation and failed trials to find an optimal approach to automated testing.

Let's conclude this chapter with example testing strategy:

- heavy unit-tests for the entire module, with faked *Repositories* and mocked *Ports*
- no unit tests for individual classes inside the Clean Architecture modules, unless they are things like calculators or validators and it is very impractical to test them with the entire module
- higher-level tests for *Facade-based* modules, without test doubles for database
- always have tests to check external providers, at minimum mimic the real usage of it, e.g. charge payment card, then capture
- at least three API-level tests for each *Use Case / Facade* method to check positive scenario, negative scenario and to check if authorization is working
- a couple of UI tests that will cover several endpoints at once
- do not do automated acceptance testing on the level of UI

Remember that fine test suite should empower developers and encourage them to introduce changes. It should be a kind of a safety net that whispers to their ears "I got your back!" even when they are to touch most critical, money-making parts of the system. In the long run, a test suite helps maintain a high velocity of delivery. Bear in mind that your test suite can only be as good as code that is being tested. One has to pay attention to design and code quality to guarantee high testability.

FINAL WORDS

```
from: sebastian@cleanarchitecture.io
to: the dearest reader
subject: Personal thanks
```

Dear reader,

you did it!

Thank you for bearing with me throughout the lecture of this book. I sincerely hope reading it was at least as enlightening experience for you as writing it was for me.

I must admit I thought it would be relatively easy to write about the Clean Architecture when you implemented it twice and taught a few dozens of people how to do it.

Throughout writing the book, I tried to look at things from different angles. I was not looking for absolute truth or *best practices* (again, I wouldn't say I like that term - *best practices are context dependant*). That is how it works in software development - it is a game of trade-offs. Universal solutions, appropriate for everyone does not exist. A developer chooses a solution not to have a particular problem they do not like. At the same time, they accept several less significant issues they have to learn how to live with.

That is why I want you to evaluate every recipe shown in this book before using it. Try to find out if it will really benefit you, your team, your company and your project.

Regardless of whether you will use 5% or 100% advice from the book, I wish it would help you advance your career.

Yours truly,

Sebastian

APPENDIX A: MIGRATING FROM LEGACY

SHOULD I EVEN MIGRATE?

What good for are techniques if they are only applicable to green-field projects?

Working with legacy (also known as brown-field) projects is much more probable scenario for most of us. Hence, a migration strategy would be most welcomed. Frankly, there is little to no chance that you will be able to migrate your entire project to the Clean Architecture. Not only because not all code would benefit from it, but also because some areas are not actively developed. Rewriting them just to make it finally look good is art for art's sake. Unless it is a non-critical part, you decide to rewrite to gain an understanding of applying the Clean Architecture.

You may neglect refactoring to the Clean Architecture if your project is undergoing major changes. Unstable conditions will make it considerably harder (and more expensive) to make progress.

HOW TO DO IT?

First things first - we have to have an idea about different modules of our project. If there is no notion of modules in the code, we have to sketch it, using imagination and a whiteboard. Do not jump to code if you are not sure where modules' boundaries are. Talk to people. Gather domain knowledge. Invite someone that works longer than you to support your efforts. Ideally, it would be best if you came up with a DDD Context Map, but rough sketches would also do.

Now when you have identified components, try to mark their complexity/importance on the scale from 1 to 3, where 1 is the most important and 3 is the least important. 1 is for components that are either company's competitive advantage, are most complex, are main source of revenue or are just complex. On the other hand, 3 is often a glue code for 3rd party solution used by a business which is not changed very often, if at all.

Now when you understand how things work, the next step is to do some groundwork. Setup dependency injection library if you do not have one already.

Identify the module boundary and formalize it, using *Use Cases* or *Facade*. There is no need to start big-bang rewrite from identifying *Entities*, writing *Repositories* for them etc. Hold your horses, at least for now. At the moment, just wrap up the module and find how it is used. Then expose an API for it. Then start using it in other code areas.

Do not forget about testing. It would be ideal if you had a set of higher-level tests you can run locally to make sure your refactorings are safe.

In the next step, identify code coupled with external providers and start moving it to a newly created *Adapter*. Create a *Port* and configure a binding in IoC container.

In the final step, start identifying *Entities* that will protect your business rules. Create repositories for them. That is going to be the most challenging and long-lasting part of the migration.

"I CANNOT STOP DELIVERING NEW FEATURES"

Frankly, it would be better to not even think about freezing new features. Refactoring codebase is a serious undertaking. Since it involves so many changes, it is risky. It would be best to take an approach like Branch By Abstraction that will let you do changes along with usual development and deploy changed code.

Do not try to do a big-bang refactoring and definitely do not jump into code unless you have come up with a plan. Instead, adopt a steady approach to evolve codebase in the desired direction. It is not an easy refactoring to do since you will be probably learning two things at the same time - the Clean Architecture AND arcana of your project. You do not have to strive for perfection at first attempt. Perfectionism is not an ally of a software developer. You are not building a cathedral that is supposed to last ages. Know where you want to be and slowly go there, step by step.

APPENDIX B: INTRODUCTION TO EVENT SOURCING

WHAT IS EVENT SOURCING?

Let’s consider e-commerce Order. It might hold current status (new, confirmed, shipped, etc.) and summaries – total price, shipping and taxes. Naturally, Order does not exist on its own. We usually wire it with another entity, OrderLine that refers to a single product ordered with quantity information. This structure could be represented in a relational database in the following way:

```
orders
-[ RECORD 1 ]---------
id          | 1
status      | NEW
total_price | 169.9900


order_lines
-[ RECORD 1 ]---
id         | 1
order_id   | 1
product_id | 512
quantity   | 1
-[ RECORD 2 ]---
id         | 2
order_id   | 1
product_id | 614
quantity   | 3
```

By storing data this way, we can always cheaply get the CURRENT state of our Order. We store a dump of the serialized object after the latest changes. Any data mutation, for example switching status from new to shipped causes data overwrite. We irreversibly lose old state. What if we need to track all changes…?

Let's see how that fits in another database table:

```
order_history
-[ RECORD 1 ]--------------------------------
id         | 1
order_id   | 1
event_name | Created
created_at | 2018-08-09 23:00:09.22674+02
data       | {}
-[ RECORD 2 ]--------------------------------
id         | 2
order_id   | 1
event_name | LineAdded
created_at | 2018-08-09 23:01:03.47922+02
data       | {"product_id": 512, "quantity": 1}
-[ RECORD 3 ]--------------------------------
id         | 3
order_id   | 1
event_name | LineAdded
created_at | 2018-08-09 23:37:06.93112+02
data       | {"product_id": 614, "quantity": 3}
-[ RECORD 4 ]--------------------------------
id         | 4
order_id   | 1
event_name | Confirmed
created_at | 2020-08-09 23:52:03.08832+02
data       | {"status": "confirmed"}
```

Such a representation enables us to firmly tell what was changed and when. However, `order_history` plays second fiddle. It is merely an auxiliary record of `orders`, added just to fulfill some business requirement. We still reach to original `orders` table when we want to know the exact state of any **`Order`** in all other scenarios. In particular, we rely on `orders` whenever we make any changes (e.g. changing quantity) or reading state in most cases (e.g. telling what total price of the order is).

However, note that `order_history` is as good as `orders` table when we have to get current **`Order`** state. We just have to fetch all entries for given **`Order`** and 'replay' them from the start. In the end, we'll get exactly the same information that is saved in the `orders` table. So should we still treat `orders` table as our source of truth? Event Sourcing takes a different approach. We can safely get rid of the table or at least no longer rely on it in any situation that would actually change **`Order`**.

To sum up, Event Sourcing comes down to:

- Keeping your business objects (from now on called *Aggregates*) as a series of replayable events. A collection of these events is called an event stream
- Never deleting any events from a system, only appending new ones
- Using events as the only reliable way of telling in what state a given *Aggregate* is
- If you need to query data or present them in a table-like format, keep a copy of them in a denormalized format. This is called projection
- Designing your *Aggregates* to protect certain vital business invariants, such as `Order` encapsulates costs summary. A good rule of thumb is to keep aggregates as small as possible

What you get in return:

- A complete history what was changed when and by who (if you enclose such information in an event)
- Time-travel debugging, allowing to recreate the state of the system in any given moment
- Possibility of creating specialized read models of your data for high performance
- Append – only write model that is also easier to scale

SIMPLE EXAMPLE OF AN EVENT SOURCING AGGREGATE

ORDER AS ENTITY

Consider the following **Order** class written as if it was a classic *Entity*:

```
class Order:
    def __init__(
        self,
        uuid: UUID,
        customer_id: CustomerId,
        lines: Optional[List[OrderLine]] = None,
        status: OrderStatus = OrderStatus.NEW,
    ) -> None:
        if lines is None:
            lines = []
        self._customer_id = customer_id
        self._status = status
        self._lines = lines

    def confirm(self) -> None:
        if self._status != OrderStatus.NEW:
            raise IllegalStatusChange(
                "Only NEW order can be confirmed!"
            )
        self._status = OrderStatus.CONFIRMED
```

To create an instance, we just need a `customer_id` and `uuid`. There are default arguments provided for `lines` and `status`. Order guards a very simple domain invariant, namely makes sure only new orders can be confirmed.

INTRODUCING EVENTS

Let's rewrite **Order** class using Event Sourcing. First, we need events that will represent any state mutations:

```
@dataclass(frozen=True)
class Event:
    _subclasses: ClassVar[Dict[str, Type]] = {}

    created_at: datetime
    version: int

    def __init_subclass__(cls) -> None:
        """Register subclasses by name."""
        Event._subclasses[cls.__name__] = cls
    @classmethod
    def subclass_for_name(cls, name: str) → Type:
        """Get event subclass to deserialize."""
        return cls._subclasses[name]

    def as_dict(self) → dict:
        """Dump to dict for serialization."""
        dict_repr = asdict(self)
        dict_repr.pop("created_at")
        dict_repr.pop("version")
        return dict_repr

@dataclass(frozen=True)
class OrderDrafted(Event):
    customer_id: CustomerId


@dataclass(frozen=True)
class OrderConfirmed(Event):
    pass
```

THESE ARE NOT DOMAIN EVENTS WE SAW BEFORE!

A resemblance between Event Sourcing events and Domain Events used mostly for integration purposes in previous chapters is visible to the naked eye. Both stand for a significant fact within a domain.

However, they are not the same and should not be used interchangeably! Event Sourcing events should never cross a boundary of a module containing an *Aggregate* emitting these events. *Domain Events* may be used outside, published and used for integration. Event Sourcing events absolutely not. The latter is more like persistence details and are not meant to be used outside of the module.

For more details, read *Why Event Sourcing is a microservice communication anti-pattern* by Olivier Libutzki https://dev.to/olibutzki/why-event-sourcing-is-a-microservice-anti-pattern-3mcj

In such a simple example, there are only two events. Note that their naming is as specific for orders as possible and familiar to a domain expert. One should avoid creating general event classes to spare keystrokes like `StatusChanged` with a `status` field. The first event, `OrderDrafted`, is a standard way of starting any event stream for `Order`. The second event, `OrderConfirmed`, represents an act of confirming `Order`. **Such event classes only carry as much information as is it required for rebuilding the state of an aggregate. This is a significant difference between Event Sourcing events and *Domain Events* used for integration.** For following `Order` calls:

```
# instance created using @classmethod serving as a factory
order = Order.draft(uuid4(), customer_id=1)
order.confirm()
```

…a corresponding *event stream* looks like this:

```
event_stream = [
    OrderDrafted(customer_id=1, created_at=...),
    OrderConfirmed(created_at=...),
]
```

ORDER AS AGGREGATE

In the end, we should be able to load `Order` state using these events. There goes a rewritten version of `Order` class that is a full-fledged ES aggregate. It will accept an instance of `EventStream` as an argument:

```
@dataclass(frozen=True)
class EventStream:
    uuid: UUID
    events: List[Event]
    version: int
```

This is a simple data structure with UUID of the *Aggregate*, a list of past events to rebuild the state from and a version of *Aggregate* that will come in handy a bit later.

```
class Order:
    def __init__(
        self, event_stream: EventStream
    ) -> None:  # 1
        self._uuid = event_stream.uuid
        self._version = event_stream.version

        self._customer_id = 0  # 2
        self._status = OrderStatus.NEW
        self._lines: Dict[ProductId, int] = {}

        for event in event_stream.events:  # 3
            self._apply(event)

        self._new_events = []  # 4

    @property
    def uuid(self):
        return self._uuid

    @property
    def _next_version(self):
        """Useful for events creation"""
        return self._version + 1

    @property
    def changes(self) -> AggregateChanges:  # 5
        return AggregateChanges(
            self._uuid,
            self._new_events[:],
            self._version,
        )

    def _apply(self, event: Event) -> None:  # 6
        if isinstance(event, OrderDrafted):
            self._customer_id = event.customer_id
            self._status = OrderStatus.NEW
        elif isinstance(event, OrderConfirmed):
            self._status = OrderStatus.CONFIRMED
        else:
            raise ValueError(
                f"Unknown event {event}"
            )
    @classmethod
    def draft(
        cls, uuid: UUID, customer_id: CustomerId
    ) -> "Order":
        instance = cls(
            EventStream(
                uuid=uuid, version=0, events=[]
            )
        )
        instance._new_events = [
            OrderDrafted(
                datetime.now(tz=pytz.UTC),
                0,
                customer_id,
            )
        ]
        return instance

    def confirm(self) -> None:  # 7
        if self._status != OrderStatus.NEW:
            raise IllegalStatusChange(
                "Only NEW order can be confirmed!"
            )

        event = OrderConfirmed(
            datetime.now(tz=pytz.UTC),
            self._next_version,
        )  # 8
        self._apply(event)
        self._new_events.append(event)  # 9
```

1. To instantiate `Order` one has to give it a list of events it should be initialized with
2. Before initializing class, we sometimes may need to set default values for fields knowing they will be overridden
3. Each event is applied, causing state mutation
4. The Aggregate will keep new events created after initialization due to method calls…
5. … to be later obtained to be saved. Here, we use a `@property` + list copy trick to make sure no one will mangle with recorded changes from the outside
6. A heart of every Event Sourcing *Aggregate* is `_apply` method. Inside it we mutate state. **Real-life implementation should not use if-elif + isinstance calls.** This would look much better in a language that supports polymorphic methods overloading.

7. Finally, a public method which is responsible for the business logic of confirming orders
8. Instead of directly altering state, we create an event and apply it immediately
9. Right after it, we append it to `_new_events`.

TESTING AGGREGATES

Testing Event Sourcing *Aggregates* is trivial. Arrange (Given) step comes down to instantiating an *Aggregate* with given events. Act (When) step involves calling a public method on the aggregate while Assert (Then) checks either for expected events or exception:

```python
@freeze_time("2019-01-14")
def test_order_newly_created_confirmation_changes_status(self):
    now = datetime.now(tz=pytz.UTC)
    order = Order(
        EventStream(
            uuid=uuid4(),
            events=[OrderDrafted(customer_id=1, created_at=now)],
            version=1
        )
    )

    order.confirm()

    assert order.changes.events == [OrderConfirmed(now)]

@freeze_time("2019-01-14")
def test_order_newly_created_cannot_be_confirmed_twice(self):
    now = datetime.now(tz=pytz.UTC)
    order = Order(
        EventStream(
            uuid=uuid4(),
            events=[
                OrderDrafted(customer_id=1, created_at=now),
                OrderConfirmed(created_at=now)],
            version=1
        )
    )

    with self.assertRaises(IllegalStatusChange):
        order.confirm()
```

PERSISTENCE IN EVENT SOURCING

APPEND-ONLY EVENT STREAMS

Since events are first-class citizens required to rebuild the state of an *Aggregate*, we need to be able to retrieve them using aggregate id and append new ones to the existing event stream.

These functionalities could be provided by a class inheriting from such an abstract class:

```python
class EventStore(abc.ABC):
    @abc.abstractmethod
    def load_stream(
        self, aggregate_uuid: UUID
    ) → EventStream:
        pass

    @abc.abstractmethod
    def append_to_stream(
        self, changes: AggregateChanges
    ) → None:
        pass
```

`load_stream` method returns **EventStream** instance - simple data structure with a list of events needed for *Aggregate*'s initialization, UUID, and current version:

```python
@dataclass(frozen=True)
class EventStream:
    uuid: UUID
    events: List[Event]
    version: int
```

We will use version to protect against concurrent updates using optimistic locking. `append_to_stream` method accepts **AggregateChanges** - another simple data structure:

```python
@dataclass(frozen=True)
class AggregateChanges:
    aggregate_uuid: UUID
    events: List[Event]
    expected_version: int
```

It consists of *Aggregate*'s UUID, expected version (the same value we obtained from `load_stream`) and a list of events our *Aggregate* produced. Please note that we do not have to store old events, only new ones. This is possible because we are not allowed to delete any events ever, so this is an append-only structure.

Aforementioned `expected_version` parameter serves for protection against concurrent updates. If such a situation occurs, this method should raise an exception:

```
class ConcurrentStreamWriteError(RuntimeError):
    pass
```

Implementation details depend on a chosen database engine.

Another critical question, what database should be used? Some people state that almost any is fine. I find it hardly a valid answer. Is transactional database like MySQL a good choice? What to pay attention to? To precisely answer this question, one has to consider the nature of events stream and how they are used to reconstruct *Aggregates*.

RETRIEVING STRATEGY

To rebuild an aggregate, we need all events that were ever emitted by it. Our *Aggregates* will usually have a unique ID, in particular UUID. In other words, we should be able to query our event store by *Aggregate*''s ID. This requirement can be easily met by many substantially varying database engines.

Redis

This is a popular data-structure DB that can store data in a few handy structures, such as:

- strings
- lists
- sets
- sorted sets
- hashes (also known as dictionaries or maps)

Assuming Redis is our choice, we would store events using lists and use UUID *Aggregate* as a part of a key name. To retrieve all events for a given UUID, we would use the following query:

```
LRANGE event_stream_f42d9a33-81da-45ba-a066-32de5e747067 0 -1
```

WARNING: Redis' implementation of lists causes such queries to lower performance linearly with an increase of a number of events for the given *Aggregate*[52]. This means Redis is not an optimal choice.

RDBMS (PostgreSQL, MySQL, etc)

A natural way of modeling event stream using RDBMS is to create a table for all of them. Although we are going to store many types of events with different fields, it is not feasible to create a separate table for each one. Firstly, the performance of querying will suffer. Secondly, it will make queries more complicated to implement and maintain. Thirdly, events may evolve over time, including additional data. Of course, we can not change events from the past, but somehow we would have to store new ones with altered structure alongside old ones. So staying with the one-table-for-all-events design is the right thing to do.

```
events
--------------------------------------------------------
| uuid | aggregate_uuid | name | data | <sort_column> |
--------------------------------------------------------
```

On the other hand, we may consider creating a separate table for each *Aggregate* type to get logical shards of data. Due to the nature of *Aggregates* (they are separated from each other), one has also a possibility to shard by `aggregate_uuid`.

Each event has also its own `uuid`. `aggregate_uuid` is a column allowing us for easily querying by it. **We should put an index on it.** `name` is self explanatory (e.g. **`OrderConfirmed`**). Then we have a flexible part – `data`. We will store JSON-encoded details inside. Depending on a chosen database engine, we would use a dedicated data type (PostgreSQL supports JSON columns) or simply TEXT. Finally, we need a column to sort by. Depending on the chosen design, it may be either `created_at` timestamp or `version`. At least one of them has to be assigned to events when they are created in code.

Querying is trivial:

```
SELECT * FROM events
WHERE aggregate_uuid = 'f42d9a33-81da-45ba-a066-32de5e747067'
ORDER BY created_at;
```

[52] Redis LRANGE command documentation https://redis.io/commands/lrange

Document-oriented (MongoDB, RethinkDB etc)

Analogously to RDBMS, we should be able to use a single collection (MongoDB) or a table (RethinkDB) to retrieve all events. Beside data JSON part, we would also add `aggregate_uuid` and put an index on it for faster reads.

Querying is also very simple:

```
# MongoDB
db.events.find(
    {aggregate_uuid: 'f42d9a33-81da-45ba-a066-32de5e747067'}
)

# RethinkDB
r.table('events').get_all(
    'f42d9a33-81da-45ba-a066-32de5e747067', index='aggregate_uuid'
).run()
```

STORING STRATEGY

As I have already mentioned, we never delete events. Event Sourcing does not limit the number of events by *Aggregate*, so we should be prepared that our events table/collection will grow indefinitely. Therefore, we need a database that is able to scale and maintain approximate read/write times regardless of the number of events (up to some extent, of course).

Events are our source of truth, so we can not afford any data loss. Thus, we need a database with strong consistency guarantees.

Event Sourcing assumes that only one *Aggregate* should be saved within one business operation. Saving a single *Aggregate* has to be atomic. We do not need to have a full-fledged all-or-nothing guarantee as it is with relational SQL databases that spans entire HTTP request or whatever. We just need to make sure that once we attempt to save changes to *Aggregate* and bump up its version, it will be an atomic operation.

Protection against concurrent updates is not something I can pass over in silence. Frankly, I find it bizarre that most of articles about Event Sourcing implementation do not say a word about these issues and possible solutions. A commonly used READ - MODIFY - WRITE approach is an invitation for race conditions. It would be great if a database engine provided means to implement an optimistic clocking to prevent bad things from happening. Of course, we could work around this problem using pessimistic locking and Redis, yet it makes implementation harder.

REQUIREMENTS WRAP-UP

- Efficient access to all events for a given *Aggregate*
- Strong consistency guarantees
- Possibility of implementing optimistic locking/different protection without external services
- Nice to have: enable horizontal scalability by sharding/partitioning

EXAMPLE IMPLEMENTATION USING POSTGRESQL

Choosing PostgreSQL to be a weapon of choice should not be a surprise. A mature, battle-proven open source solution can be a perfect base for an event store. One-table-for-all-events is a good approach for providing efficient read and writes. It can be scaled using table partitioning or sharding (more or less manual, but still – within reach). PostgreSQL has strong consistency guarantees, so this requirement is also met. Protection against concurrent updates using optimistic locking is quite easy to implement. It seems like a perfect choice!

Table design

Not much changed for events table design. We have both `created_at` and `version` columns, though for sorting it `version` will be used:

```
events
--------------------------------------------------------------
| uuid | aggregate_uuid | name | data | created_at | version |
--------------------------------------------------------------
```

However, to get simple protection against concurrent updates, we are going to use one extra table:

```
aggregates
------------------
| uuid | version |
------------------
```

A version will be bumped up by one every time we have some events to save. Using additional condition in UPDATE query and a returned number of affected rows, we can easily tell if we won the race or not:

```
-- 1 - expected version
UPDATE "aggregates"
    SET version = 2
    WHERE "aggregate_uuid" = 'f42d9a33-81da-45ba-a066-32de5e747067'
    AND "version" = 1  # expected version check
```

If this query returns affected rows count equal 1 – we are good to go. Otherwise, it means someone changed history in the meantime and we should raise `ConcurrentStreamWriteError`.

Code for creating both tables:

```
CREATE TABLE "aggregates" (
    uuid UUID NOT NULL PRIMARY KEY,
    version int NOT NULL DEFAULT 1
);

CREATE TABLE "events" (
    uuid UUID NOT NULL PRIMARY KEY,
    aggregate_uuid UUID NOT NULL,
    name VARCHAR(50) NOT NULL,
    data json,
    created_at timestamp with time zone NOT NULL,
    version int NOT NULL
);

-- Do not forget about the index!
CREATE INDEX aggregate_uuid_idx ON "events" (“aggregate_uuid”, “version”);
```

Note there are no foreign key constraints present. They were omitted on purpose as a form of optimization.

There are numerous ways of getting data from these tables to Python. One of them is using an ORM. Mapping in SQLAlchemy can look like this:

```
class AggregateModel(Base):
    __tablename__ = "aggregates"

    uuid = Column(
        postgresql.UUID, primary_key=True
    )
    version = Column(BigInteger, default=1)
```

```python
class EventModel(Base, EventMixin):
    __tablename__ = "events"
    __table_args__ = (
        Index(
            "ix_events_aggregate_version",
            "aggregate_uuid",
            "version",
        ),
    )
    uuid = Column(
        postgresql.UUID, primary_key=True
    )
    aggregate_uuid = Column(
        postgresql.UUID,
        ForeignKey("aggregates.uuid"),
    )
    name = Column(VARCHAR(50))
    data = Column(postgresql.JSON, nullable=True)
    created_at = Column(DateTime(timezone=True))
    version = Column(BigInteger)

    aggregate = relationship(
        AggregateModel,
        uselist=False,
        backref="events",
    )
```

Event Store implementation

Relying on this code, we can implement `load_stream` method of **PostgreSQLEventStore**:

```python
class PostgreSQLEventStore(EventStore):
    def __init__(self, session: Session) → None:
        # we rely on SQLAlchemy, so we need Session to be passed for future usage
        self._session = session
    def load_stream(
        self, aggregate_uuid: UUID
    ) -> EventStream:
        events_query: List[
            EventModel
        ] = self._session.query(
            EventModel
        ).filter(
            EventModel.aggregate_uuid
            == str(aggregate_uuid)
        ).order_by(
            EventModel.version
        ).all()

        if not events:
            raise NotFound

        aggregate_uuid = events[0].aggregate_uuid
        aggregate_version = events[-1].version
        events_objects = [
            self._to_event_object(model)
            for model in events
        ]

        return EventStream(
            aggregate_uuid,
            events_objects,
            aggregate_version,
        )

    def _to_event_object(
        self, event_model: EventModel
    ) -> Event:
        event_cls = Event.subclass_for_name(
            event_model.name
        )
        return event_cls(
            created_at=event_model.created_at,
            version=event_model.version,
            **event_model.data
        )
```

The only magic part is in `_to_event_object` method. This uses class method of `Event` that keeps all known subclasses in an internal dictionary. After getting the right class by name, we may easily reconstruct it using unpacking with double asterisk.

One should consider abandoning rich ORM capabilities in favor of lighter SQLAlchemy core or even raw queries to gain some performance boost. ORM does not add much value here and is definitely less efficient than other methods.

Since we are implementing optimistic locking, `append_to_stream` is more complicated to implement:

```python
class PostgreSQLEventStore(EventStore):
    ...

    def append_to_stream(
        self,
        changes: AggregateChanges,
    ) -> None:
        if not changes.events:
            raise NoEventsToAppend

        if changes.expected_version:
            self._perform_update(changes)
        else:
            self._perform_create(changes)

        self._insert_events(changes)
```

```python
def _perform_update(
    self, changes: AggregateChanges
) -> None:
    stmt = (
        AggregateModel.__table__.update()
        .values(
            version=changes.expected_version
            + 1
        )
        .where(
            (
                AggregateModel.version
                == changes.expected_version
            )
            & (
                AggregateModel.uuid
                == changes.aggregate_uuid
            )
        )
    )
    connection = self._session.connection()
    result = connection.execute(stmt)

    if result.rowcount != 1:
        # optimistic lock failed
        raise ConcurrentStreamWriteError

def _perform_create(
    self, changes: AggregateChanges
) -> None:
    stmt = AggregateModel.__table__.insert().values(
        uuid=str(changes.aggregate_uuid),
        version=1,
    )
    self._session.connection().execute(stmt)
def _insert_events(
    self, changes: AggregateChanges
) -> None:
    connection = self._session.connection()
    for event in changes.events:
        connection.execute(
            EventModel.__table__.insert().values(
                uuid=str(uuid4()),
                aggregate_uuid=str(
                    changes.aggregate_uuid
                ),
                name=event.__class__.__name__,
                data=event.as_dict(),
                created_at=event.created_at,
                version=event.version,
            )
        )
```

The method works in one of two modes. If it is a first save of an *Aggregate*, then we do insert to `aggregates` table. Otherwise, we increment the version by one using conditional UPDATE. One can tell from a number of updated rows if someone has not updated aggregate in the meantime and if so, raise `ConcurrentStreamWriteError`.

PostgreSQL-based Event Store in action

Whenever you want to load an aggregate from *Event Store* you need its UUID:

```
event_store = PostgreSQLEventStore(session)
event_stream = event_store.load_stream(
    UUID("36b89a56-cbf1-45f2-9f94-b7958481e3d1")
)
order = Order(event_stream)
```

...then you call methods you need...

```
order.confirm()
```

...and finally, save *Aggregate* changes to an append-only stream:

```
event_store.append_to_stream(order.changes)
```

What to do when ConcurrentStreamWriteError is raised?

It depends on business requirements and scenario. In case of an `Order` instance, we may imagine that someone accepts the order and some other guy concurrently tries to cancel it. We could try to fetch aggregate again and resolve the conflict manually, but much simpler (and in most cases sufficient!) approach is to just retry the entire operation that touches our *Aggregate*.

This is an example using Python retrying[53] library:

```
@retry(
    retry_on_exception=lambda exc: isinstance(
        exc, ConcurrentStreamWriteError
    )
)
def cancel_order(
    event_store: EventStore, order_uuid: UUID
) -> None:
    event_stream = event_store.load_stream(
        order_uuid
    )
    order = Order(event_stream)
    order.cancel()
    event_store.append_to_stream(order.changes)
```

Assume there is a race condition between setting two statuses, cancelled and confirmed. If latter wins, then code above will raise `ConcurrentStreamWriteError` during execution of `event_store.append_to_stream`. `@retry` decorator will take care of rerunning the whole thing and reloading the entire *Aggregate* with the most recent version. Provided there are no more concurrent updates we finally are able to cancel our newly confirmed order OR raise another exception if our business rules do not allow for cancelling an order that is confirmed. It is crucial to have a limited number of retries and not try to retry indefinitely. In highly concurrent systems this may lead to unpleasant long-running retry races.

Hiding Event Store behind a Repository

Although *Event Store* itself is an abstraction we can hide it further by using a familiar *Repository* pattern and treat `Order` *Aggregate* as an *Entity* without knowing it uses Event Sourcing:

```
class OrdersRepository:
    def __init__(
        self, event_store: EventStore
    ) -> None:
        self._event_store = event_store
```

[53] retrying library https://pypi.python.org/pypi/retrying

```
    def get(self, aggregate_uuid: UUID) → Order:
        event_stream = self._event_store.load_stream(
            aggregate_uuid
        )
        return Order(event_stream)

    def save(self, order: Order) → None:
        self._event_store.append_to_stream(
            order.changes
        )
```

SNAPSHOTS

The whole idea with rebuilding the state of an *Aggregate* from a potentially very long event stream may sound risky from a performance point of view. In case of **`Order`** *Aggregate*, it is rather unlikely that this is going to be a real problem since orders are relatively short-living (minutes, hours, maybe days) so their event stream should not exceed several events in length.

In case of a longer-living *Aggregates* with ever-growing history, it may be a much more serious issue. Consider **`BankAccount`**. Even though its public interface has merely two main methods - `deposit` and `withdraw`, there might be tens of thousands of them. The solution is to use state snapshots - events-like data structures that can be used to load the state of an without having to iterate over old events. We do not remove the latter, but just use snapshots to limit the number of events needed to rebuild the state of an *Aggregate*.

Taking a snapshot is as simple as copying an internal state of an *Aggregate* to a designated event-like data class:

```
@dataclass(frozen=True)
class OrderSnapshot(Event):
    customer_id: CustomerId
    status: OrderStatus
    lines: Dict[ProductId, int]
```

```python
class Order:
    ...

    def take_snapshot(self) → OrderSnapshot:
        if self._new_events:
            # for snapshot creation after
            # command execution
            version = self._next_version
        else:
            # for background snapshot creation
            version = self._version

        return OrderSnapshot(
            datetime.now(tz=pytz.UTC),
            version,
            self._customer_id,
            self._status,
            self._lines.copy(),
        )
```

Restoring a state using a snapshot is as simple as applying events. One extends `_apply` method of the *Aggregate*:

```python
class Order:
    ...

    def _apply(self, event: Event) → None:  # 6
        ...
        elif isinstance(
            event, OrderSnapshot
        ):  # not in the first snippet!
            self._customer_id = event.customer_id
            self._status = event.status
            self._lines = event.lines.copy()
        else:
            raise ValueError(
                f"Unknown event {event}"
            )
```

Naturally, snapshots need to be kept somewhere. We potentially might be keeping them in the same table, but due to certain implementation advantages and a different nature of snapshots (they can be removed/regenerated while events are immutable) we would rather use a separate table for snapshots. It will be identical to `events` table, though.

The next step is to extend the logic of fetching event stream for a given *Aggregate*'s UUID in *Event Store*. The algorithm is pretty simple - first, one looks for the newest snapshot. If there is none, one fetches all existing events. Otherwise, we fetch only events that are newer than our snapshot + the snapshot itself.

```
class PostgreSQLEventStore(EventStore):
    ...

    def load_stream(
        self, aggregate_uuid: UUID
    ) -> EventStream:
        events_query: List[
            EventModel
        ] = self._session.query(
            EventModel
        ).filter(
            EventModel.aggregate_uuid
            == str(aggregate_uuid)
        ).order_by(
            EventModel.version
        )

        try:
            latest_snapshot = (
                self._session.query(SnapshotModel)
                .filter(
                    SnapshotModel.aggregate_uuid
                    == str(aggregate_uuid)
                )
                .order_by(
                    SnapshotModel.version.desc()
                )
                .limit(1)
                .one()
            )
        except exc.NoResultFound:
            events = events_query.all()
        else:
            # for this to work, snapshot has to
            # be created AFTER the last command
            newer_events = events_query.filter(
                EventModel.version
                > latest_snapshot.version
            )
            events = [
                latest_snapshot
            ] + newer_events.all()

        if not events:
            raise NotFound
```

```python
        aggregate_uuid = events[0].aggregate_uuid
        aggregate_version = events[-1].version
        events_objects = [
            self._to_event_object(model)
            for model in events
        ]

        return EventStream(
            aggregate_uuid,
            events_objects,
            aggregate_version,
        )
```

In this simple example, snapshots were kept in the same table as events. In contrast to events, snapshots can be deleted. Actually, they are fully disposable since one can always take another one. Hence, it would make sense to keep them in a separate table and keep just one, the latest snapshot of any *Aggregate*.

The only remaining question is when and how to create snapshots? There are several options. One might generate them asynchronously by looking for *Aggregates* that have more than N events newer than its latest snapshot. Another option is to put the logic of snapshotting in a *Repository* or *Event Store*. In the latter case, we would have to change the signature of `append_to_stream` method, so it accepts an entire *Aggregate*.

For example, we might take a snapshot of an *Aggregate* every time version is a multiplication of 100:

```python
class OrdersRepository:
    ...

    def save(self, order: Order) -> None:
        changes = order.changes
        self._event_store.append_to_stream(
            order.changes
        )
        if changes.expected_version % 100 == 0:
            self._event_store.save_snapshot(
                order.uuid, order.take_snapshot()
            )
```

Above snippet uses a new method of `EventStore` which will store our snapshot:

```
class EventStore(abc.ABC):
    ...

    @abc.abstractmethod
    def save_snapshot(
        self,
        aggregate_uuid: UUID,
        snapshot: Event,
    ) -> None:
        pass
```

The corresponding concrete implementation is fairly simple; we just save our special `Event` subclass to a designated table:

```
class PostgreSQLEventStore(EventStore):
    ...

    def save_snapshot(
        self,
        aggregate_uuid: UUID,
        snapshot: Event,
    ) -> None:
        self._session.connection().execute(
            SnapshotModel.__table__.insert().values(
                uuid=str(uuid4()),
                aggregate_uuid=str(
                    aggregate_uuid
                ),
                name=snapshot.__class__.__name__,
                data=snapshot.as_dict(),
                created_at=snapshot.created_at,
                version=snapshot.version,
            )
        )
```

Be aware that this implementation of snapshots creating and storing does certain assumptions. For example, the version on the latest snapshot is always equal to version of the newest event in `AggregateChanges`. This is reflected also in code for `load_stream` method that has to have a way to order snapshot and events correctly to successfully restore *Aggregate*'s state.

PROJECTIONS

Even though having a complete history may be invaluable, querying data or displaying anything to an end-user would be a nightmare if all we had were bare events. Since the latter is a source of truth, we may use them to generate tailored read models - so-called projections. A projection is a super simple concept - it can be as simple as a function that takes an event stream and produces a document meant to be displayed.

First, let's imagine we have a following projection of our `Order` *Aggregates*:

```
class OrderProjection(Base):
    __tablename__ = "projection_orders"
    uuid = Column(
        postgresql.UUID, primary_key=True
    )
    version = Column(BigInteger)

    customer_id = Column(Integer)
    status = Column(VARCHAR(64))
    total_quantity = Column(Integer)
    lines = Column(postgresql.JSON)
    updated_at = Column(DateTime(timezone=True))
```

Then, we need actual projection function which will iterate over *Events* from the stream and appropriately create/update the flattened view of our *Aggregate*:

```
def project_order(
    session: Session,
    aggregate_uuid: UUID,
    events: List[Event],
) → None:
    def _update_version_and_ts(
        projection: OrderProjection, event: Event
    ) → None:
        projection.version = event.version
        projection.updated_at = event.created_at

    @singledispatch
    def project(event: Event) → None:
        raise ValueError(
            f"Unknown event: {type(event)}"
        )
```

```python
@project.register
def _(event: OrderDrafted) → None:
    projection = OrderProjection(
        uuid=str(aggregate_uuid),
        customer_id=event.customer_id,
        status="NEW",
        total_quantity=0,
        lines={},
    )
    _update_version_and_ts(projection, event)
    session.add(projection)
    session.flush()

@project.register
def _(event: OrderConfirmed) → None:
    projection = session.query(
        OrderProjection
    ).get(str(aggregate_uuid))
    projection.status = "CONFIRMED"
    _update_version_and_ts(projection, event)
    session.flush()

@project.register
def _(event: NewProductAdded) → None:
    projection = session.query(
        OrderProjection
    ).get(str(aggregate_uuid))
    projection.lines = {
        **projection.lines,
        **{
            str(
                event.product_id
            ): event.quantity
        },
    }
    projection.total_quantity += (
        event.quantity
    )
    _update_version_and_ts(projection, event)
    session.flush()
@project.register
def _(
    event: ProductQuantityIncreased,
) -> None:
    projection = session.query(
        OrderProjection
    ).get(str(aggregate_uuid))
    lines_idx = str(event.product_id)
    projection.lines = {
        **projection.lines,
        **{
            lines_idx: projection.lines[
                lines_idx
            ]
            + event.quantity
        },
    }
    projection.total_quantity += (
        event.quantity
    )
    _update_version_and_ts(projection, event)
    session.flush()

for event in events:
    project(event)
```

This implementation is the simplest example, tightly coupled with SQLAlchemy Core. One might also imagine splitting this into two stages. The first would be technology agnostic and would produce simple data structures, like dictionaries.

The second stage would be infrastructure-specific and would save calculated projections into designated places:

```
class AccountBalance(TypedDict):
    account_uuid: UUID
    balance: Decimal
```

```python
def project_account_balance(
    events: List[Event],
) -> AccountBalance:
    result = {
        "account_uuid": events[0].account_uuid,
        "balance": Decimal("0.00"),
    }
    for event in events:
        if isinstance(event, CashDeposited):
            result["balance"] += event.amount
        elif isinstance(event, CashWithdrawn):
            result["balance"] -= event.amount

    return result
```

Then, the infrastructure-specific counterpart of the projection has to persist such it. For example, if one was to use PostgreSQL, they could use UPSERT to update the read model conveniently.

The advantage of such a solution is that it increases unit-testability of a module it lives in. Assuming that the first stage (transforming events into simple data structures) is an application-specific thing, one could call *Use Cases* and test if projections change as expected.

Projections can be generated within the same transaction/process, but nothing stands against processing them in the background. That is even better in terms of scalability, though also means making friends with eventual consistency - read model data is not consistent with the write model.

Last but not least, projections are disposable just like snapshots. We should design them in such a way that it should be possible to regenerate it without extra fuss.

EVENT SOURCING VERSUS A MODULAR APPLICATION

EVENT SOURCING IS A PRIVATE DETAIL OF A MODULE (INCLUDING TESTING)

Knowledge about using Event Sourcing should not leak outside of a module that uses it. No code outside a module should know about Event Sourcing events. Additionally, one should be able to unit-test such a module as a whole without inspecting what ES events were generated. For testing, we have virtually the same possibilities like for non-ES module - we call its *Use Cases* and check the behavior of the module. The biggest advantage of ES module is that we can treat projections as part of the module's API if only we split them into the two-phase process as it was described a few paragraphs earlier.

We still may be checking for events when we write tests at the level of *Aggregates*, though.

NEED INTEGRATION? USE DOMAIN EVENTS ALONGSIDE EVENT SOURCING EVENTS

Domain Events may be used in combination with Event Sourcing modules. If there is an event that should trigger action in other modules, we may use *Domain* Events in our *Aggregates* as an addition to ES events. The implementation may be the same as for *Entities* - *Aggregate* keeps *Domain Events* in its field (separately from ES events!). Later after saving an *Aggregate, Repository* picks up *Domain Events* and publishes them via *Event Bus*.

BIBLIOGRAPHY

Andrea Saltarello, Dino Esposito, Microsoft .NET: Architecting Applications for the Enterprise, Second Edition

Andrew Hunt, David Thomas, The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition, 2nd Edition

Bert Bates, Eric Freeman, Elisabeth Robson, Kathy Sierra, *Head First Design Patterns*

Bertrand Meyer, Object-Oriented Software Construction

Brian Foote, Joseph Yoder, *Big Ball of Mud* http://www.laputan.org/mud/mud.html#BigBallOfMud

Eric Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software

Fred Brooks, The Mythical Man-Month

Gerard Meszaros, xUnit Test Patterns: Refactoring Test Code 1st Edition

Gregor Hohpe, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions

Herberto Graça, *The Software Architecture Chronicles* https://herbertograca.com/category/development/series/the-software-architecture-chronicles/

Jorge Herrera, *Python microlibs* https://medium.com/@jherreras/python-microlibs-5be9461ad979

Joshua Bloch, Effective Java 3rd Edition

Kent Beck, Test Driven Development: By Example

Kirk Knoernschild, Java Application Architecture: Modularity Patterns with Examples Using OSGi

Nat Pryce, Steve Freeman, Growing Object-Oriented Software, Guided by Tests

Mark Seemann, Service Locator is an Anti-Pattern https://blog.ploeh.dk/2010/02/03/ServiceLocatorisanAnti-Pattern/

Martin Fowler, *Branch By Abstraction* https://martinfowler.com/bliki/BranchByAbstraction.html

Martin Fowler, Patterns of Enterprise Application Architecture

Martin Fowler, Refactoring: Improving the Design of Existing Code 2nd edition

Mary Poppendieck, Tom Poppendieck, Leading Lean Software Development: Results Are Not the Point

Matthias Noback, Object Design Style Guide

Miguel Grinberg, Flask Web Development, 2nd Edition

Mike Cohn, Succeeding with Agile: Software Development Using Scrum

Nick Chamberlain, Applying Domain-Driven Design with CQRS and Event Sourcing https://buildplease.com/products/fpc/

Olivier Libutzki, *Why Event Sourcing is a microservice communication anti-pattern* https://dev.to/olibutzki/why-event-sourcing-is-a-microservice-anti-pattern-3mcj

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

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

Sandi Metz, 99 Bottles of OOP: A Practical Guide to Object-Oriented Design

Thomas J. McCabe, *A Complexity Measure* http://www.literateprogramming.com/mccabe.pdf

Vaughn Vernon, Implementing Domain-Driven Design

Vaughn Vernon, Domain-Driven Design Distilled