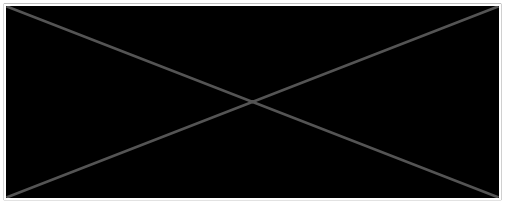
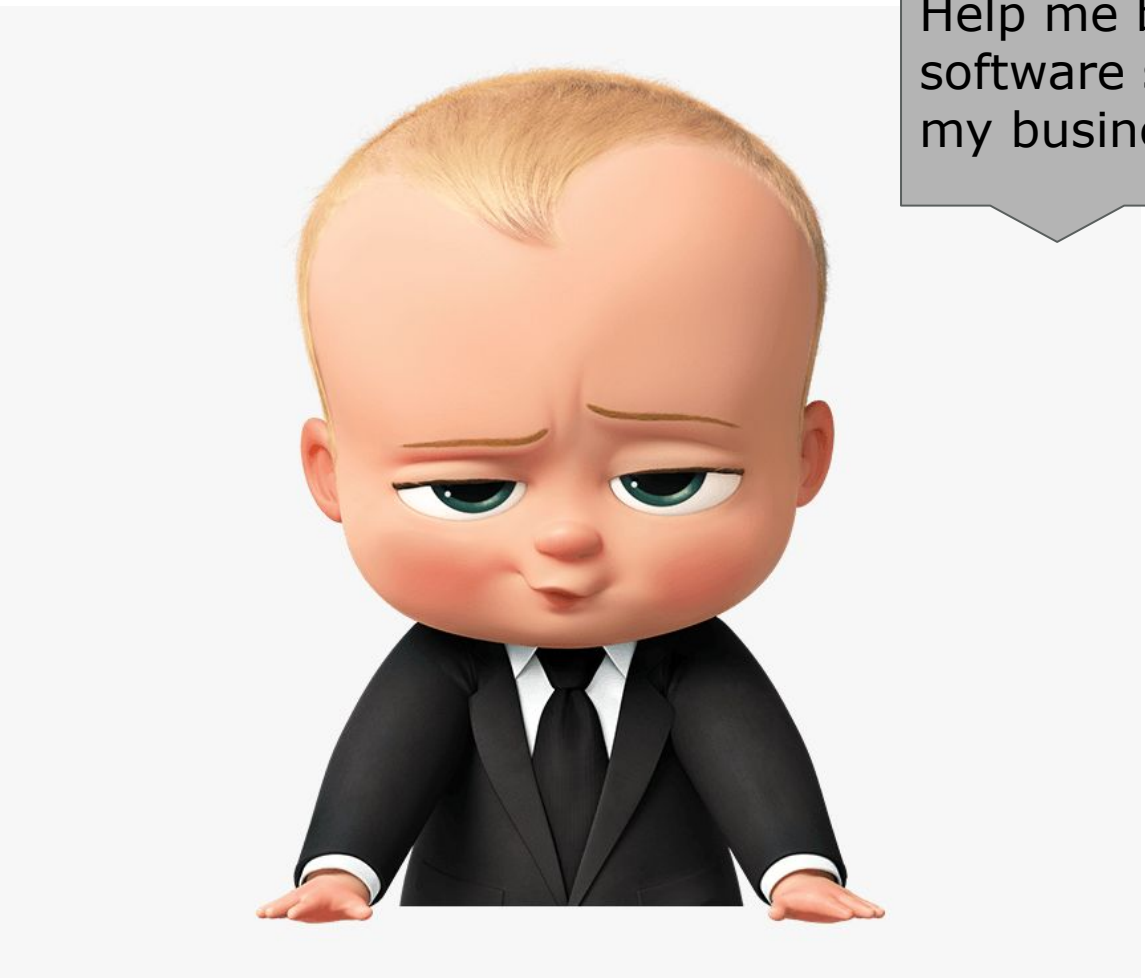
Programming 6

Recap DDD



Recap



Help me build a software system for my business!

Recap

How complex is your domain? Maybe we should approach this strategically.

Sound great, make it happen!

Recap

Software Entropy

Tackling the complexity in the heart of the software

Domain Driven Design

- Organisational boundaries
- Software already in place
- Ubiquitous language
- Business Decisions/clustering

Strategic design
Multiple subdomains

- Core *
- Supporting
- Generic

Context mapping

- Published language
- Shared kernel
- ACL
- Open/Host Service

Solution space
Tactical Design
Bounded Contexts

- Value Objects
- Entities
- Aggregates
- ….

Recap

Entity

- unique identifier
- state
- behavior

e.g: Person

```
public static void main(String[] args) {
    Person jeffrey = Person.born( name: "Jeffrey");
    Person sarah = Person.born( name: "Sarah");
    jeffrey.ages( years: 20);
    sarah.ages( years: 20);

    jeffrey.marry(sarah);

    System.out.println(jeffrey);
    System.out.println(sarah);
}
```

```
Person{name='Jeffrey', age=20, spouse=Sarah}
Person{name='Sarah', age=20, spouse=Jeffrey}
```

Value Object

- immutable
- state

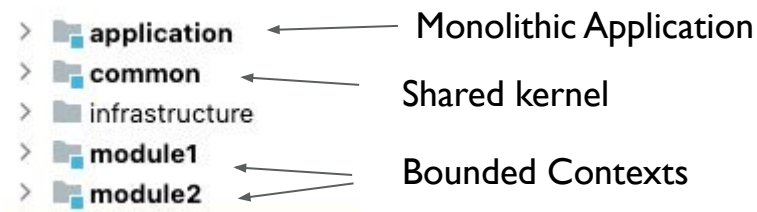
e.g: Address

Aggregate

- consistent boundary
- Root entity
- Referenced external

e.g: Person, Family

Modular Monolith



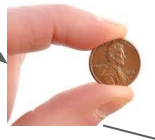
Recap

Let’s dig a little deeper in an Entity

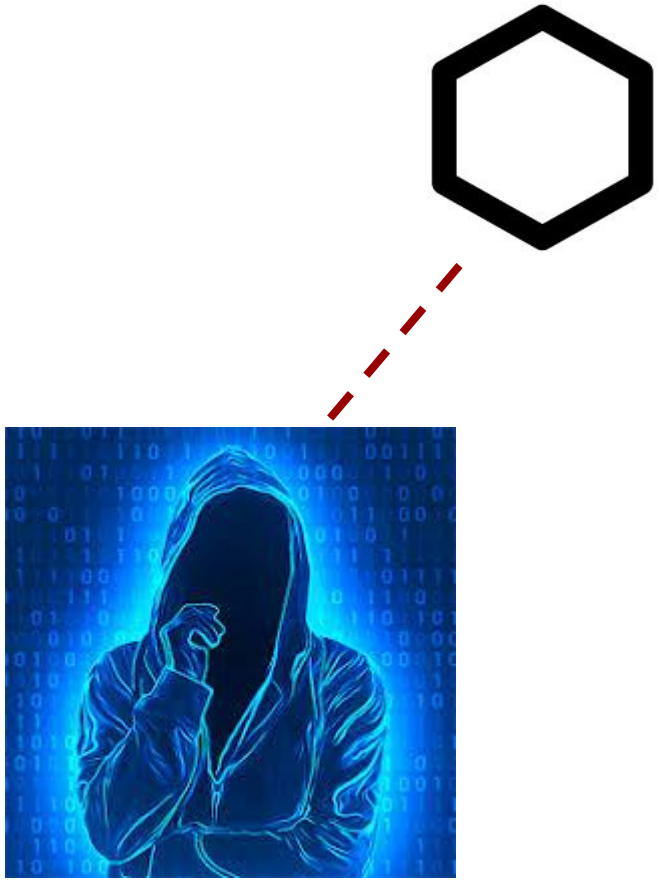
A person is an entity.

- Has a unique identifier
- Has a natural identifier
- An address
- A name
- Might be married -> behaviour!

Putting it all together



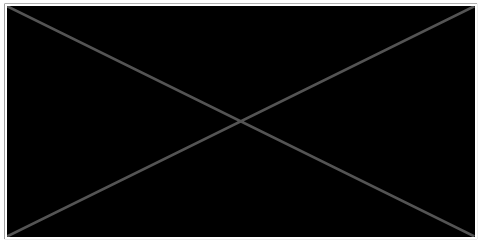
Recap



All great and all, but how are you going to implement this?

Programming 6

Hexagonal Architecture



Architecture goals

- Make it easy to develop
- Make it easy to deploy
- Make it easy to configure
- Make it easy to modify
- Make it easy to understand

Are these really the goals?

- Software is expensive!
  - Software will need to be changed

Ultimate goal:

Minimize the total cost of ownership of the software!

The goal!

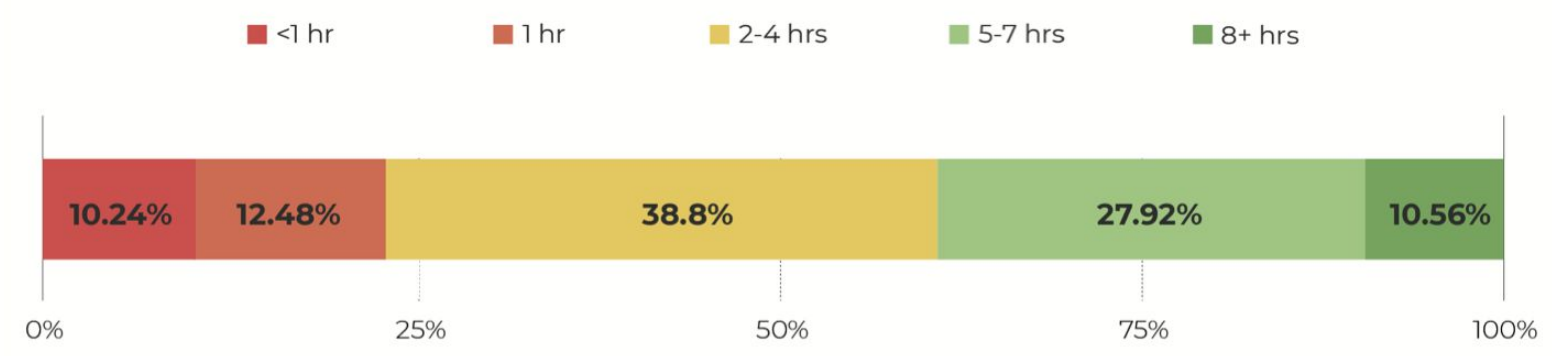
ActiveState

Developer

Survey

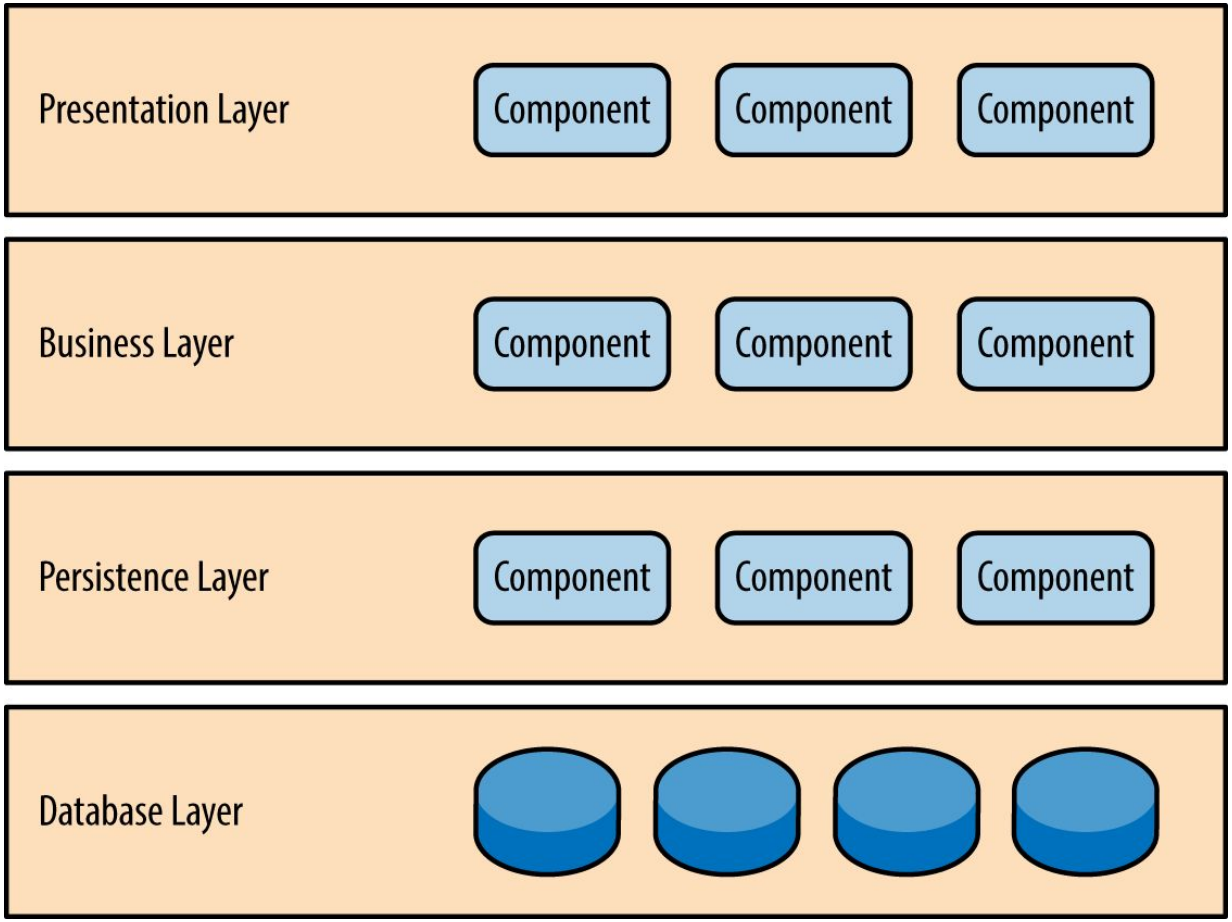
```
// Dear programmer:
// When I wrote this code, only god and
// I knew how it worked.
// Now, only god knows it!
//
// Therefore, if you are trying to optimize
// this routine and it fails (most surely),
// please increase this counter as a
// warning for the next person:
//
// total_hours_wasted_here = 254
//
```

Programming Hours



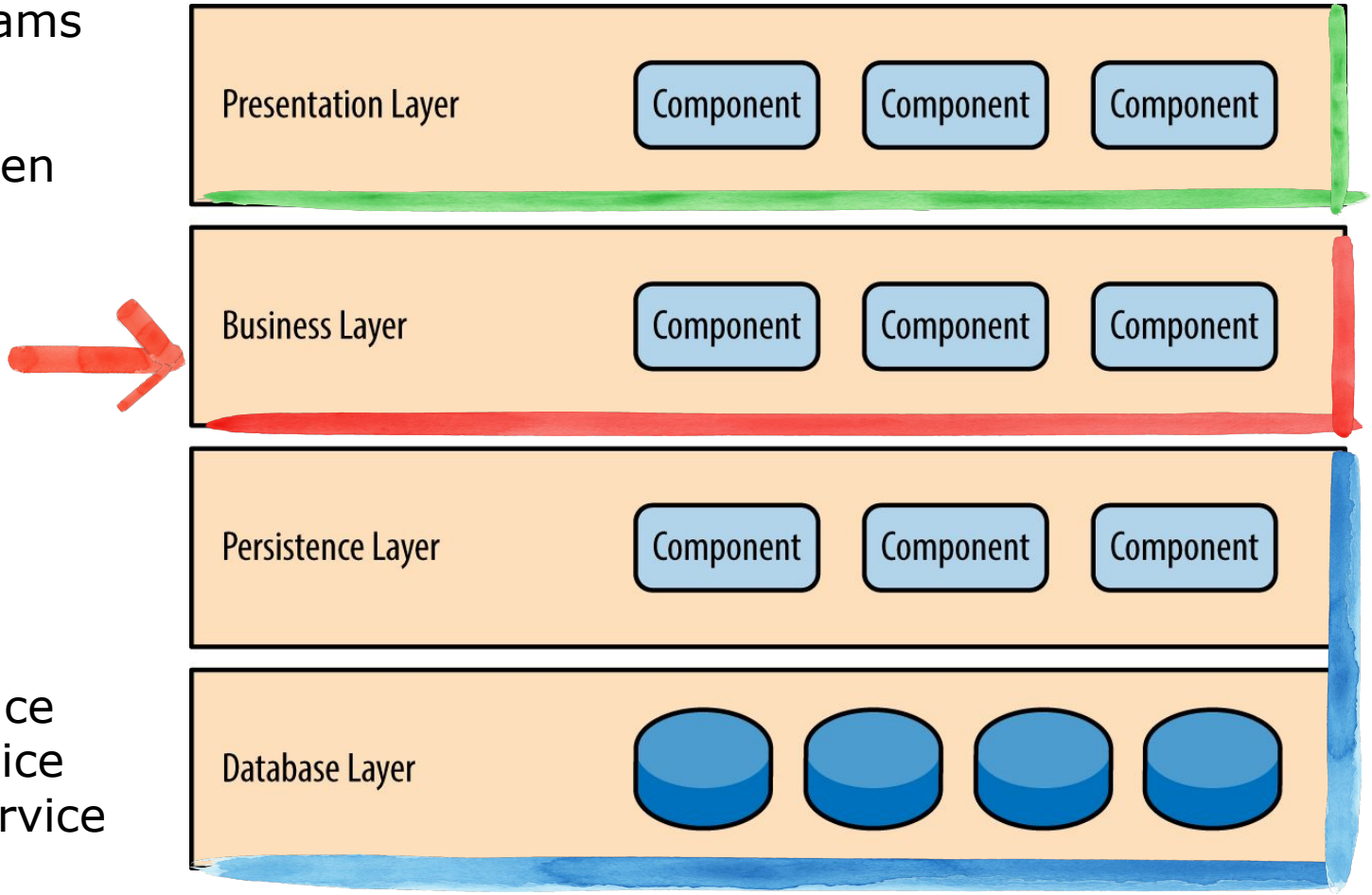
Disclaimer: Layered architecture

It’s a solid pattern (pun intended).



Critique of a layered architecture

- Hidden functionality
  - Partitioning based on actors
- Working in teams
  - Conflicts
  - Merge hell
- Database driven design
- **S**.o.l.**I**.d



e.g.:

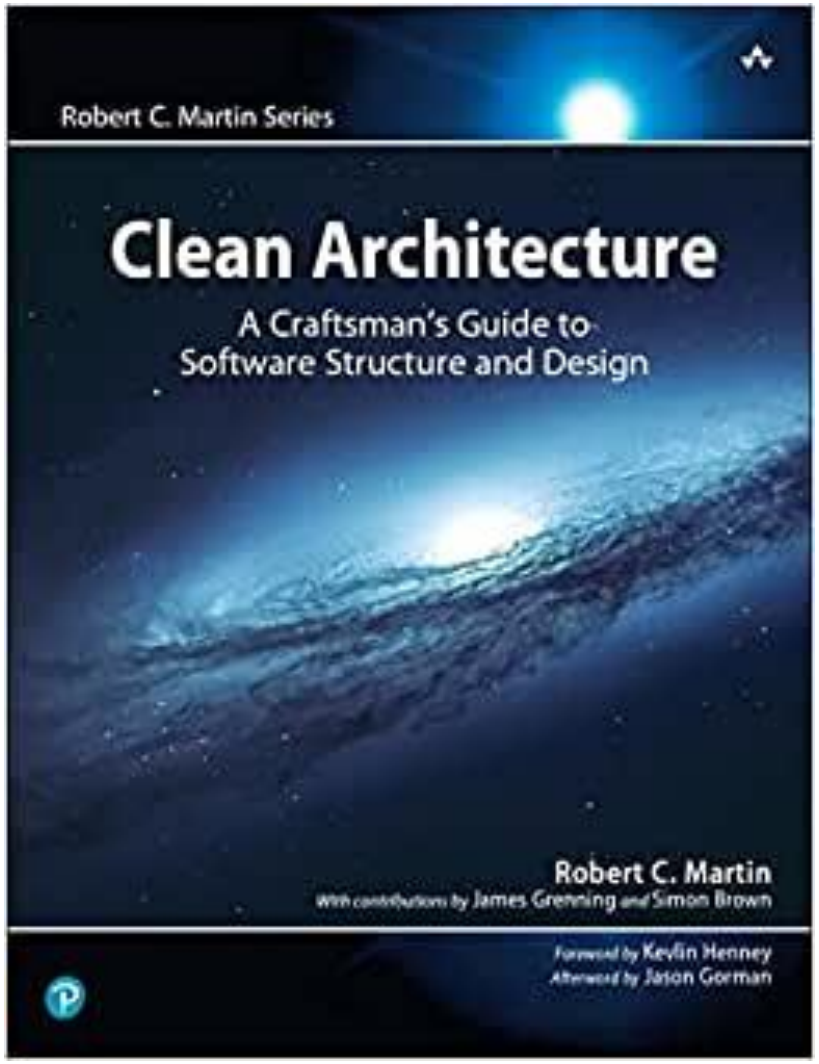
- Rejected invoice
- Accepted invoice
  InvoiceService

Drivers

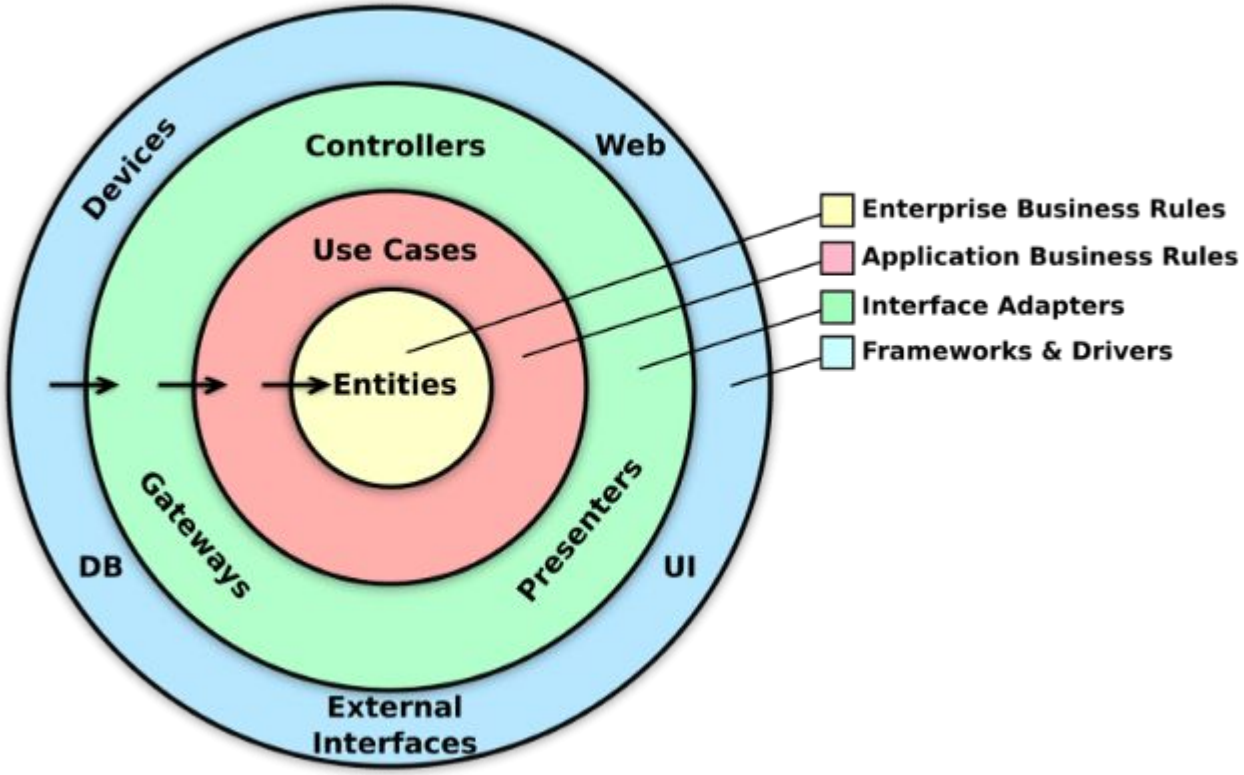
Uncle Bob - Clean Architecture

- Treat the database as an external resource
- Treat the web as an external resource

  …..
- Keep every framework at an arm length!
- The domain logic is the core

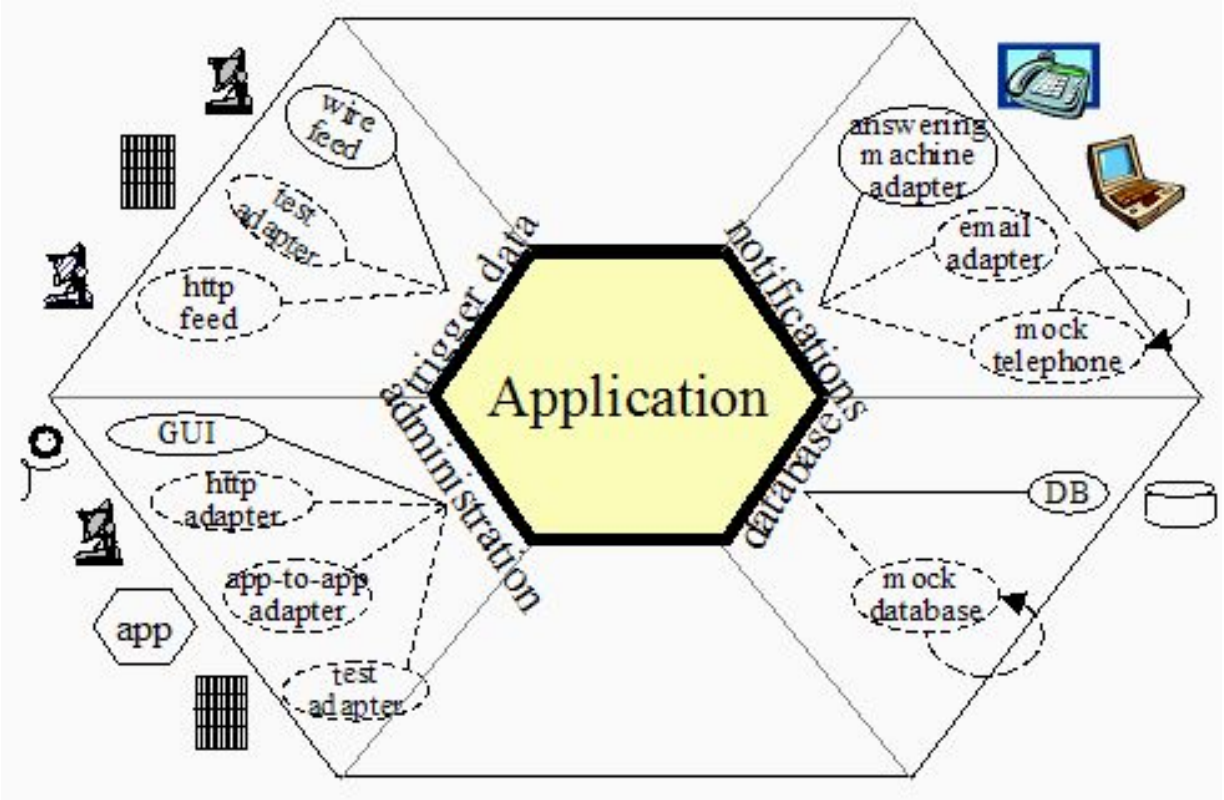


A clean architecture

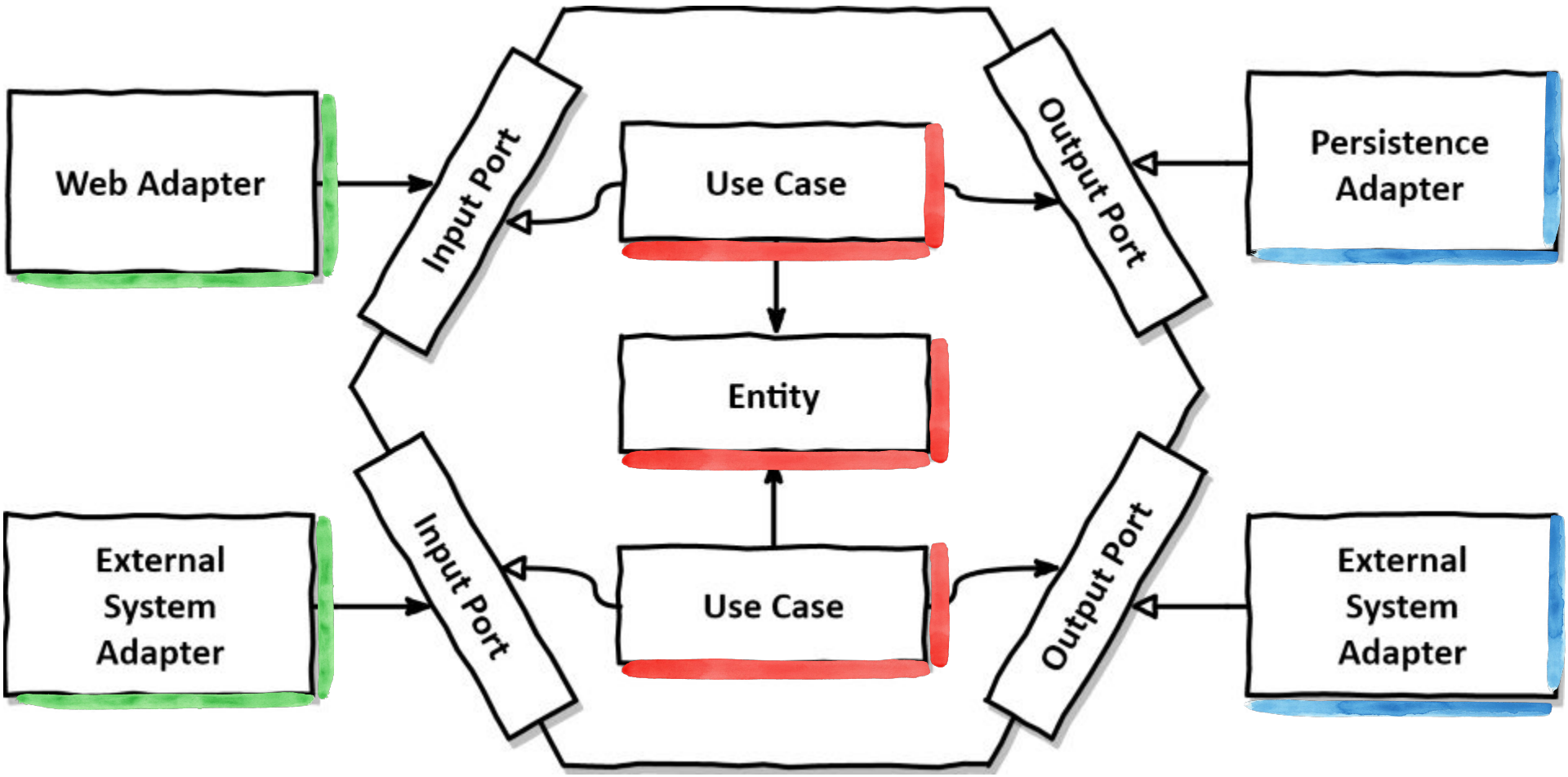


Enter: Hexagonal Architecture

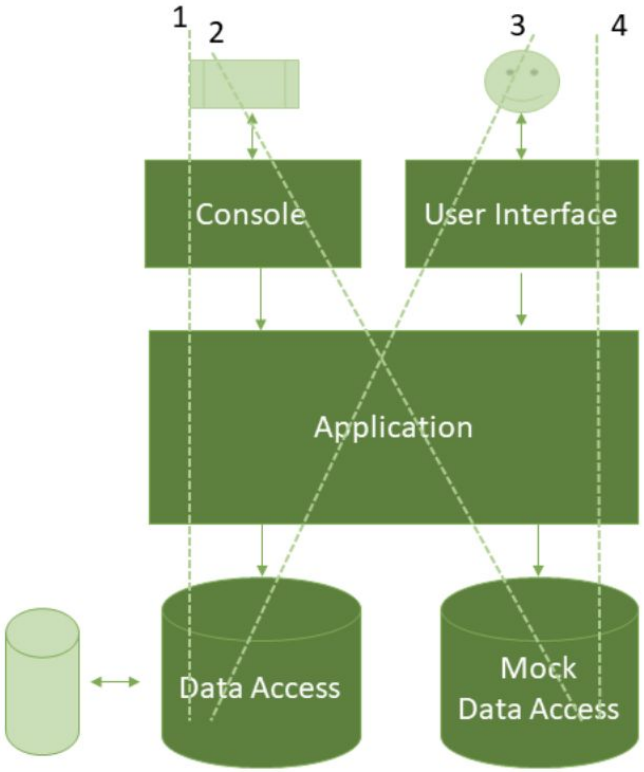
- Very abstract, how on earth can we implement this?
- Layered architecture
- Onion architecture
- OR..
- Alistair Cockburn - Paper about hexagonal Architecture
- https://alistair.cockburn.us/hexagonal-architecture/



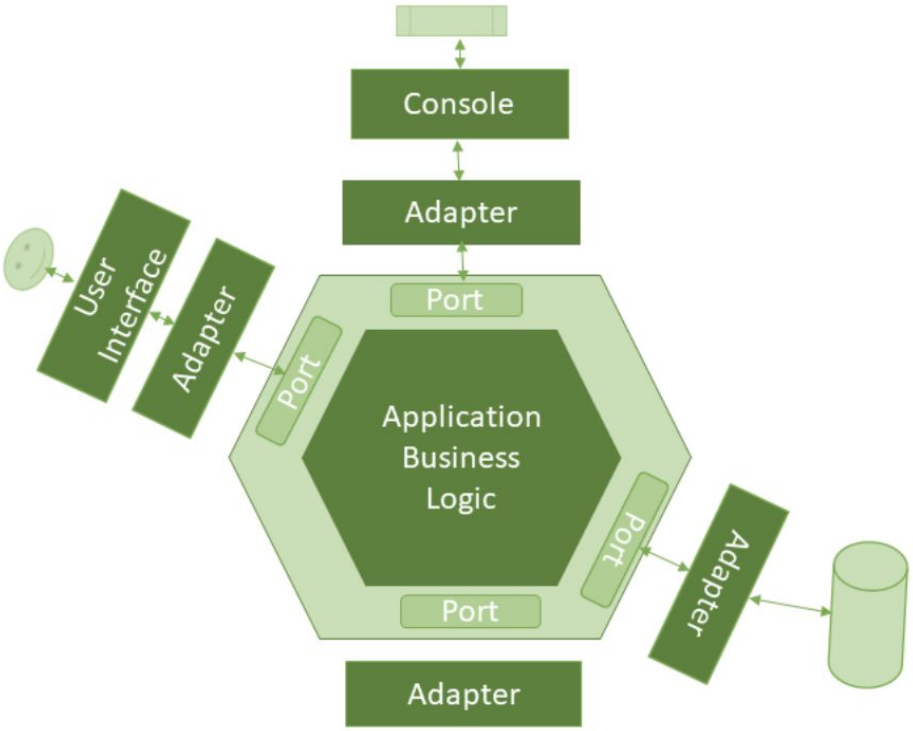
Hexagonal Architecture



Layered vs Hexagonal architecture



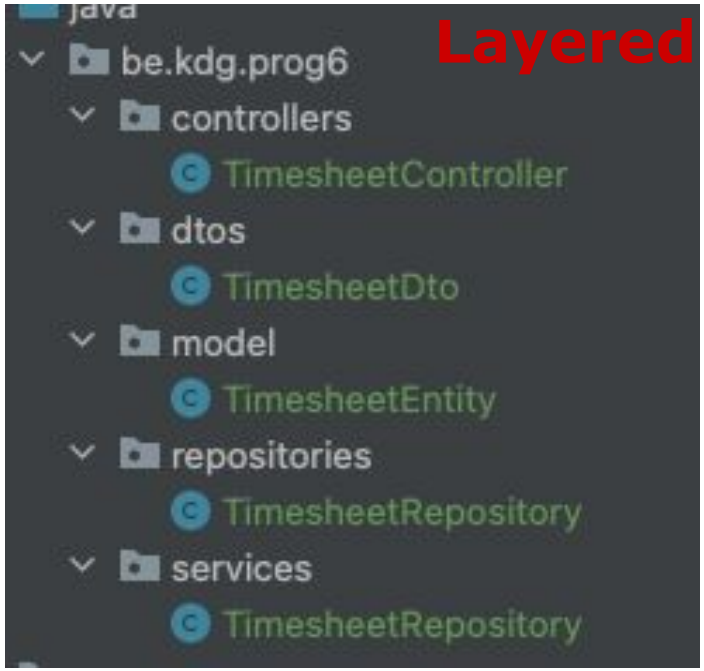
Three Layer Architectural drawing



Hexagonal Architectural drawing

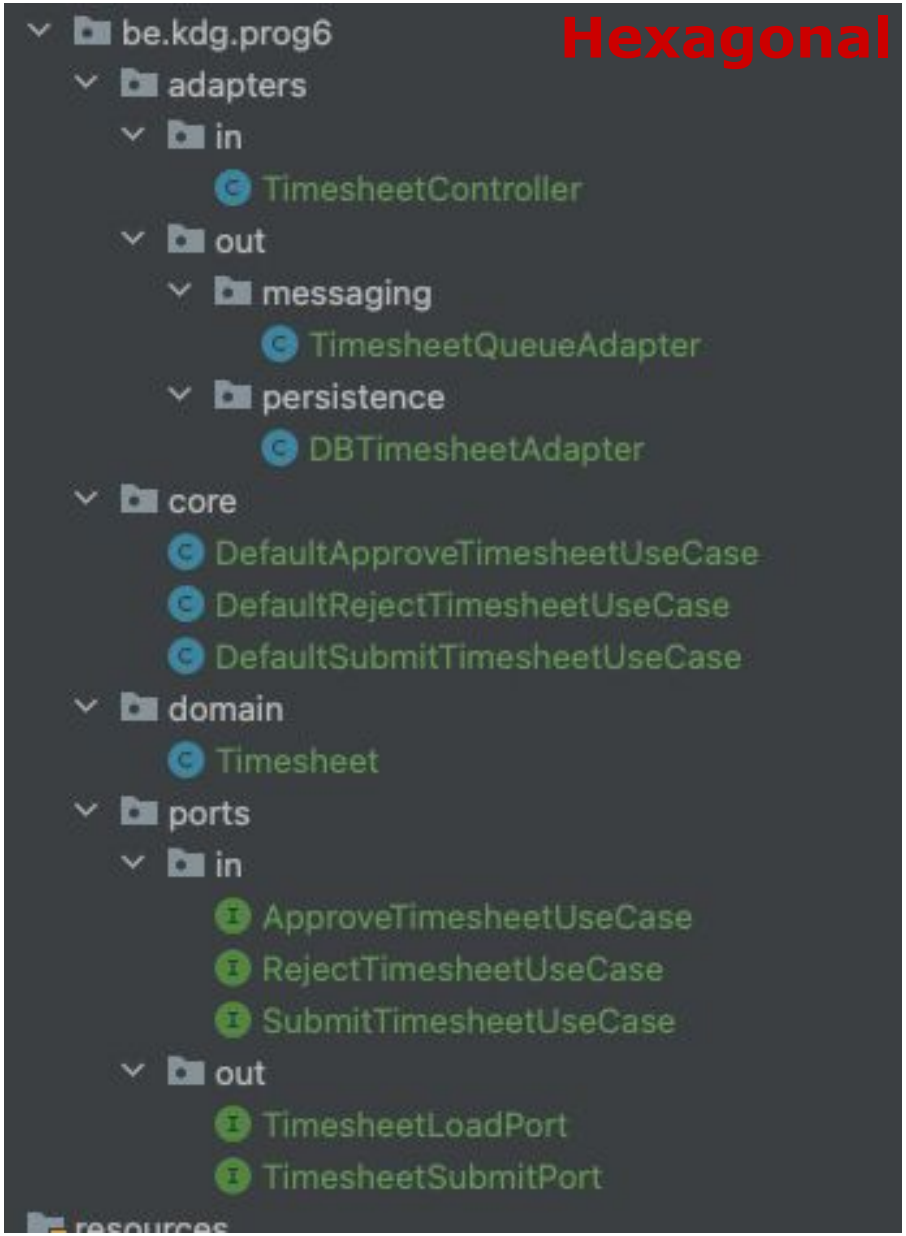
Timesheet app

What does it do?



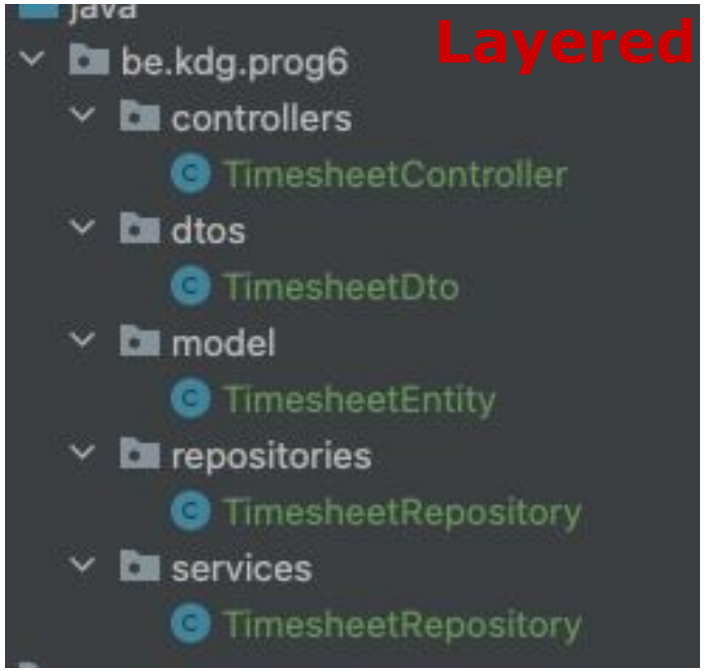
Timesheet app

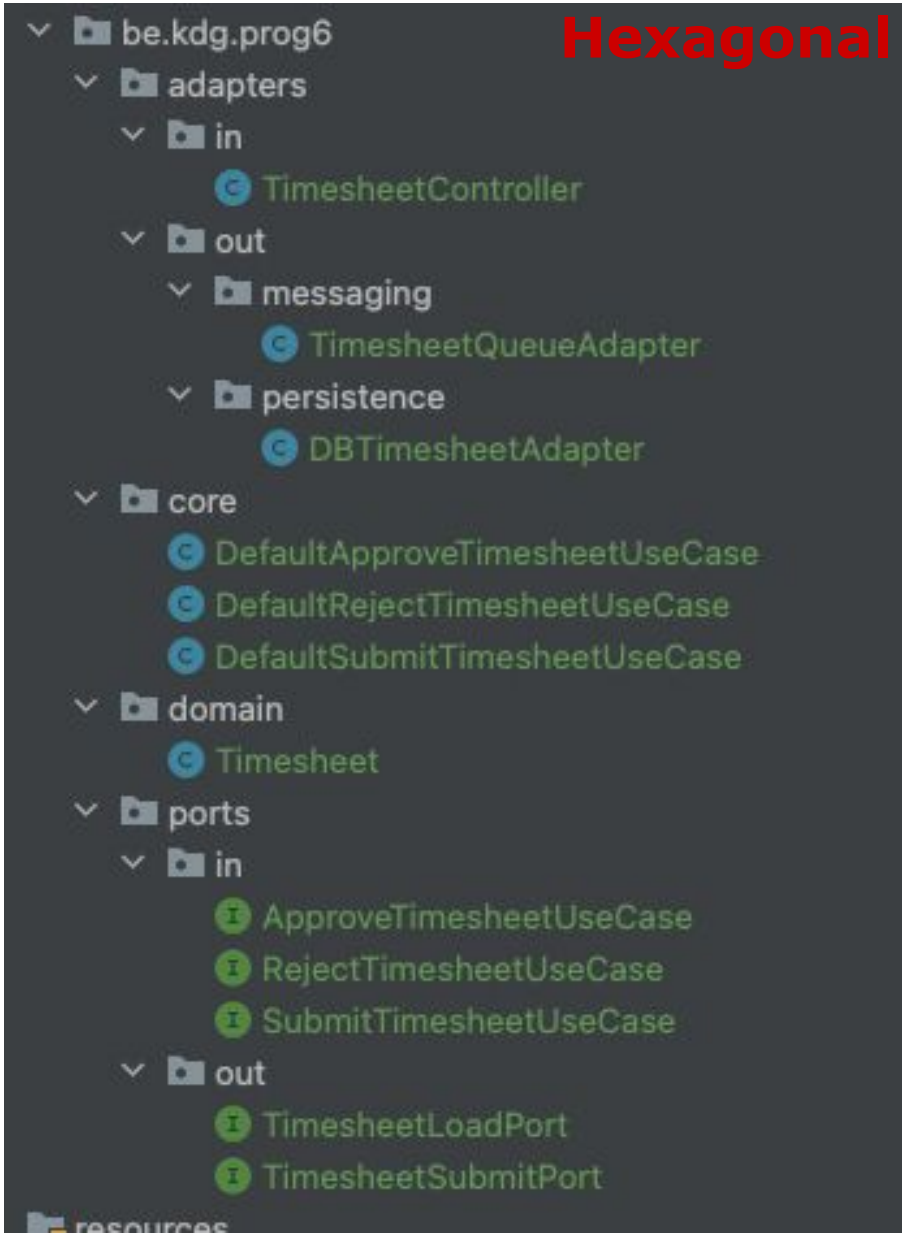
What does it do?



Timesheet app

What does it do?



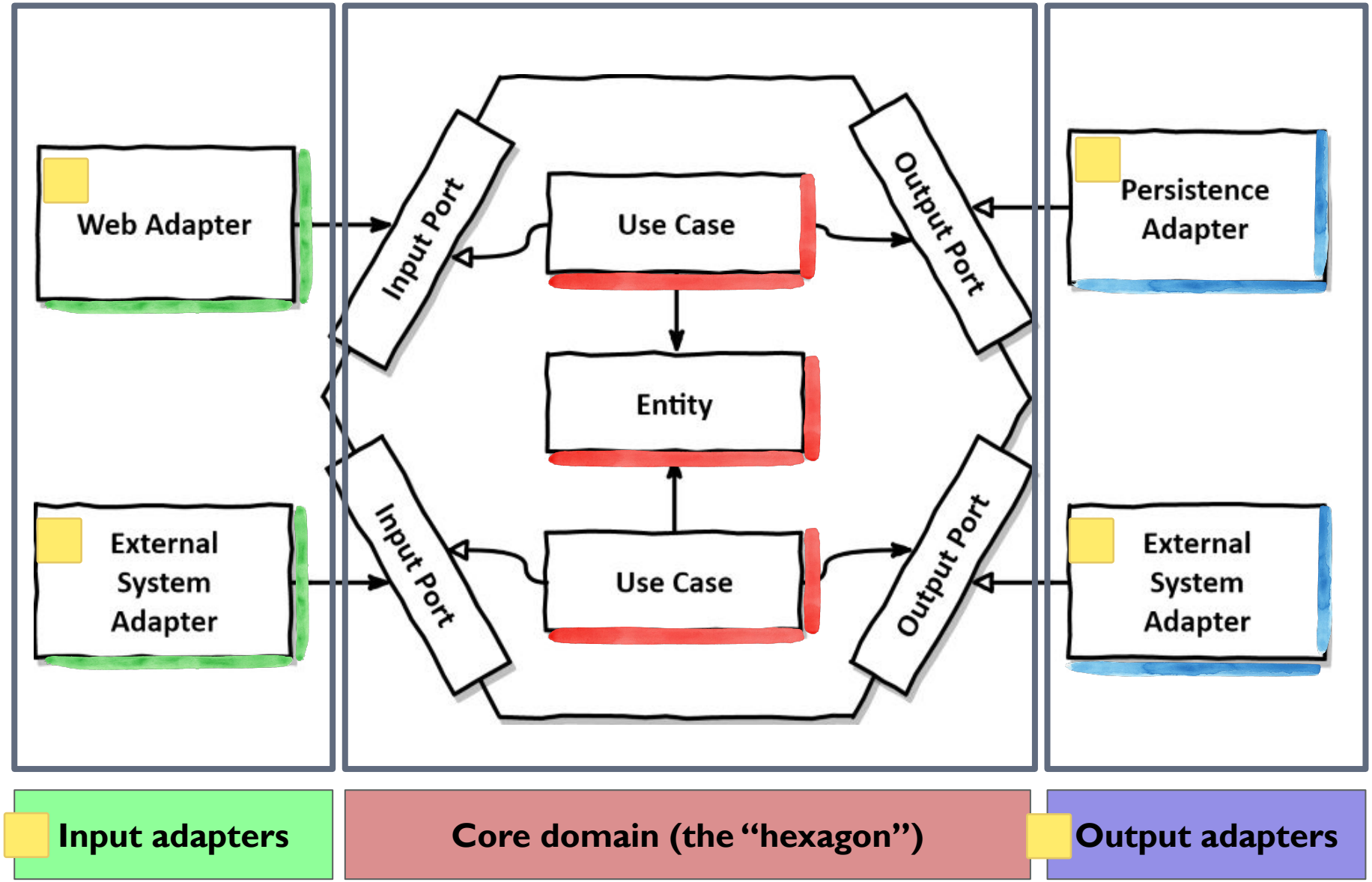


Pros and cons

- Ideal for complexer domains (large enterprise apps)
- Very expressive
- Infrastructure agnostic
- Easier to test
- Easy to evolve the architecture
- Easier to work with in larger teams
- Highly maintainable
- Clean code/clean architecture

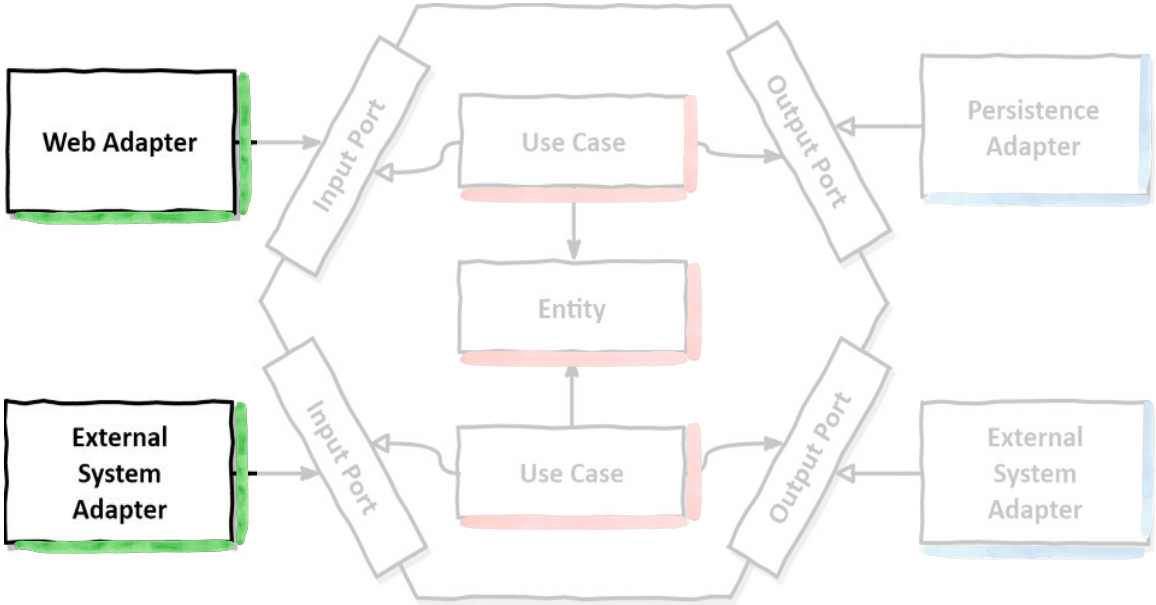
- Can become complex
- Cost to build abstractions
- YAGNI
- Mapping hell
- Rich model is preferable although anemic model also possible
- How many ports and adapters?
- Lots of variations of hexagonal architecture

Ubiquitous language of Hexagonal Architecture



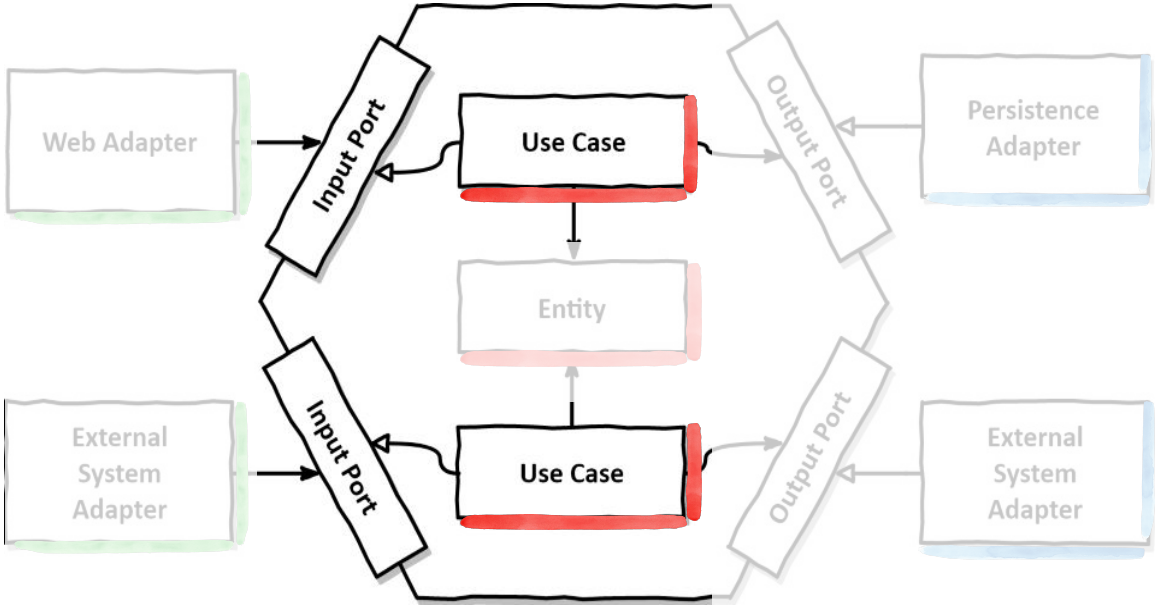
Ubiquitous language of Hexagonal Architecture

- Input adapter
  - Technology **specific** entry point
  - Could be an (MVC) controller, messaging queue listener, command line interface, etc.
  - Calls an input port



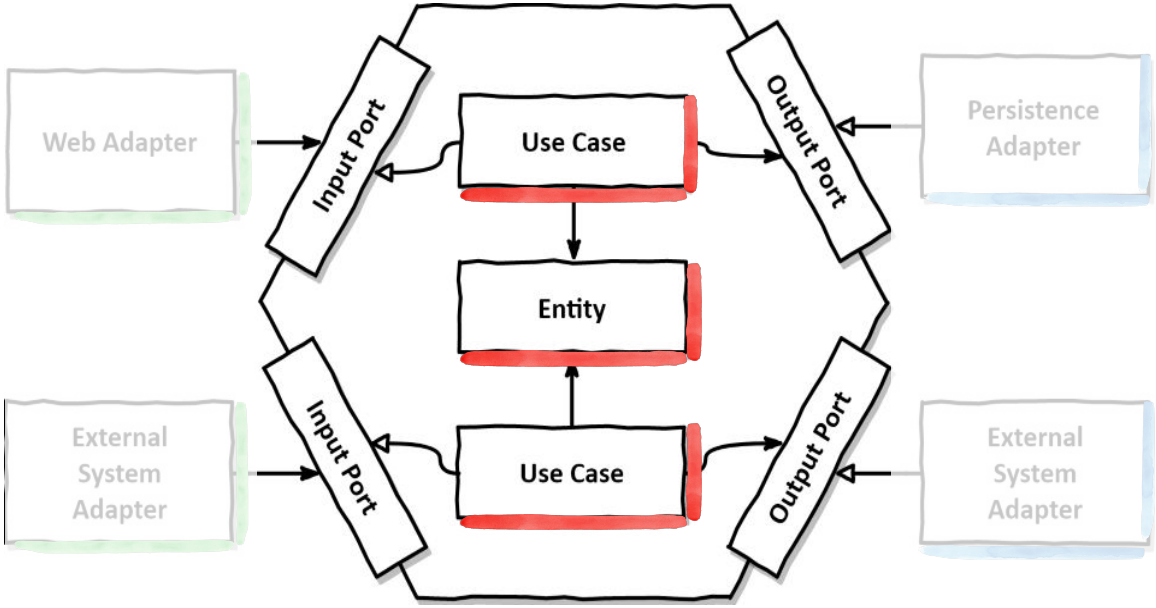
Ubiquitous language of Hexagonal Architecture

- Input port
  - Part of the core domain
  - Technology **agnostic** entry point for business functionality
  - Always an interface
  - The implementation? A **use case**



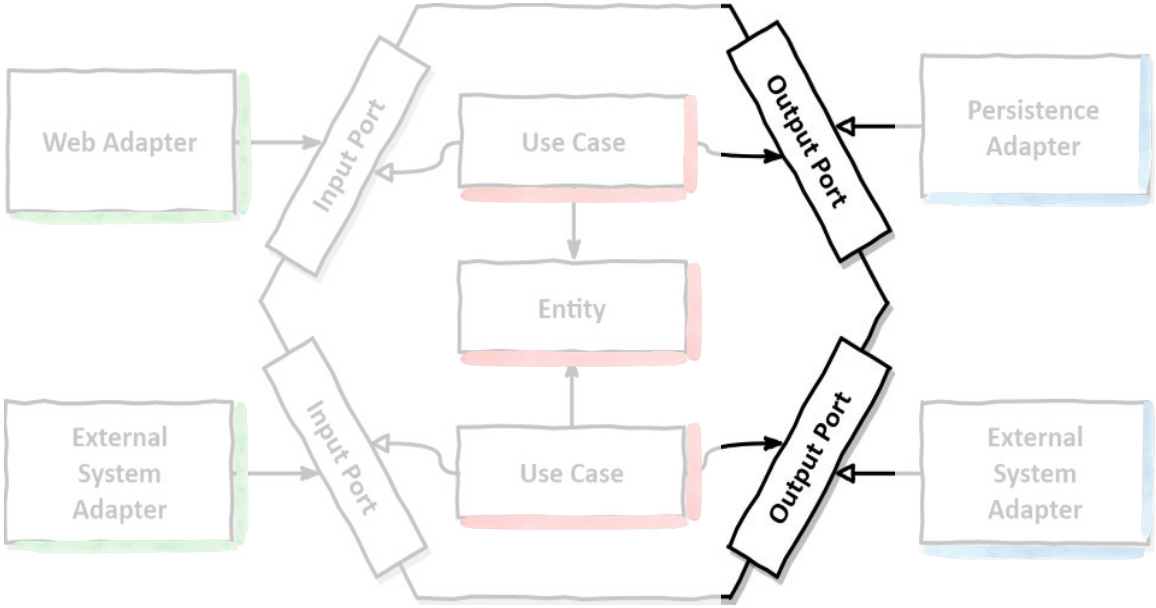
Ubiquitous language of Hexagonal Architecture

- Core domain
  - Application services that implement the actual functionality: use cases
  - A use case **implements** an input port
  - In a domain driven design the domain rules will go in the domain.
    - In the diagram below this is an "Entity" (not JPA entity).
  - These use cases will always call another output port, **never** another input port!



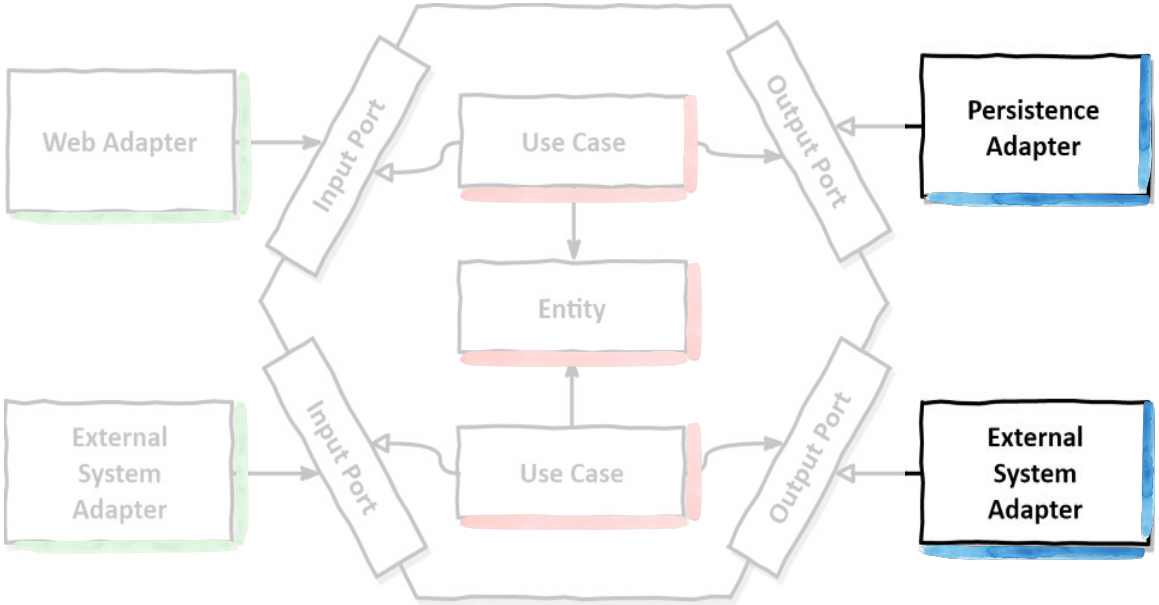
Ubiquitous language of Hexagonal Architecture

- Output port
  - Part of the core domain
  - Hides infrastructure from the use cases (the core domain)
  - Always an interface
  - The implementation? **<u>An output adapter</u>**



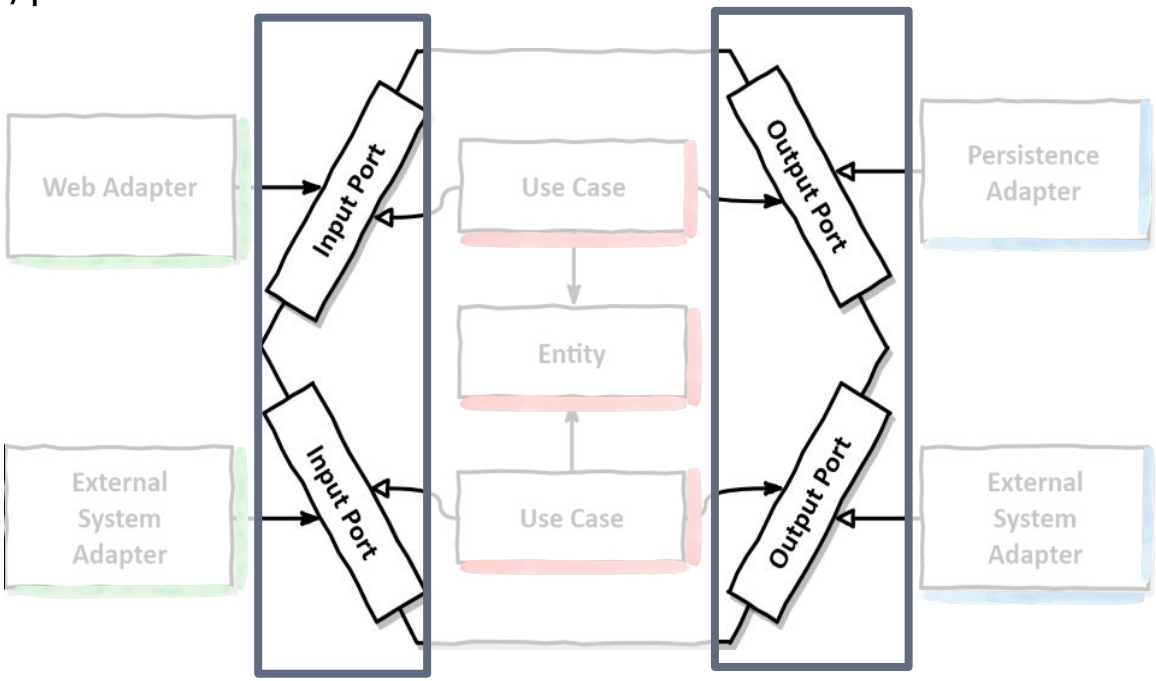
Ubiquitous language of Hexagonal Architecture

- Output adapter
  - Technology specific communication with “a device” (database, etc.)
  - Output adapters always implement an output port
  - Output adapters never call another input adapter or output adapter



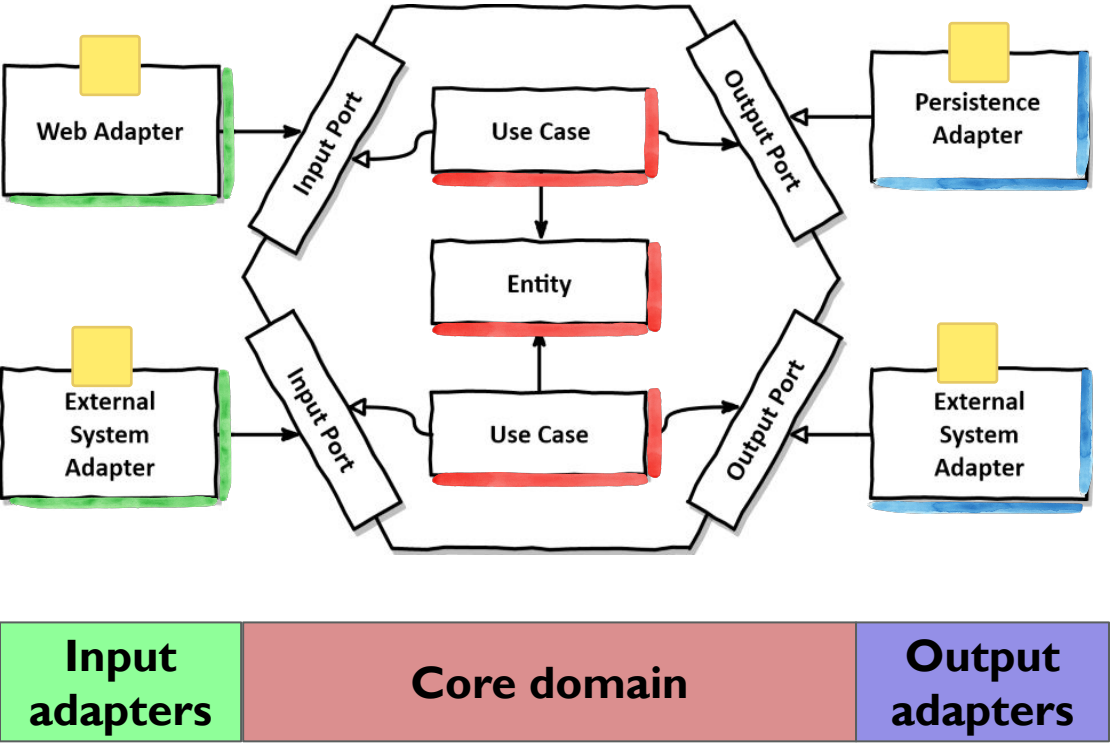
Ubiquitous language of Hexagonal Architecture

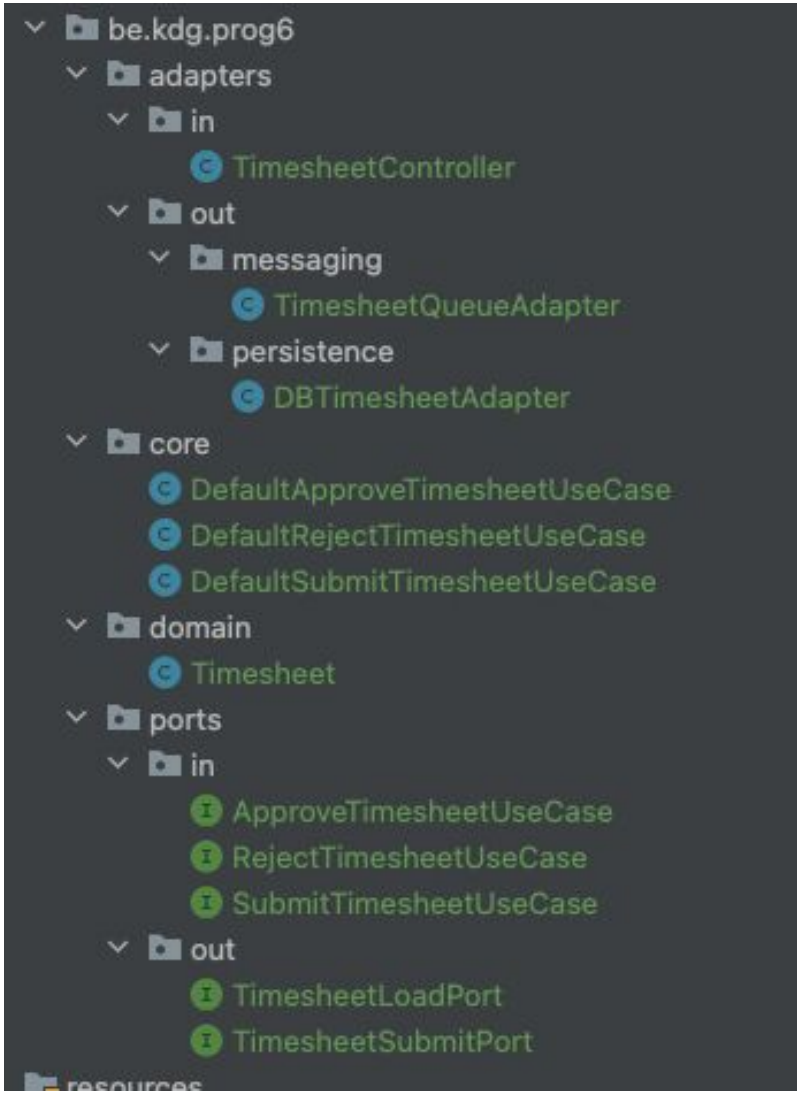
- Driving side
  - Alistair Cockburn calls these the primary adapters
  - These are on the left side, external actors will initiate the interactions with driving adapters only
  - "In(put)"-adapters/ports
- Driven side
  - Alistair Cockburn calls these the secondary adapters
  - Are under control of the application, or the applications steers the communication here.
  - "Out(put)"-adapters/ports



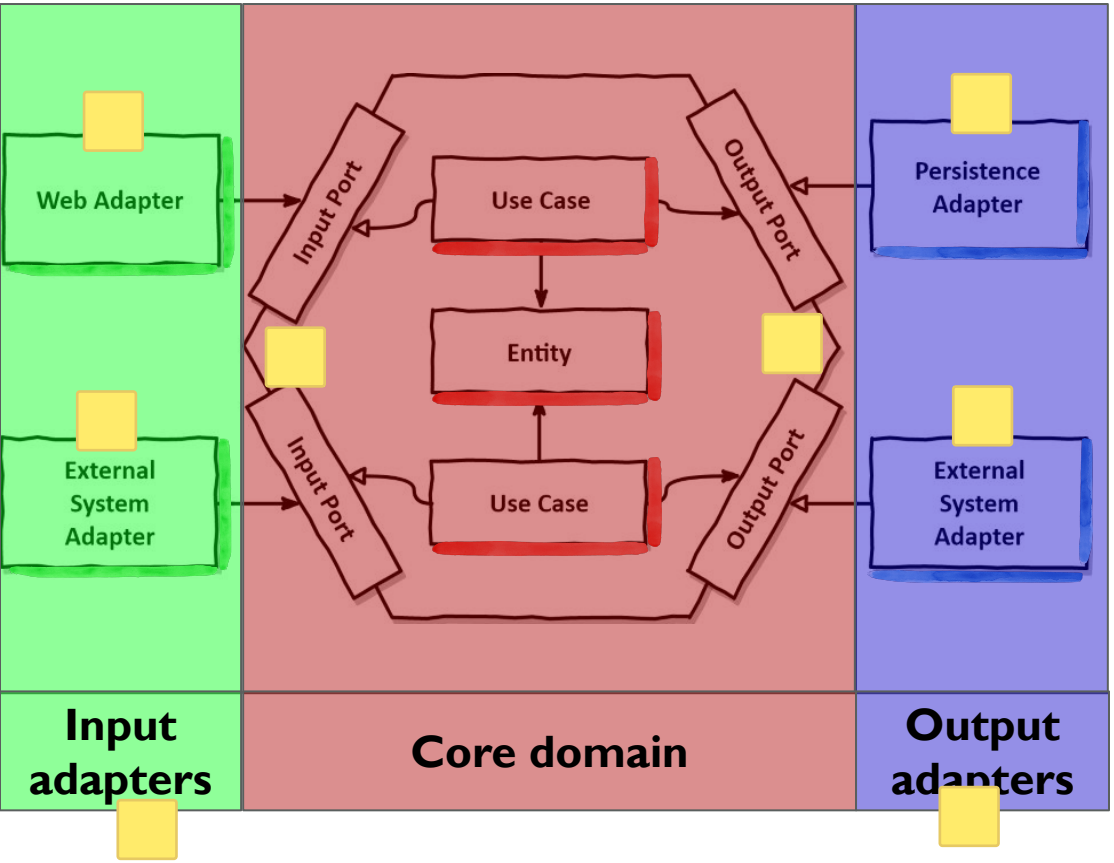
Informational!

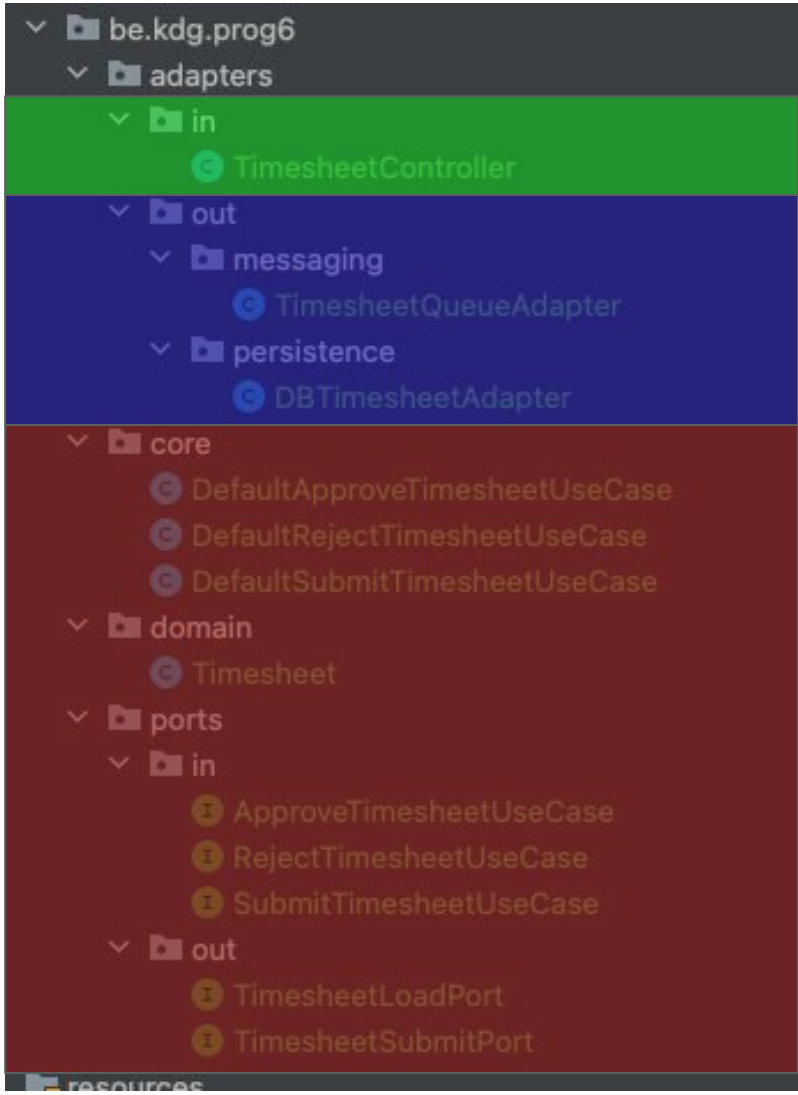
Timesheet app





Timesheet app



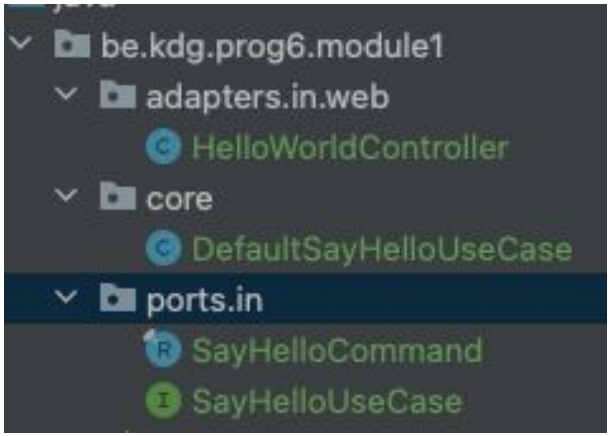


Next steps

- Web adapter
- Persistence adapter
- Other adapters (gateways)
- Mapping between layers
- Core
  - Use cases
  - Queries
- Messaging adapters
- Configuration
- Bounded Contexts

Web adapter

- Always an input adapter
- Gets adapter specific input
  - For web: HTTP message
- Maps this domain specific input to steer the use case
  - A “command”
- The command does the input validation
- The command is immutable
- Calls the input port (use case)




```java
@RestController
public class HelloWorldController {

    2 usages
    private final SayHelloUseCase sayHelloUseCase;

    public HelloWorldController(SayHelloUseCase sayHelloUseCase) {
        this.sayHelloUseCase = sayHelloUseCase;
    }

    @PostMapping("/hello/{name}")
    public String sayHello(@PathVariable String name){
        return sayHelloUseCase.sayHello(new SayHelloCommand(name));
    }
}
```

```java
5 usages
public record SayHelloCommand(String name){

    1 usage
    public SayHelloCommand{
        Objects.requireNonNull(name);
        StringUtils.hasText(name);
    }
}
```

Use Cases

Types of Use Cases.

- Query the domain
- Modify the domain
- Project data (explained later)

- A Use Case is part of the domain and can modify it
- It calls an output port if state has changed.
- We will end the class with a use case to make this clear.
- Each use case is an implementation of an **input port**
- *Strategy pattern can be used when for instance we have multiple use cases.*

A Use Case that only queries the data ends with Query. It is a good practice to separate the Queries from the actual Use Cases as it facilitates concepts like CQRS (explained later).

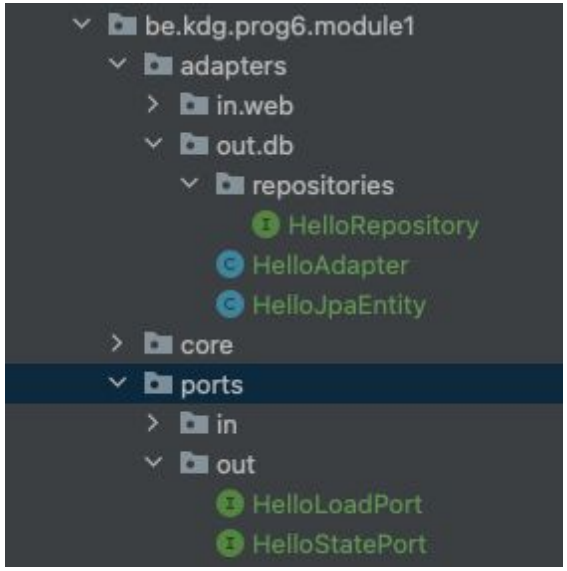
Database/Persistence Adapter

A database adapter implements at least one output port.

A database adapter is always an **out** adapter.

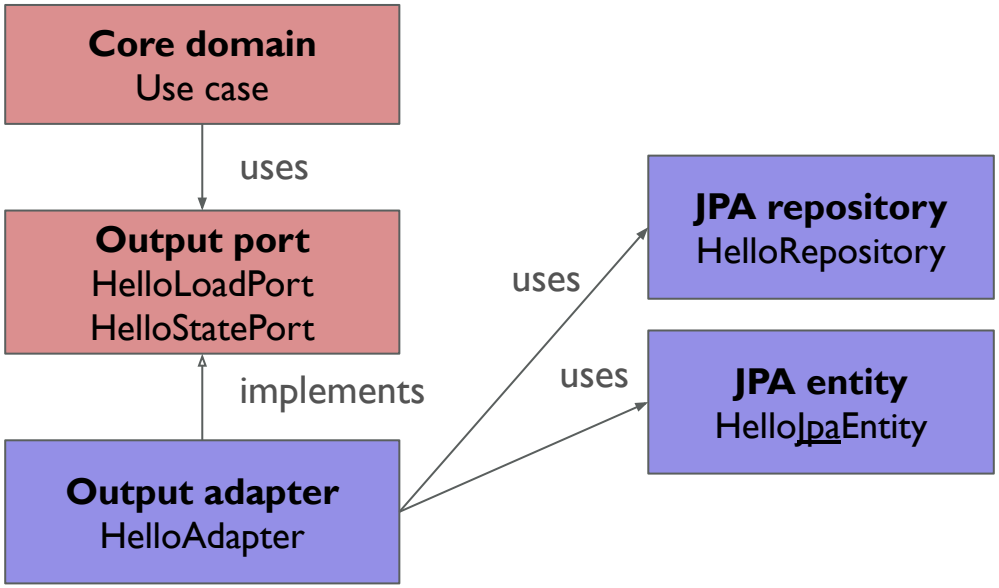
It will receive a domain object, map it to its necessary database layout and then save it.

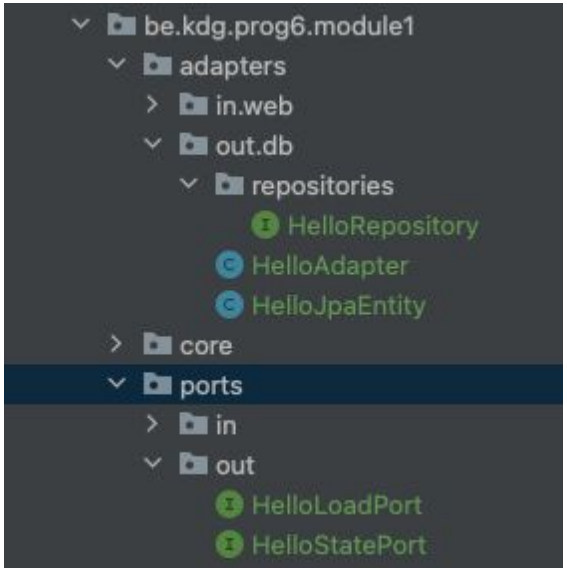
Slicing of out ports and in ports can be cumbersome, but remember the **I** in the SOLID acronym.



Database/Persistence Adapter

In the screenshot a Repository is a JPA interface, we inject this in the concrete implementation (output adapter) `HelloAdapter` that implements the necessary ports.





Important:

1. The core domain (use case and output port) does **<u>not</u>** know about JPA!
2. Most of the time you'll have 1 persistence adapter per entity (in DDD terms).

REST Adapter/Gateway

This is an **<u>out</u>** adapter that consumes an external rest service.

You can use `RestTemplate` in the adapter.

The port it implements doesn't really look different from e.g. a port that is implemented by a database adapter.

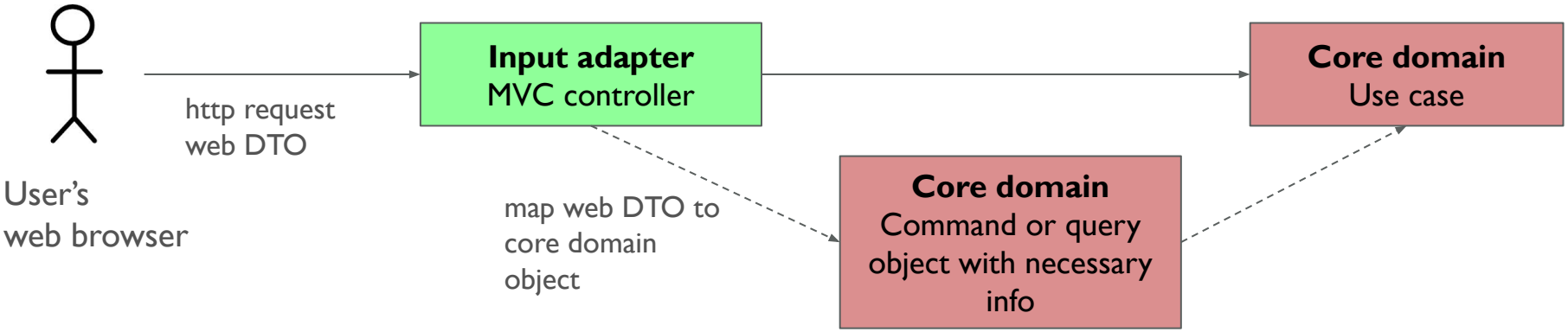
In this adapter you can build in resilience.

For instance, if the external Rest Service is down, you can build in a retry mechanism.

We will talk about resilience patterns in the next weeks.

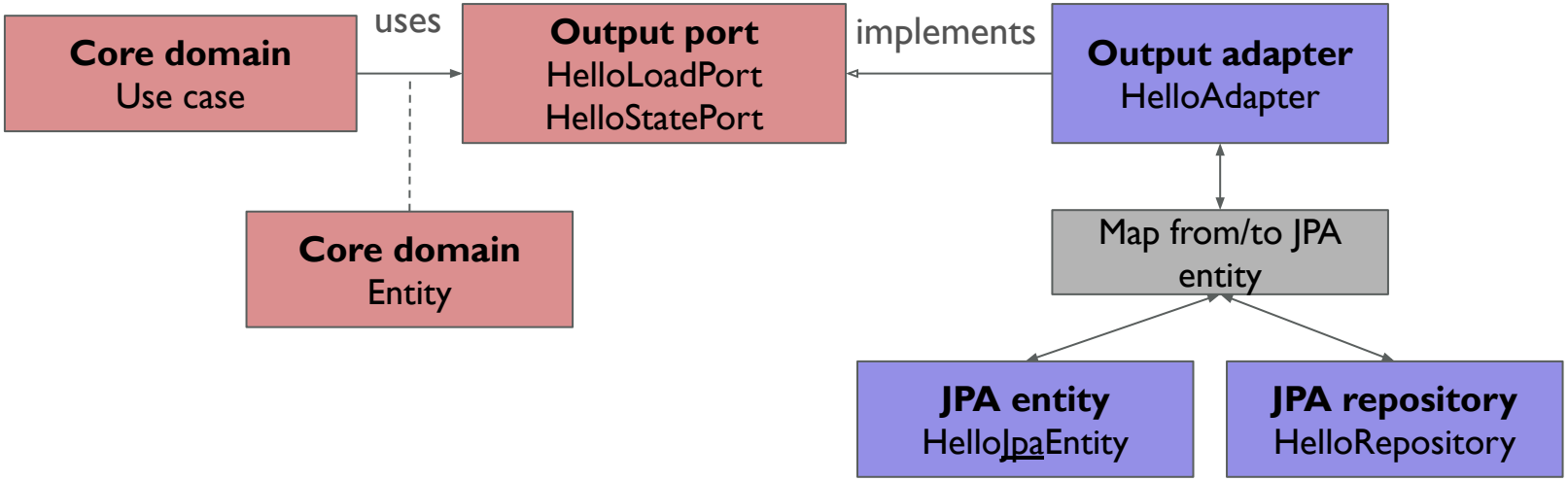
Communication: input adapter to core domain

- An input adapter will **<u>always</u>** communicate with the core domain with command or query objects, and those are also part of the core domain.
- If this is an HTTP POST with a Web DTO, this will need to be mapped to the command or query object.
  Do **not** pass Web DTOs to the core domain!
- There can be a lot of mapping going on…
- You can take shortcuts and make abstractions when needed, but take shortcuts consciously!
  - In order to facilitate this, you can have dedicated mappers in order to help you.
  - e.g.: mapstruct, dozer, modelmapper, selma, ...



Communication: core domain to output adapter

- The core domain needs to talk to the database somehow
- But it can't know the database exists!
- The use case uses an output port, which is database agnostic (so it doesn't know about JPA!) to make changes or query the database.
- The output adapter **always** accepts and/or returns a domain entity
- Only the adapter can know how to map to the technology-specific representation: mapped behind the scenes!



Communication (DDD)

- Commands and events is the way to communicate between Bounded Contexts.

- Commands are things that need to happen,
  events are things that happened and are considered facts.

- Commands convey **information and intent.**
  Events convey **information and context.**

- Events can be handled by all interested parties, while a command is sent to a specific consumer/handler.
  Commands can be refused!

- Sometimes events are considered "**thin**" or "**fat**"
  Both have advantages and disadvantages.
  Read more:
  https://codeopinion.com/thin-vs-fat-integration-events/

Command or event?

If there is a state change to an aggregate this will **always** result in an **event**!

Avoid generic events such as PersonUpdatedEvent instead use PersonMarriedEvent, PersonRetiredEvent…

If there is something technical that has to be done. Send an email, integrate with an external system (for example a payment engine such as Mollie or PayPal) this will **always** be **a command**!

Why? Because a payment engine cannot take the **business decision.**

Who encapsulates the **business capability** or who takes the business **decision**!

-> Is it you? Send command. (so called **process managers**)

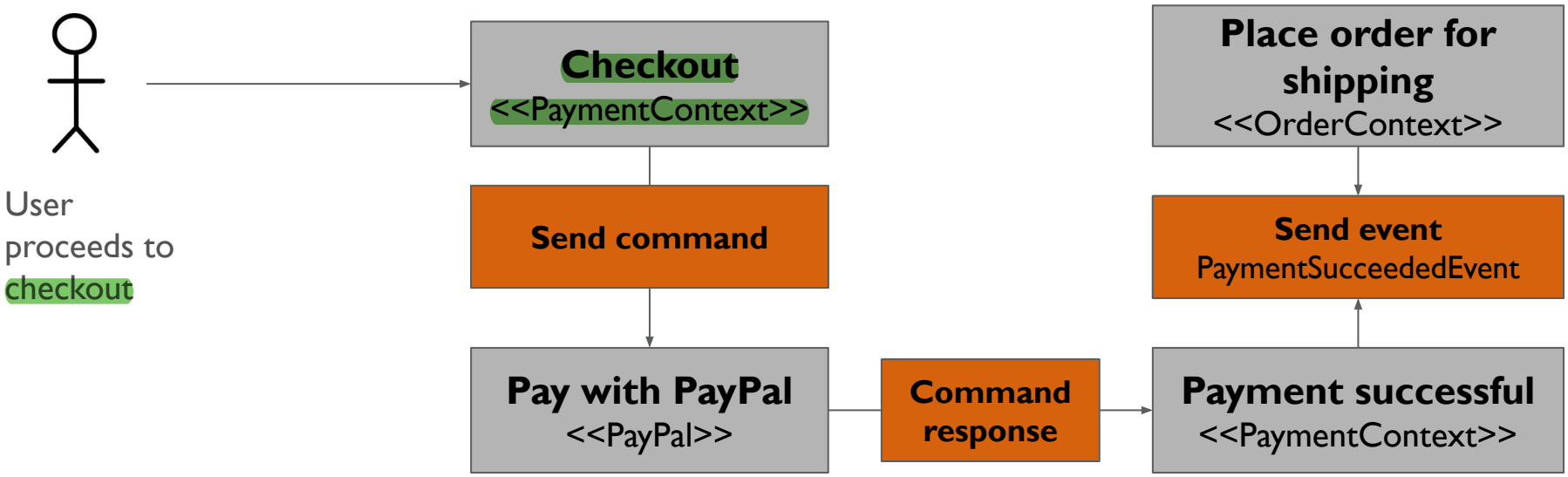
-> Is it not you? Somebody will need to react on an event.

Command or event?

Who encapsulates the **business capability** or who takes the business **decision**!

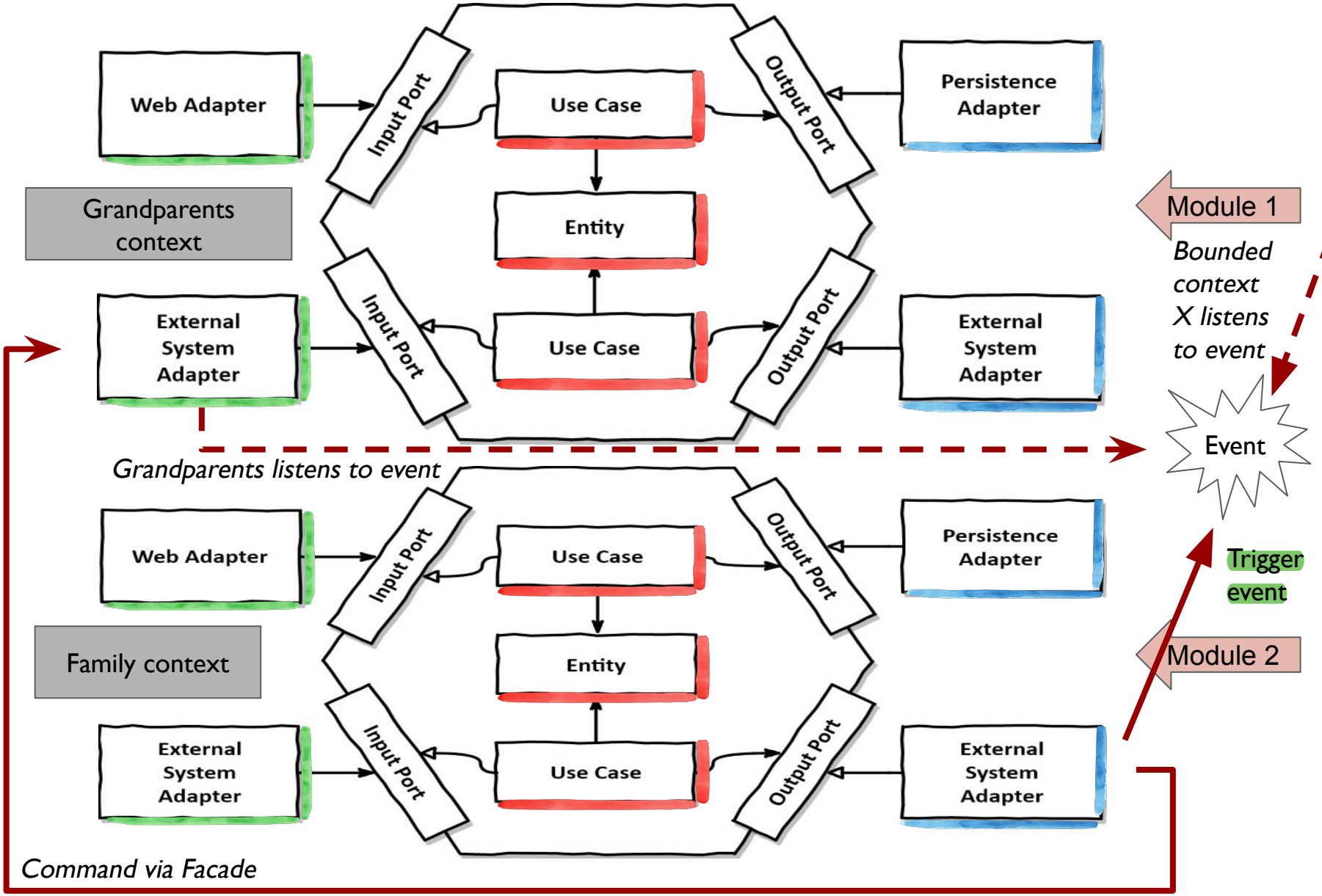
-> Is it you? Send command. (so called **process managers**)

-> Is it not you? Somebody will need to react on an event.



Command or event?

Communication across bounded contexts



Communication across bounded contexts

Events: Spring application events

Commands: called via a *Facade*

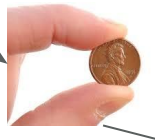
Where to put these?

In a monolithic application a *Shared Kernel* is the logical choice.

- When we send a command to another bounded context we will:
  - Create a "Facade" in the Shared Kernel
  - An output adapter from bounded context A will **call** this Facade
  - An input adapter in bounded context B will **implement** this Facade

- When we publish an event with Spring application events:
  - The domain event needs to be found by both bounded contexts and also put in the Shared Kernel.
  - Bounded context A will **publish** the event
  - Bounded context B will **subscribe** to this event with an event listener

Putting it all together



Questions?