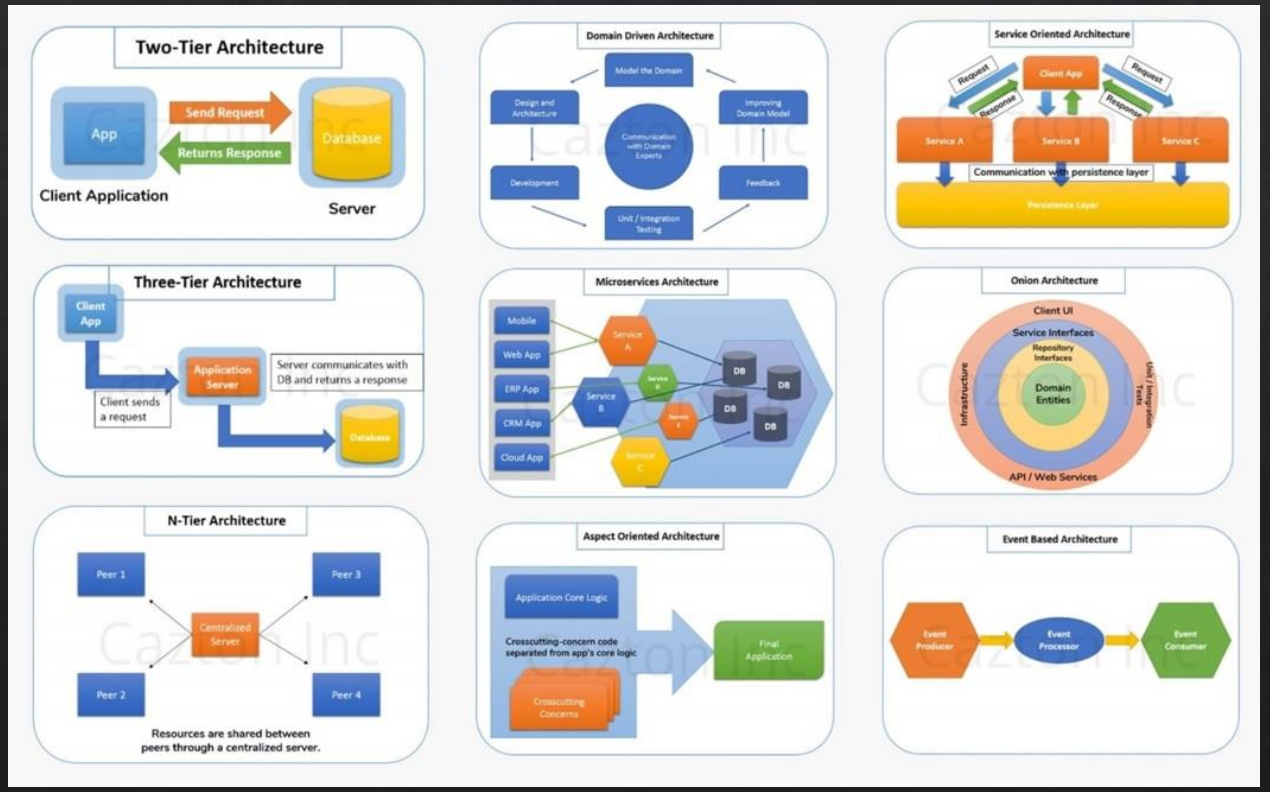
The clean architecture

Marcos Tobias Muñiz (uo270930)

Irene Bello Díaz (uo269570)

Ideas regarding the architecture of systems

- Hexagonal Architecture
- Onion Architecture
- Screaming Architecture
- …

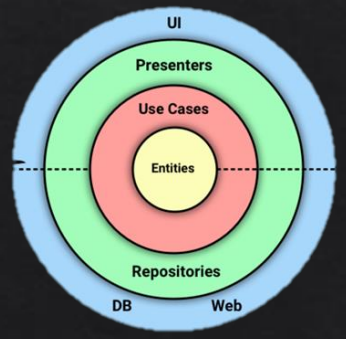


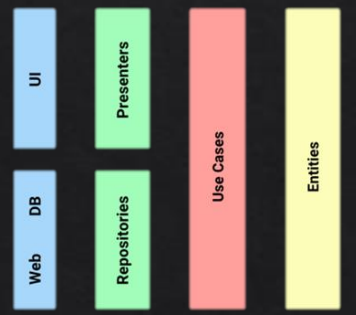
These all have something in common:

SEPARATION OF CONCERNS

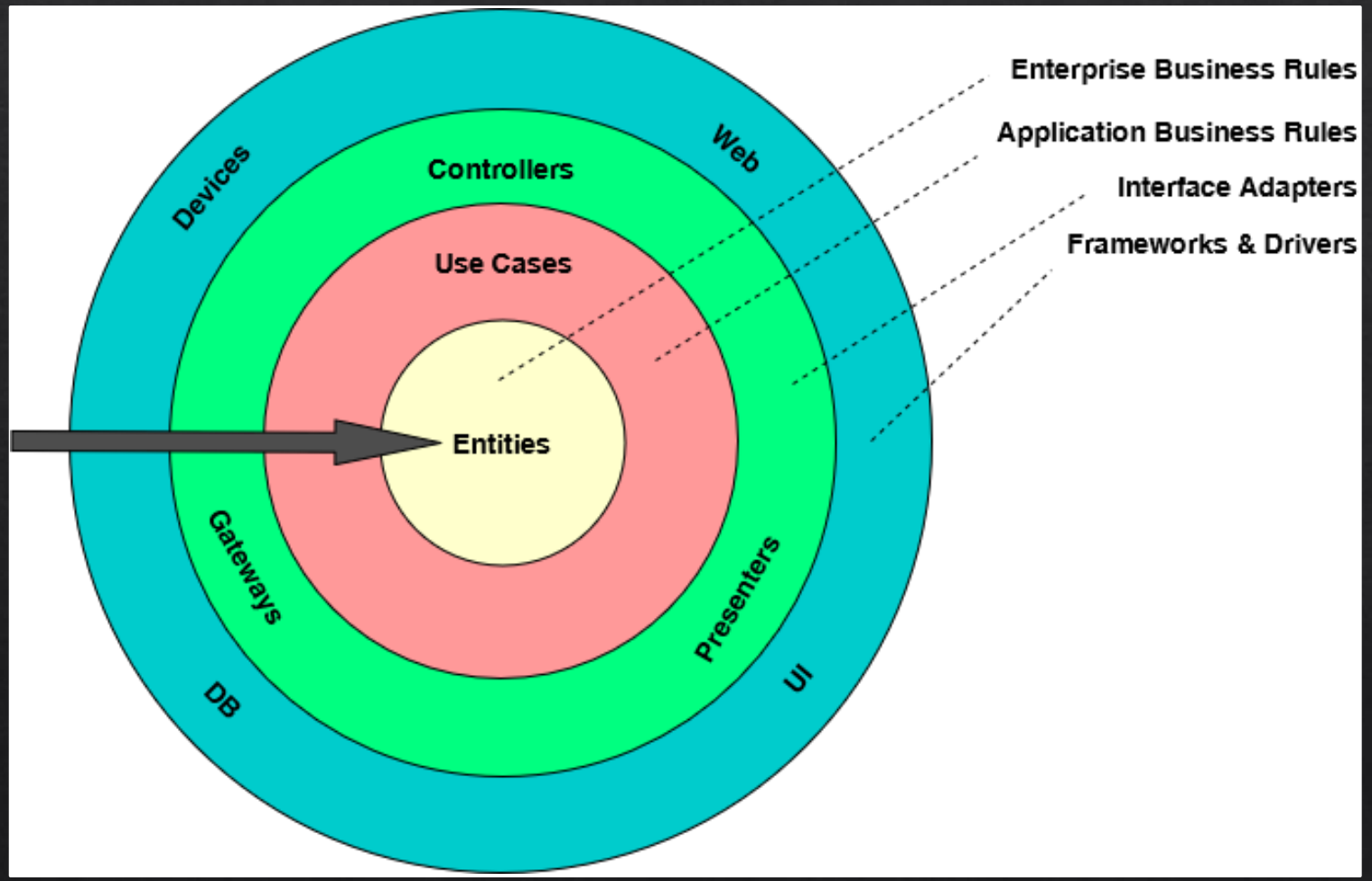
Similarities

1. Independent of Frameworks
2. Testable
3. Independent of UI
4. Independent of Database
5. Independent of any external agency





What is clean architecture?

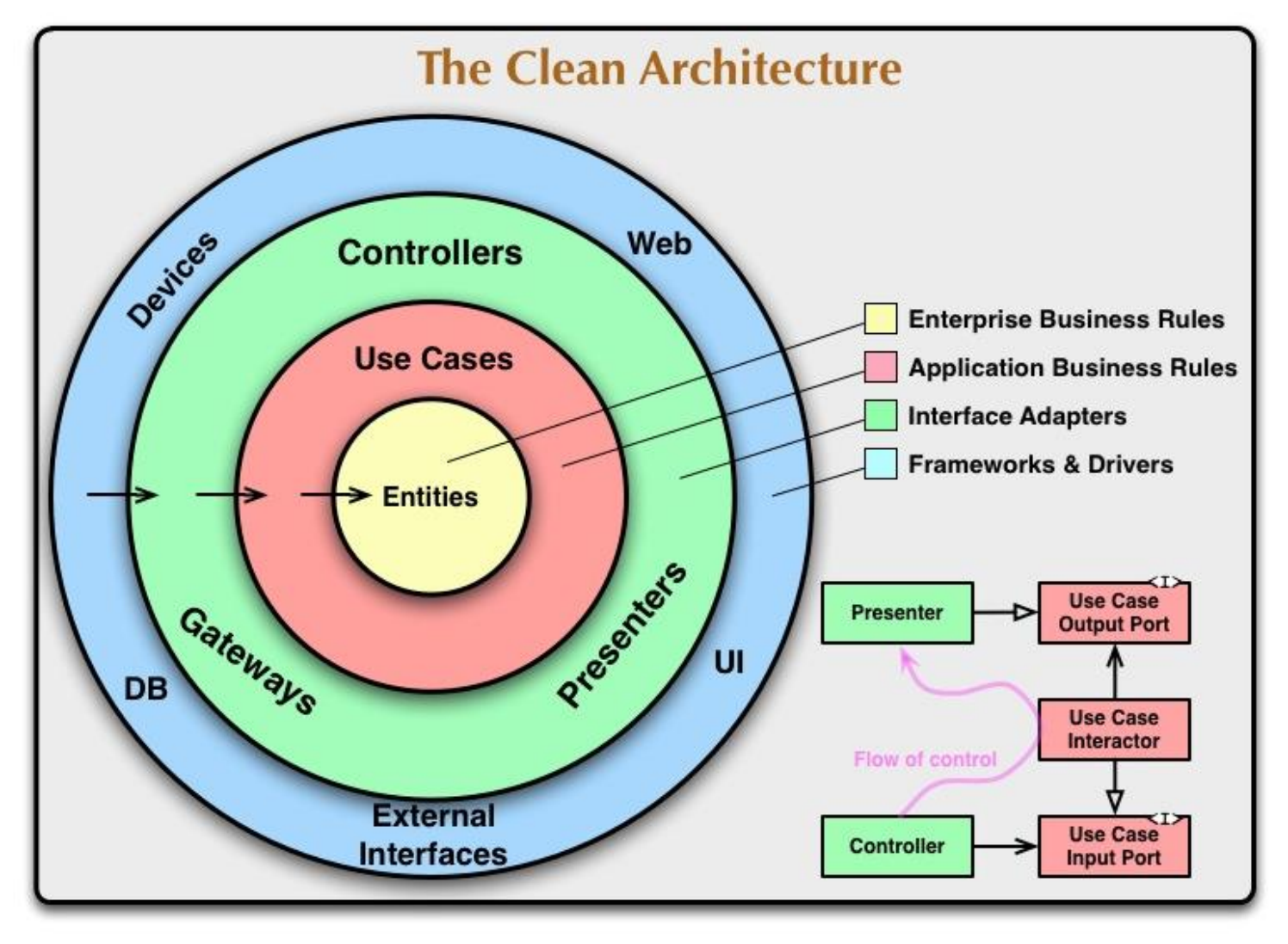


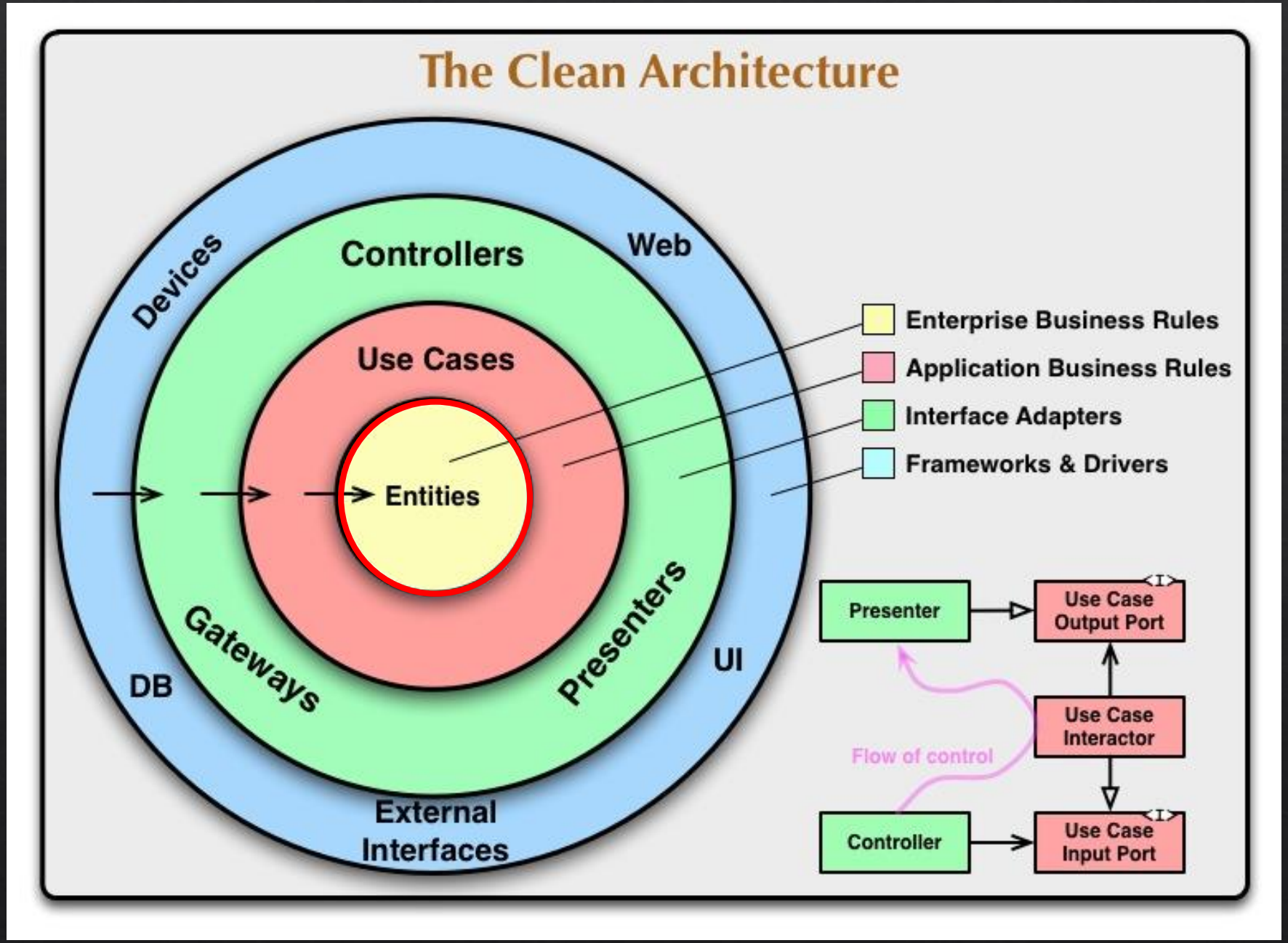
- Software design philosophy
- Design separated in ring levels
- Main rule: Dependency rule. Source code dependencies can only point inwards.

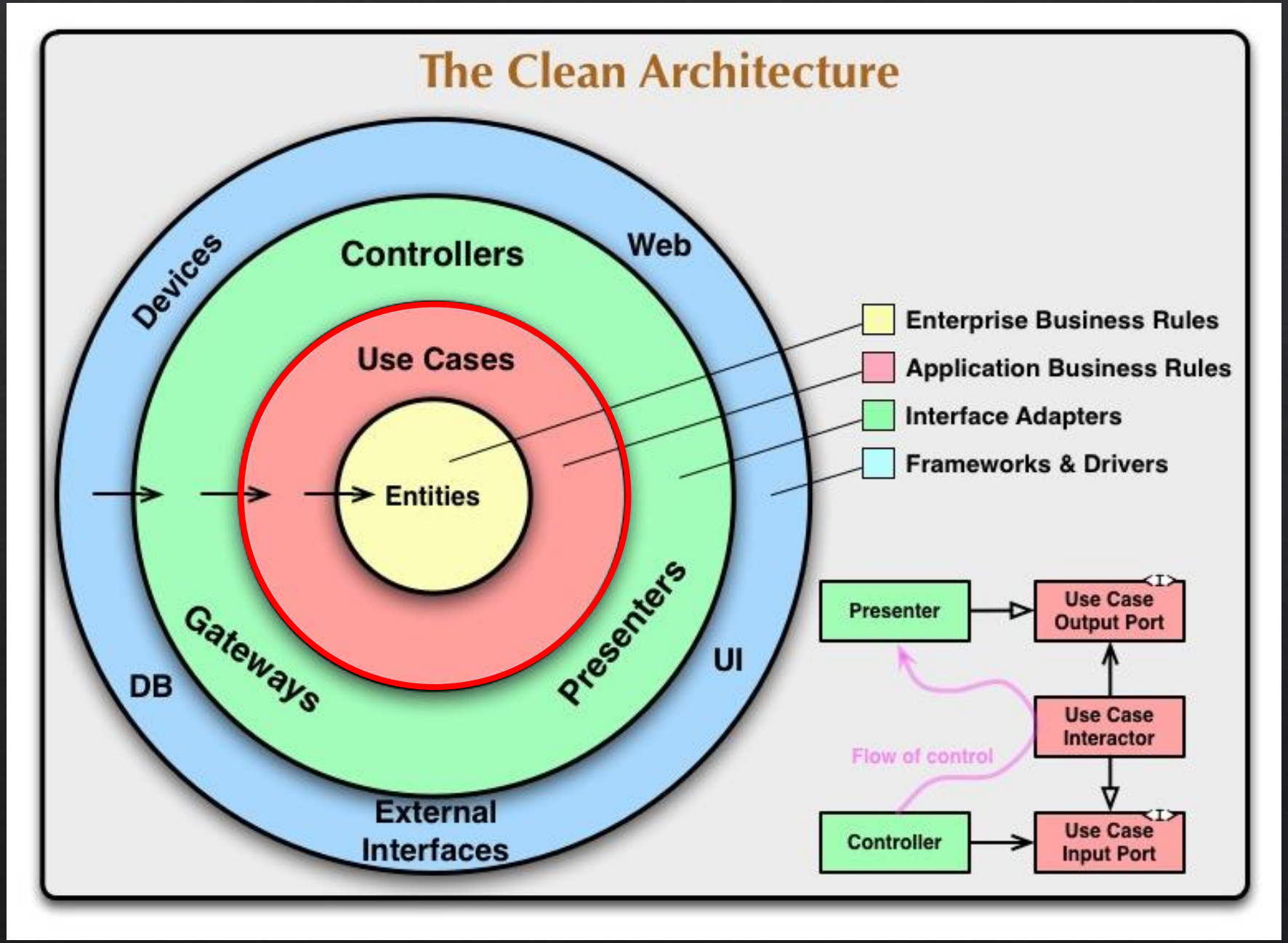
Parts of the ring

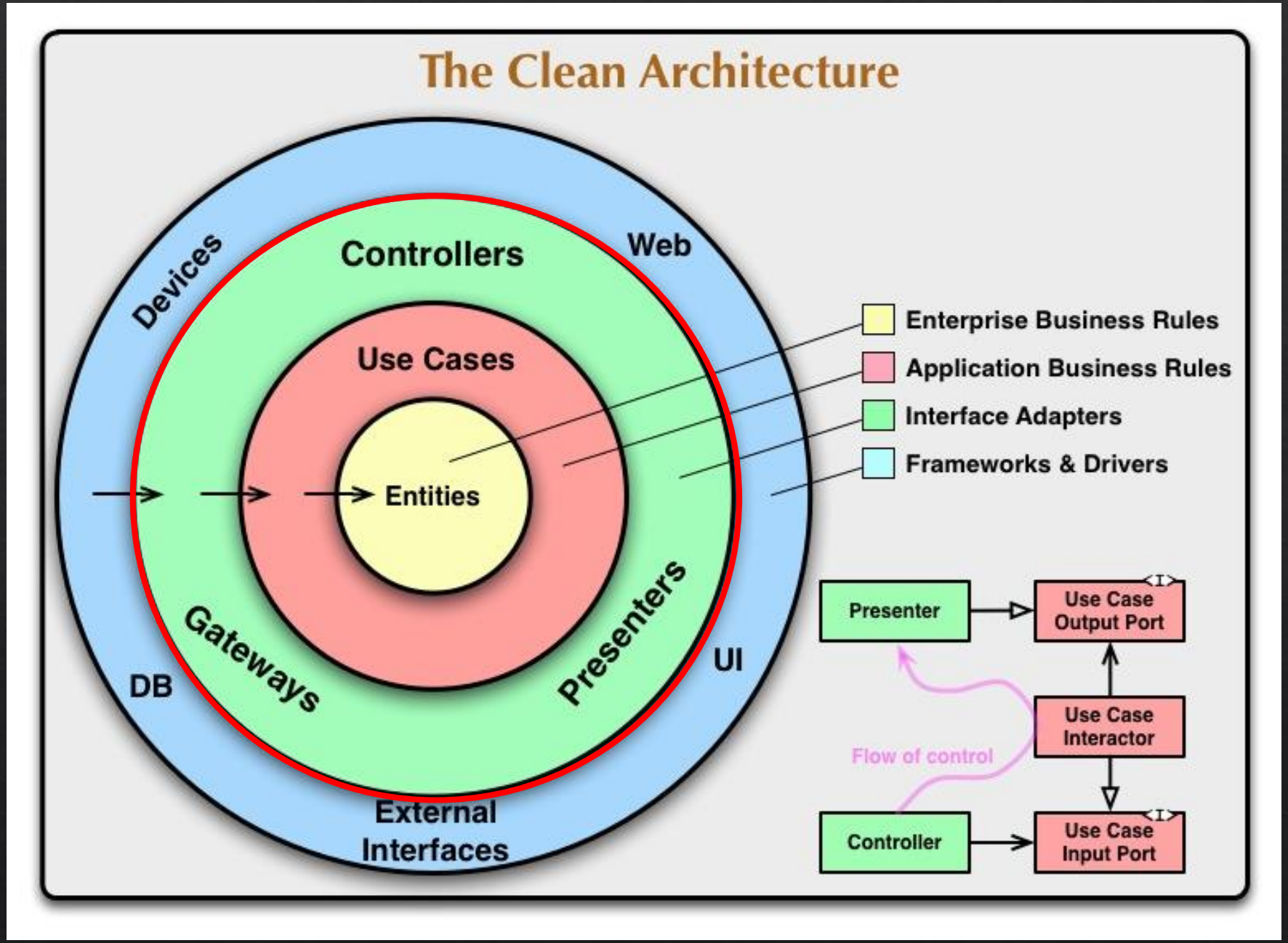
- Entities
- Use cases
- Interface Adapters
- Frameworks and Drivers
- …

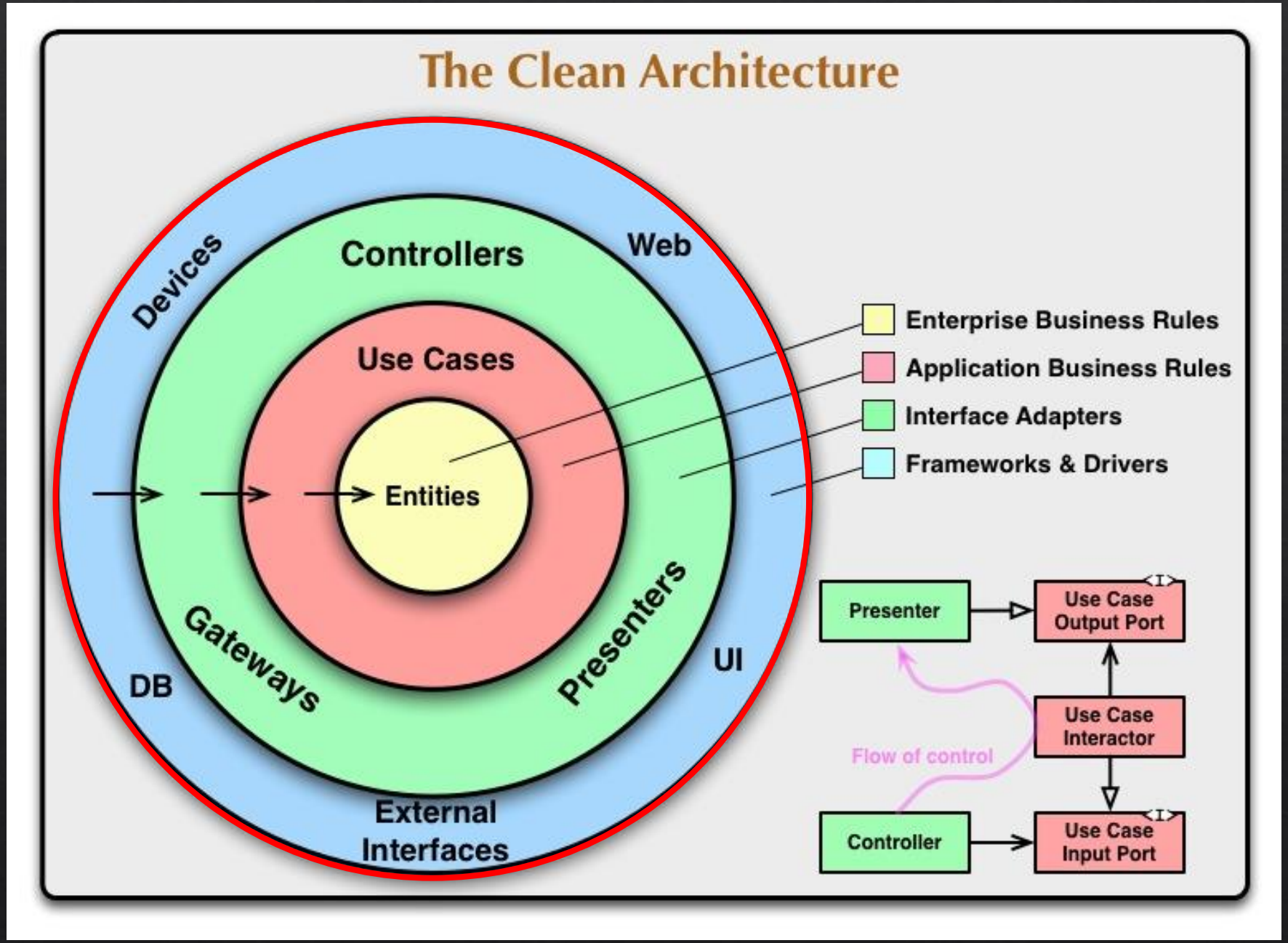
There's no rule that says you must always have just these four. However, *The Dependency Rule* always applies




The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port

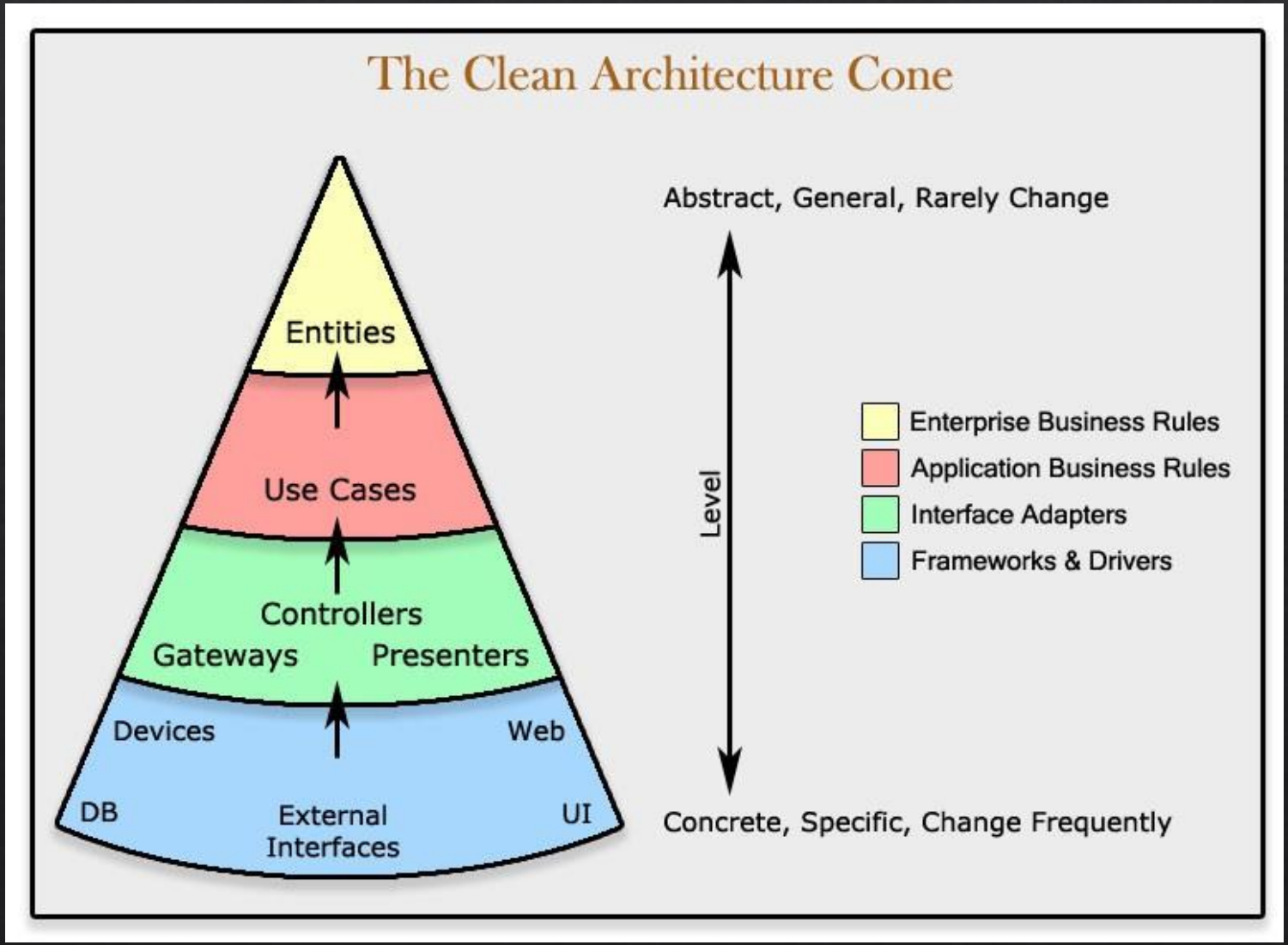

The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port


The Clean Architecture
Devices
Controllers
Web
Use Cases
Entities
Gateways
Presenters
DB
UI
External
Interfaces
Enterprise Business Rules
Application Business Rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case
Output Port
Use Case
Interactor
Flow of control
Controller
Use Case
Input Port

What things travel from one layer to another?

- We use the dependency inversion principle.
- Structs/Data transfer objects … (simple data structures)



Goals of the code architecture

Modular, scalable, maintainable and testeable appliaction.

Separation of concerns.

Flexible to change

React as an implementation detail

First:

- We should not design for React, we should use React for implementing our design
- Do not tie the data with the component
- Aim at usability and resilience to changes

```
function AElement({elements}) {
    <Typografy>
        {elements.map(element => element.title).filter(title => title.startsWith('9'))};
    </Typografy>
}
```

- What is elements?
- What data is it expecting?

```
<Component1>
  <Component2>
    <Component3>
      <Component4>
        <Component5>
          <Component6>
            <Component7>
              <Component8>
                <Component9>
                  <Component10>
                    <Component11>
                      <Component12>   <-- (your component probably)
                        <Component13>
                          ...
                           <Component-N>
```

How could we avoid this?

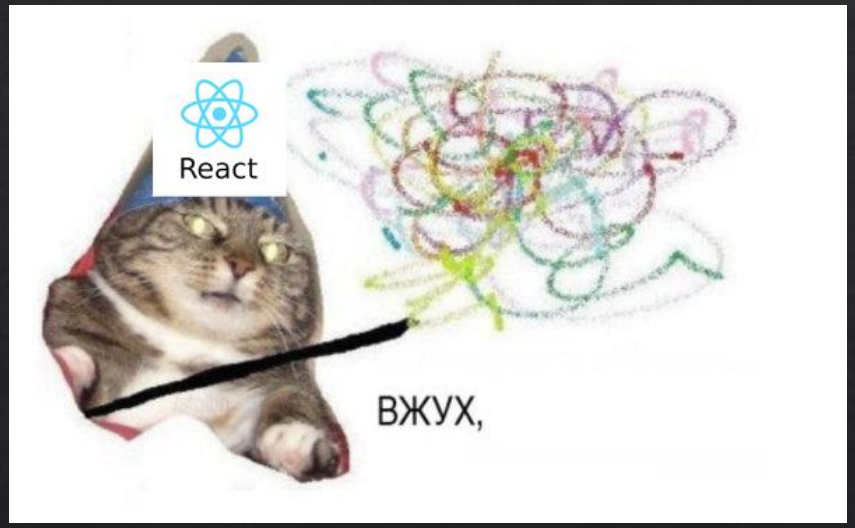
- Design without React

How would a component look following this approach?

- Reusable
- Independent
- Resilient to changes
- Text does not matter

```
function AElement({text}) {
    <Typografy>
        {text};
    </Typografy>
}
```

React does not do magic



Slow down!

Data modelling

Let’s think of a simple application

```
state = {
    form: {
        name: "Phone number",
        value: "",
        type: Text,
        placeholder: "Phone number",
        error: ""
    },
    numbers: []
}
```

- Represent the data
- Save other information that would be useful

Modeling events

```
const button = document.querySelector(".btn")

button.onclick = function() {
  console.log("Hello!");
};

// OR

button.onclick = () => {
  console.log("Hello!");
};
```

Functions for data management

```
const addNumber = (state, number) => {
    _.assign({}, state, {numbers: [...state.numbers, number]});
}
```

- Still no React!
- Add, delete and modify data
- Testing becomes easier!

Application layer

- Business logic
- What happens when a button is pressed?

Presentation layer

- The data is still raw!

Recap until now

We have:

- Data
- How to manage said data
- Business logic of our application

We still need to process the data for displaying it!

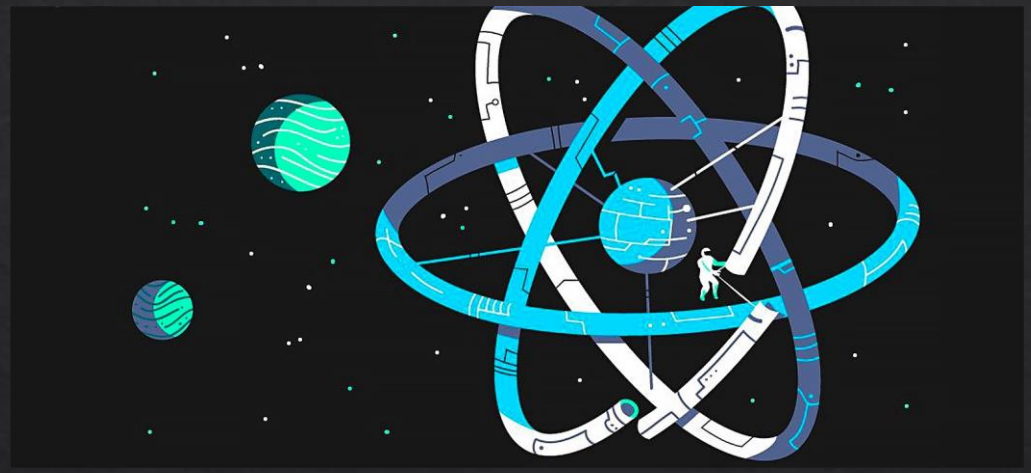
Data processing

```
const orderedNumbers = (numbers) => {
    numbers.sort();
}

const only9 = (numbers) => {
    numbers.map(number => number.title).filter(number => number.startsWith('9'));
}
```

- Now it is the time for the function
- We can add as many as we want

React



There are still two sides:

- Domain side
- View side

Domain side

- We handle states
- States change throughout the component chain
- A change in the state fires a new render of the component

View side

- Completely independent, generic
- Resilient to changes
- Reusable

```
function AElement({text}) {
    <Typografy>
        {text};
    </Typografy>
}
```

Time for questions