Clean Architecture

There are many ideas about the architecture of systems, as we can see in the slide there are several, for example the hexagonal architecture, the onion architecture …
These are not equal to one another they have some differences, but they have also the same objective and this is the separation of concerns. This is achieved by dividing the software into layers.

The architectures that we said produce system that have some similarities.
They are independent of frameworks. This means that the architecture does not depend on a library.
Testable. The code that implements the business rules can be tested without any external element.
Independent of UI (user interface). This means that the ui can change without propagating the changes into the rest of the system. For example, we change form a web app to a mobile app the ui but the business rules are the same. They did not have to change.
Independent of the database. We can change the database for one another for example change from oracle to mongo our business rules do not change.
Independent of any external agency. This means that our business rules do not know nothing about the outside world as we are going to see in the future.

Now, what is really clean architecture?
Clean architecture is a software design philosophy that separates the elements of a design into ring levels.
The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes.
There is a rule we need to follow if we want the clean architecture to work. This rule is the dependency rule.
It says that the source code dependencies can only point inwards, as we see in the arrow in the picture. Nothing in the inner circle knows about he outer circle. This means that something we declare in the outer circle can not be mentioned in the code of the inner circle.

As we can see in the slide there are some parts in the architecture.
Entities,
Use cases
Interface adapters
Frameworks and drivers.
There is no rule that says we must always have four. However, we have to take into account that the dependency rule.
Now we are going to talk about all the rings in the clean architecture.
Entities
The inner ring is the entities.
This encapsulates Enterprise wide the business rules. As we know, an entity can be an object with some methods or it can be a set of data structures or functions. These are the business objects of the application.
These are the least likely to change when something changes. So to have a good clean architecture, if we change something of the outer rings the entities should not change.
Uses cases

This layer contains application specific business rules. Here we implement all the use cases of the system.
We don't expect that the changes in this layer affect the entities one, we also do not expect that the changes of the outer layer change this. This layer is isolated for the external ones.
The change we can have here are the ones that happen when we change an operation of the application. This means that if we change some details of an use-case, then some code on this layer will certainly be affected too.

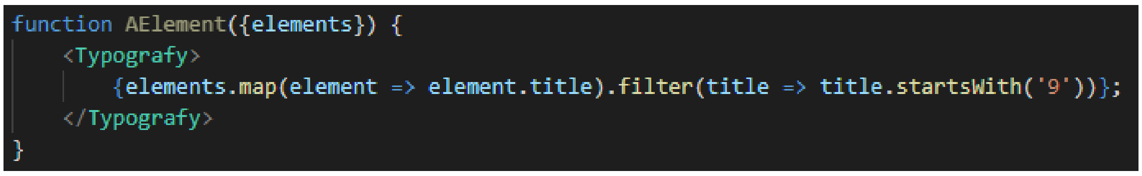
Interface adapter
The next layer we can see is the interface adapter ones.
This layer has a set of adapter which main concern is to convert the data from the format that is most convenient for the user and entities to the format that is most convenient to the database.
In this layer we will have for example all the modelo-vista-controlador architecture (presenter, views and controllers)
This layer is the one used to talk to the database all the rings that are inside this one should not know nothing about the database.

The outermost layer is generally composed of frameworks and tools such as the Database, the Web Framework, etc.
This is the layer where the programmer writes the less code.
This is the ring where the details go.
The database is a details as well as the web, so this two things should appear here.
We keep this thing in the outermost ring where they can do little harm.

Crossing boundaries.
Obviously, the layers need to talk to each other sometime. We have to take into account how the data flows, in begins in the controller, moves through the use case and then winds up in the presenter.
This communication contradiction is solved using the Dependency inversion principle. As we saw in other courses, in java, we could arrange interfaces and inheritance relationships such as the code dependencies oppose the flow of control.
We take advantage of dynamic polymorphism to create source code dependencies that oppose the flow of control so we can conform to the dependency rule.
We have to be careful to for the data that crossed the boundaries. Normally they are simple data structures.
As we saw in information repositories last semester, we use data transfer object DTO, this are isolated, simple, data structures that are able to cross the boundaries. The main use of this object is because we don't want to pass the entities or the database rows.
We have to remember when passing information between layers, that we have to pass them in the form that is most convenient for the inner circle without violating the dependency rule.
To sum up, the main goals of this architecture are the following, modular, scalable, maintainable and testable. Also to separate the concerns and to be flexible to change.

React as an implementation detail
Our purpose should be trying to make our architecture as independent as possible. We are going to talk about it from an implementation point of view with React.
I, and probably some of you here had got into the lab project without knowing anything about React, and that surely didn't help for writing good and structured code. So I will try to explain how could we improve said code and consequently our performance as well
First:

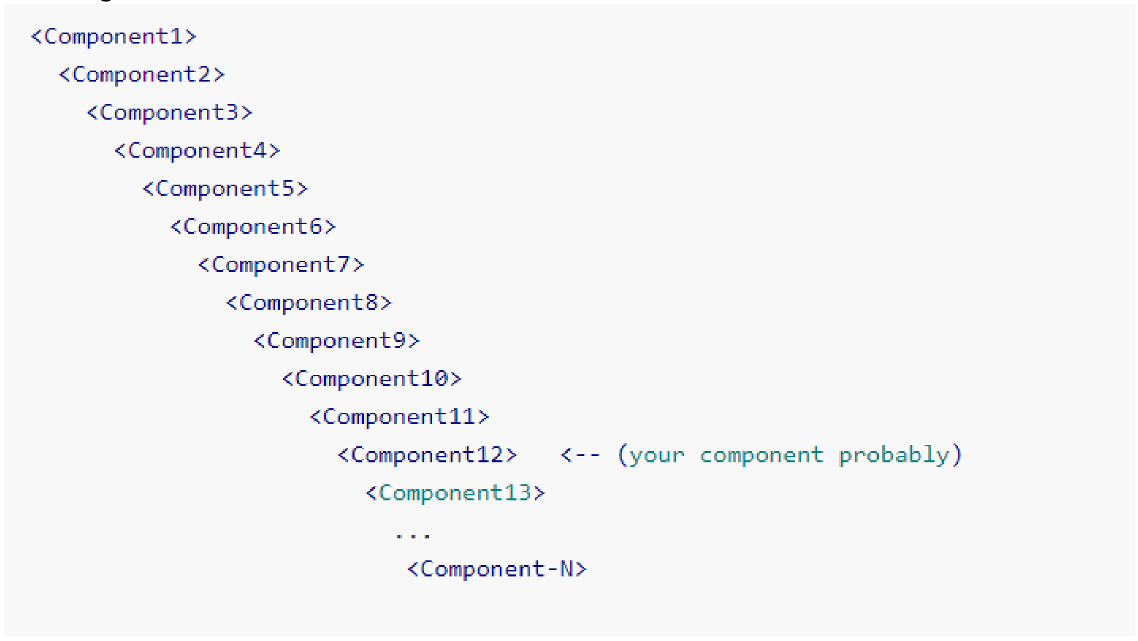
We shouldn’t design for React, we should use React for implementing our design.
What do I mean with that?
React main strength is the ability to create reusable components. In some cases we will find ourselves implementing components that are not resilient to changes neither reusable.
Components that we use as merely functions



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

For example this element. The reusability is practically none. What is elements? What happens if we want to select elements that start with another number, or a letter? Or elements that does not have a title but do have a name or an author? We would need to create more components.

This alone is a reasonable problem, but things start to get really messy when we do this frequently. We start to have more and more components that are specific for one thing, one insider another.
Resulting in an structure like this



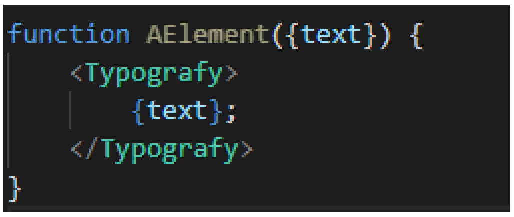
```
<Component1>
  <Component2>
    <Component3>
      <Component4>
        <Component5>
          <Component6>
            <Component7>
              <Component8>
                <Component9>
                  <Component10>
                    <Component11>
                      <Component12>   <--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

Funny, but awful.

So, what could we do to avoid this?
First and foremost, the easiest way of keeping React independent from our design is to design without React.

We know the drill, make layers, and keep them reasonably isolated. React does not need to know anything about our implementation. The business layer should take care of that, and the result of those calculations can be passed to react in order to display them.

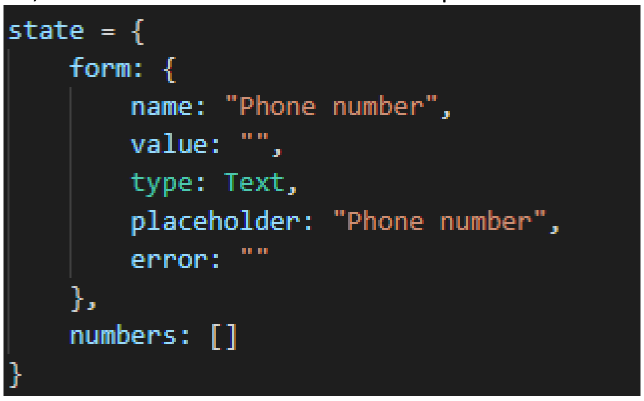
So, How would a component look following this approach? Taking the previously shown component it would look like this:



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

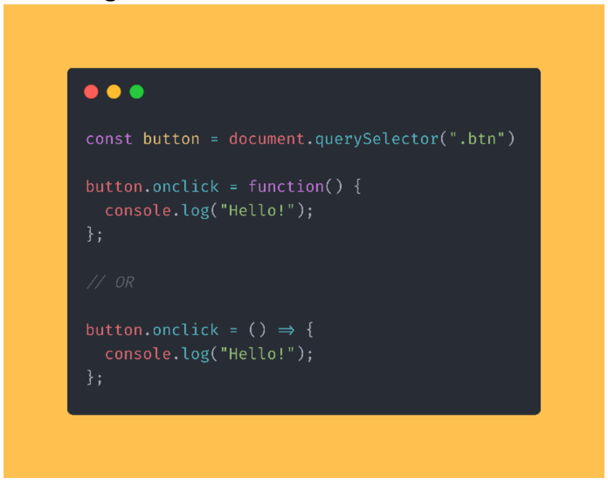
As it is apparent, this component is very reusable: you could use it anytime you want to display some formatted text. Independent and you can change it as much as you want, as you don't need to know anything about the data that it is getting and how should it be handled. It just displays.

But obviously React does not do magic.
The implementation of the management of the data must be somewhere else.
But let's slow down.
This problem goes back to the design of the architecture of our application.
We must do all the steps properly. The first one being data modeling.
This is a very important step. Data is completely independent and self-describing. It does not matter the language, context or anything. Data is Data.
So, let's think we want to build a simple form for storing phone numbers



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

We don't only represent the value of the data, but also some other information that will be useful in the future.
So now we have our data. What's next
Modeling events



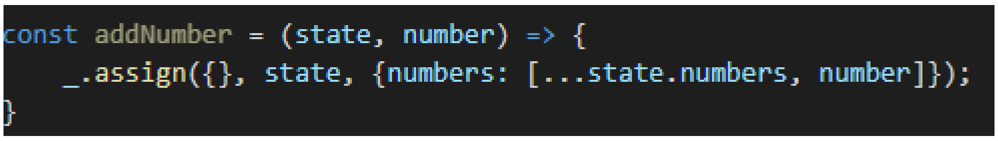
```
const button = document.querySelector(".btn")

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

// OR

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

And React is still nowhere to be seen
We want to add data, modify data, delete data. How? Functions



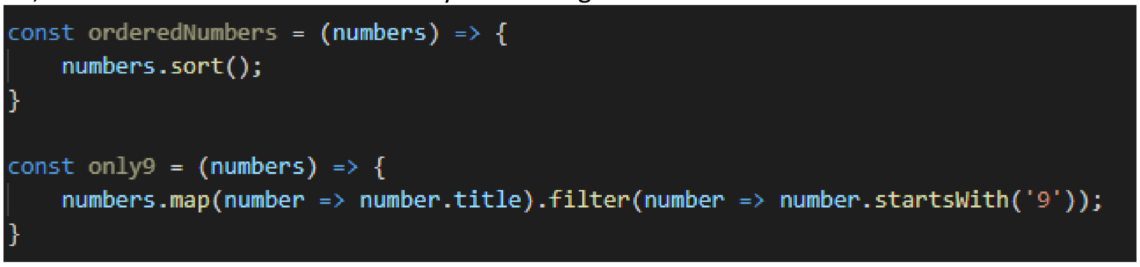
```
const addNumber = (state, number) => {
    _.assign({}, state, {numbers: [...state.numbers, number]});
}
```

(Don't mind the syntax, it basically add the number to the array in the state provided)
But it is independent. What is state, or number, we don't know. We don't NEED to know. We just add one to the other. That's it.

And the benefits go beyond independency. Testing becomes easier!
We no longer need to test the display of the text, managing complicated React components. We can just test how the state changes when we call the function. ->

Application "layer"
We have the data and how to modify it. Now it's time to start handling said data according to our needs. And this is where we do it. Every piece of business logic is done here.
What happens when we do x or y? What is this button going to do when pressed? Those interactions are defined here and later shared to the component
And finally, what we all have been waiting for.

Presentation Layer
but the data is still raw – you might say. And it is true
But first, a little review of what we have seen so far. ->We have the data, some utilities for handling it, and the business rules of our application. ->But we have not yet decided how the data is going to be displayed, and as we now know we should not let that functionality be in the react component itself.
So, here we create some functionality for this regard.



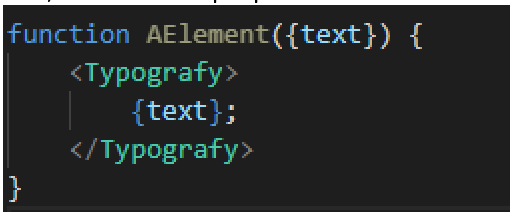
```
const orderedNumbers = (numbers) => {
    numbers.sort();
}

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

For example the function that I showed in the first slide.

And finally, React
Inside React we find two more sides, the domain side and the view side.
We defined states with the data, and we give that to the domain side. Instead of modifying or toying with the data, we just pass one state, something happens and the state changes. We are not sharing data through the component chain, we share states.
Here is important to remember that each time React detects a change on the state, the component affected is rendered again, thus showing the changes.

Now the view side is generic. Going back to the original example it would look something like this, which is our purpose.



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