VOLVO

Functional Thinking is SOLID

How Functional Thinking behind your solutions lead to SOLID and clean code base

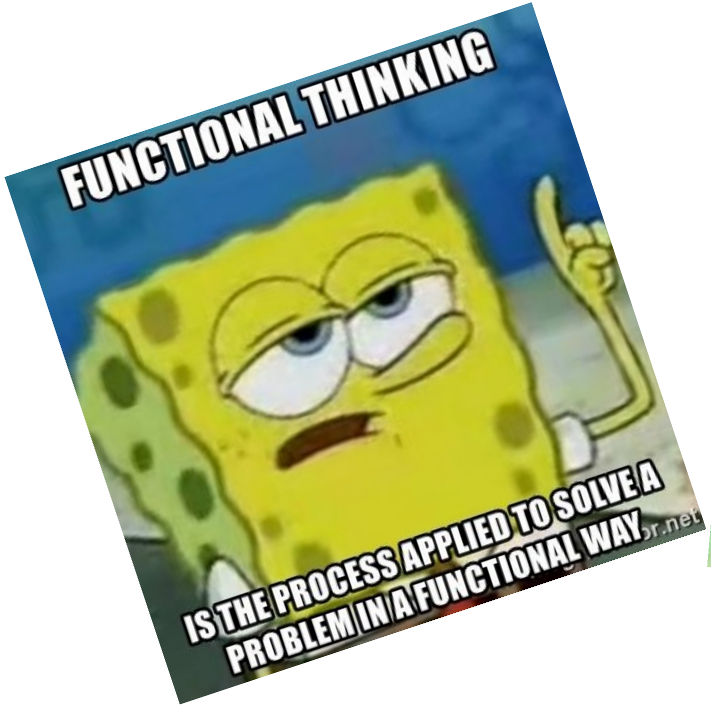
VOLVO

@chiaradiajm

@chiara-jm

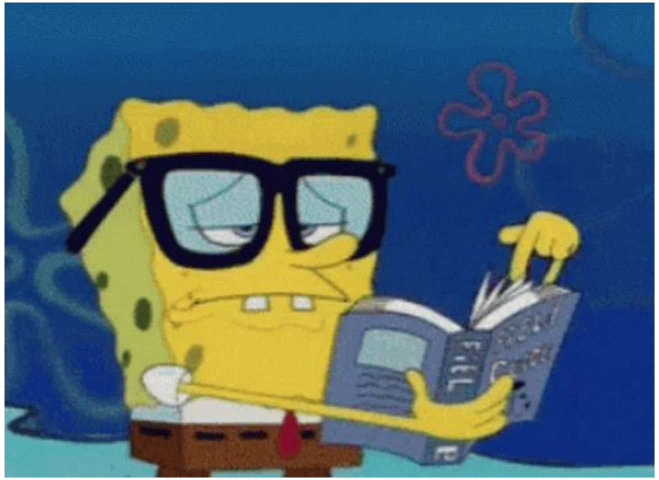
@chiara-jm

What is Functional Thinking?


FUNCTIONAL THINKING
IS THE PROCESS APPLIED TO SOLVE A PROBLEM IN A FUNCTIONAL WAY

VOLVO

Functional programming



Functional programming is a programming paradigm where programs are constructed by ***applying*** and **composing functions**.

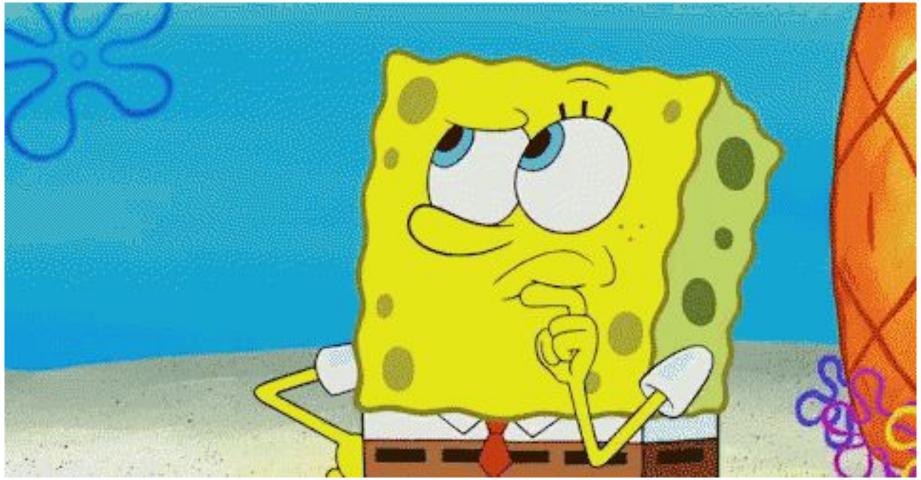
It is a declarative programming paradigm in which function definitions are **trees of expressions** that **map values to other values**, rather than a sequence of imperative statements which update the running state of the program.

Source: Wikipedia

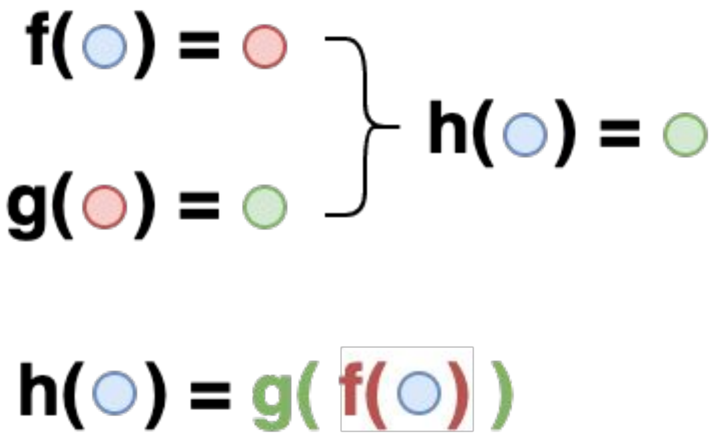
VOLVO

What is Functional Thinking?

- Apply and compose functions
- Work with declarative expressions
- Avoiding side effects

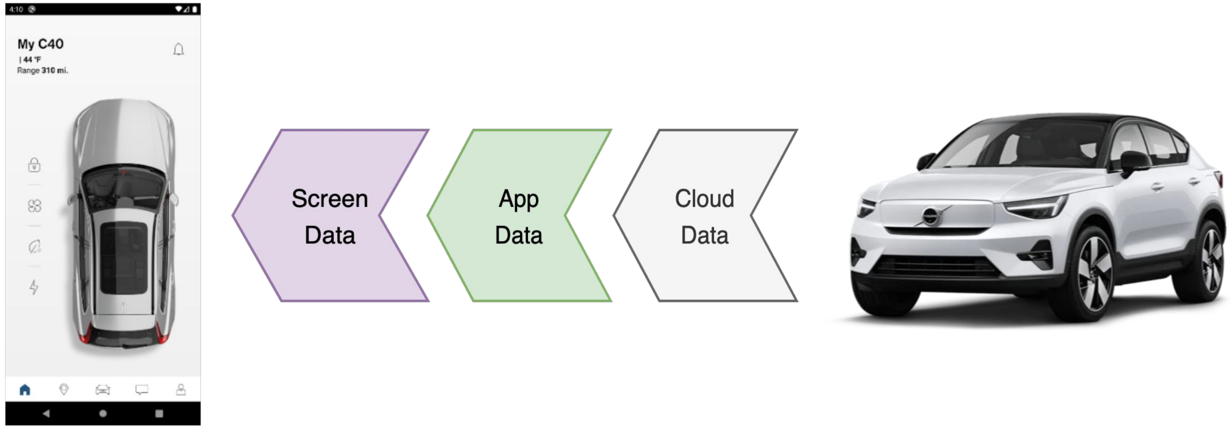


Functional Composition



Functional Composition

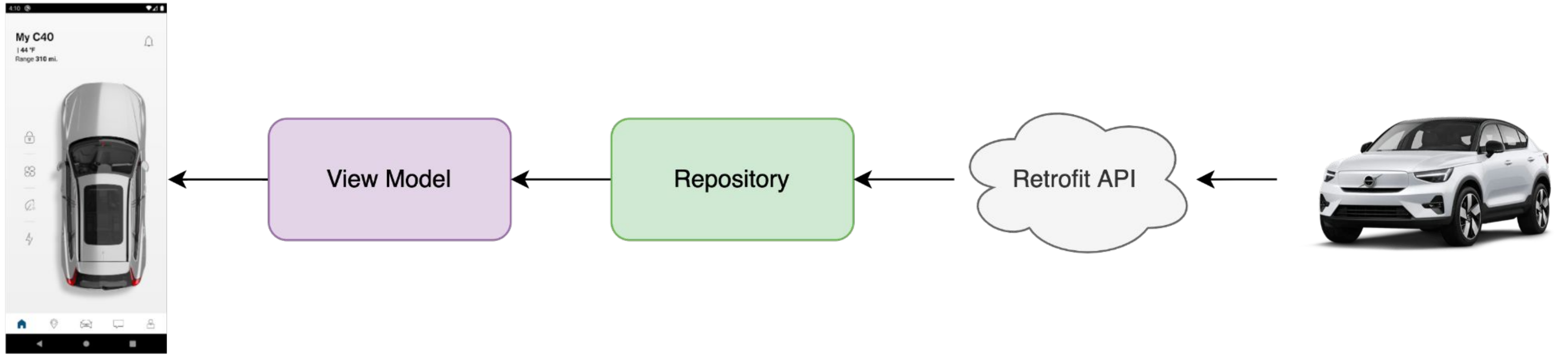
Functional Composition



Functional Composition

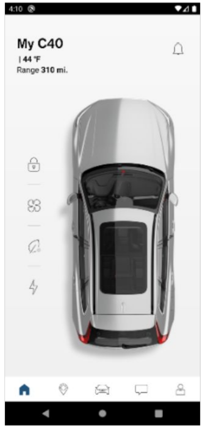
```
CarUI(carId) = CarView(
    getCarSnapshot(
        getAppCarState(
            getCloudCarState(carId)
        )
    )
)
```

Functional Composition



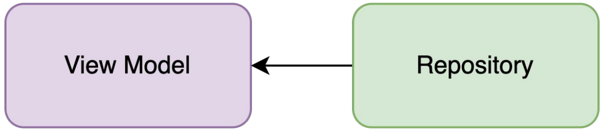
VOLVO

Functional Composition



```
CarView(carSnapshot = carSnapshot)
```

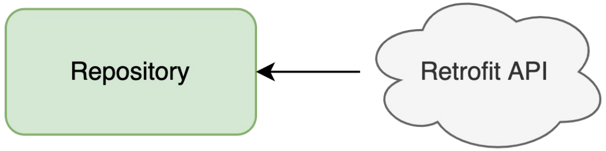
Functional Composition



```
val carSnapshot get() =
    repository.getCarState(carId).toCarSnapshot()
```

```
val carSnapshot by viewModel.carSnapshot.collectAsState()
CarView(carSnapshot = carSnapshot)
```

Functional Composition



```
carCloud.getState(carId).toCarState()
```

Functional Composition

```
val carSnapshot by viewModel.carSnapshot.
CarView(carSnapshot = carSnapshot)
```

```
val carSnapshot get() =
    repository.getCarState(carId).toCarSn
```

```
carCloud.getState(carId).toCarState()
```

Partial Application

f(🔵 🔴) = 🟢

g(🔴) = f(🟦 🔴) = 🟢

Partial Application

```
carSnapshot(carId) =
    getCarSnapshot(
        getAppCarState(
            getCloudCarState(carId)
        )
    )
```

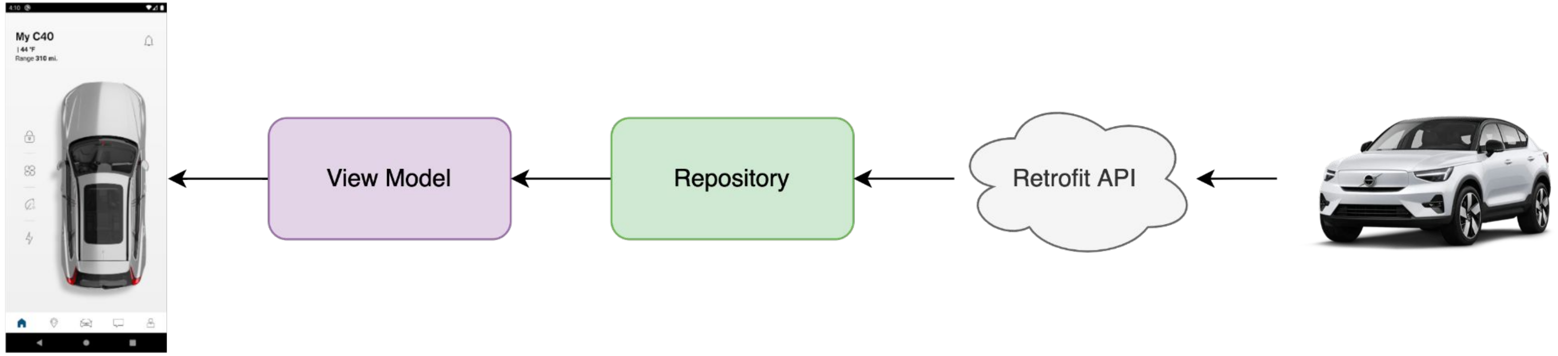
Partial Application

```
val carSnapshot by viewModel.carSnapshot.collectAsState()
CarView(carSnapshot = carSnapshot)
```

```
val carSnapshot get() =
    repository.getCarState(carId).toCarSnapshot()
```

```
carCloud.getState(carId).toCarState()
```

Partial Application



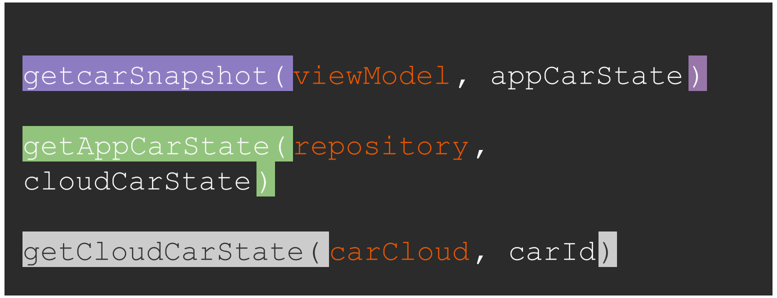
VOLVO

Partial Application

- Hide implementation details
- Identify dependencies
- Dependency Injection

VOLVO

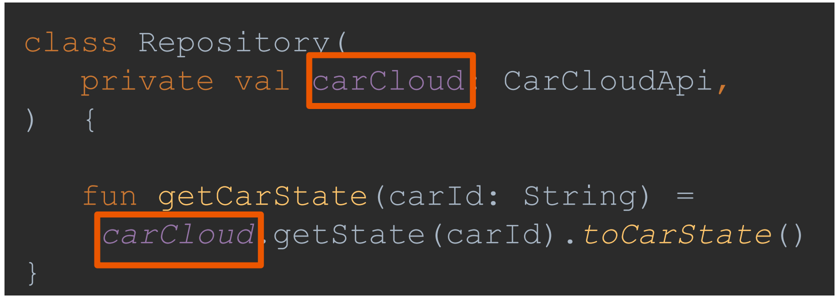
Partial Application



```
getcarSnapshot(viewModel, appCarState)

getAppCarState(repository,
cloudCarState)

getCloudCarState(carCloud, carId)
```



```
class Repository(
    private val carCloud: CarCloudApi,
) {

    fun getCarState(carId: String) =
     carCloud.getState(carId).toCarState()
}
```

High-Order Functions

What?

A function used as a parameter

Why?

Abstract behaviour

High-Order Functions

```
carSnapshot(
  getcarSnapshot,
  getAppCarState,
  getCloudCarState,
  carId,
) = getcarSnapshot(
        getAppCarState(
          getCloudCarState(carId)
        )
      )
```

High-Order Functions

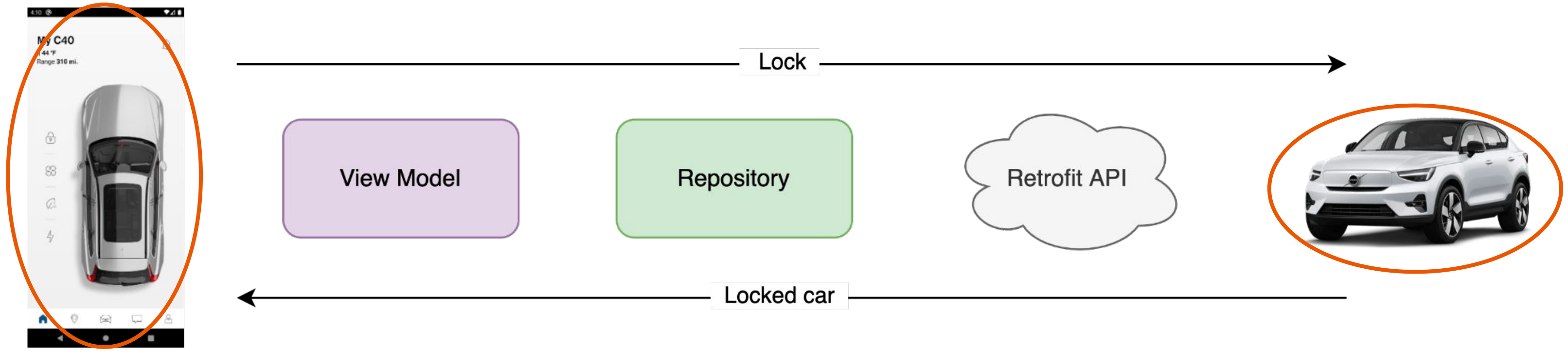
```
class GetCarState(
    private val repository: Repository
) : (String) -> CarState by repository::getCarState


val viewModel = ViewModel(
    getCarState = GetCarState()
)


val viewModel = ViewModel(
    getCarState = repository::getCarState
)
```

VOLVO

Side Effects



Side Effects

```
LockButton { viewModel.lock() }
```

```
lock(carId)
```

```
carCloud.lock(carId)
```

VOLVO

Side Effects

```
val carSnapshot by viewModel.carSnapshot.
CarView(carSnapshot = carSnapshot)
```

```
val carSnapshot = getCarStateFlow(carId).
```

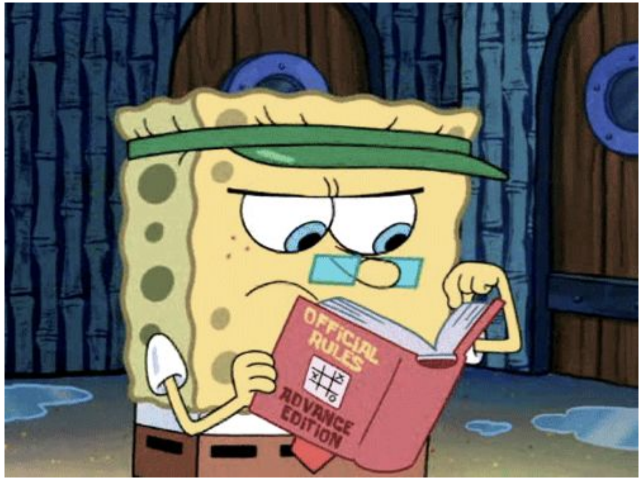
```
carCloud.lock(carId)
      .onSuccess { publishCarState() }
```

Homework

- Reactive Programing
- Immutability
- Separate **data** and **behaviour**
- Monadic Error Handling

`kotlinx.coroutines.flow`	
`val myField`	
`data class`	`lock(carId)`
`kotlin.Result`	

How does all this relate to SOLID?



Single Responsibility

An object should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the object.

The compositional nature of functional programing will allow us to focus on one responsibility at a time.

A function converts the given input into the expected output

The lack of side-effects (or limiting them to the ends of our layered architecture) enforces the SRP.

A function calculates a value or generates a side effect, but not both.

Single Responsibility

```
val carSnapshot by viewModel.carSnapshot.collectAsState()
CarView(carSnapshot = carSnapshot)
```

```
val carSnapshot get() = getCarState( carId).map { it.toCarSnapshot() }
```

```
carStateFlow.emit(carCloud.getState(carId). toCarState())
```

Open Close Principle

"Software entities ... should be open for extension, but closed for modification."

```
class ViewModel(
   private val carId: String,
   private val getCarState: (String) -> CarState,
)
```

- High-Order Functions together with Dependency Injection allow us to extend without modifying.
- Reactive programming + immutability (Homework)

Liskov Substitution

An object (such as a class) and a sub-object (such as a class that extends the first class) must be interchangeable without breaking the program

```
class ViewModel(
   private val carId: String,
   private val getCarState: (String) -> Result<CarState>,
)
```

- Avoid side-effects
- Monadic error handling
- Immutability

Interface Segregation

Many client-specific interfaces are better than one general-purpose interface.

```
class ViewModel(
   private val carId: String,
   private val getCarState: (String) -> CarState,
)
```

Thinking in terms of High-Order functions as dependencies ensures that the *client* only depend on what it needs and not more.

VOLVO

Dependency Inversion

High level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions

The abstraction is given by the High-Order functions.

```
getCarState:(String) -> CarState
```

The presentation layer consumes the HOF definition

```
ViewModel(getCarState:(String) -> CarState)
```

The data layer will implement that definition

```
repository::getCarState
```

The High-Order function definition `(String) -> CarState` is agnostic from both presentation and data layers.

Closing notes

- OOP and FP can be “best-friends”
- You do not need to go functional all the way
- It can take some practice to break old routines
- FP composition is a real powerful tool

VOLVO

One final note

- There is a SpongeBob gif for everything

Questions?

About FP and Kotlin?

About working at **Volvo**?

About me?