Some design concepts reviewed

MVC, Delegator, MVVM

Reminder on the MVC design pattern

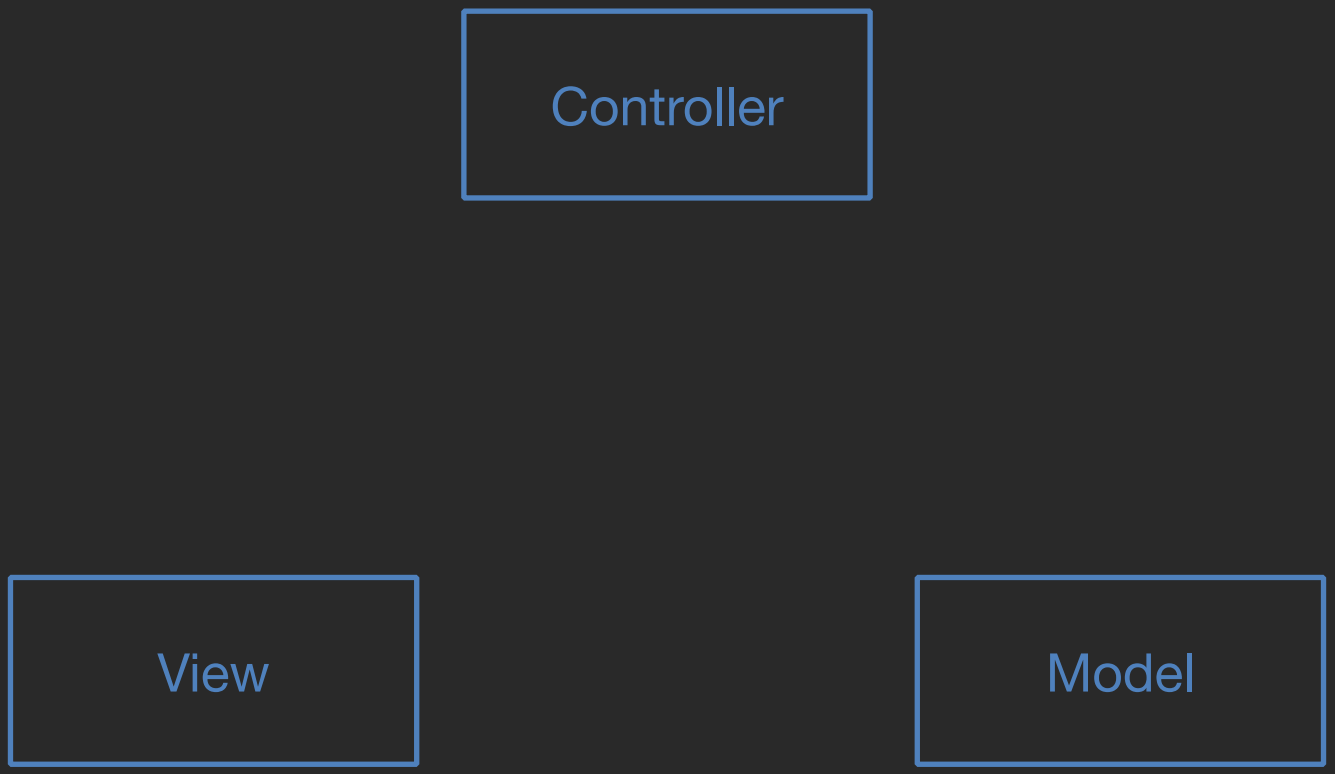
Model-View-Controller

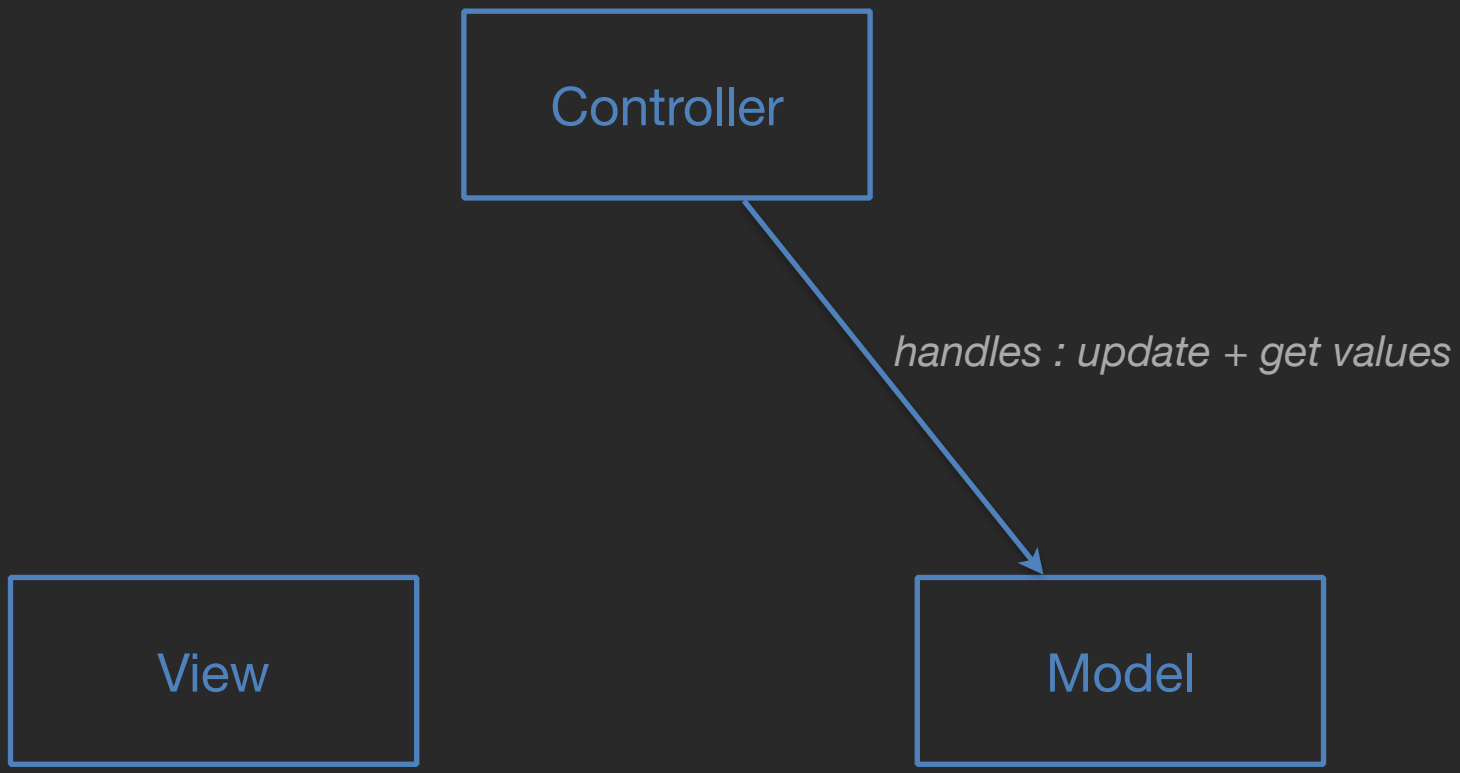
The MVC is the basis of different design patterns for UI Design

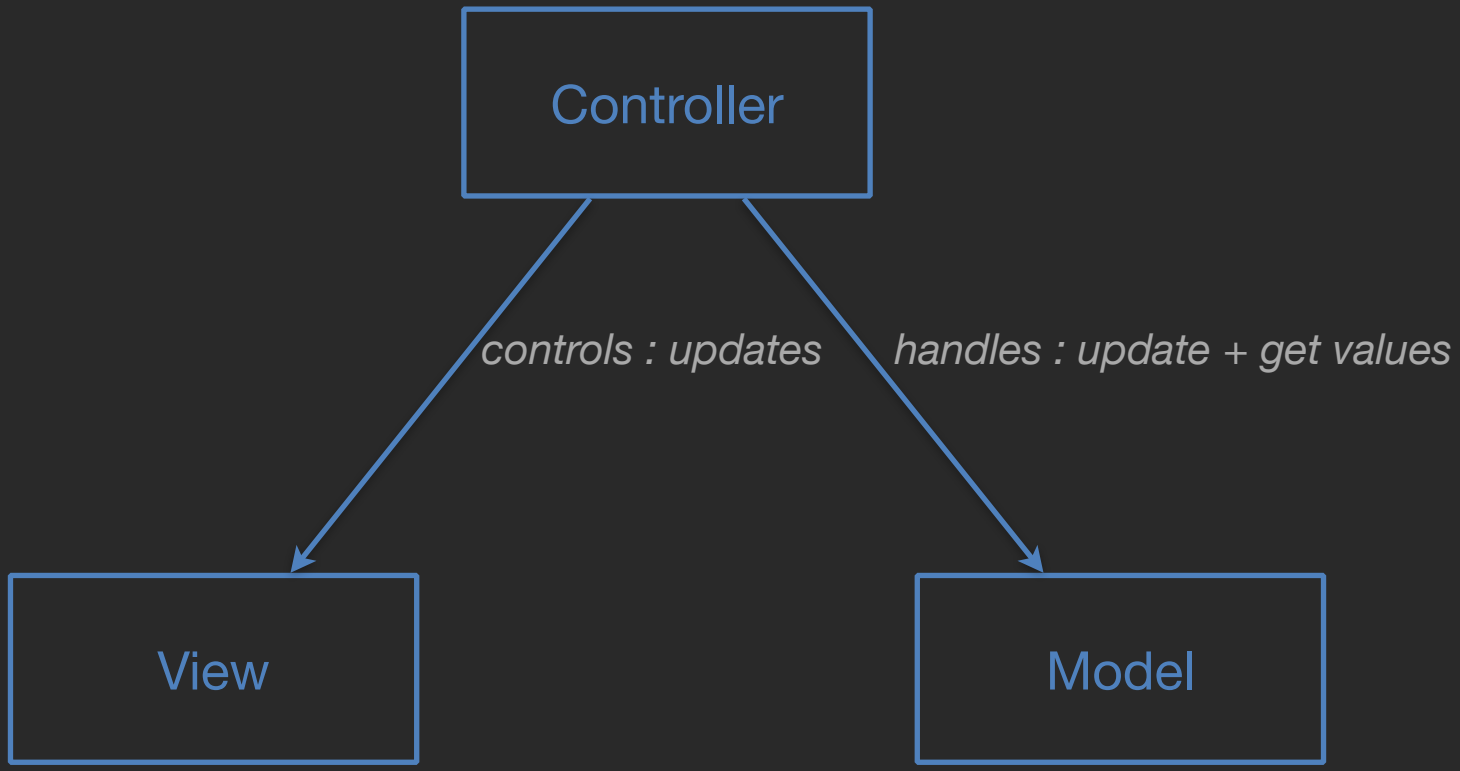
It was the first to formalize the separation of interface and business code

Its principles are the following:

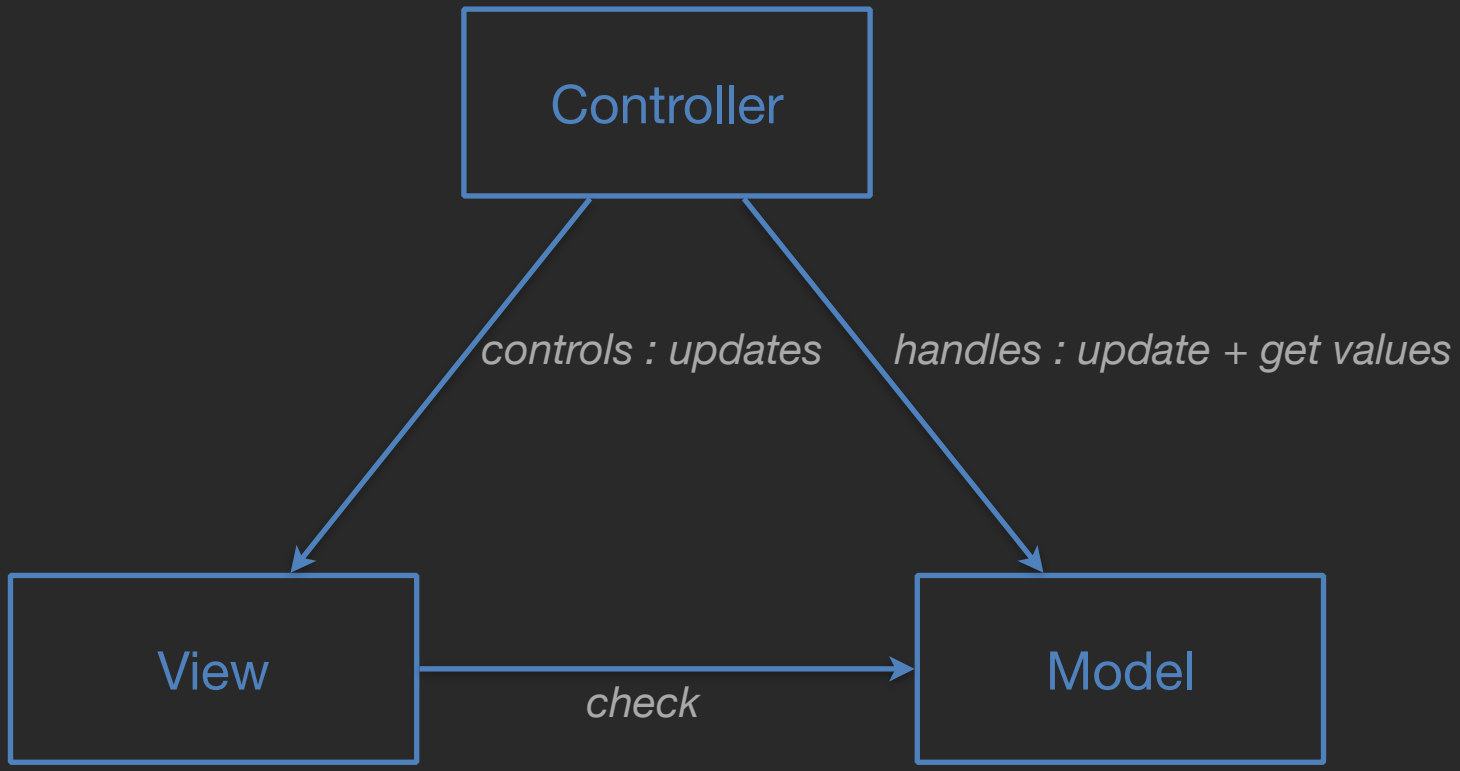
- a model (business class) that contains and manages the data
- a view that contains the presentation of the data
- a controller that manages the interactions between the view and the model

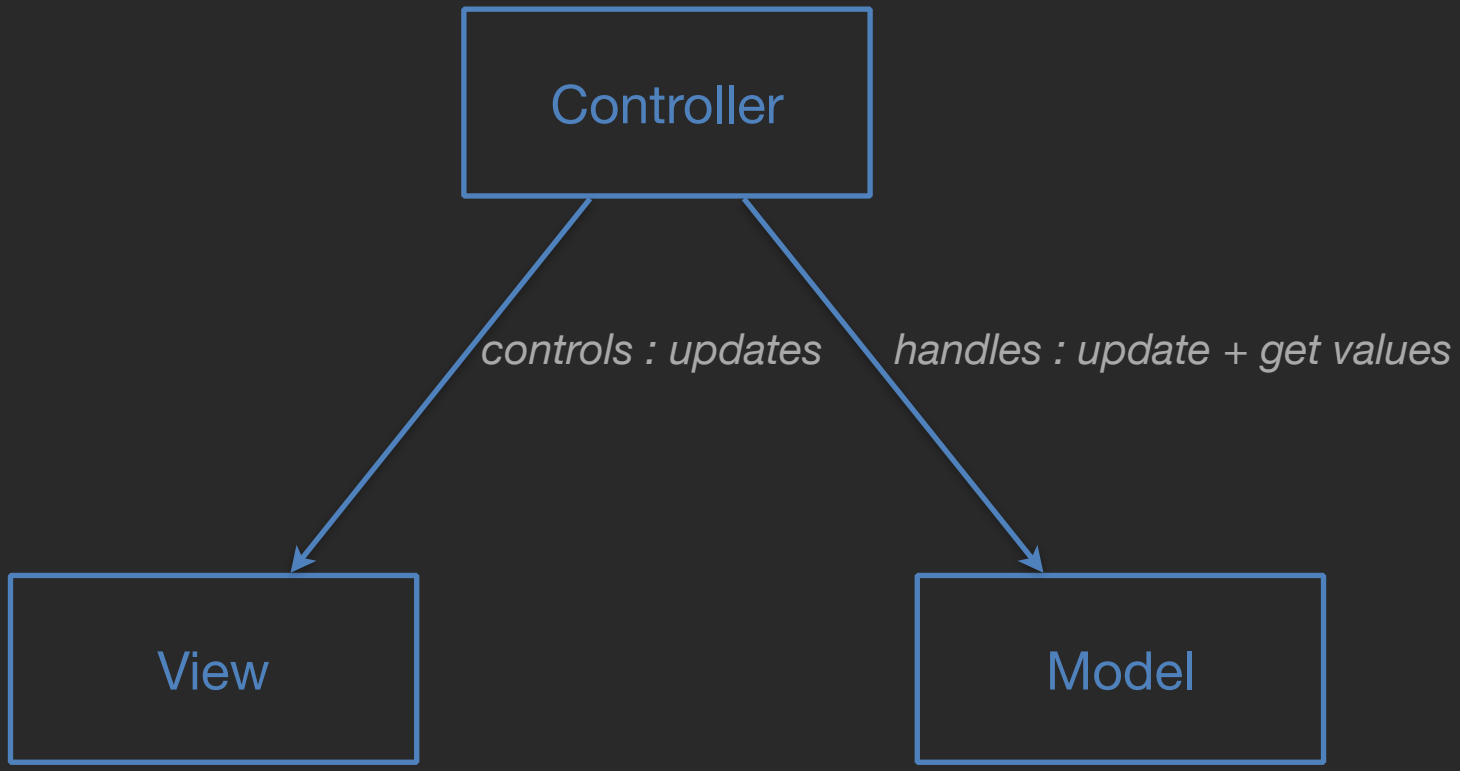

Controller
View
Model


Controller
handles : update + get values
View
Model

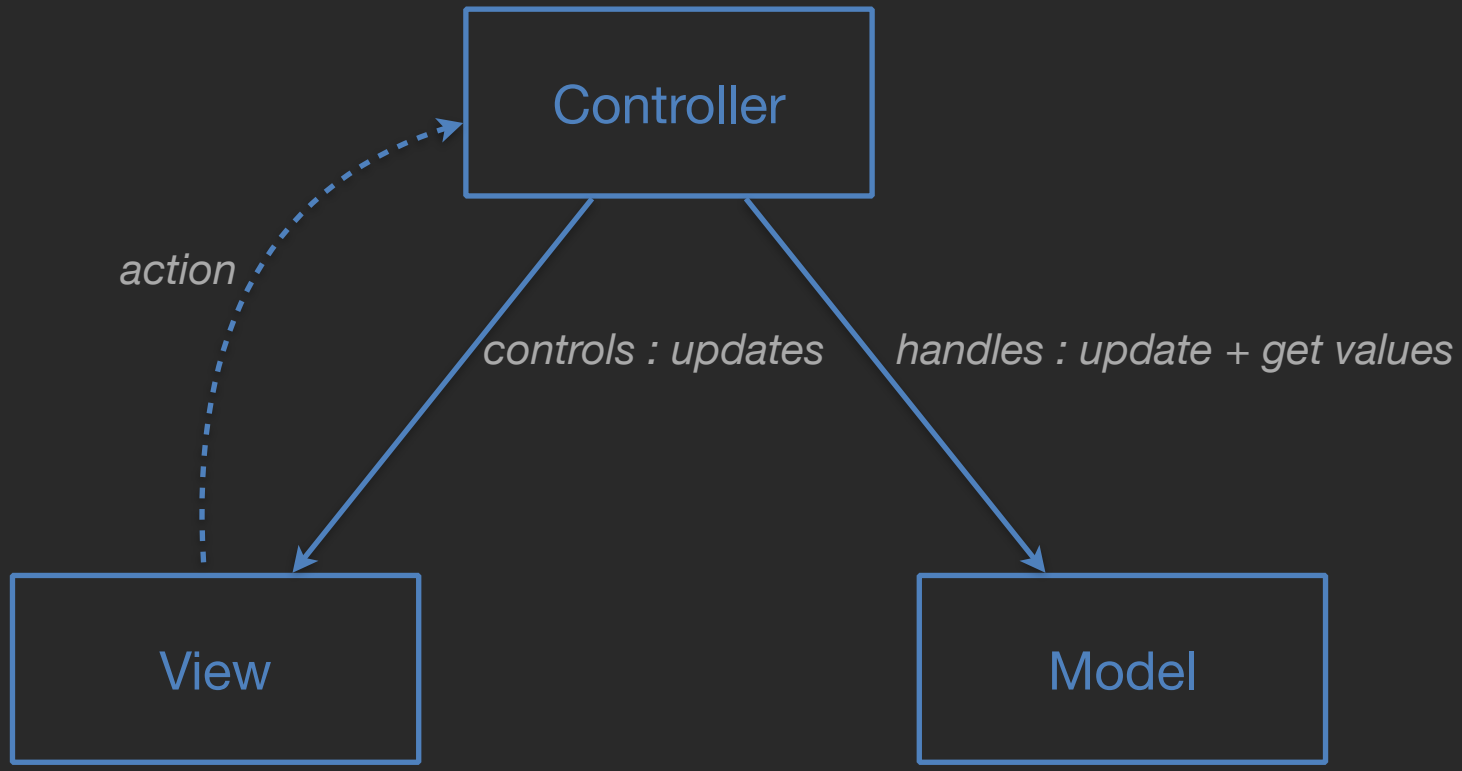


Model-View-Controller

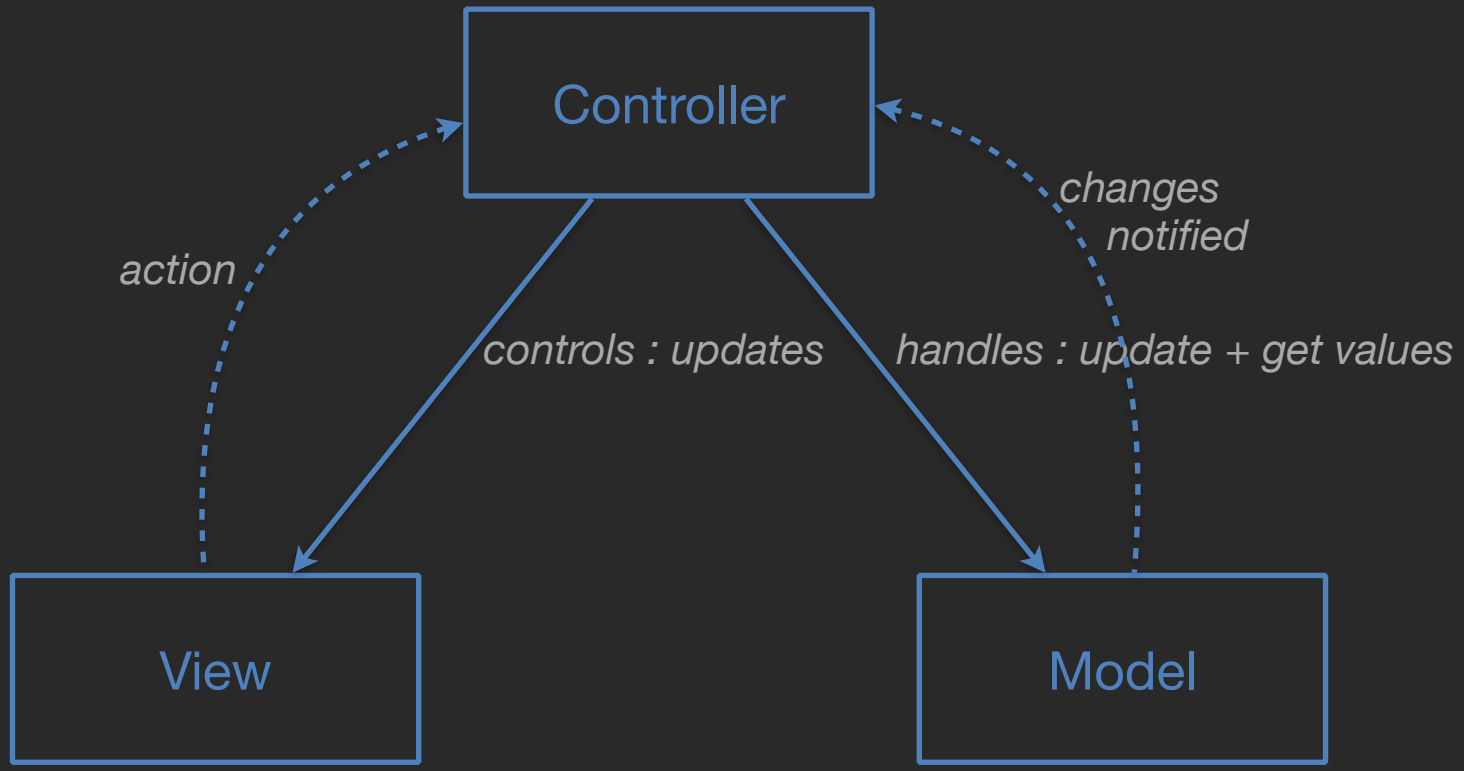

Controller
controls : updates
handles : update + get values
View
check
Model



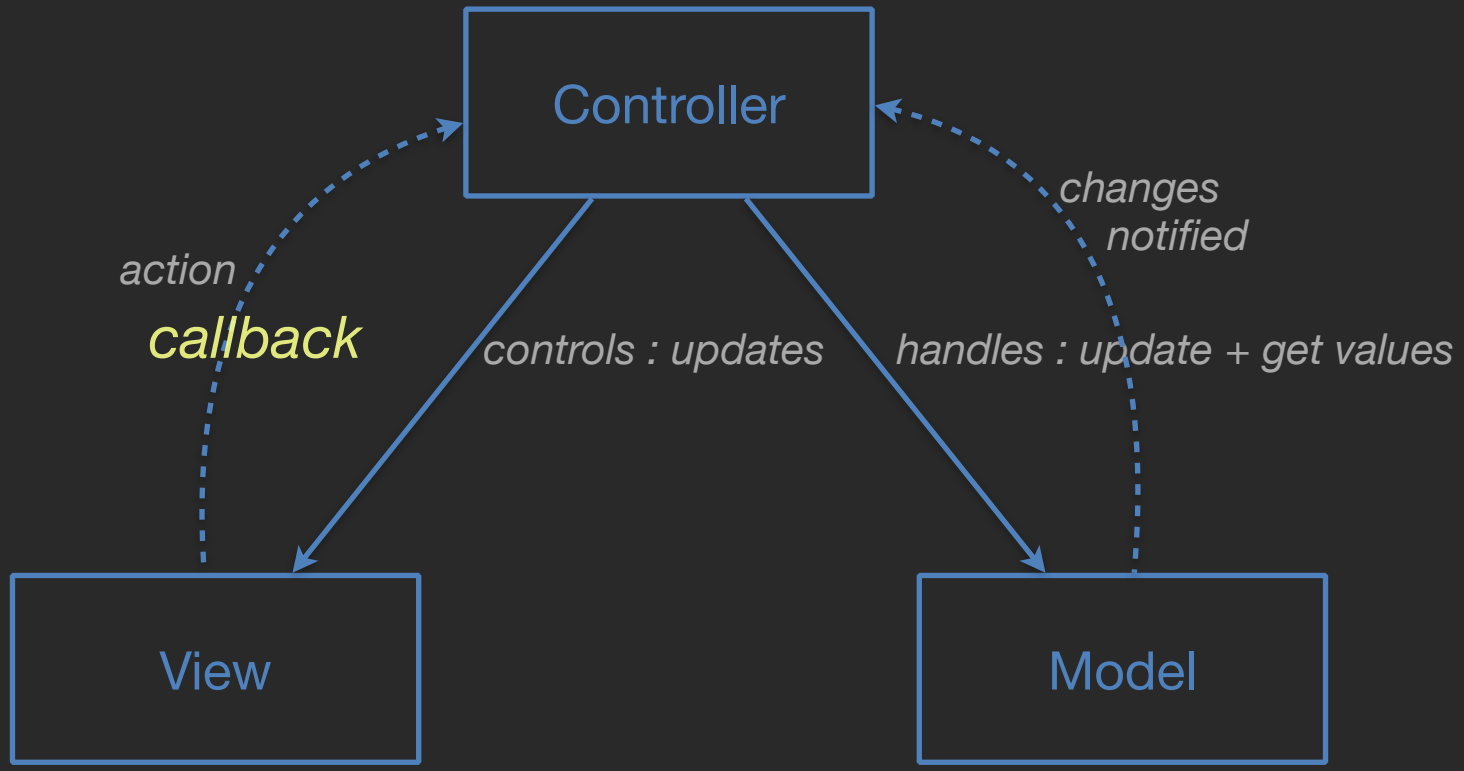
Model-View-Controller



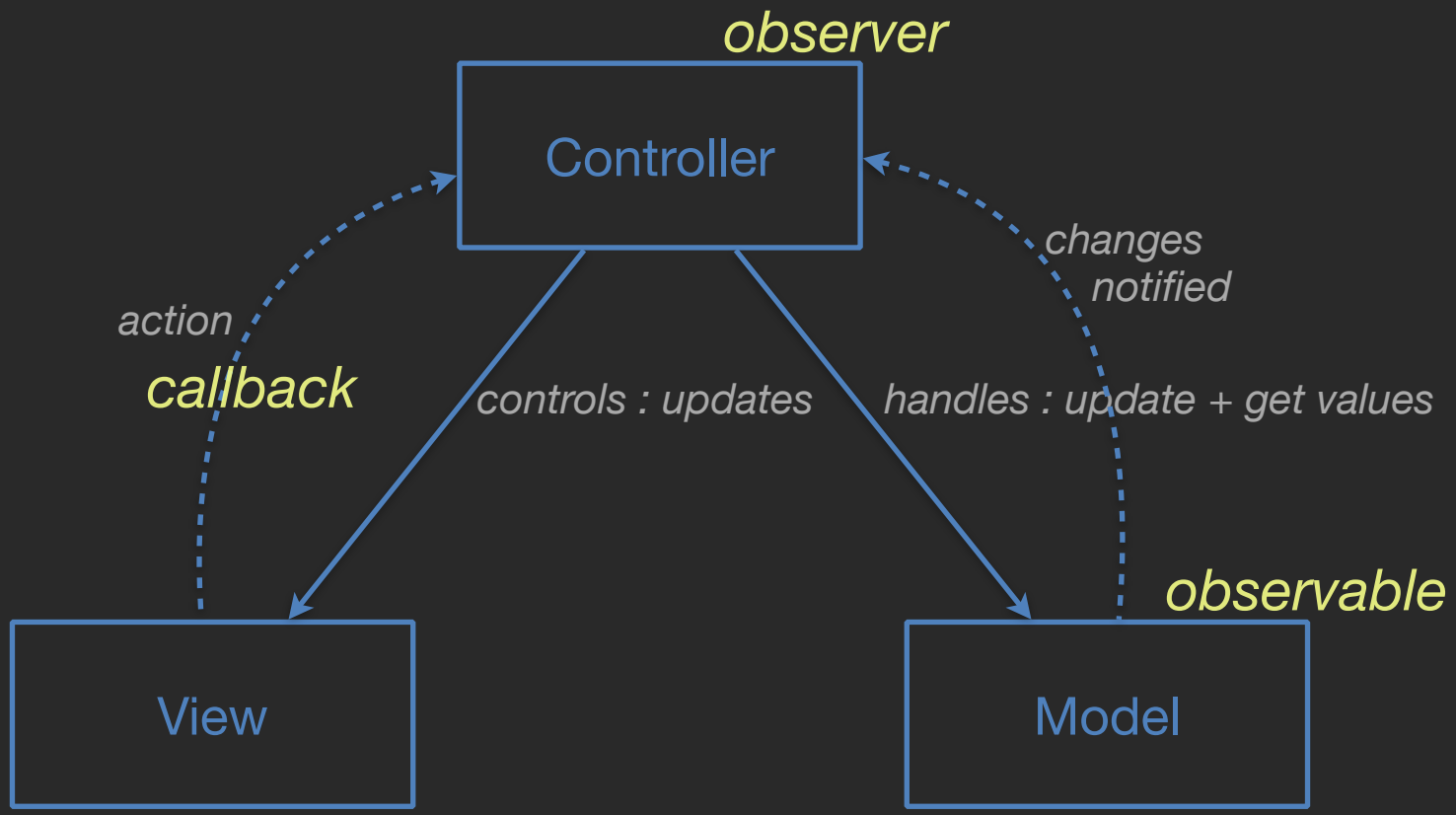
Model-View-Controller



Model-View-Controller



Model-View-Controller



Model-View-Controller

Main question: how model communicates with controller?

Adding a link from model to controller or better the view ⇒ break the independence of model

Other solution: using delegate or observer design pattern:

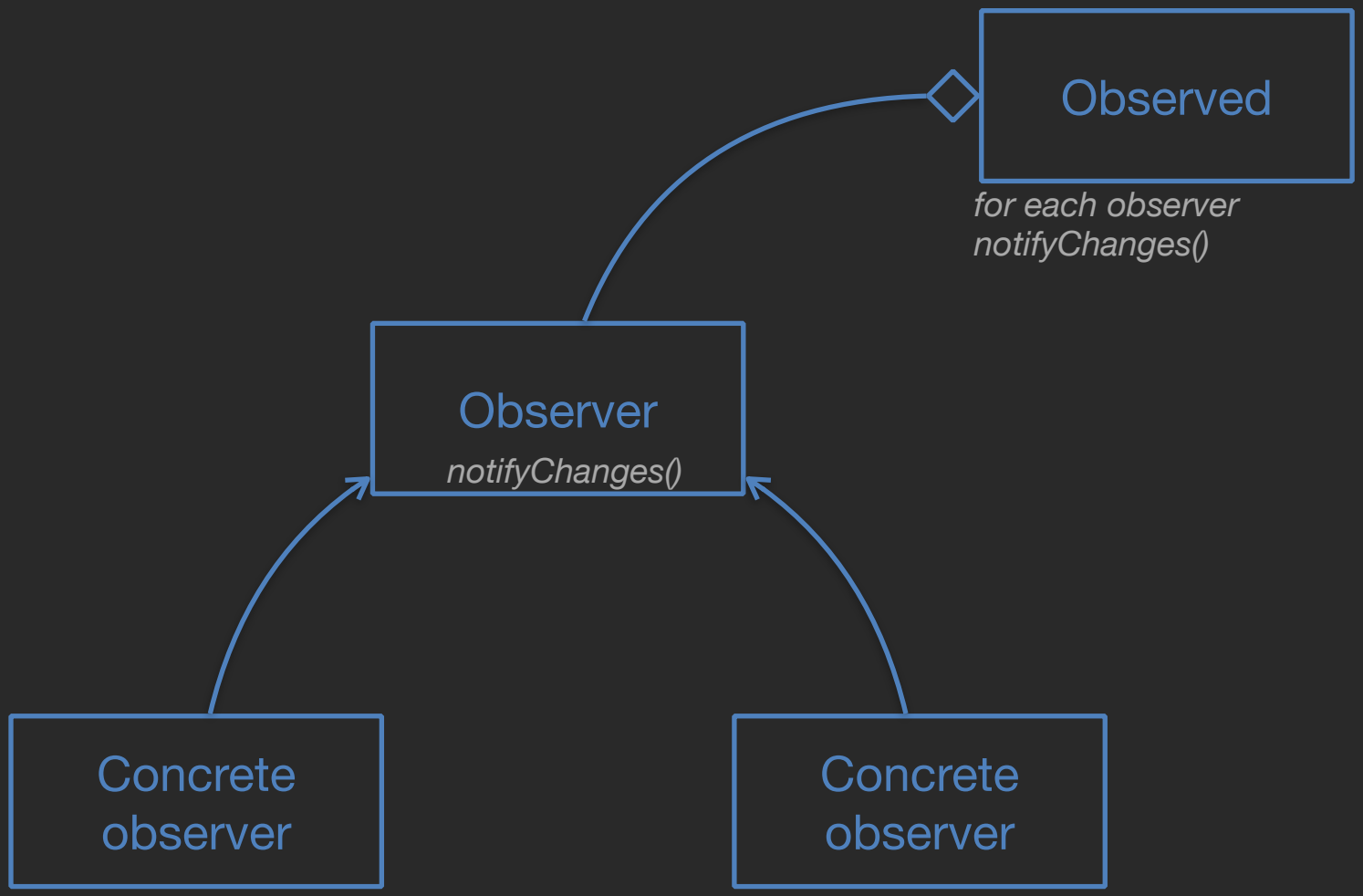
- model will be the subject
- the controller will be the observer
- protocol will be used to define abstract classes of delegate

Main question: how model communicates with controller?

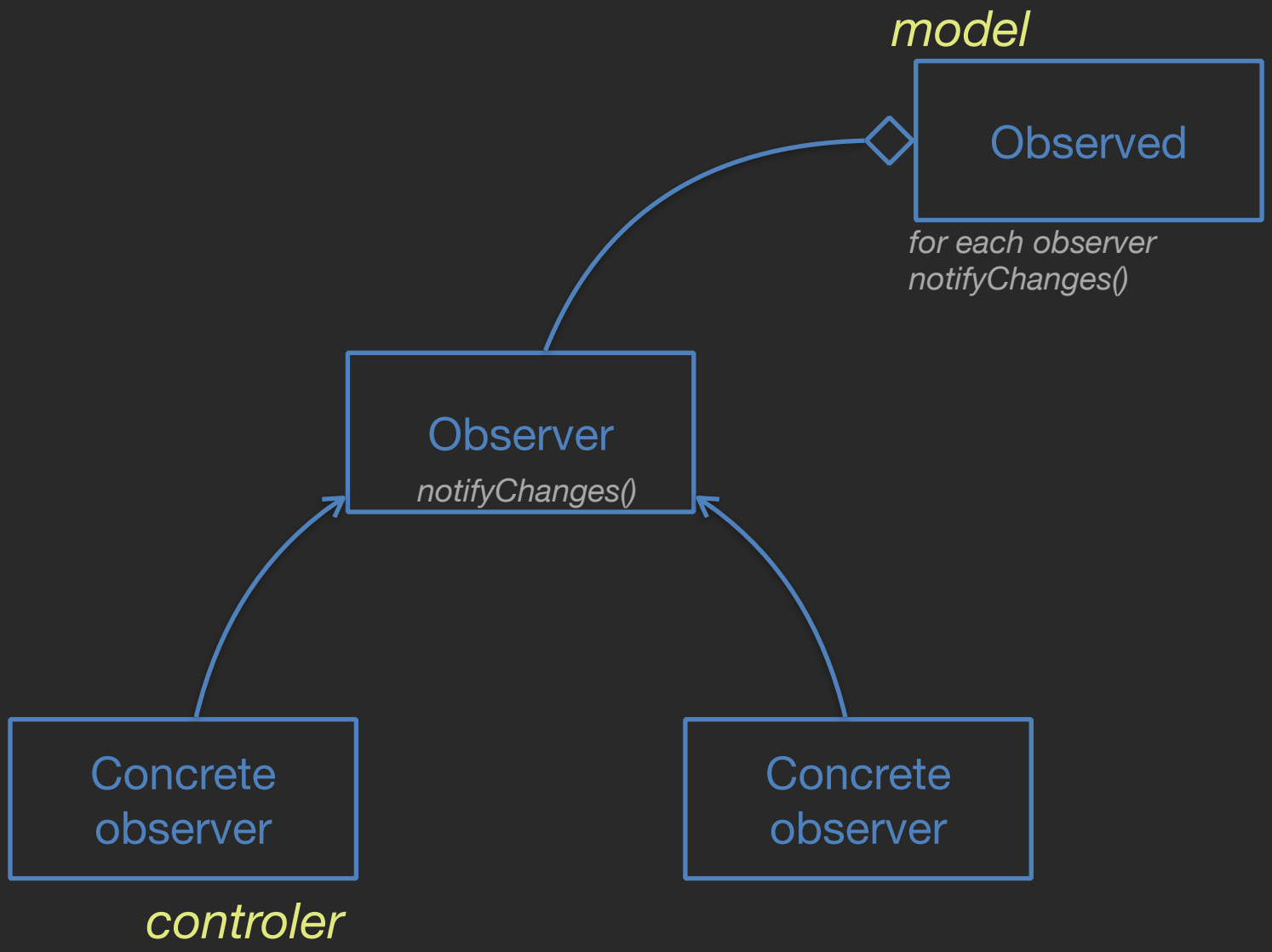
Adding a link from model to controller or better the view ⇒ break the independence of model

Other solution: using delegate or observer design pattern:

- model will be the subject
- the controller will be the observer
- protocol will be used to define abstract classes of delegate

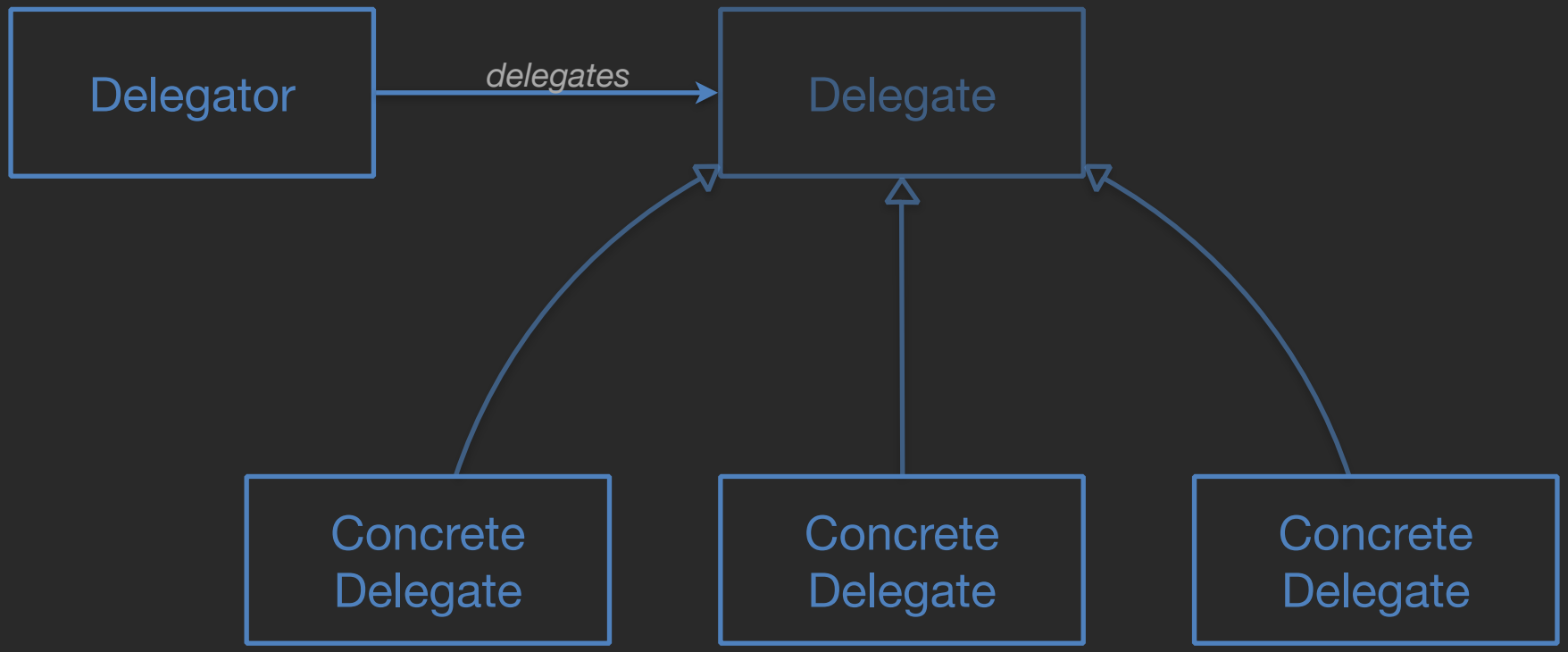


how model communicates with controller?

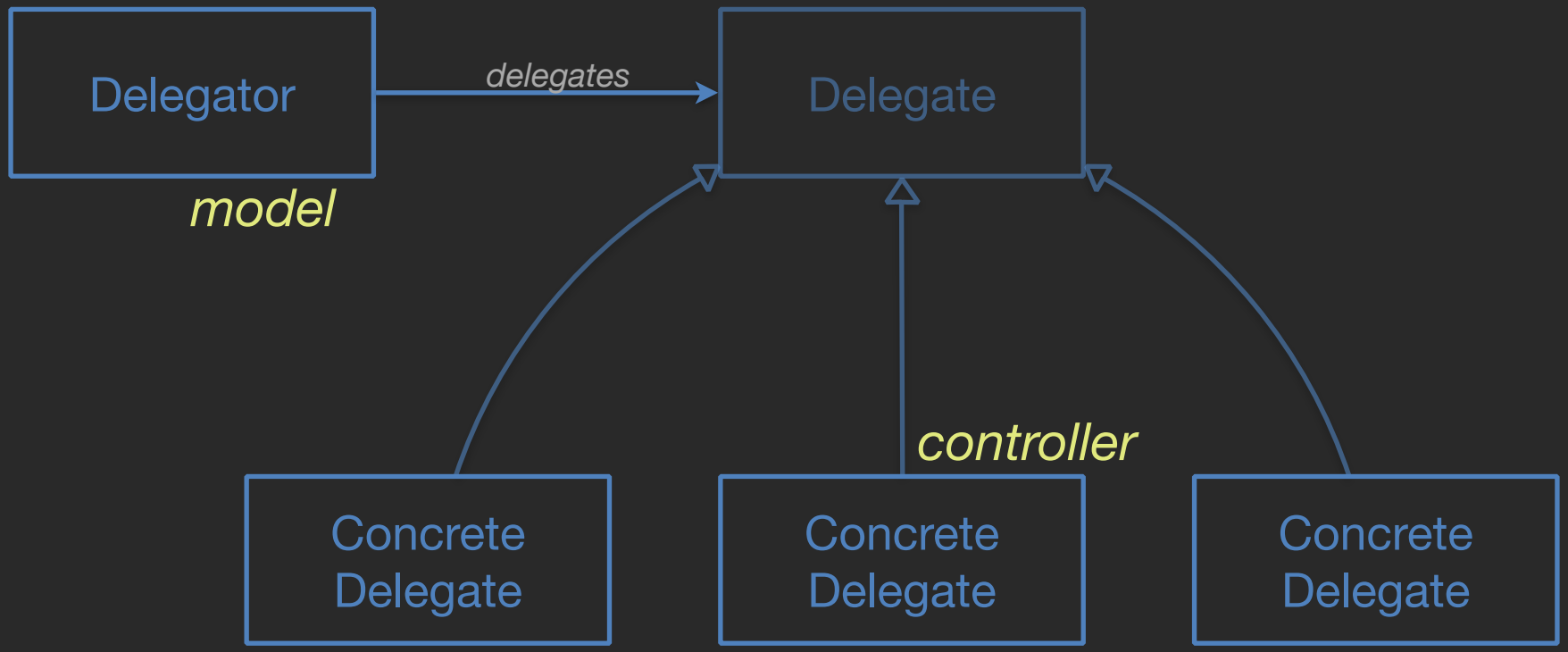


how model communicates with controller?

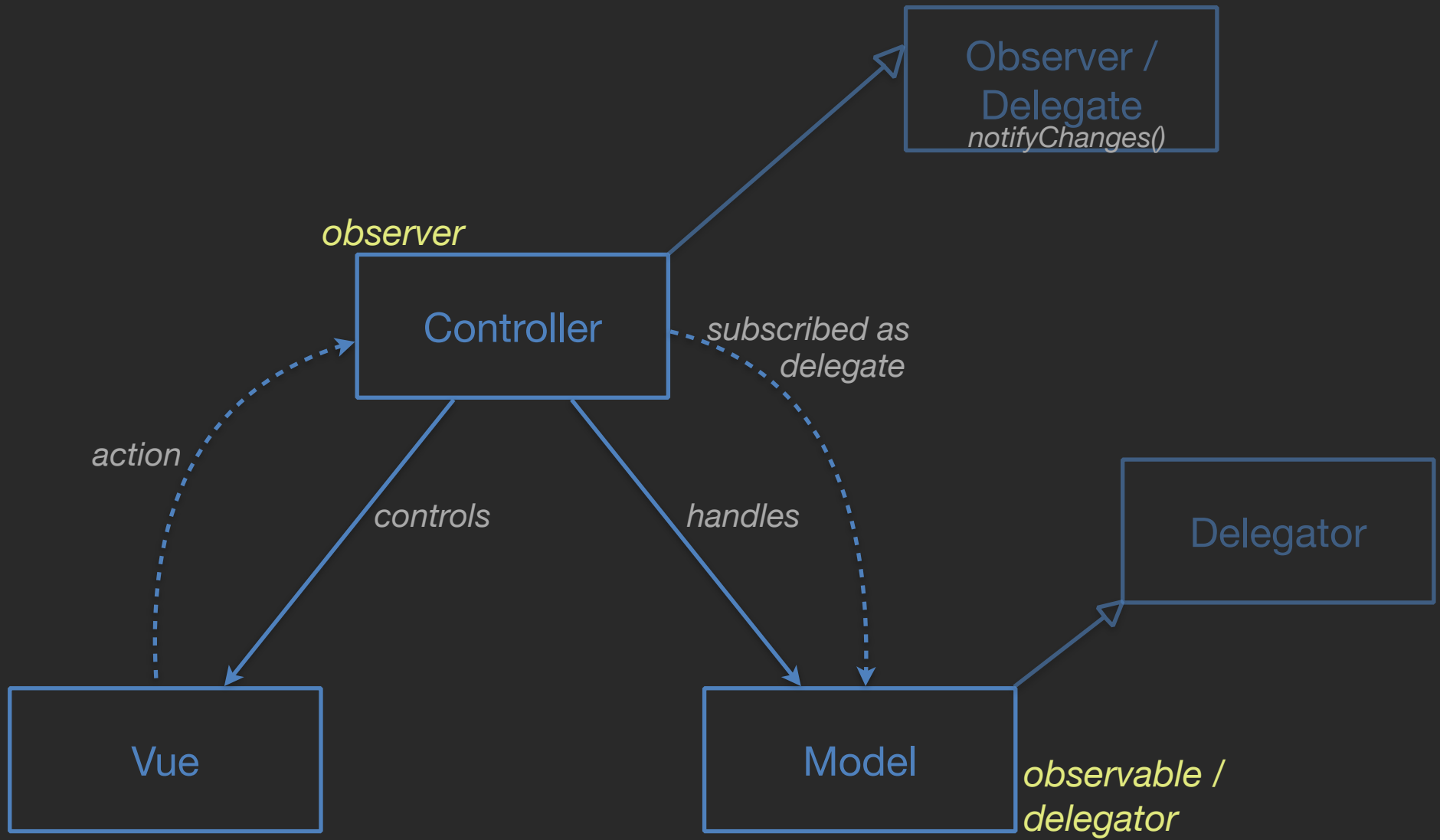
Design pattern delegate



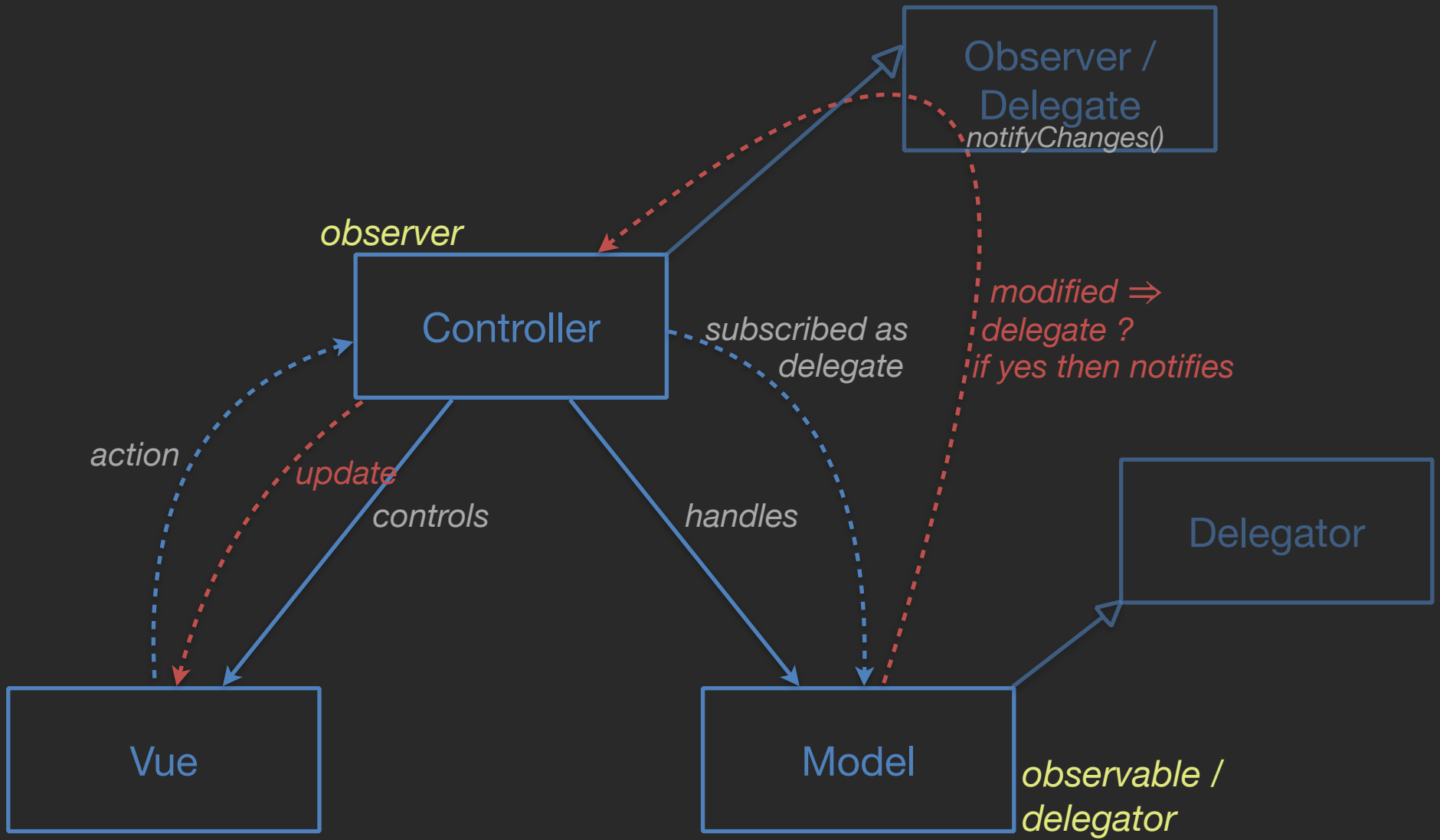
Design pattern delegate



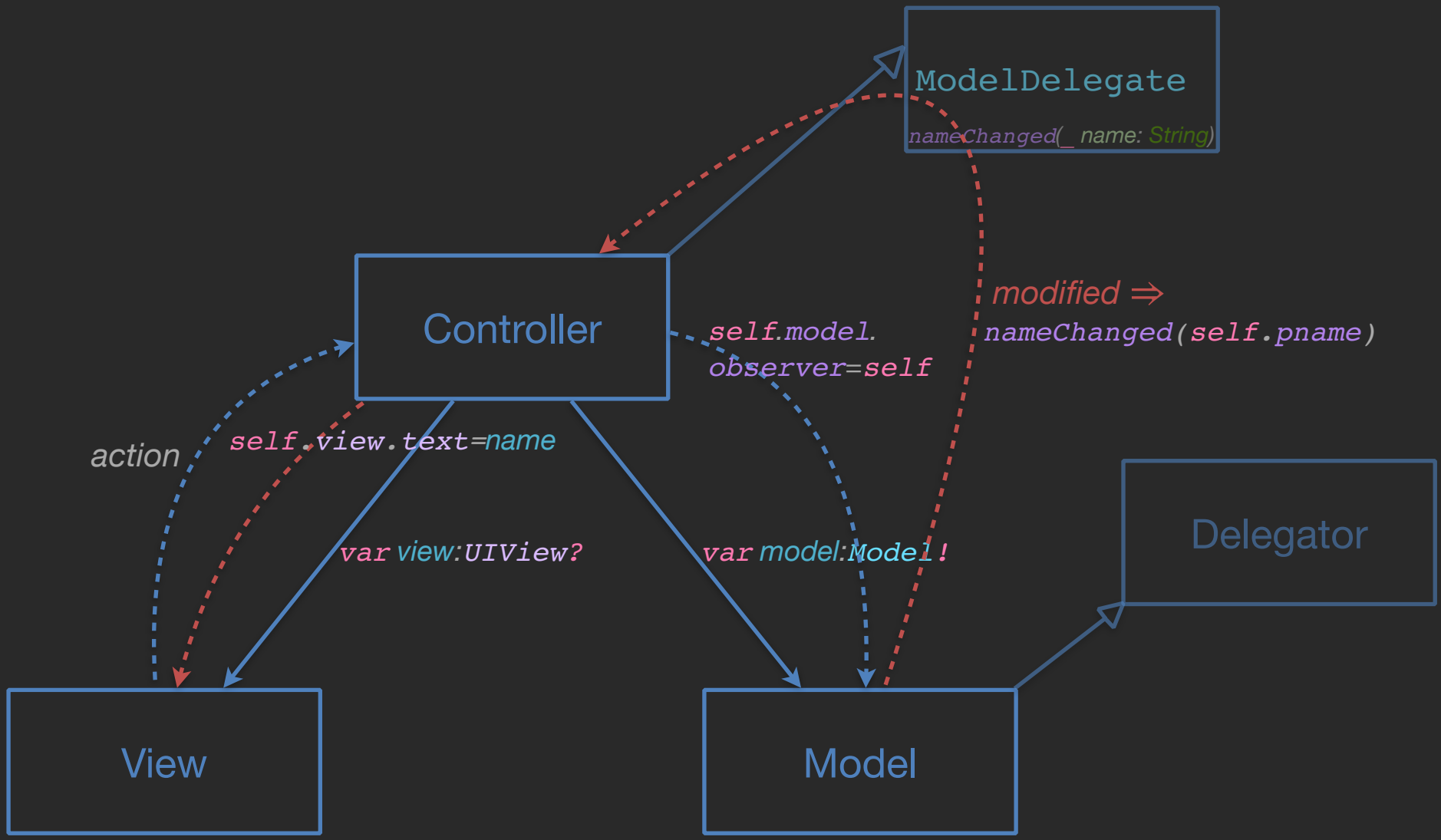
Full implementation



Full implementation



Mise en œuvre complète



A critical review

Advantages:

- we have a good separation between the view, the model and the controller ;
- the model knows nothing about the view and is therefore independent;
- the controller makes the link between the model and the view;
- the controller is automatically warned at each change of properties it observes.

But:

- the observed properties must have been foreseen as being able to be observed by the model;
- obligation to define an abstract type ModelDelegate for these properties;
- multiplication of delegates ;
- what about some properties needed by the views (e.g. `Identifiable`) or the addition of visualized properties such as revenue?

Solution: MVVM ?

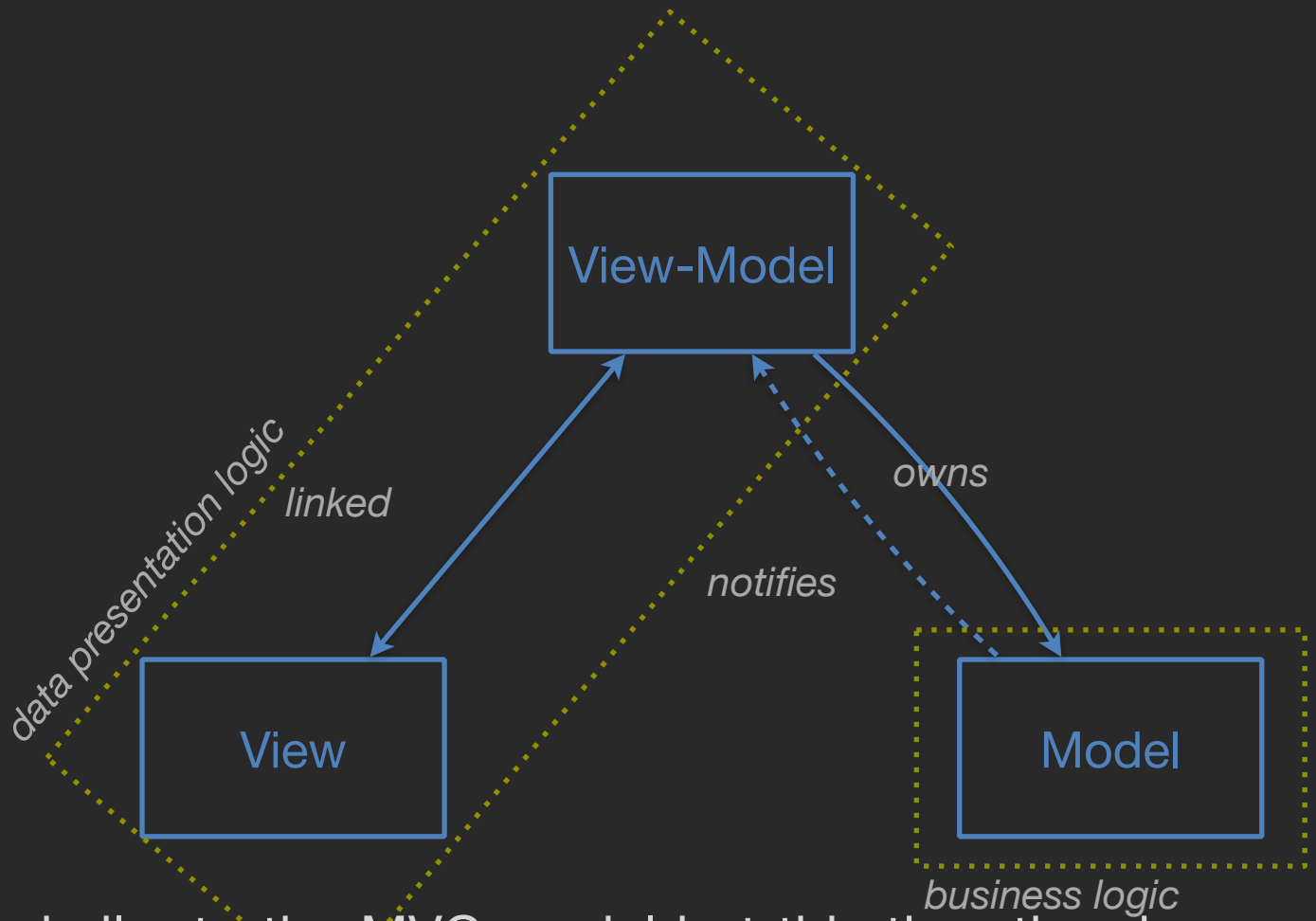
MVVM Design Pattern

Introduction to MVVM Design Pattern

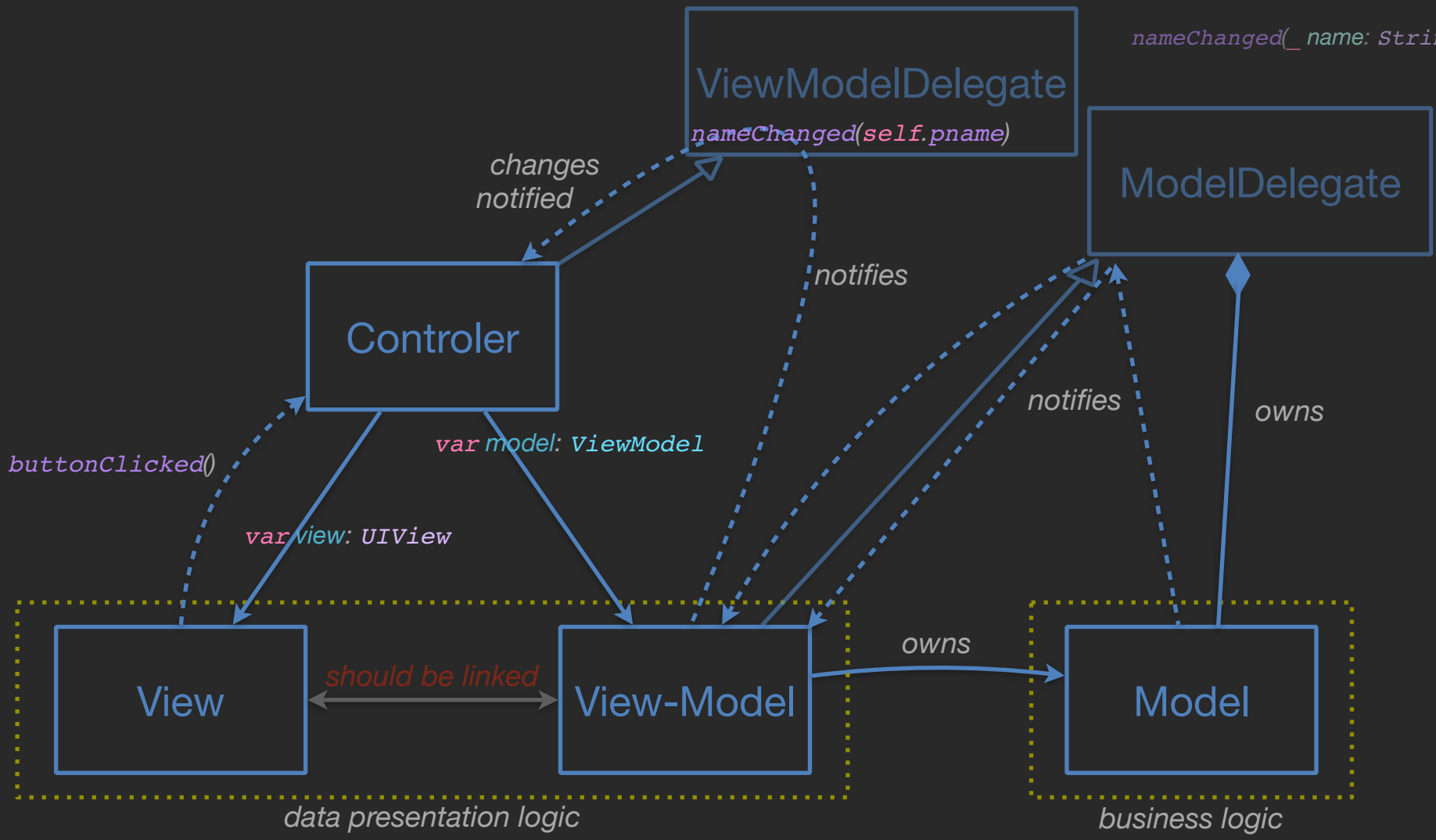
The MVC design pattern has found its limits with the appearance of richer and richer GUI libraries, but imposing certain conditions (types, properties) to the models to be displayed, such as

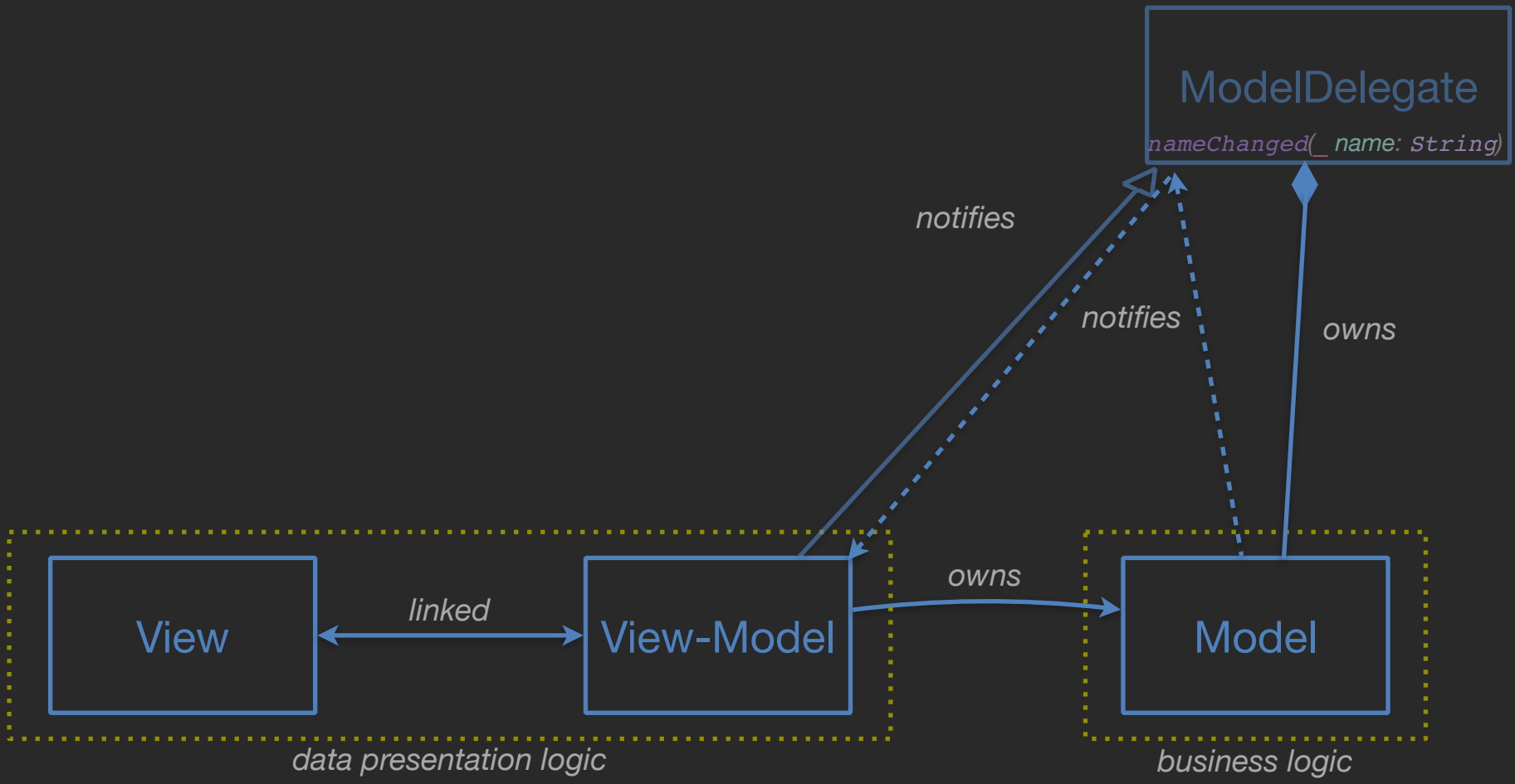
- a unique id for models to be displayed in list
- a particular type for the model: `Encodable`, `Equatable`, etc...

Proposed by Microsoft, the model-view-model proposes to define an intermediary: the model-view or ViewModel



In fact similar to the MVC model but this time the view-model is not a controller and only handles the presentation logic and allows to manage the constraints (particular types, id, ...) imposed by the interface system.


nameChanged(_ name: Stri
ViewModelDelegate
nameChanged(self.pname)
ModelDelegate
changes
notified
notifies
Controler
notifies
owns
buttonClicked()
var model: ViewModel
var view: UIView
View
should be linked
View-Model
owns
Model
data presentation logic
business logic



With SwiftUI, the *Combine* framework (reactive programming) allows a direct and adapted application of this design pattern.

Combine Framework

Introduction

SwiftUI has chosen to use a declarative interface associated with *Combine* framework which allows to introduce reactive programming.

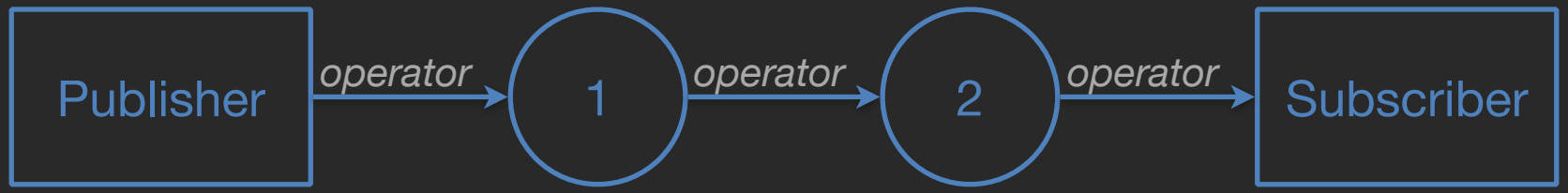
We will not describe here in detail this framework but rather the shortcuts introduced by SwiftUI, introduced by decorators, that allow to use its features simply.

The framework allows more finesse, especially to chain treatments with operators like `map`, `pipe`, etc... similar to those of javascript for asynchronous treatments; the counterpart is a less simple implementation.

Combine framework and SwiftUI

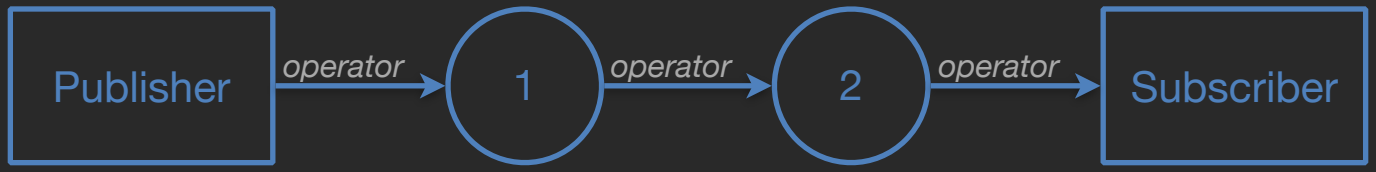
The combine framework provided with Swift allows to introduce reactive programming.

It is easily used with SwiftUI thanks to default implementations of Publisher and Subscriber



Publishers are set by using following operators:

1. `@State` : allows a property to be read and set by SwiftUI
2. `@StateObject` : allows a property to instantiate an *observable* object
3. `@ObservedObject` : allows a property to subscribe itself as an observer of an *observable* object and to trigger update of UI when necessary
4. `@EnvironmentObject` : allows to use an instance of an *observable* object from different views



As soon as a property is declared as a *Publisher* via one of the previous decorators, SwiftUI become a *Subscriber* and will therefore trigger evaluation of `body` property and so update of the view.

Important remarks:

- *user interface* is declared as the *evaluation* of a *computed property*
- there is no *business logic*, just the evaluation of a `View` value
- the view is *updated* only when the `body` property is *evaluated*
- `body` property is evaluated
  - when screen/view is created
  - *each time a publisher is modified* ⇒ so modification must be detected to trigger update of the view

Properties @State, TextField and Stepper

`@State` : local and private property

Any change in value trigger redisplay of the view

Example :

```
struct ContentView: View {
    @State private var remerciement = true
    var body: some View {
        VStack {
            Toggle(isOn: $remerciement) {
                Text("Message de bienvenue")
            }.padding()
            if remerciement {
                Text("Hello World!")
            }
        }
    }
}
```

Properties @State, TextField and Stepper

`@State` : local and private property

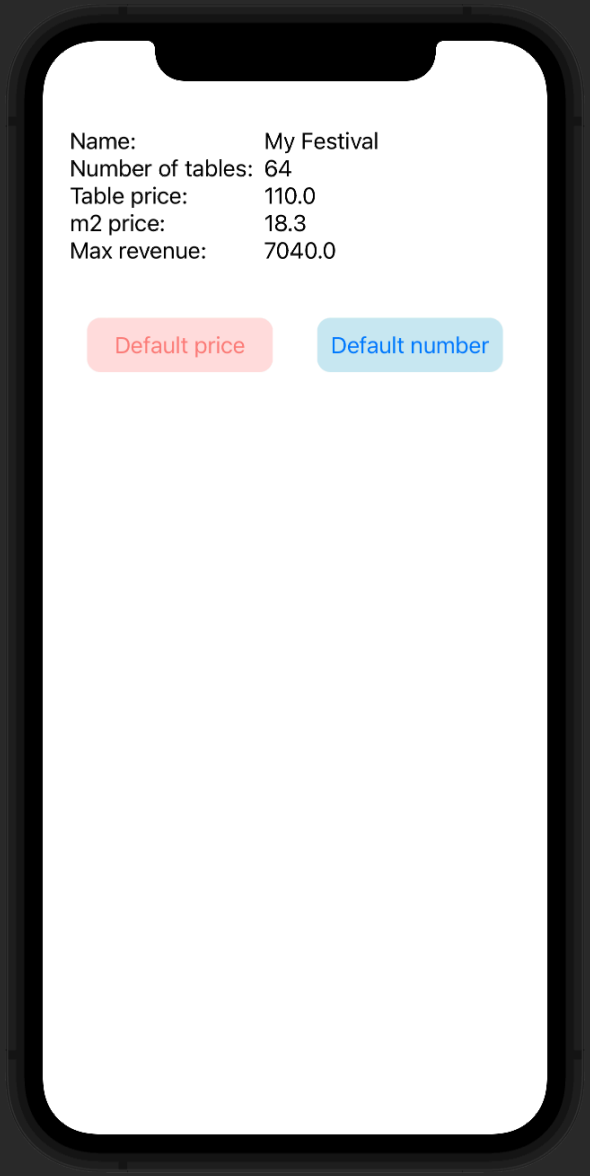
Any change in value trigger redisplay of the view

Example :

```
struct ContentView: View {
    @State private var remerciement = true
    var body: some View {
        VStack {
            Toggle(isOn: $remerciement) {
                Text("Message de bienvenue")
            }.padding()
            if remerciement {
                Text("Hello World!")
            }
        }
    }
}
```

Exercise 6: @State

Using your interface from exercise 3, 4 or 5, make sure that pressing buttons changes the values of the number of tables or the price of the table (fixed values)



TextField and Stepper

Some components allow you to enter values.

These components expect a link to a `Publisher` property (i.e. with `@State` decorator), link indicated by the `$` decorator in front of the property name.

Using a `TextField` and a `Stepper` :

```
TextField("placeholder", text: $property)
Stepper("Label", value: $property, in: range, step:
step)
```

It is possible to specify a function for incrementing and decrementing:

```
Stepper("Label", value: $property, onIncrement: {},
onDecrement: {})
```

TextFields allow to enter only text, any text; but it is possible to use a Formatter to limit the input.

Example :

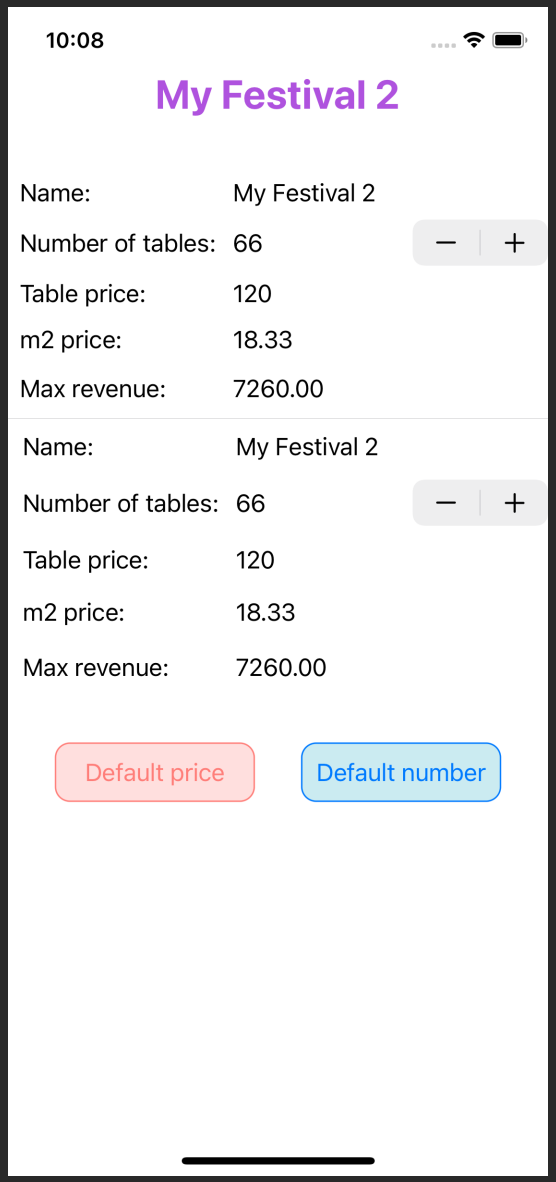
```
let formatter: NumberFormatter = {
  let formatter = NumberFormatter()
  formatter.numberStyle = .decimal
  formatter.minimumFractionDigits = 0
  formatter.maximumFractionDigits = 2
  return formatter
}()
@State var nval: Double = 0
TextField("Enter decimal value", value:$nval,
          formatter: formatter)
```

It is also possible to trigger an action upon validation of an entered value:

```
TextField("Title", value:$nval).onSubmit{ }
```

Exercise 7 : Entering values

1. Make a new interface using your solutions from exercise 3 and 5 and add a second interface that displays the values twice
2. Add a title at the top of the screen (a centered Text() and .font(.title)
3. Add Stepper() to change the number of tables and a TextField to enter the table price
4. Make the input of these values adjust the display correctly



@StateObject

The disadvantage of working with `@State` is that our view manages all the data and ultimately deals with business logic. By Apple's own point of view:

> *You might also find this kind of storage convenient while you prototype, before you're ready to make changes to your app's data model.*

Conclusion :

- The `@State` must be reserved for variables useful only to the interface
- Instead, go back to basics and use a model: `Festival`

Exercise 8: Why not use@State with our own model?

Let's take our example from exercise 7 and:

define a `Festival` model class by making sqmprice and maxRevenue computed properties

replace the `@State` properties by an `@State var` festival property

Test your app

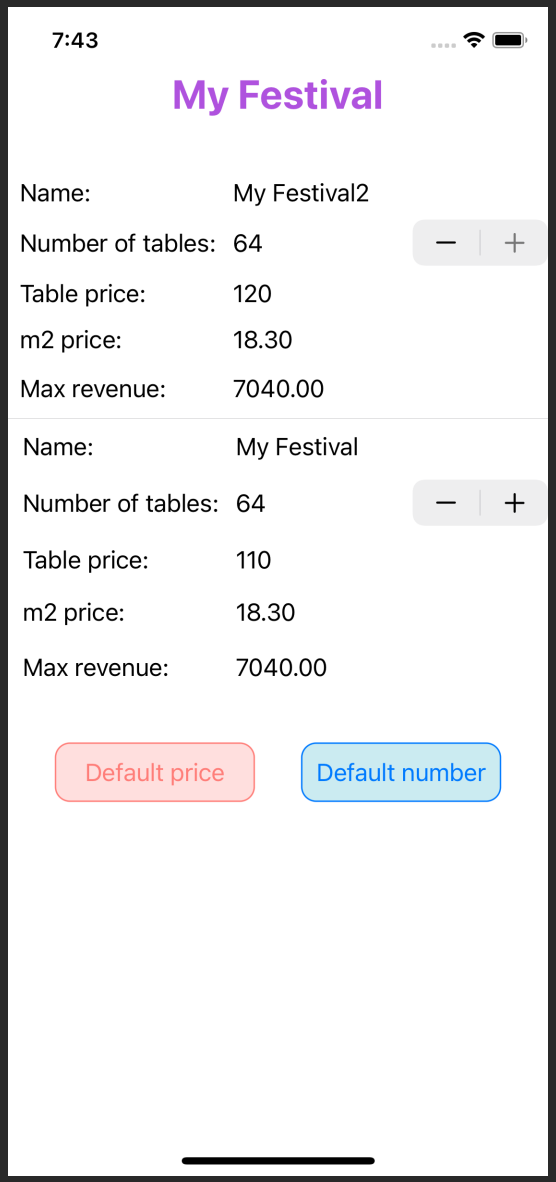
Exercise 8: Why not use@State with our own model?

Let's take our example from exercise 7 and:

define a `Festival` model class by making sqmprice and maxRevenue computed properties

replace the `@State` properties by an `@State var` festival property

Test your app



Conclusion from exercise :

- touching the `Stepper` does not cause any change
- entering a new name works but does not update the other displays of the name
- same thing for the price of the table, which also does not change the price per m2 and the maximum income

Why ?

Conclusion from exercise :

- touching the `Stepper` does not cause any change
- entering a new name works but does not update the other displays of the name
- same thing for the price of the table, which also does not change the price per m2 and the maximum income

Why ?

The input does well change properties of festival object. But *SwiftUI* doesn't detect the change and therefore doesn't trigger `body` evaluation and therefore the display update.

Conclusion from exercise :

- touching the `Stepper` does not cause any change
- entering a new name works but does not update the other displays of the name
- same thing for the price of the table, which also does not change the price per m2 and the maximum income

Why ?

The input does well change properties of festival object. But *SwiftUI* doesn't detect the change and therefore doesn't trigger `body` evaluation and therefore the display update.

SwiftUI observes the *value of the object property*, i.e. its *reference* (reference type), which *does not change* and therefore does not require, from this point of view, an update of the view.

Solution : @StateObject

What exactly does Apple's definition mean?

> *A property wrapper type that instantiates an observable object.*

Not just any object can be used, it must be an observable object:

> *A type of object with a publisher that emits before the object has changed.*

This type of object can therefore have *Publishers*, in the sense of Combine, which *will be observed* by SwiftUI.

In practice:

- the object must be an `ObservableObject`
- the properties that shall become *Publishers* must be decorated by the `@Published`

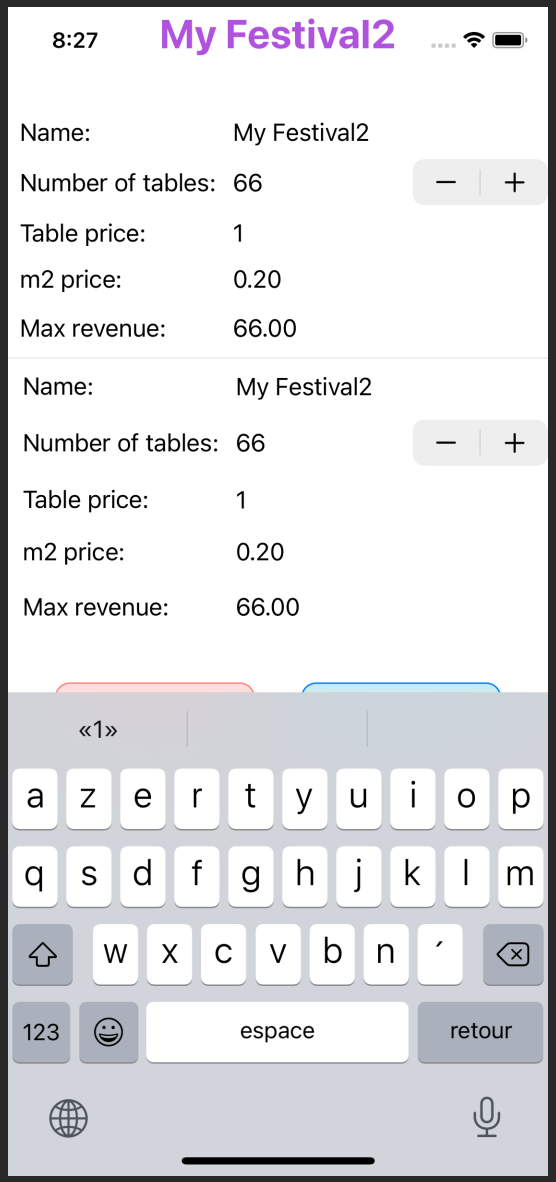
Example :

```
class Festival : ObservableObject{
  static var sqmTable : Double = 6.0
  @Published public var name: String
  @Published public var tablesmax: Int
  @Published public var tableprice: Double
  public var sqmprice: Double {
    return round(tableprice*10/Festival.sqmTable)/10 }
  public var maxRevenue : Double {return tableprice*Double(tablesmax)}
  init(name: String, tablesmax: Int = 64, tableprice: Double = 110 ){
    self.name = name
    self.tablesmax = tablesmax ; self.tableprice = tableprice
   }
}
```

Exercise 9 : Use of @StateObject

Take your solution from exercise 8, and make `Festival` of type `ObservableObject` and publish its properties (see previous example)

Test your solution



@ObservedObject & @EnvironmentObject

The `@ObservedObject` decorator acts in the same way as `@StateObject` but *does not require* the object to be instantiated at declaration.

This decorator will be used to retrieve observed objects instantiated in a previous screen.

The `@EnvironmentObject` decorator acts like an `@ObservedObject` but allows access to this type of object in different screens without having to pass them as parameters.

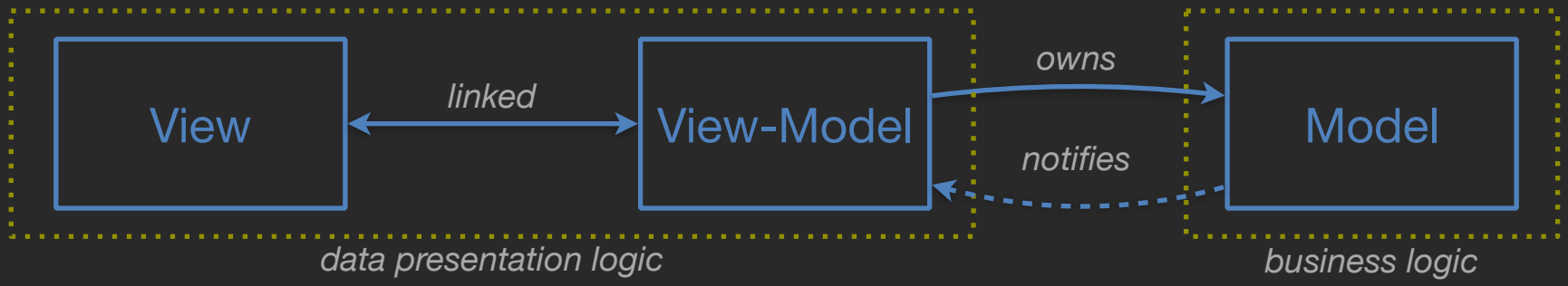
It is therefore used for data that are shared by multiple screens and that have a lifetime greater than that of a single screen, such as a shopping cart

We will see in detail these two decorators when we explain the navigation between screens.

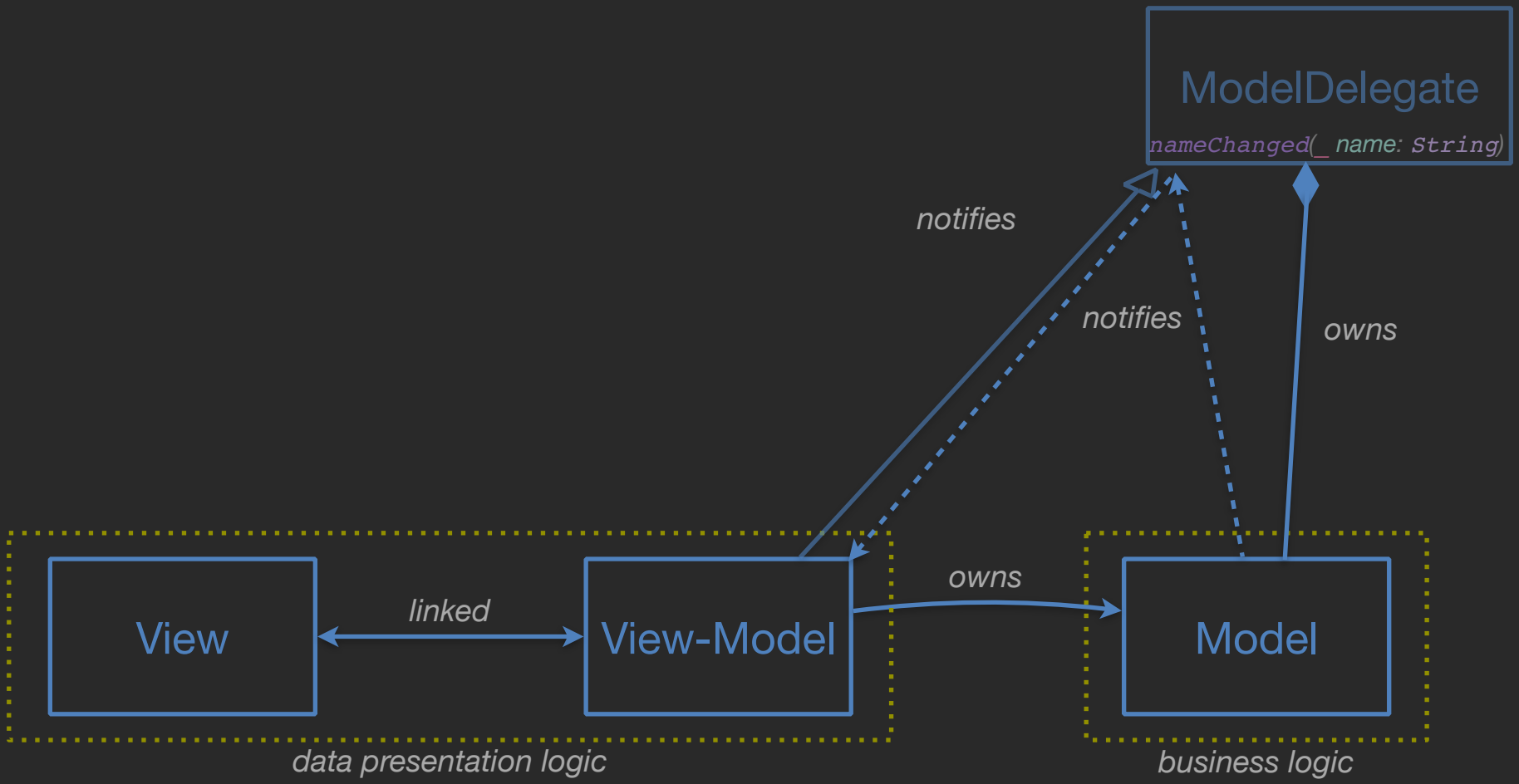
MVVM and SwiftUI

Implementation on an example

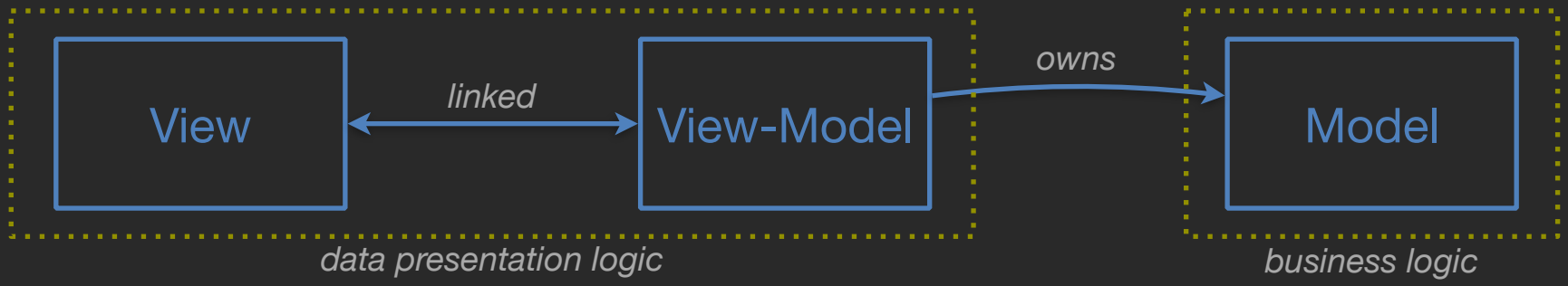
Reminder: MVVM Design Pattern



Reminder: MVVM Design Pattern



First try: simple MVVM Design Pattern



Implementing simple MVVM with SwiftUI

Implementing the MVVM design pattern is easy with SwiftUI:

- Design a `FestivalViewModel` of type `ObservableObject`
- Declare with the `@Published` decorator the properties managed by the `View`
- `FestivalViewModel` updates `Festival` when an `@Published` property is modified by using `willSet` or `didSet` property observers

Exercise 10 : go to MVVM

In fact, the `Festival` class as written in exercise 9 was a `ViewModel` after all.

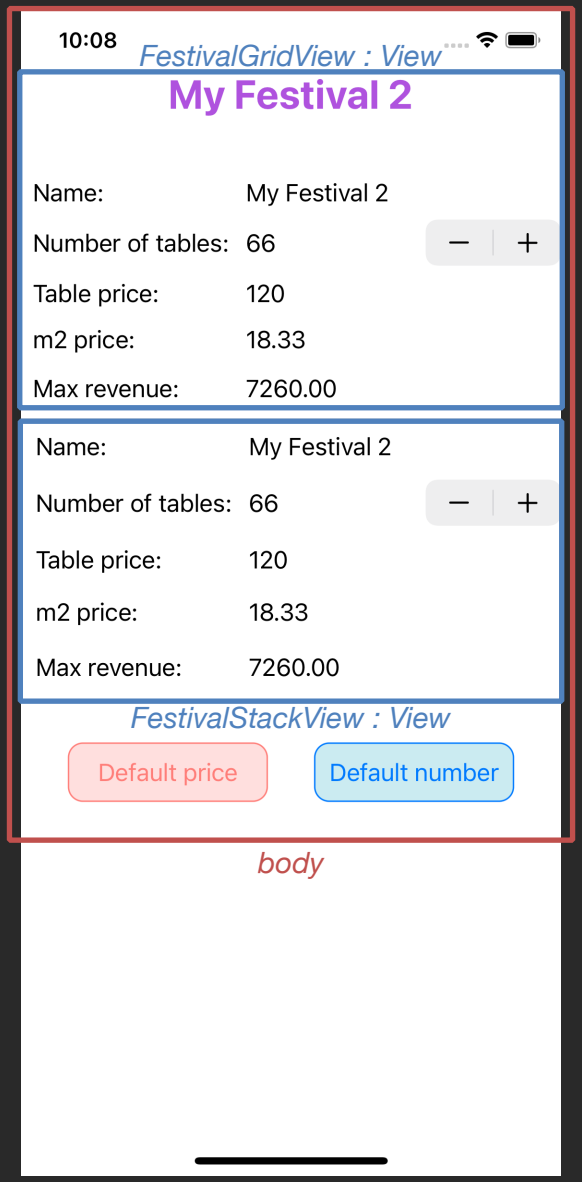
In order to visualize the implementation of the MVVM design pattern,

1. Make a `Festival` types, adding the computed property revenueMax to `Festival` and making sqmprice a stored property
2. Create a `FestivalViewModel` type; this type only needs the properties that are output (table name, number, and price). Note: `FestivaViewModel` initializes itself with an instance of `Festival` to be able to update Festival when its own properties are changed by the view.

3. Create two `View` `FestivalGridView` and `FestivalStackView` corresponding to the two parts of your interface and used in your `body`, as shown on the right.
   Each of these components will take a `Festival` as parameters when created and has a `FestivalViewModel` properties to manage the display.

4. Compose your `ContentView` by making `body` use your `FestivalGridView` and `FestivalStackView`.
   `ContentView` now takes a `FestivalViewModel` as a parameter init.

5. Test your application by making sure that each component has its own `ViewModel`



Implementing full MVVM with SwiftUI

Model should now notifies its ViewModel:

- Design a `FestivalDelegate` abstract type that has three functions: `nameChanged`, `tablePriceChanged`, `tablesNumberChanged`, `sqmPriceChanged`
- Make `FestivalViewModel` becomes of type `FestivalDelegate`
- Make Festival owns a set of FestivalDelegate so that it notifies, via the delegation functions, all its FestivalDelegate when one of its property is set.

Exercise 11 : Full implementation of MVVM design

Implement a full MVVM design so that you have well only one model, that each part of the UI has its own ViewModel and that all is updated whatever where changes are made.

@StateObject vs @State

Should we only use `@StateObject` and `ViewModel` instead of simple `@State` properties?

Of course, each time you edit a data model, you must use a `ViewModel` declared with an `@StateObject`.

But properties decorated with `@State` are still useful to, for example :

- manage variables of the interface (hidden for example)
- do input control