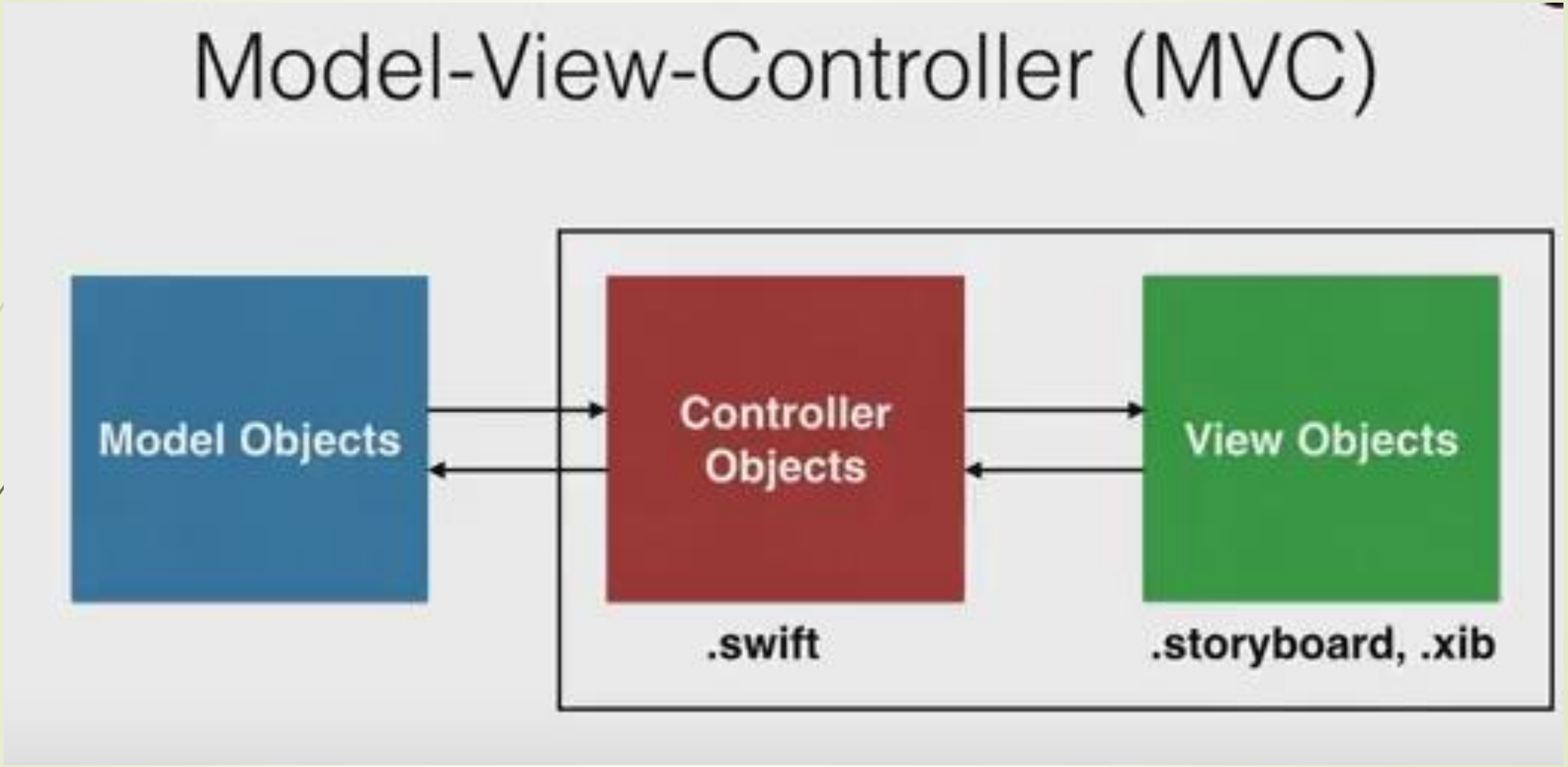
Programavimas išmaniesiems įrenginiams

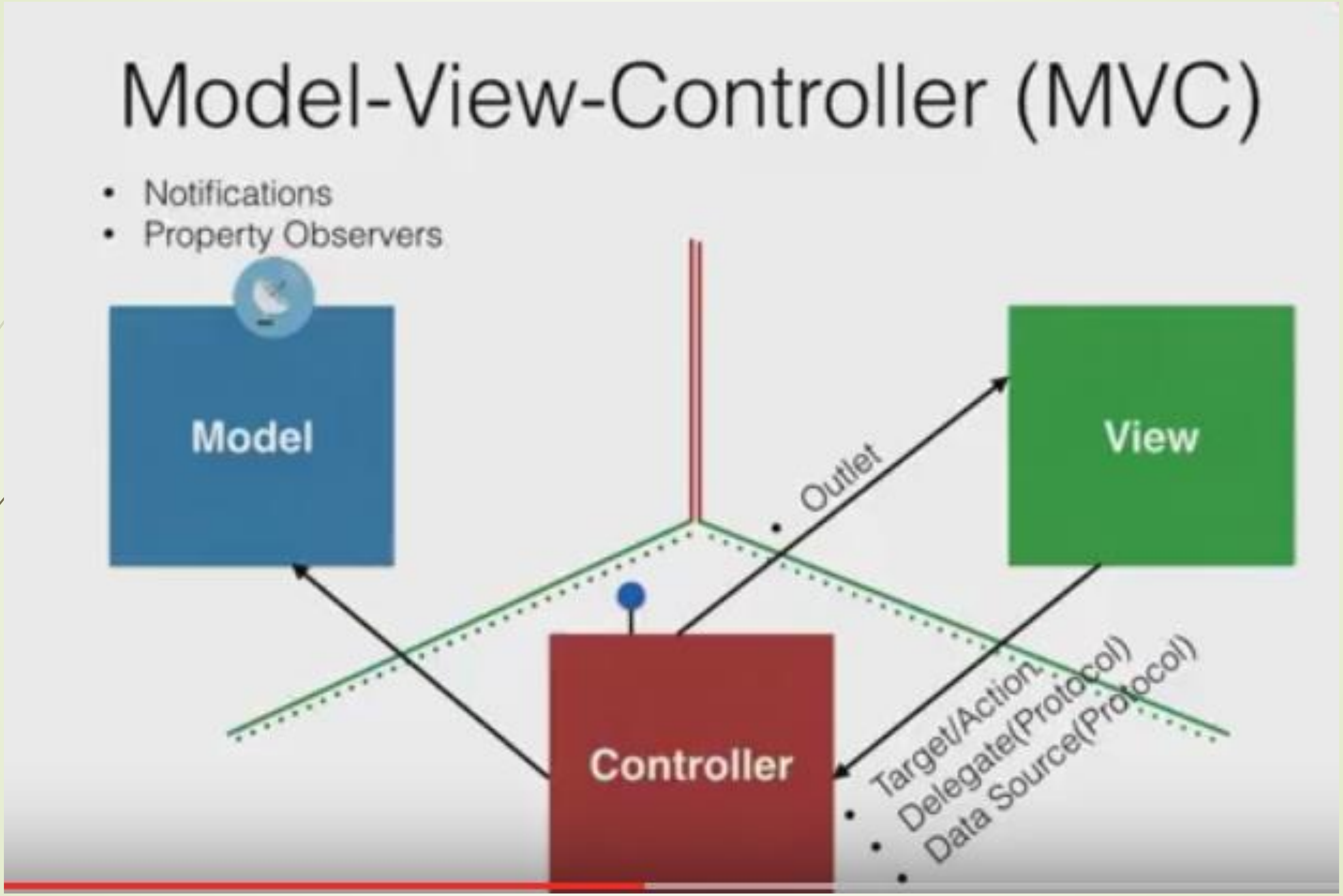
Kas yra Model – View – Controller arba MVC?

Marius Gžegoževskis

MVC

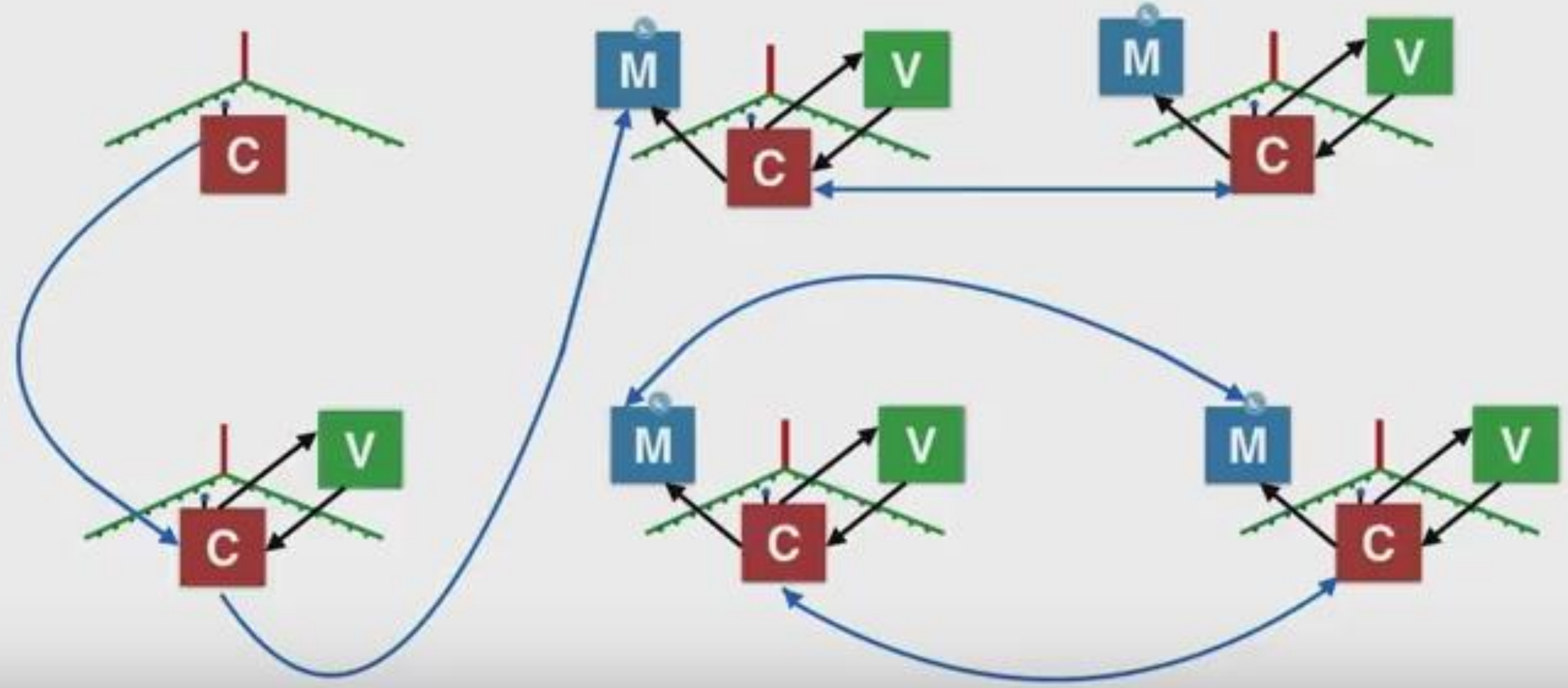
- **Model** — Tai duomenys su kuriais dirbame ir aplikacijos funkcionalumas. Kaip taisyklė gali būti atskirtas nuo pagrindinės aplikacijos. Pavyzdžiui tą pačią klasę galime naudoti tekstinio pavidalo (angl. **Text-based**) aplikacijoje arba aplikacijoje su grafine vartotojo sąsaja GUI nepakeičiant jos funkcionalumo.
- **View** — Tai vartotojo grafinis atvaizdis, kuriame jis intekraktyviai naudojasi aplikacija. Kurioje yra šio tipo komponentai: **widgets**, **controls** ir **tekstas**.
- **Controller** — Modelis ir View gali bendrauti tarpusavyje, t.y. Controlleris sujungia **Model** ir **View** tarpusavyje. Tačiau yra ir apribojimų. Modelis ir View negali tiesiogiai pakeisti nieko kas yra aprašyta Controlleryje. Controlleris gali siųsti žinutes **view** ir **model pasakant ka daryti jiems, ir gali stebėti pakeitimus.**


Model-View-Controller (MVC)
Model Objects
Controller Objects
View Objects
.swift
.storyboard, .xib

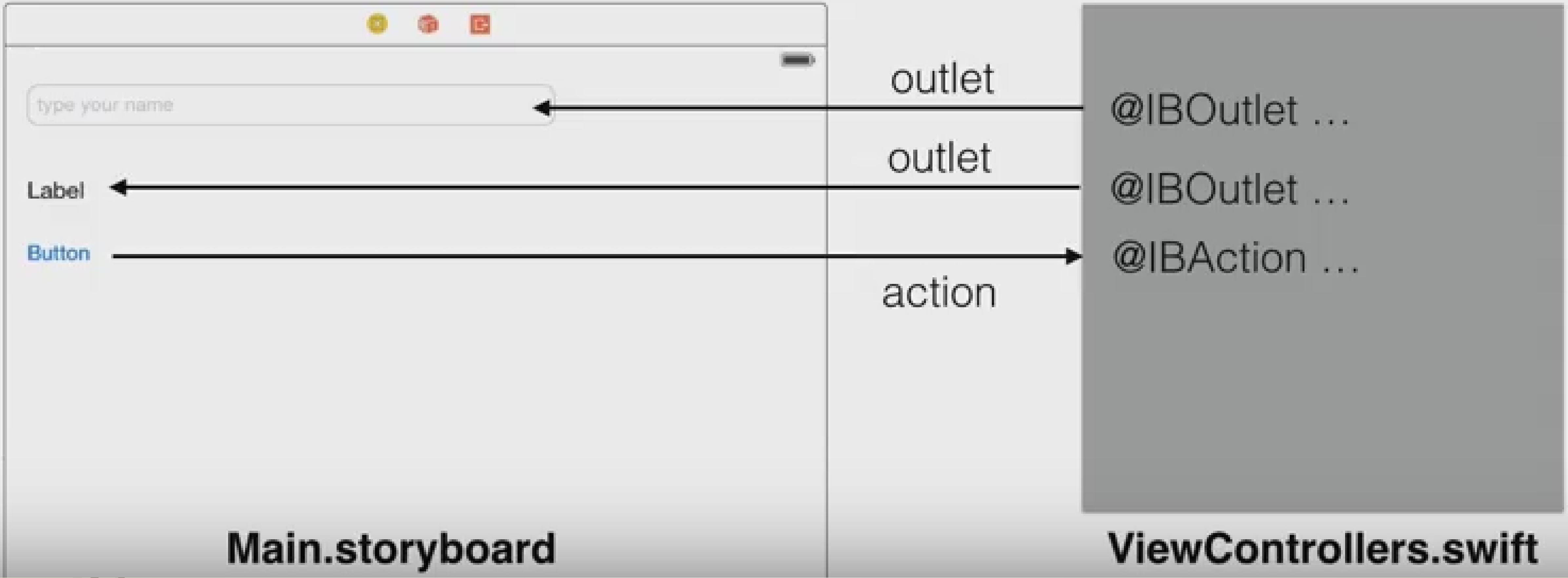


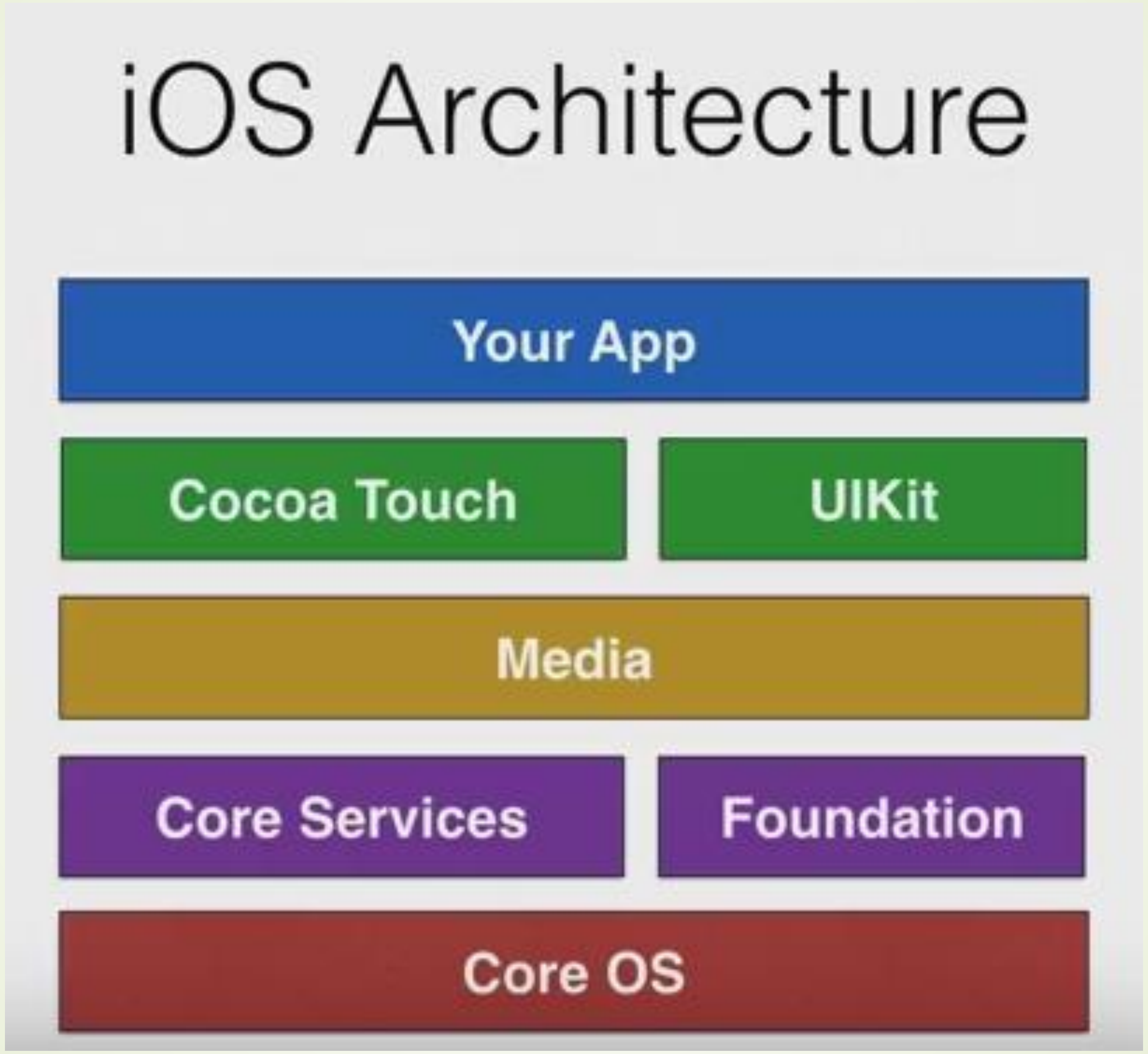
Model-View-Controller (MVC)

MVC Working together

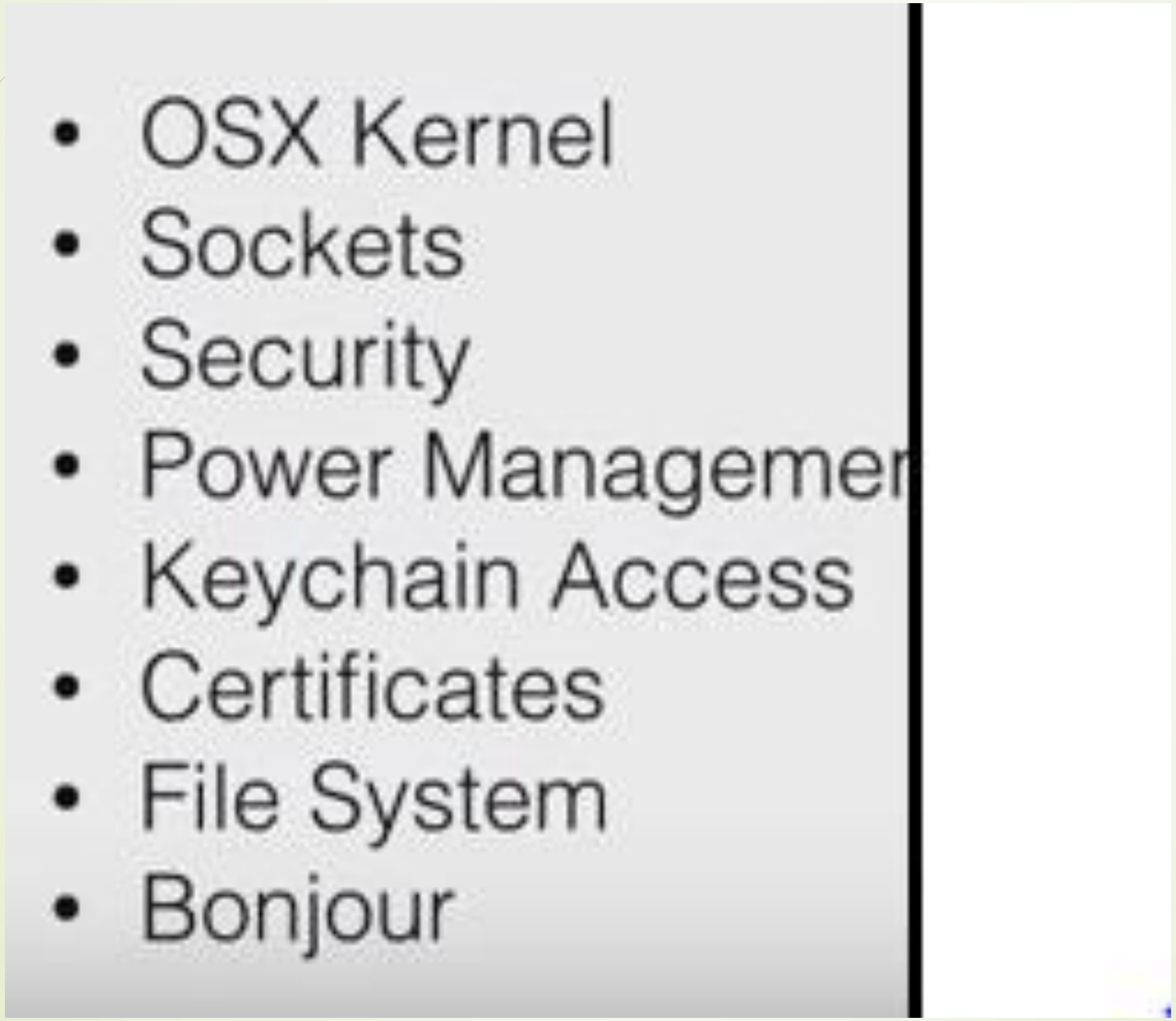


Outlets and Actions



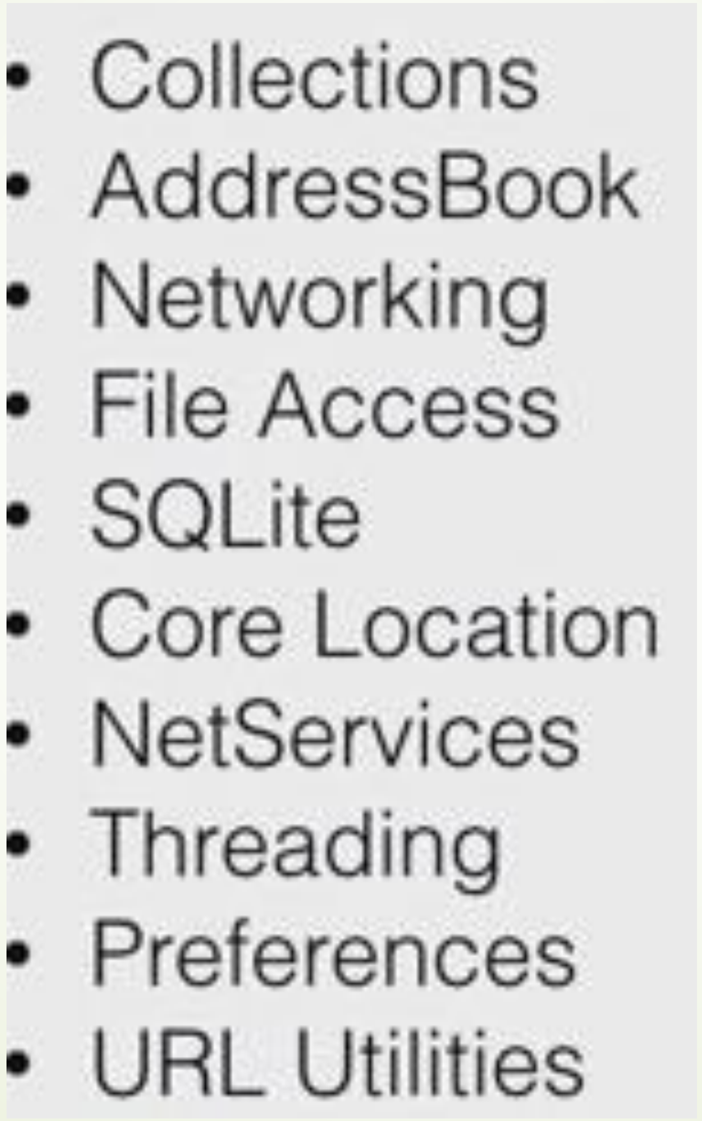

iOS Architecture
Your App
Cocoa Touch
UIKit
Media
Core Services
Foundation
Core OS

CORE OS



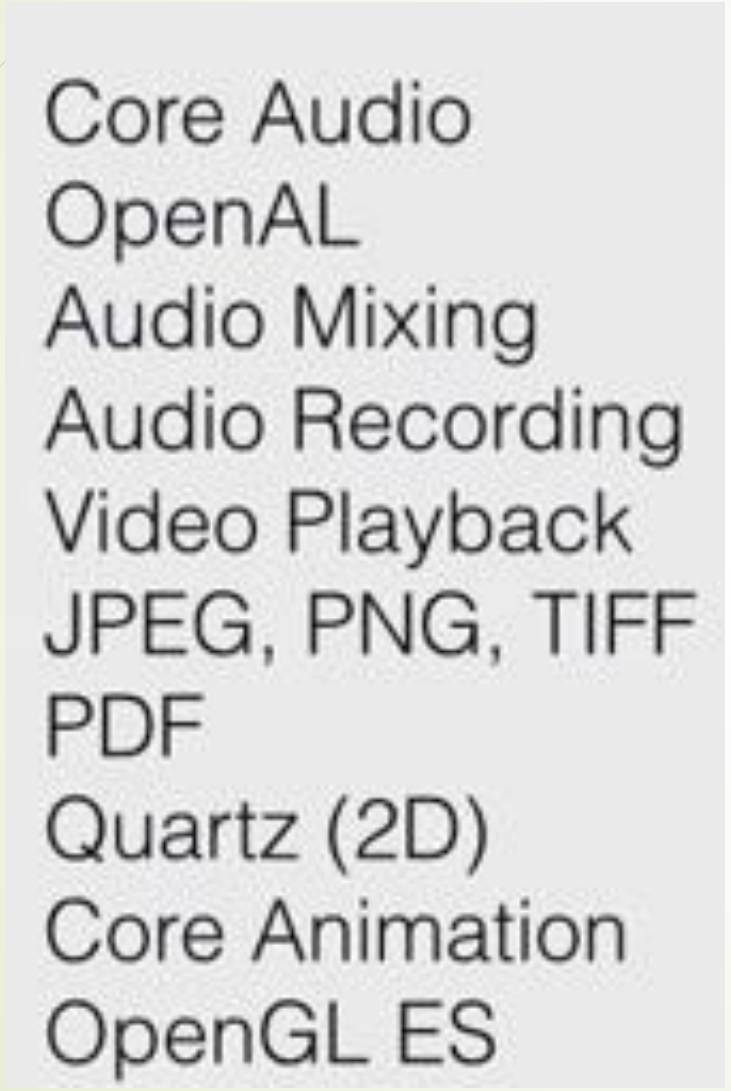
- OSX Kernel
- Sockets
- Security
- Power Managemen
- Keychain Access
- Certificates
- File System
- Bonjour

CORE SERVICES yra pasiekiami naudojant **FOUNDATION** frameworką



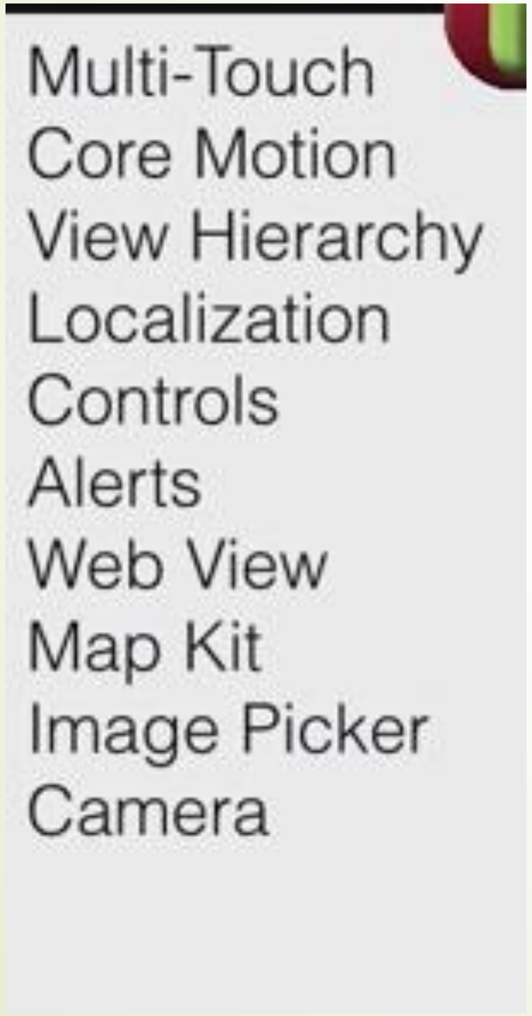
- Collections
- AddressBook
- Networking
- File Access
- SQLite
- Core Location
- NetServices
- Threading
- Preferences
- URL Utilities

MEDIA sluoksnis

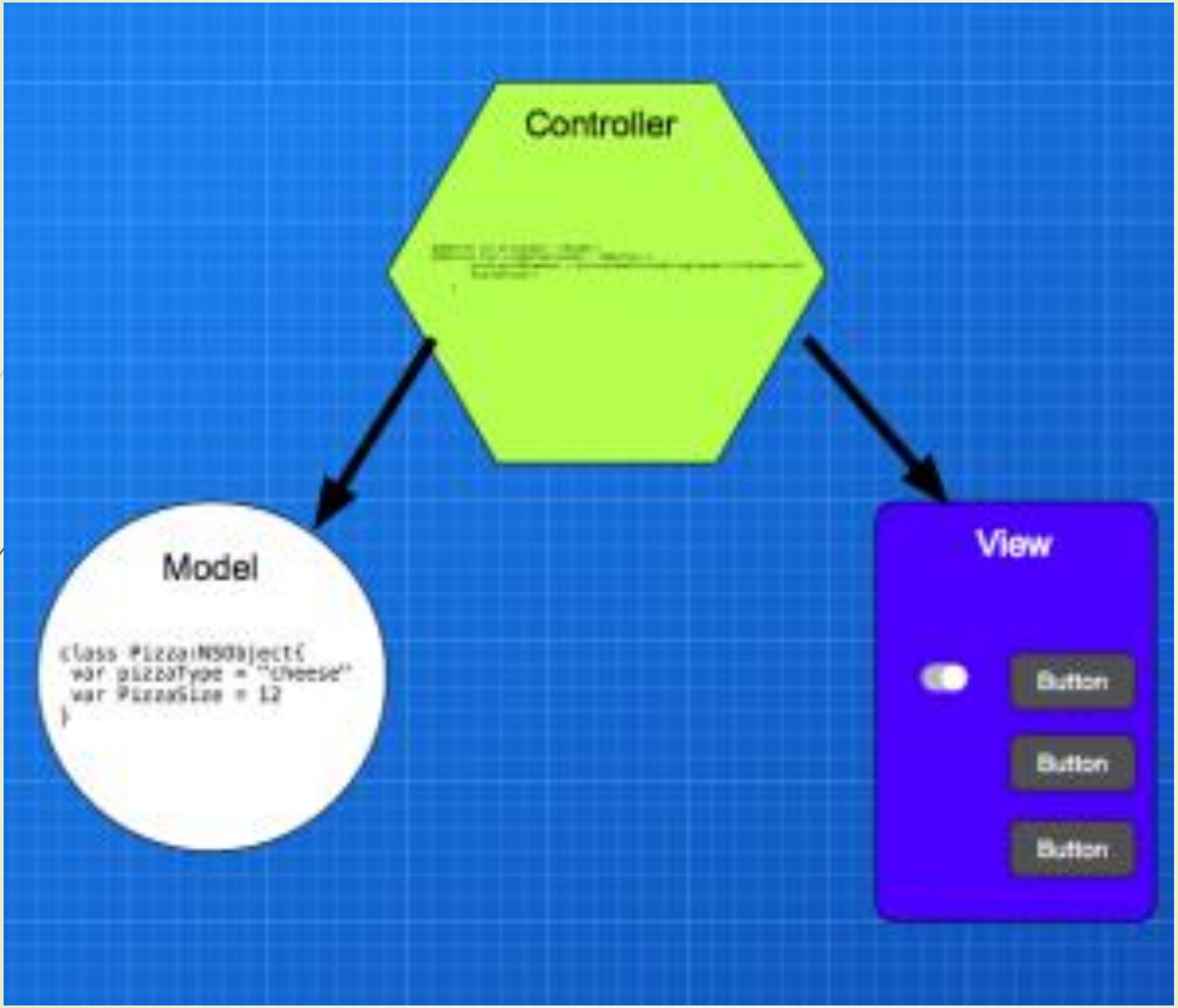


Core Audio
OpenAL
Audio Mixing
Audio Recording
Video Playback
JPEG, PNG, TIFF
PDF
Quartz (2D)
Core Animation
OpenGL ES

COCOA TOUCH yra pasiekiamas naudojant **UIKit**


Multi-Touch
Core Motion
View Hierarchy
Localization
Controls
Alerts
Web View
Map Kit
Image Picker
Camera

MVC



```
class Pizza:NSObject{
 var pizzaType = "Cheese"
 var PizzaSize = 12
}
```

MVC

- **MVC** terminas dažnai yra naudojamas dirbant su **Xcode**. Bet tai nereiškia, kad tai yra tik **iOS** arba **OS X** platformoms priklausantis. MVC yra programavimo šablonas, skirtas programoms arba applikacijoms, kuriose yra duomenų vizualus atvaizdavimas vartotojui naudojantis grafine vartotojo sąsają interaktyviai keičiantis tam tikrus parametrus.
- MVC atskiria 2 pagrindines dalis nuo aplikacijos, duomenis ir vartotojo grafinę sąsają. **Kodėl tai yra labai svarbu ?**
- Sakykime sukuriame aplikacija, kuri yra skirta **iPhone** aplikacijai tada nusprendžiame, kad mums reikia sukurti ir **iPad** versijos aplikacija, tada sugalvojame, kad norime sukurti ir **OS X** versijos.
- Naudojant **MVC,** mums tereikia pakeisti vieną iš dalių t.y. **VIEW** ir tikėtina šiek tiek kontroleryje. Programinis kodas esantis **MODEL** niekada nesikeis nepriklausomai nuo versijos, dėl kurio yra sutaupoma labai daug laiko, t.y. vienas iš pagrindinių plusų.

What is a Model

- There are parts of our program that deal with information we want to process. A pizza ordering system has a list of data giving us information about each person’s order.
- There may be other links to that data with more data about each customer, and about each pizza. In a pizza ordering system this is our model: the collection of all the data we will use in our ordering system.
- It does not in any way interact with the user. It does not display anything or does it ask for input. It is just data. Here is an example of a model made from two classes:

Modelio pavyzdys

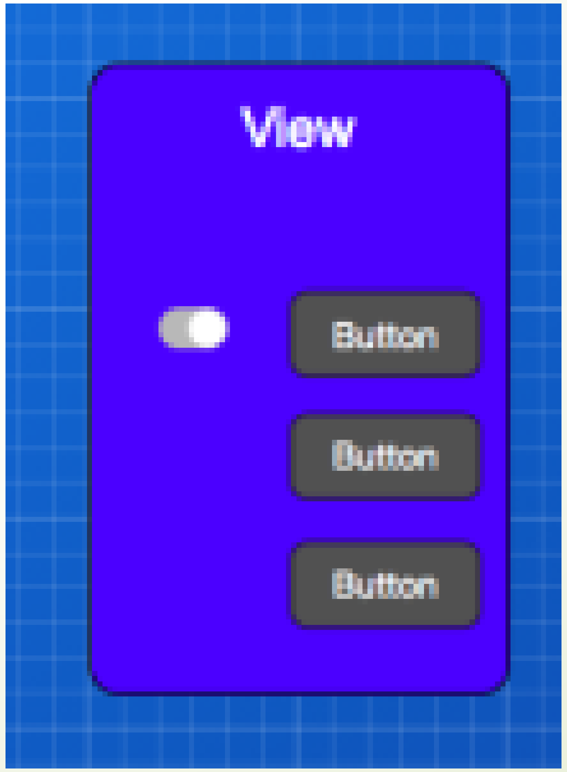
```
class Pizza{
    var pizzaTopping:String
    var pizzaCrust:String
    var pizzaSize:Double
    var pizzaRadius:Double {
    get{
    return pizzaSize / 2.0 }
    }
    let pi = 3.1415926
    let crust = 0.1
    init(topping:String,crust:String,size:Double){
        pizzaTopping = topping
        pizzaCrust = crust
        pizzaSize = size
    }
    func howMuchCheese(height:Double) -> Double{
        return  (pizzaRadius * pi * (1.0 - crust )) * height
    }
}
```

```
class PizzaOrder{
    private var pizzaArray:[Pizza] = []
    private var tableNumber:[Int] = []
    func addOrder(pizza:Pizza, table:Int){
        pizzaArray.append(pizza)
        tableNumber.append(table)
    }
    func getOrder(index:Int) -> (Pizza,Int){
        return (pizzaArray[index],tableNumber[index])
     }
    func getPizzaOrder(index:Int) -> Pizza{
        return pizzaArray[index]
    }


}
```

This model is the data for a basic pizza ordering system. An instance of class **PizzaOrder** will be a list of pizzas of class Pizza, and a list of table numbers. There is a lot more methods I should add to describe the pizza ordering process, but like **addOrder()** and **getOrder()** any method is moving data only. There is no user input or output here. They might make calculations as we do in the **howMuchCheese()** method, but again there is no user interaction here.

What is a View?

- Where all the user interaction happens is in the *view*. In **Xcode**, most people use Interface Builder either as a scene in a **storyboard** or a **.xib** file to build their views. A developer can programmatically create a view class to hold the different controls.



VIEW

- As the model never interacts with the user, the view never interacts directly with the data.
- The model doesn't do much but sit there. It might respond to a user touch with feedback such as a notifying a method somewhere, a color change when a button gets tapped or a scrolling motion at times, but that is all it does.
- The view does contain a lot of properties and methods and that tell us the state of the view. We can change the appearance and behavior of the view through methods and properties.
- The view can tell the controller that there was a change in the view, such as a button getting pressed, or a character typed. it can't do anything about it, but it can broadcast something.

Apple introduced the concept of **"storyboarding"** in iOS5 SDK to simplify and better manage screens in your app. You can still use the .xib way of development.

Pre-storyboard, each **UIViewController** had an associated **.xib** with it. Storyboard achieves two things:

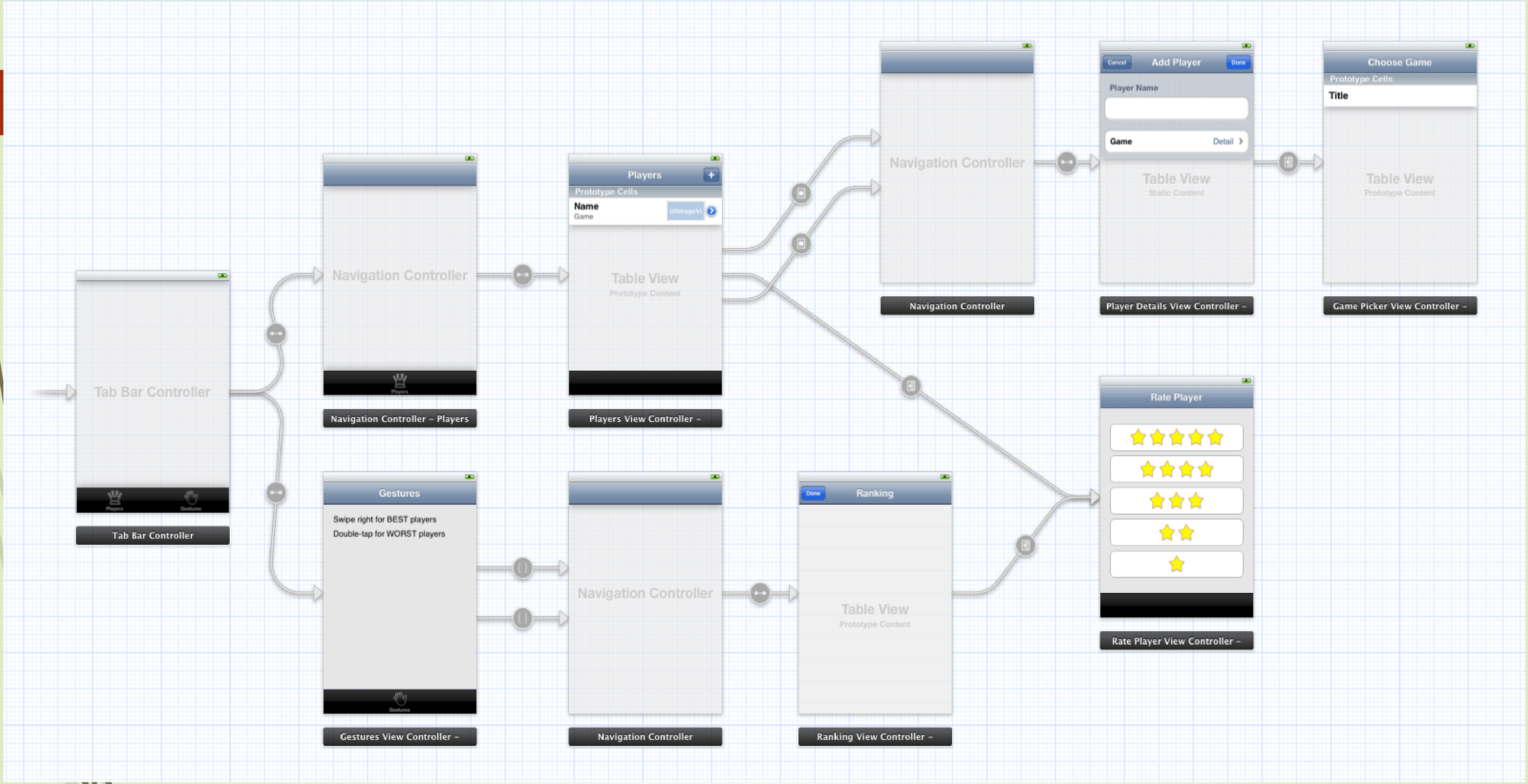
.storyboard is essentially one single file for all your screens in the app and it shows the flow of the screens. You can add **segues/transitions** between screens, this way. So, this minimizes the boilerplate code required to manage multiple screens.

Minimizes the overall no. of files in an app.

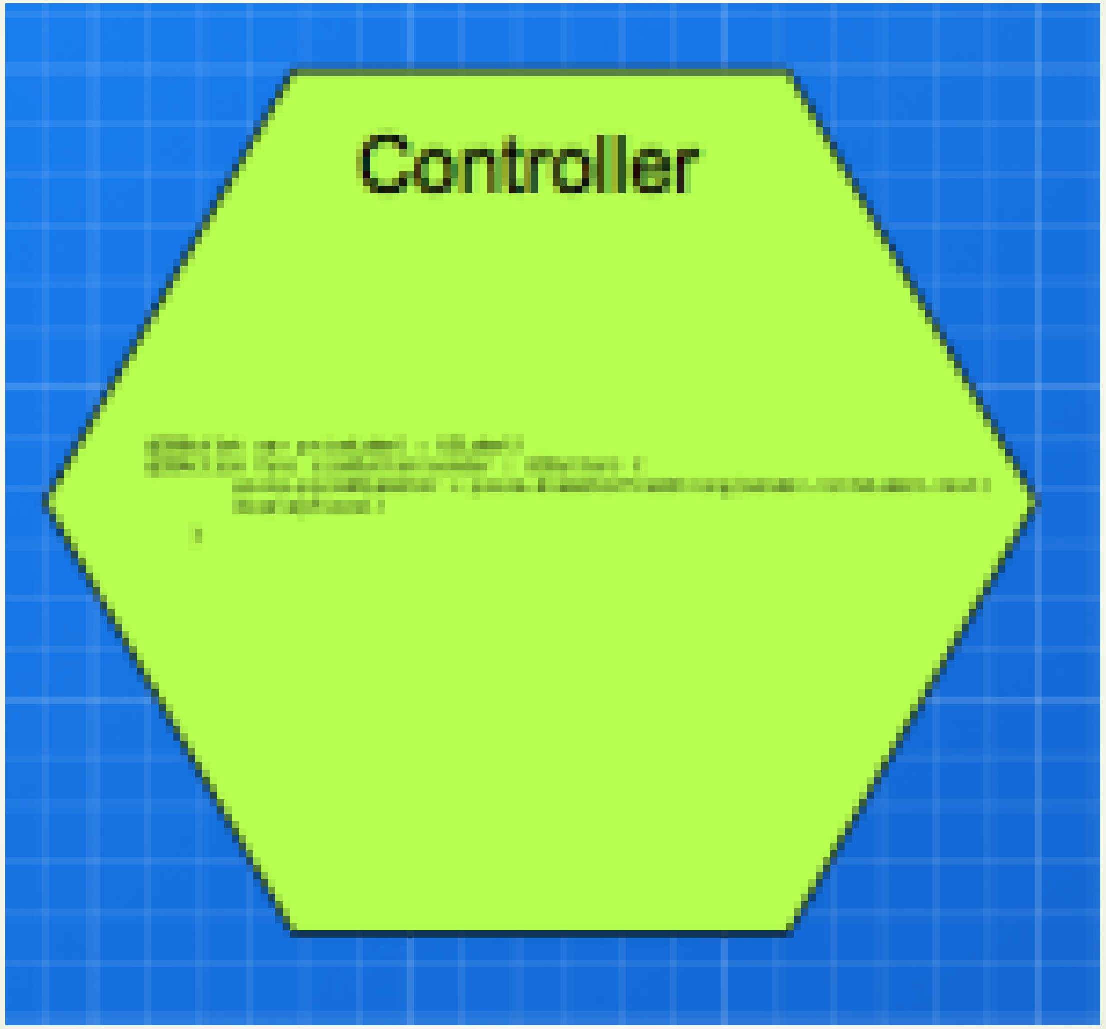
You can avoid using Storyboard while creating a new project by leaving the "Use Storyboard" option unchecked.

Storyboards have a number of advantages over regular nibs:

- With a storyboard you have a better conceptual overview of all the screens in your app and the connections between them. It’s easier to keep track of everything because the entire design is in a single file, rather than spread out over many separate nibs.
- The storyboard describes the transitions between the various screens. These transitions are called “segues” and you create them by simply ctrl-dragging from one view controller to the next. Thanks to segues you need less code to take care of your UI.
- Storyboards make working with table views a lot easier with the new prototype cells and static cells features. You can design your table views almost completely in the storyboard editor, something else that cuts down on the amount of code you have to write.


Tab Bar Controller
Tab Bar Controller
Players
Gestures
Navigation Controller
Navigation Controller – Players
Players
Prototype Cells
Name
Game
UIImageVi
Table View
Prototype Content
Players View Controller –
Navigation Controller
Navigation Controller
Cancel
Add Player
Done
Player Name
Game
Detail
Table View
Static Content
Player Details View Controller –
Choose Game
Prototype Cells
Title
Table View
Prototype Content
Game Picker View Controller –
Gestures
Swipe right for BEST players
Double-tap for WORST players
Gestures View Controller –
Navigation Controller
Navigation Controller
Done
Ranking
Table View
Prototype Content
Ranking View Controller –
Rate Player
Rate Player View Controller –

What is a Controller



```
[illegible]
```

The heart of MVC connects these two.

- Called the ***controller*** or ***view controller***, it coordinates what happens in the model and what happens in the view.
- If a user presses a button on the view, the controller responds to that event.
- If that response means sending messages to the model, the view controller does that.
- If the response requires getting information from the model, the controller does that too. in Xcode, @IBOutlet and @IBAction connect Interface Builder files containing views to the view controller.

- The key to MVC is communication. TO be more accurate, the lack of communication. MVC takes encapsulation very seriously.
- The view and the model never directly talk to each other.
- The controller can send messages to the view and the controller.
- The **view** and **controller** may do an internal action to the message sent as a method call or it may return a value to the controller. The controller never directly changes anything in either the view or the model.

As an aside we need to talk about how Xcode with **@synthesize** and Swift coddles us. For true encapsulation, our properties should all be private and we should write two special methods: **a getter** to retrieve a value and **a setter** to change a value for every property. For example the property pizzaSize might be written this way:

```
private var pizzaSize:Double = 0.0
func setPizzaSize(size:Double){
    pizzaSize = size
}
func pizzaSize()->Double{
    return pizzaSize
}
```

@synthesize

- We would have to do that for every property, which gets a bit tedious. Apple introduced @synthesize in Xcode 4.4 for Objective-C to automate this process. Instead of writing your setters and getters, all that was necessary was to add a statement in your implementation file
- @synthesize pizzaSize

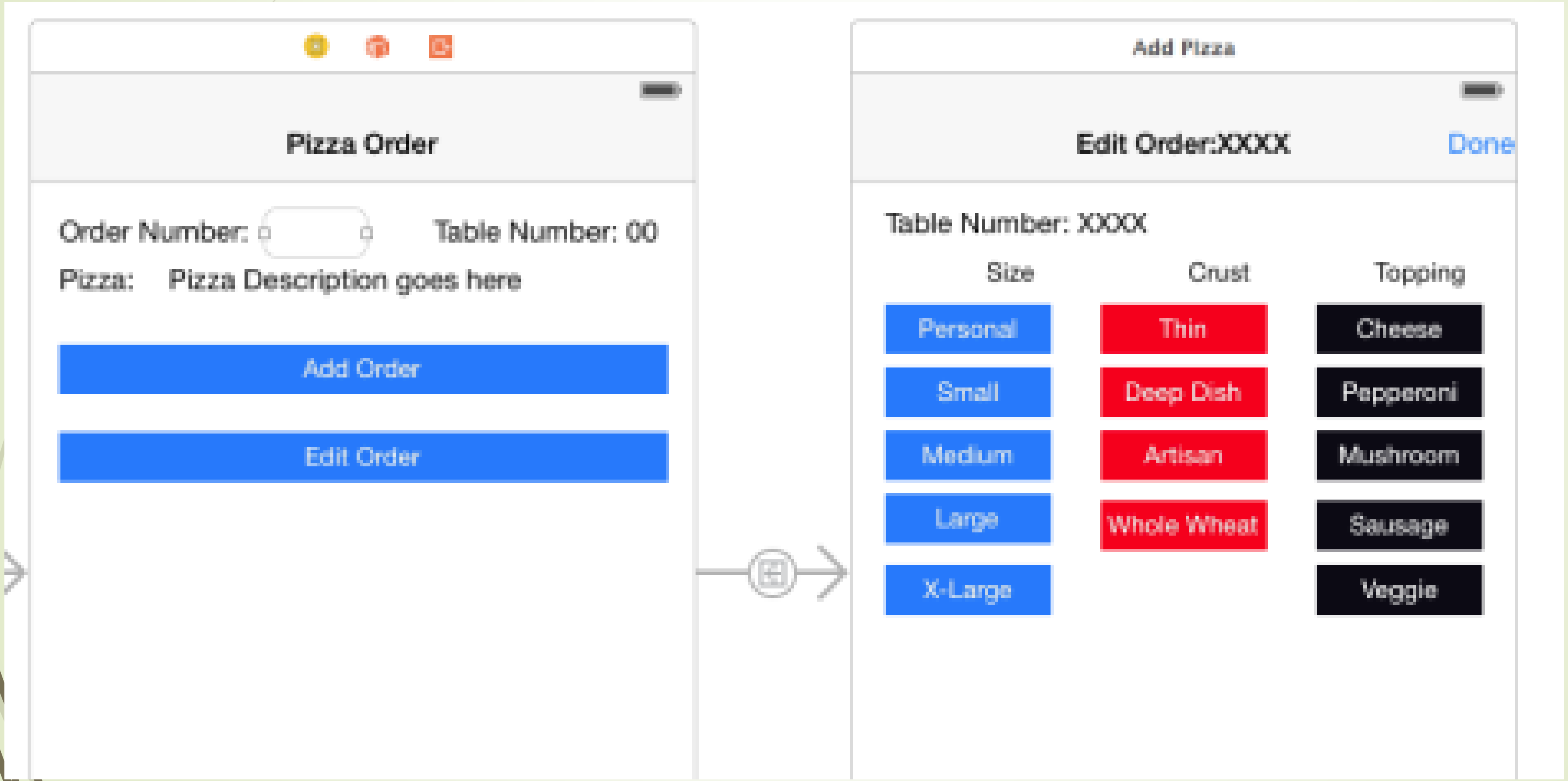
```
private var pizzaSize:Double = 0.0

func setPizzaSize(size:Double){
    pizzaSize = size
}
func pizzaSize()->Double{
    return pizzaSize
}
```

- Xcode then generates the setter and getter behind the scenes for you. Later versions of Objective-C and Swift hide this completely and create the illusion you are directly modifying a property.
- You don't need the setter or getter (though you can write your own) and in Swift **@synthesize** is not even in the language since it hides the setter and getter completely unless you specify it.

- Why I mention this is the illusion of directly modifying properties in the view and model by the controller. It isn't directly modifying anything. Behind the curtain are getter and setter methods.
- The model can not assign values in the controller. Then how do we assign values from the model to the controller? The model can return a value in a method, often a getter, called by the controller.

- So to summarize, a view, a model and a controller cannot directly change a property in each other. A view and a model may not talk to each other at all.
- A view can tell the controller there is a change in the model. A controller can send messages in the form of method calls to the view and the model and get the responses back through that method.

How Does MVC Connect to Other MVC's



- I have an Edit Pizza scene in my pizza order entry app. I need to send information to it to run properly and I would need to get information back. In the world of MVC, the controller does this.
- **But we've a small problem:** Data flows one-way from the calling controller to the new controller.

The Easy Direction

- In Xcode we have what are known as segues. Segues are a convenience to point from one view controller to another. There are alternates to segues including initializing the new view controller from the old one directly. This might be useful for popovers and modal views.
- When working with more complex navigation controllers or tab bar controllers segues keep track of some things that would become cumbersome to control ourselves in a simple way. When we move from one controller to the next, the segue tells the system to open this particular view controller, which then opens up a view and model.
- The model and view in the new MVC setup is different then the calling one. Apple includes a method **prepareForSegue()** which give us a chance to set values in the new view controller, and subsequently the new view controller's view and model.

- I might for example edit a pizza order. The user would give the order number, and the controller would ask the model for the pizza associated with that order.
- The model would return that order as an instance of Pizza and send that to a new view controller whose model is a single instance of pizza. Our **prepareForSegue()** for a navigation controller might look like this:

```
override func prepareForSegue(segue: UIStoryboardSegue!, sender:
AnyObject!){
    if segue.identifier == "editOrder" {
      var vc = segue.destinationViewController as EditPizzaViewController
       let orderNumber = orderNumberText.text.toInt()!
       vc.pizza = pizzaOrder.getPizzaOrder(orderNumber)
       vc.orderNumber = orderNumber     }
 }
```

- We first find if this is the correct segue in line 2. If so, we make the new view controller **EditPizzaViewController**. We get an order number from the text field in line 4.
- We set two properties, the order number, and the pizza which becomes the model for the new view controller. Everything moves nicely and when the Edit Pizza screen opens, everything looks good.

The Problem Direction

- We can edit the information for the pizza easily enough in the new MVC. But when we press Done to send it back to the **OrderPizzaViewController** is when problems show up. By the rules of MVC, we need a method to return a value. but where in a called instance can we go back to the class calling it? With an encapsulated class we can't.
- There is no way to send that revised model back to the original controller without breaking encapsulation or MVC. The new view controller does not know anything about the class that called it. We look stuck. If we try to make a reference directly to the calling controller, we may cause a reference loop that will kill our memory. Simply put, we can't send things backwards.

- This is the problem that delegates and protocols solve by being a little sneaky. Imagine another class, one that is really a skeleton of a class. This class contains only methods. It declares that certain methods are in this class, but never implements them.
- Generally known as abstract classes, in Swift and Objective-C they are protocols. We make a protocol class that has one method.
- That method is what you do when you are done in the **EditPizzaViewController**, and want to go back to the **OrderPizzaViewController**. It has a few parameters, things you want to pass back to the calling view controller. So it might look like this:

```
protocol PizzaEditDelegate{
    editPizzaDidFinish(pizza:Pizza)
}
```

in our new view controller EditPizzaViewController, we make an instance of this protocol.

```
delegate:PizzaEditDelegate? = nil
```

- Since we have a property of type **PizzaEditDelegate**, we can use the methods of the **PizzaEditDelegate** type, In our example, that is our method **editPizzaDidFinish**. We can stick that method call in a action for a done button:
  @IBAction doneButtonPressed(sender:UIButton!){

```
 delegate.editPizzaDidFinish(pizza:pizza)
}
```

- Since protocols are skeletons, it means any other class can adopt them. A class make the protocol methods part of its own class with a stipulation.
- As soon as a protocol gets adopted, you need to flesh out the skeleton.
- The developer has to code the required methods in the adopted class' code. We adopt a protocol by placing it after the name of the class and superclass,
- **Class OrderPizzaViewController:UIViewController,PizzaEditDelegate**

- As soon as you do that, you will get a compiler error since the protocol's method does not exist in the class. In the code for the adopting class, in our example OrderPizzaViewController, we would implement the method:
- func editPizzaDidFinish(pizza:pizza){

```
self.pizza.changePizzaOrder(orderNumber:ordernumber, pizza:pizza)
controller.navigationController.popViewControllerAnimated(true)
}
```

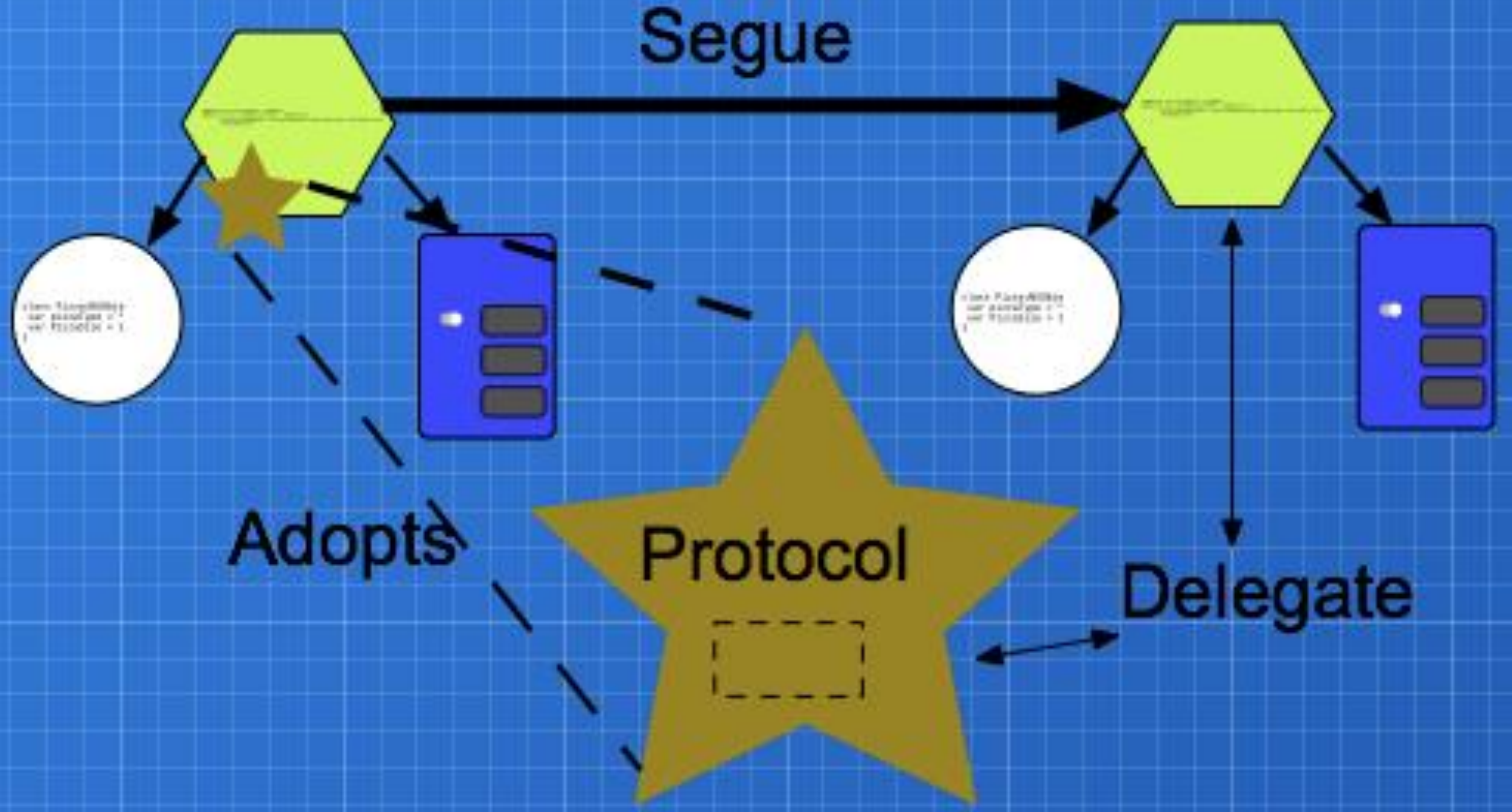
- We get the data back, place it where it belongs, and dismiss the newer controller taking the model and view with it.
- Here the data pizza from the new view controller is now a parameter, something we can work with in our model self.pizza. In line 2, I change our model via the original view controller calling a method of the model changePizzaOrder().

- One more step. While back in the destination controller EditPizzaViewController I said the delegate was an instance of the protocol, I didn't say where the delegate was. In prepareForSegue I add one more line vc.delegate = self saying the protocol is your controller

-
  ```
  override func prepareForSegue(segue: UIStoryboardSegue!, sender: AnyObject!){
   if segue.identifier == "editOrder" {
  var vc = segue.destinationViewController as EditPizzaViewController
    let orderNumber = orderNumberText.text.toInt()!
     vc.pizza = pizzaOrder.getPizzaOrder(orderNumber)
    vc.orderNumber = orderNumber
     vc.delegate = self     }
  ```

- When we tap Done, the method runs, it knows it is located in the calling controller, and calls it there where we added to the original class.
- The data is a parameter so the program can easily transfer into the controller and to the model.
- **Delegates** and **protocols** are bit sneaky but it works, and is one of the most important techniques when working with view controllers.

A diagram of how a delegate attaches to a adopted protocol.



- The original question asked why didn't Swift make this simpler. As I hope I've shown here with a Swift context, it doesn't matter what object oriented language you use. **Delegates** are part of the MVC pattern, which is a good programming practice.

- http://makeapppie.com/2014/06/20/swift-swift-adding-an-mvc-model-class-in-swift/
- http://www.raywenderlich.com/82046/introduction-to-os-x-tutorial-core-controls-and-swift-part-1