Lecture 4

Lifecycle & MVC

Attendance

tiny.cc/cis195-lec4

Today

- Followup and Piazza questions
- Swift Features: Observers and Protocols
- Life Cycle
- MVC
- Live Demo: passcode app

Logistics

Spring 2020 19x Lecture Topics

- ~~21 Jan. — Linux/Unix commands~~
- ~~28 Jan. — Version control with Git + GitHub~~
- ~~4 Feb. — HTML/CSS/Internet Basics~~

Questions

Observers

Two kinds of variables: Stored and Computed

```
struct S {
    // Stored Property
    var stored: String = "stored"

    // Computed Property
    var computed: String {
        return "computed"
    }
}
```

Stored vs Computed variables

```
struct S {
    var stored: String {
        willSet {
            // . . .
        }

        didSet {
            // . . .
        }
    }
}
```

Stored vs Computed variables

```
struct S {
    var stored: String {
        willSet {
            print("willSet was called")
            print("stored is still equal to \(self.stored)")
            print("stored will be set to \(newValue)")
        }

        didSet {
            print("didSet was called")
            print("stored is now equal to \(self.stored)")
            print("stored was previously set to \(oldValue)")
        }
    }
}
```

Credit

https://nshipster.com/swift-property-observers/

(has a lot of extra info - most of which you don't need)

Protocols

Protocols

- “A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. “

Protocols

- “A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. “
- The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements.

Protocols

- “A protocol defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality.
- The protocol can then be adopted by a class, structure, or enumeration to provide an actual implementation of those requirements.
- Any type that satisfies the requirements of a protocol is said to conform to that protocol.”

Example: FullyNamed

```
protocol FullyNamed {
    var fullName: String { get }
}
```

Example: FullyNamed

```
protocol FullyNamed {
    var fullName: String { get }
}

struct Person: FullyNamed {
    var fullName: String
}
```

Note: protocols can also have required functions

Example: FullyNamed

```
protocol FullyNamed {
    var fullName: String { get }
}

struct Person: FullyNamed {
    var fullName: String
}

let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

Note: protocols can also have required functions

FullyNamed Demo

How do we add a function to our protocol?

```
protocol FullyNamed {
    var fullName: String { get }
    func printName()
}

struct Person: FullyNamed {
    var fullName: String
}

let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

FullyNamed Demo

How do we add a function to our protocol?

```
protocol FullyNamed {
    var fullName: String { get }
    func printName()
}

let john = Person(fullName: "John Appleseed”)
john.printName()

// printName MUST be implemented by the Person struct
```

Life Cycle

Life Cycle

- **What are some of the life cycle methods?**

Life Cycle

- **What are some of the life cycle methods?**
- We can safely ignore:
  - *init(coder:)* — for creating objects and custom classes
  - *loadView* — called when a VC is created programmatically
  - *didReceiveMemoryWarning* — called when the system is out of memory (probably due to our app)

Life Cycle

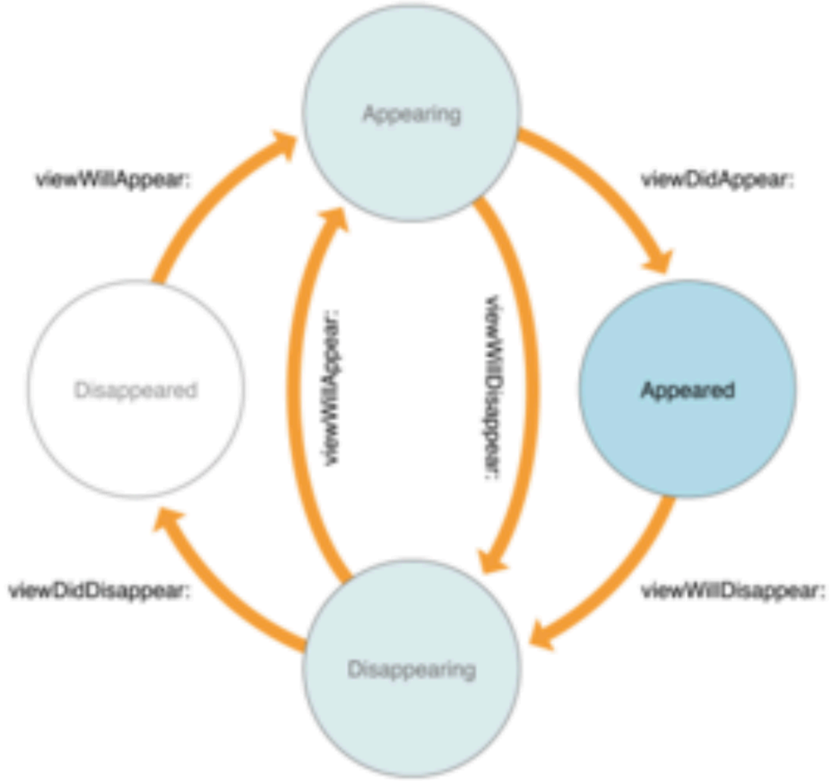
- **Time for a timeline!**

- Say we open an app, see **Screen A**, and then tap a button to move to **Screen B**. In what order are Screen A's life cycle methods called?

Life Cycle

- We have ViewController **A** and ViewController **B**
- Say we load and display **A**, press a button to transition to **B**, and then push *another* button on **B** to transition back to **A**.
- **How many times will A call viewDidLoad? viewDidAppear?**

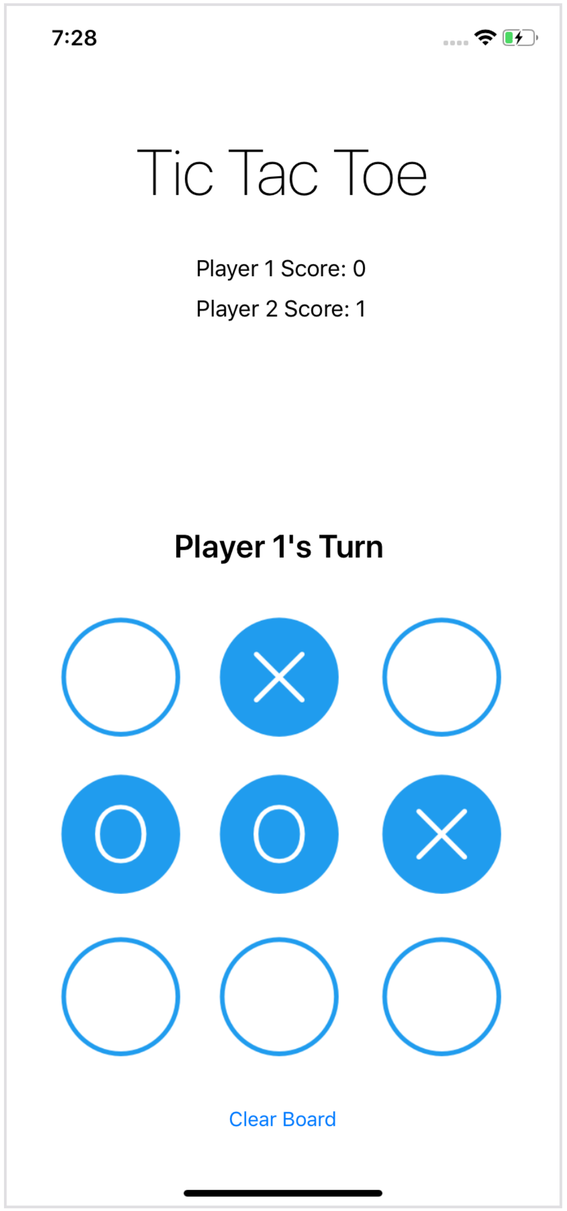
Life Cycle

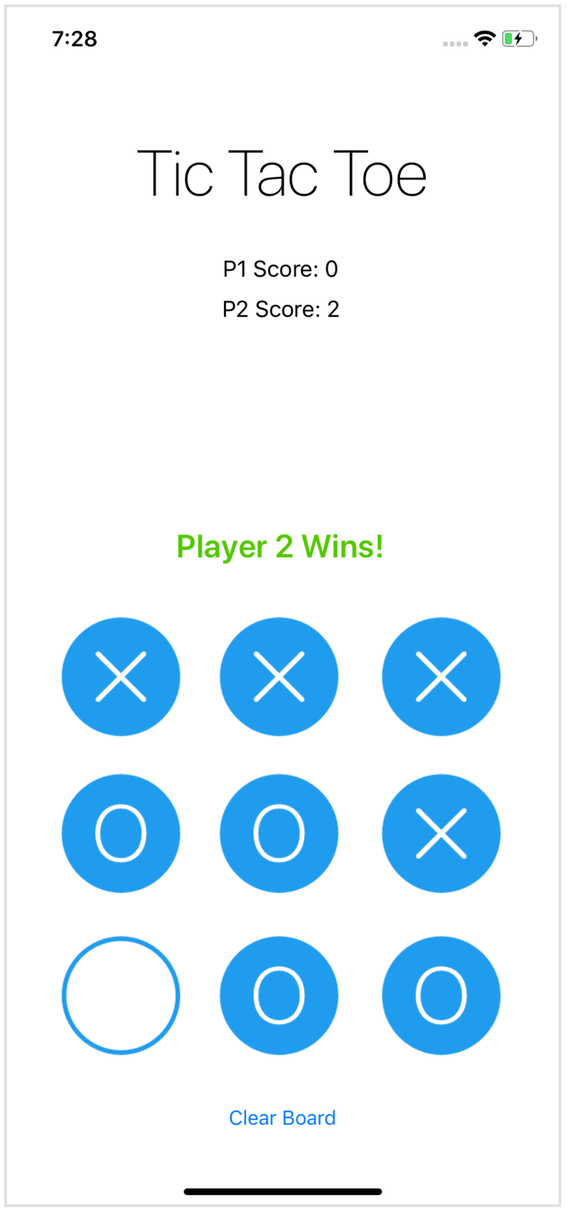


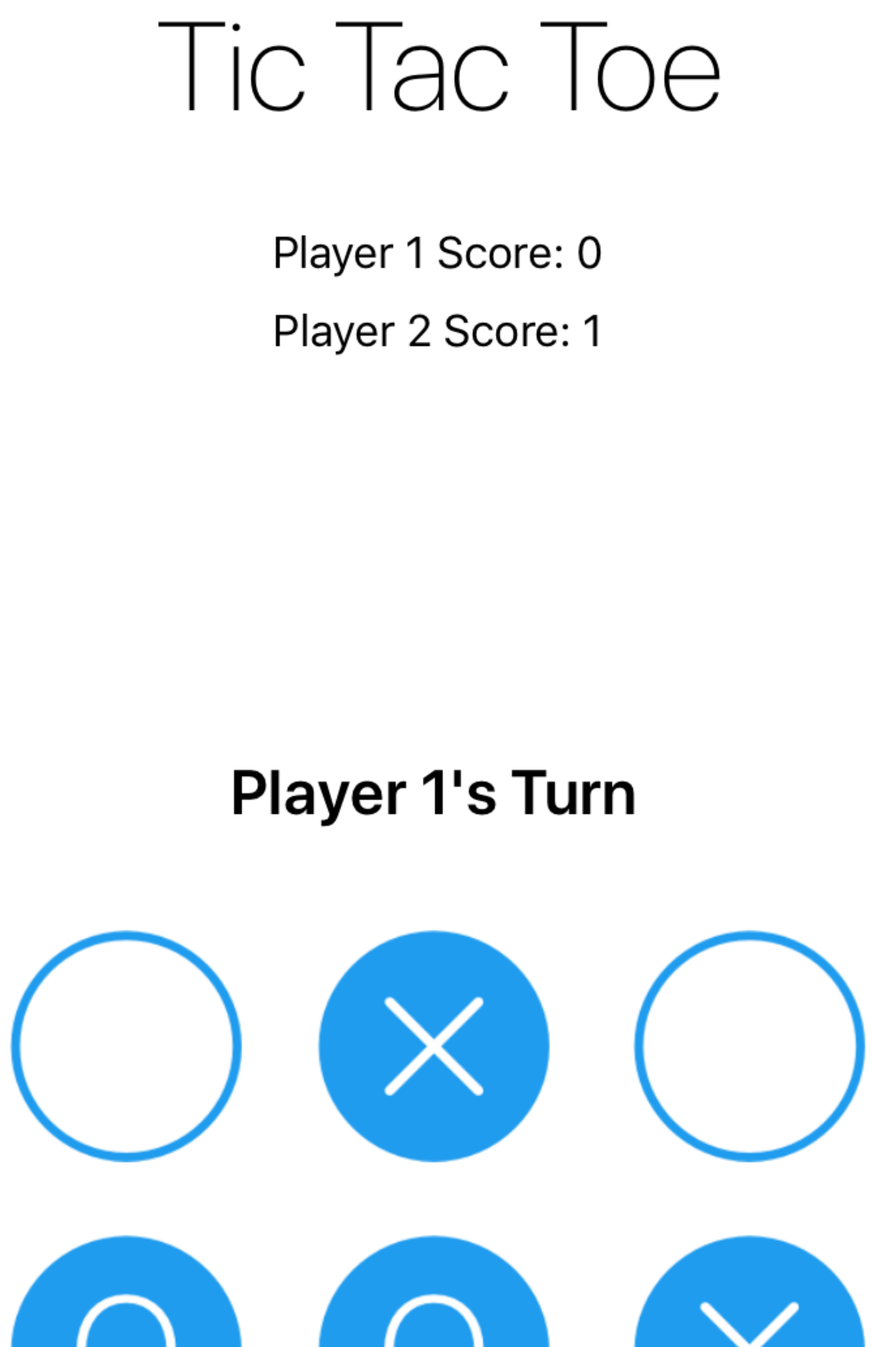
- **viewDidLoad -** initial states, setup outlets, setup objects
- **viewWillAppear -** update your views since they were last on screen
- **viewDidAppear** - animations, observers, on-screen prompts
- **viewWillDisappear -** view-related cleanup tasks
- **viewDidDisappear** - cleanup tasks

MVC

App 1: Tic Tac Toe

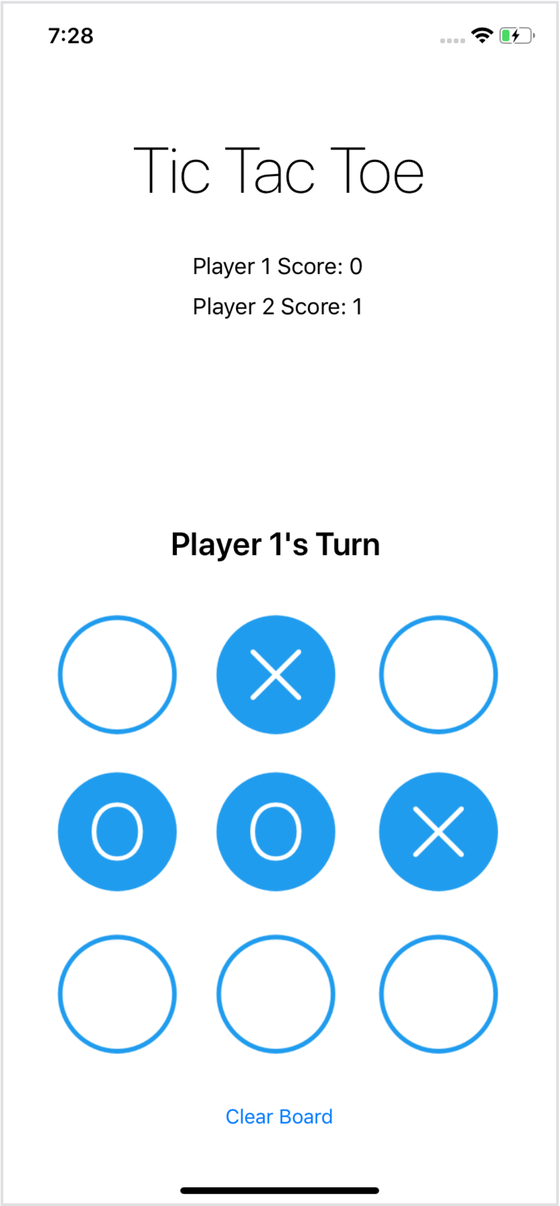






App 1: Tic Tac Toe

- Due next Thursday at 4:20pm:
  - The model and logic components
  - Refinements to the UI
- Released tonight



What is MVC?

- **A design pattern that cleanly separates your code into 3 roles: model, view, and controller**
- **Model** — manages data, logic, and rules. Game logic encoded here.
- **View** — visual layer, should know how to *look* (and that's it!)
- **Controller** — Accepts input and can *control* both the model and view.

What's the point?

- **A clean separation of concerns. Simple, modular, easy-to-read code**
- **Model** — doesn't know anything about what the game *looks* like, or anything about the UI or interaction. Only knows the state and rules.
- **View** — just knows *how to look* and who to tell about button presses. Doesn't know what the game rules are.
- **Controller** — Knows how to interface between the model and view. Knows how to "route" interaction from the view to appropriate model methods. **Knows the current state of both model and view**.

UIViewController

- **A bit of a hybrid** between controller and view.
- Receives actions from buttons etc... very much a controller!
- Also configures and modifies views directly... kind of a view?

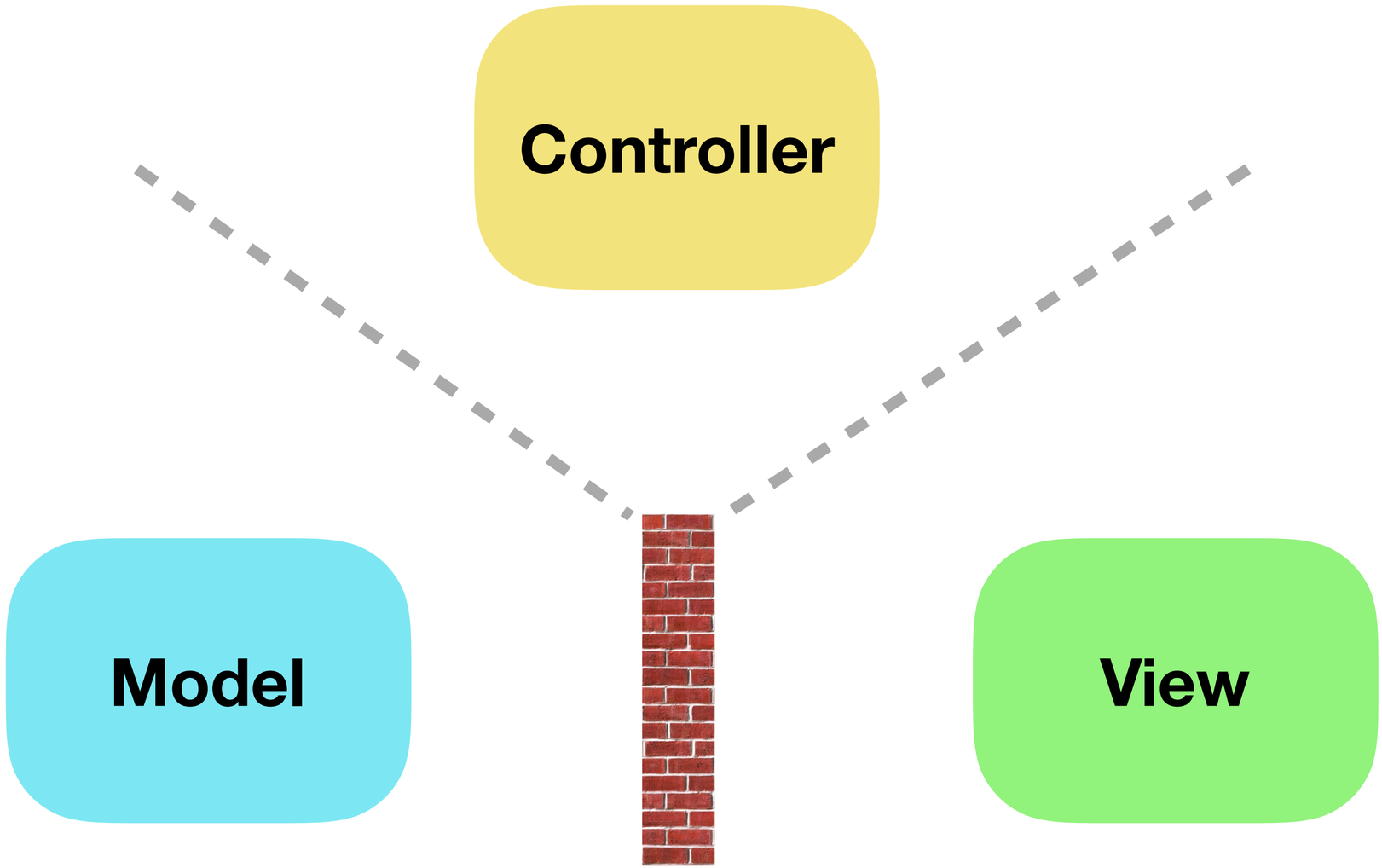
Remember, this is all arbitrary

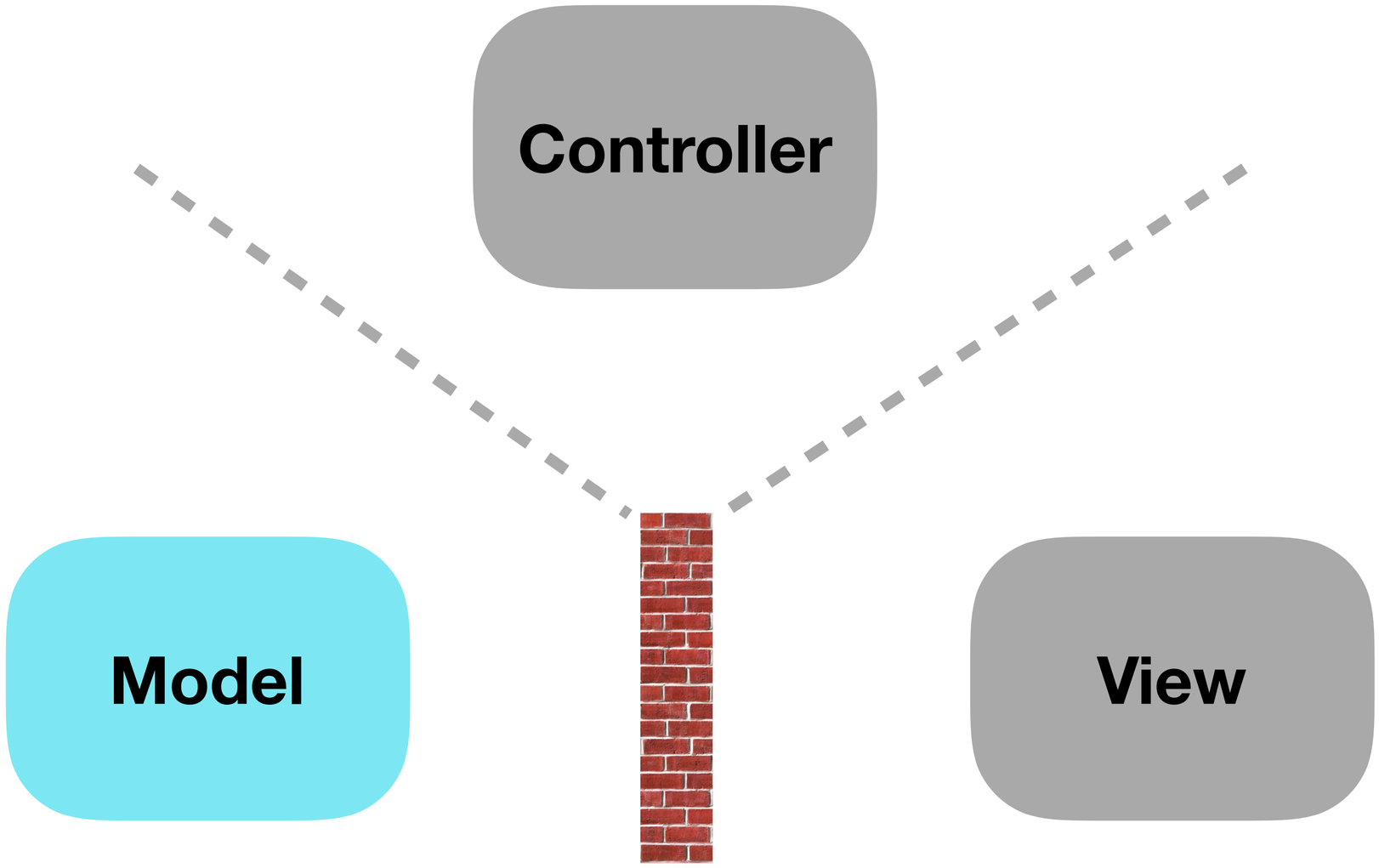
- Nothing (apart from the graders) stopping you from having a model in your ViewController.
- MVC Is just a way to organize your code. Dominant paradigm on iOS (for now)... but there are others (VIPER, MVVC, etc.)
- MVC can lead you into thinking *everything* is one of the 3...
    - This leads to fear. Fear leads to hate, hate leads to anger, anger leads to... “Massive View Controller”

Pitfalls

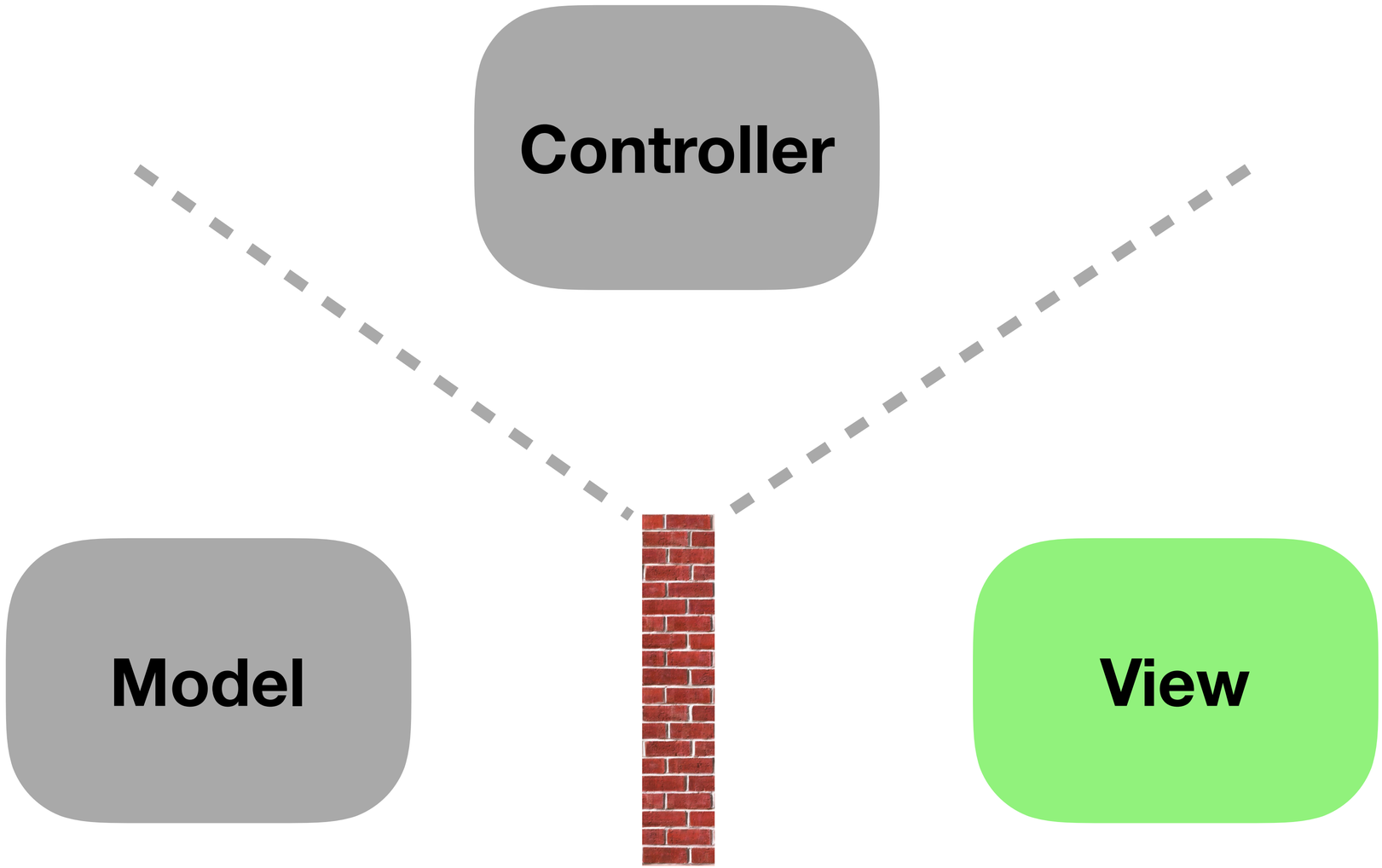
- MVC can lead you into thinking *everything* is one of the 3...
- What about Networking code? Complex views that delegate?
- Unchecked MVC leads to confusion. Confusion leads to fear. Fear leads to unmaintainable code:
  - MVC: “Massive View Controller”

MVC in iOS

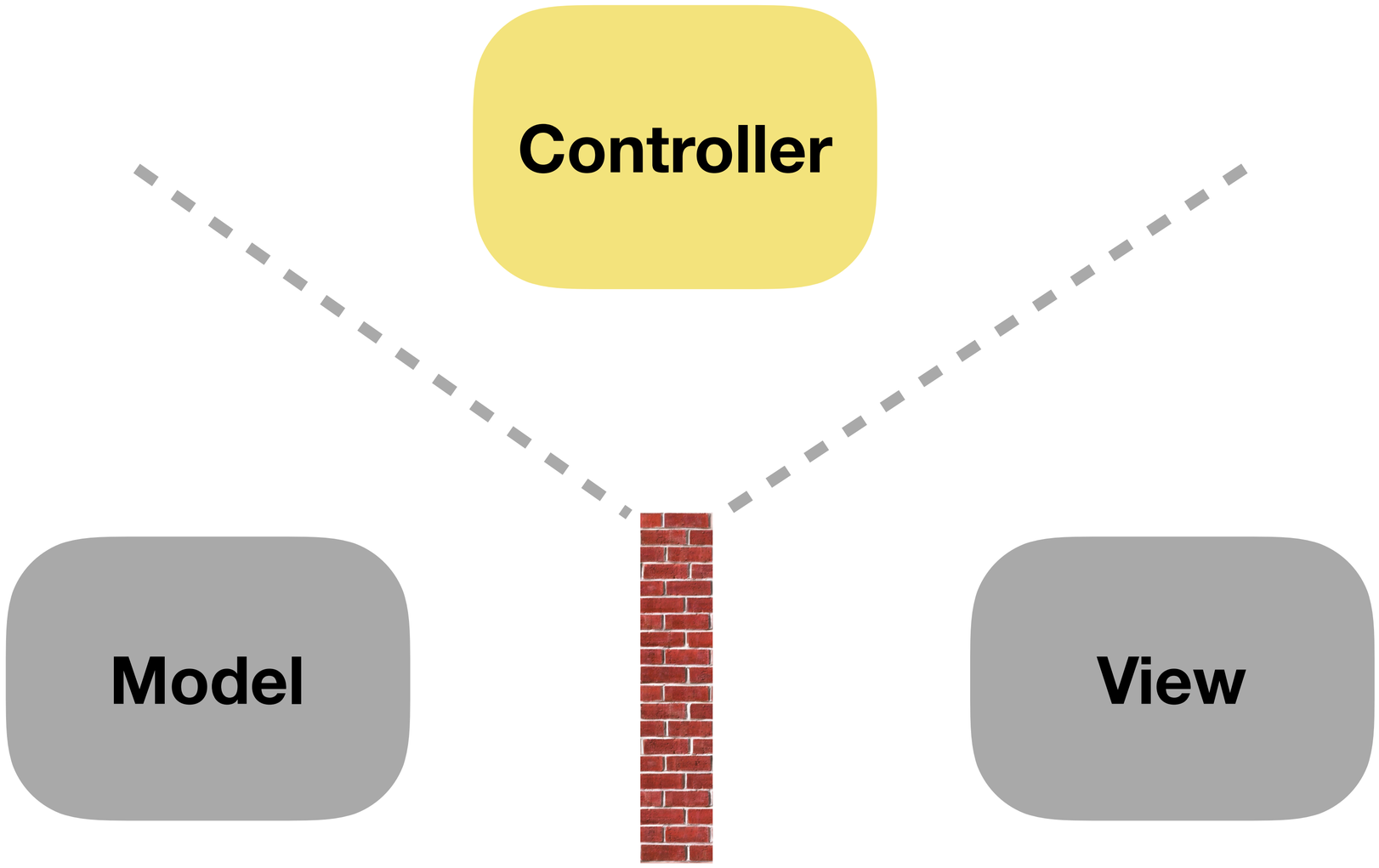

Controller
Model
View



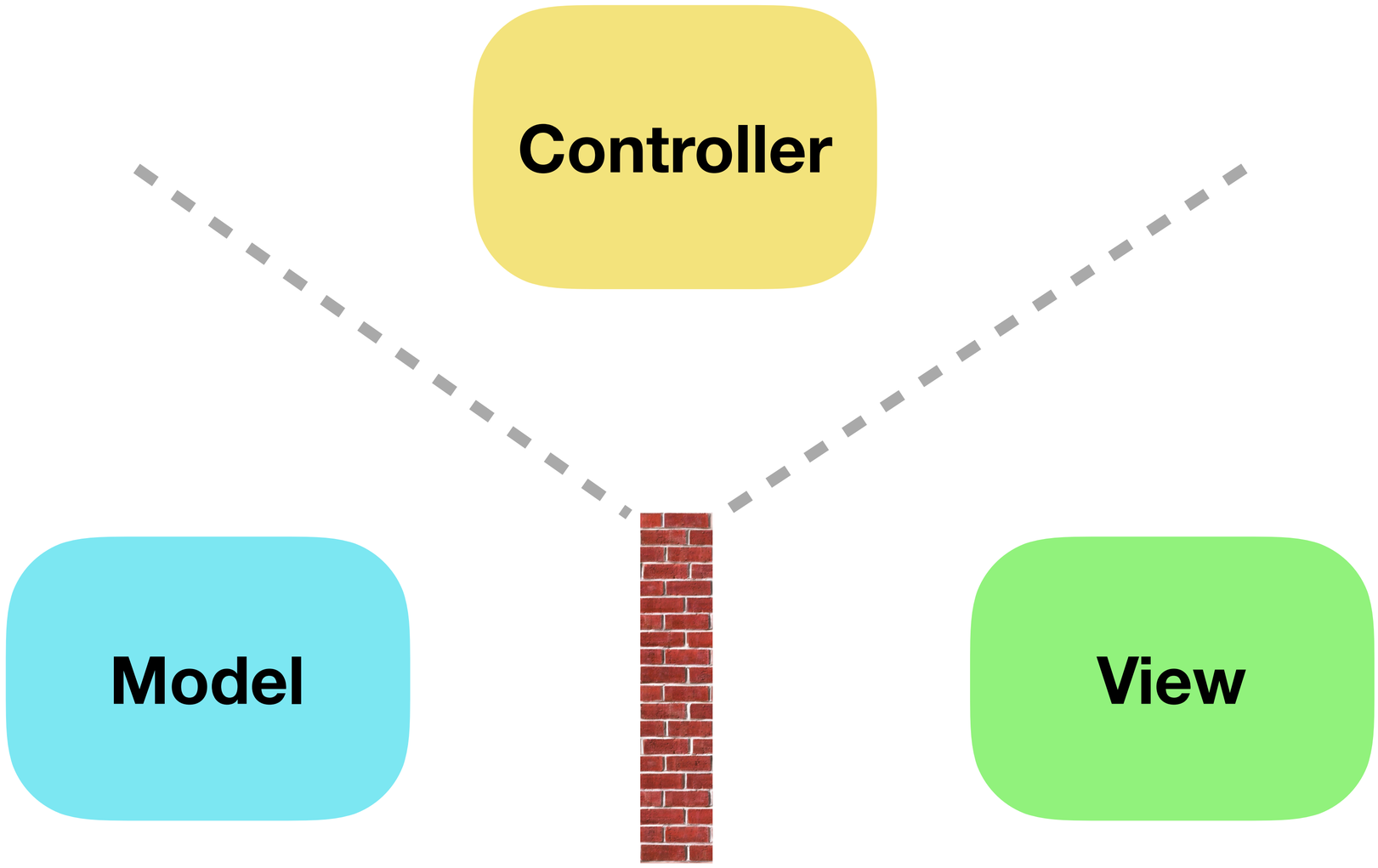
Model holds data and logic (knows how to play Tic Tac Toe).



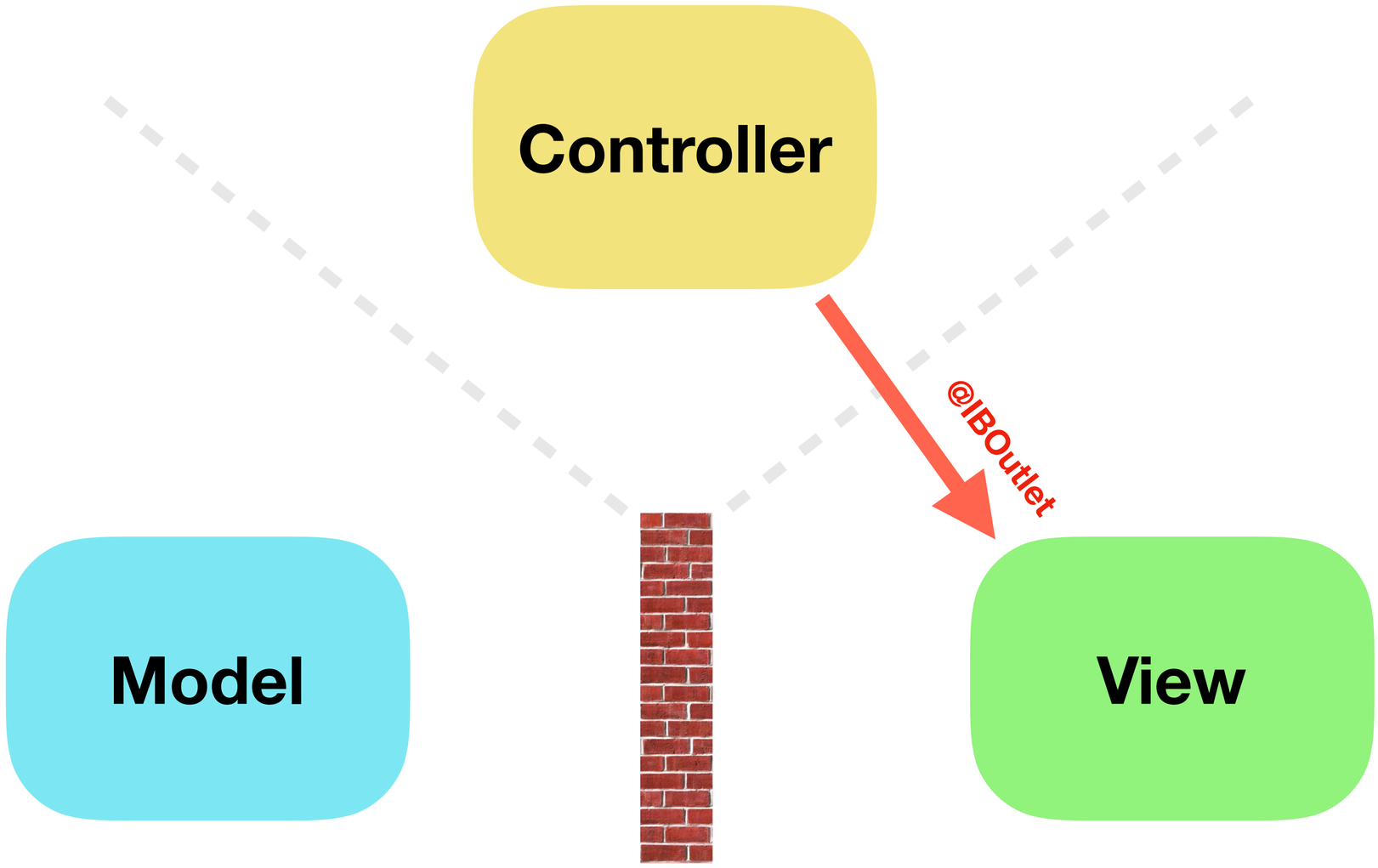
View has the UI elements the user interacts with (the buttons, the labels).



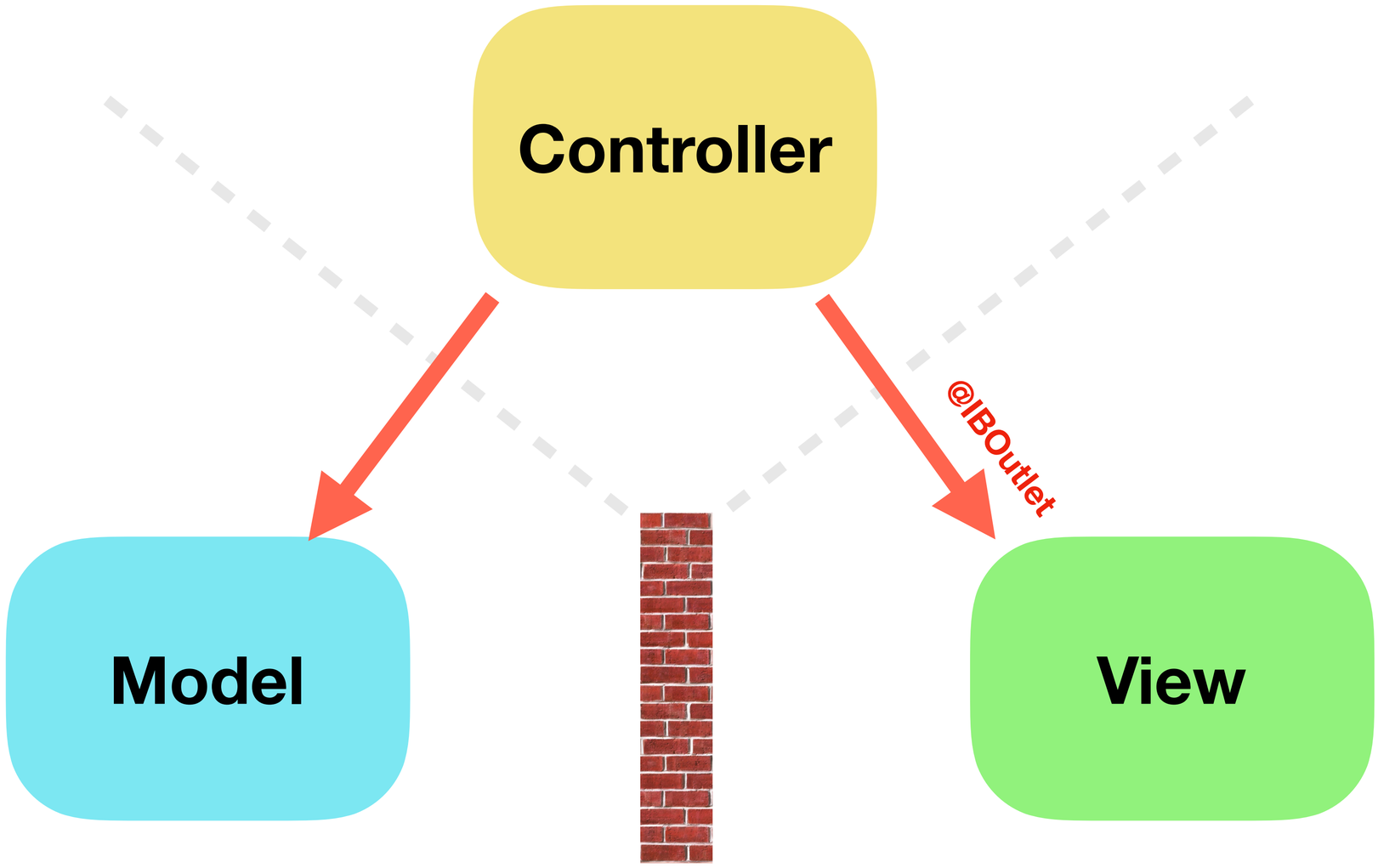
Controller connects these pieces (what happens when a view is clicked).



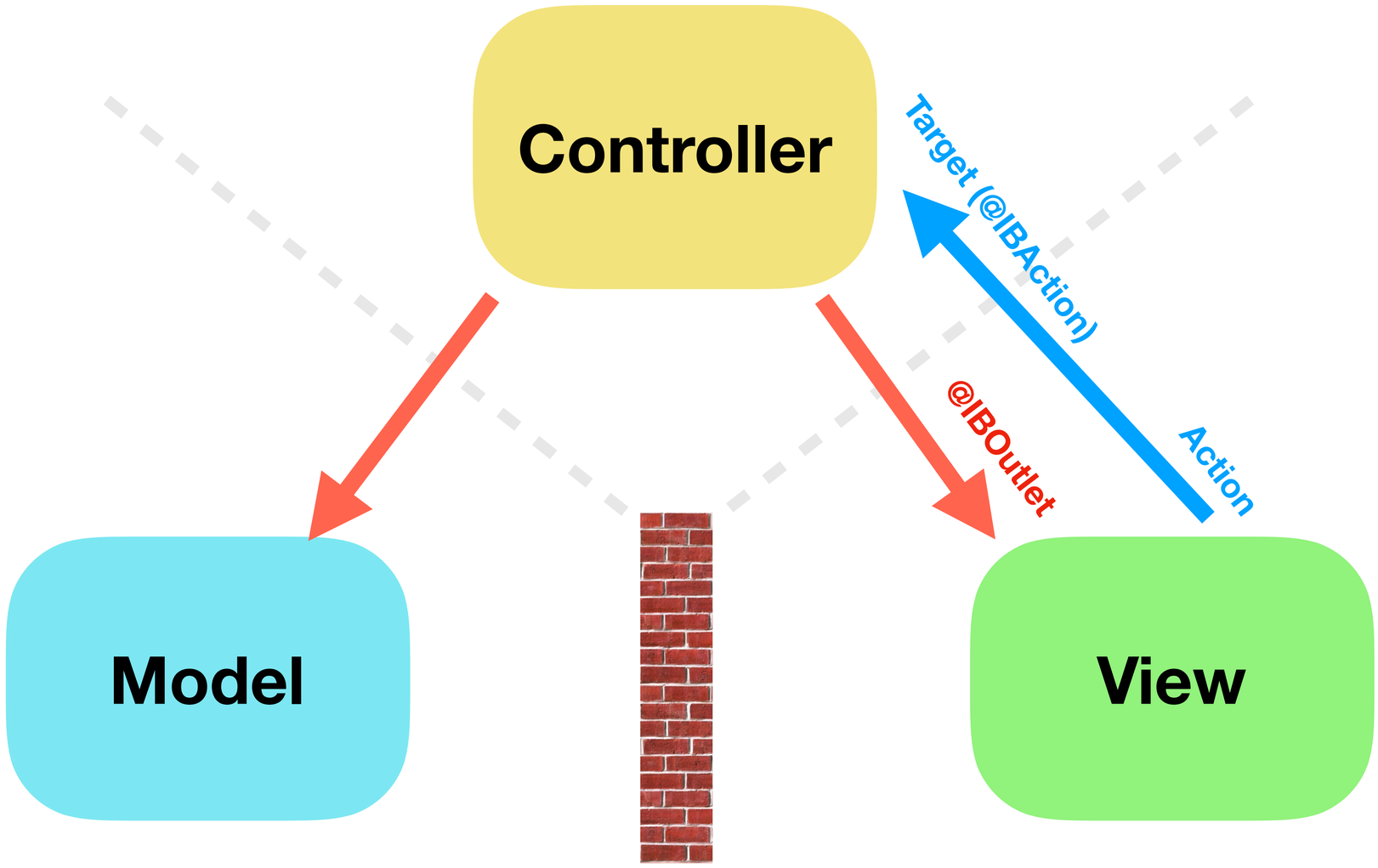
How do they communicate?



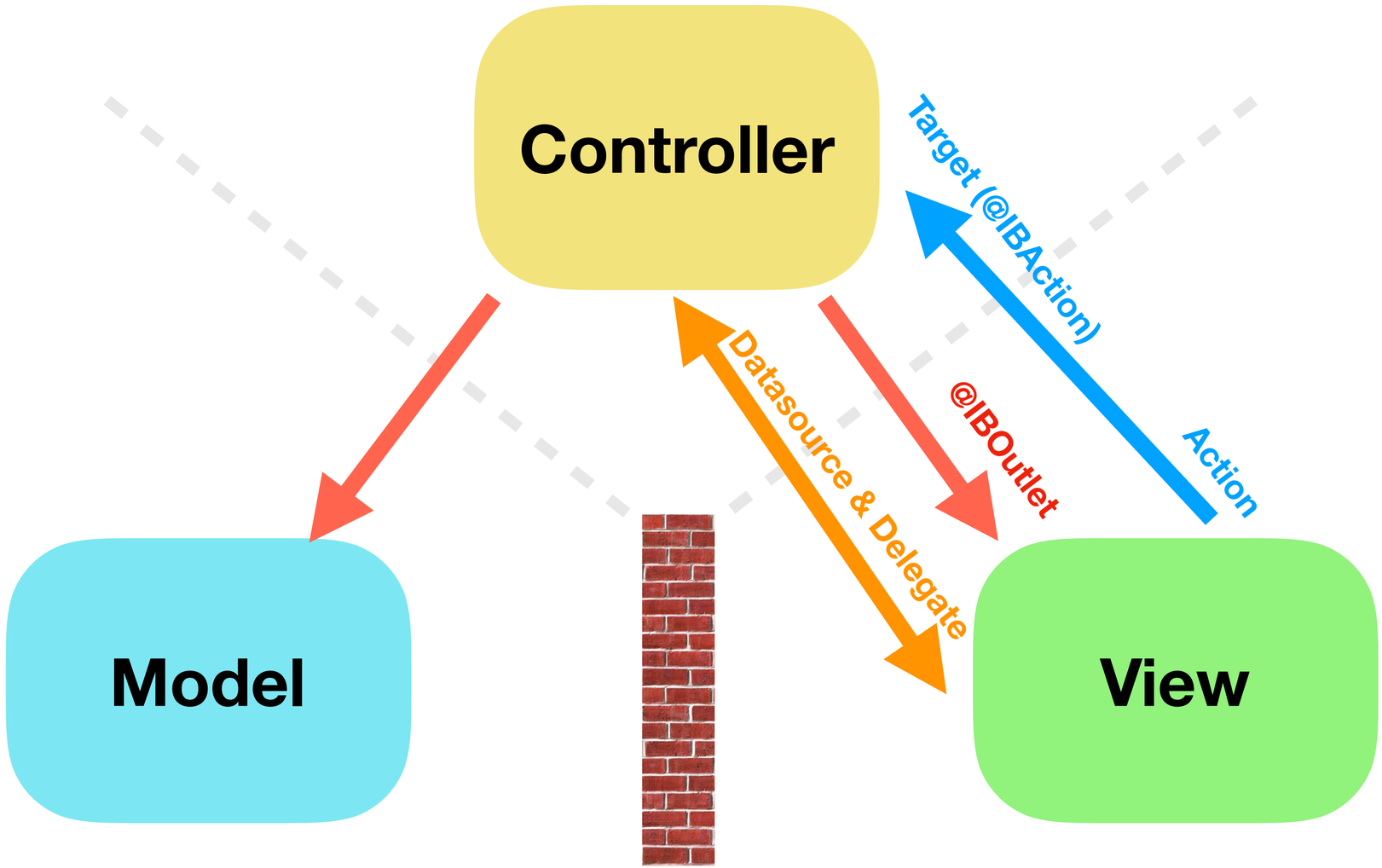
Controller talks directly to View with **IBOutlets.**



Controller talks directly to Model with… code.



View notifies Controller of changes with Target-Actions (**@IBAction**)



Next week: Datasource & Delegate methods.

Live Demo: Passcode Improvements

Due Before Next Class

- App 2: Tic Tac Toe (Part B)
- Tutorial 2: Table Views & Delegation

Links

- Survey: tiny.cc/cis195-lec4
- Piazza: tiny.cc/cis195-piazza