Developing Applications for iOS

Lecture 2:
MVC Design Concept

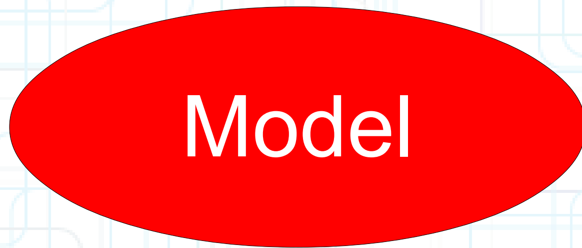
Radu Ionescu
raducu.ionescu@gmail.com
Faculty of Mathematics and Computer Science
University of Bucharest

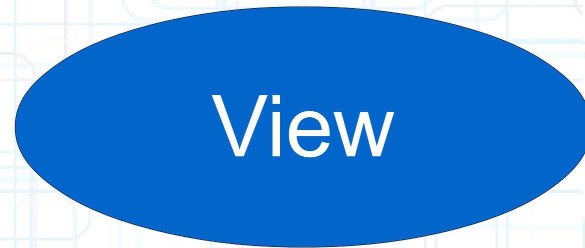
Content

- MVC Design Concept
- Introduction to Swift

MVC Design Model

Controller





MVC Design Model

Model

View

- Divide objects in your program into 3 camps.

MVC Design Model

Model

View

- Model = What your application is (but not how it is displayed)

MVC Design Model

Model

View

- Controller = How your Model is presented to the user (UI logic)

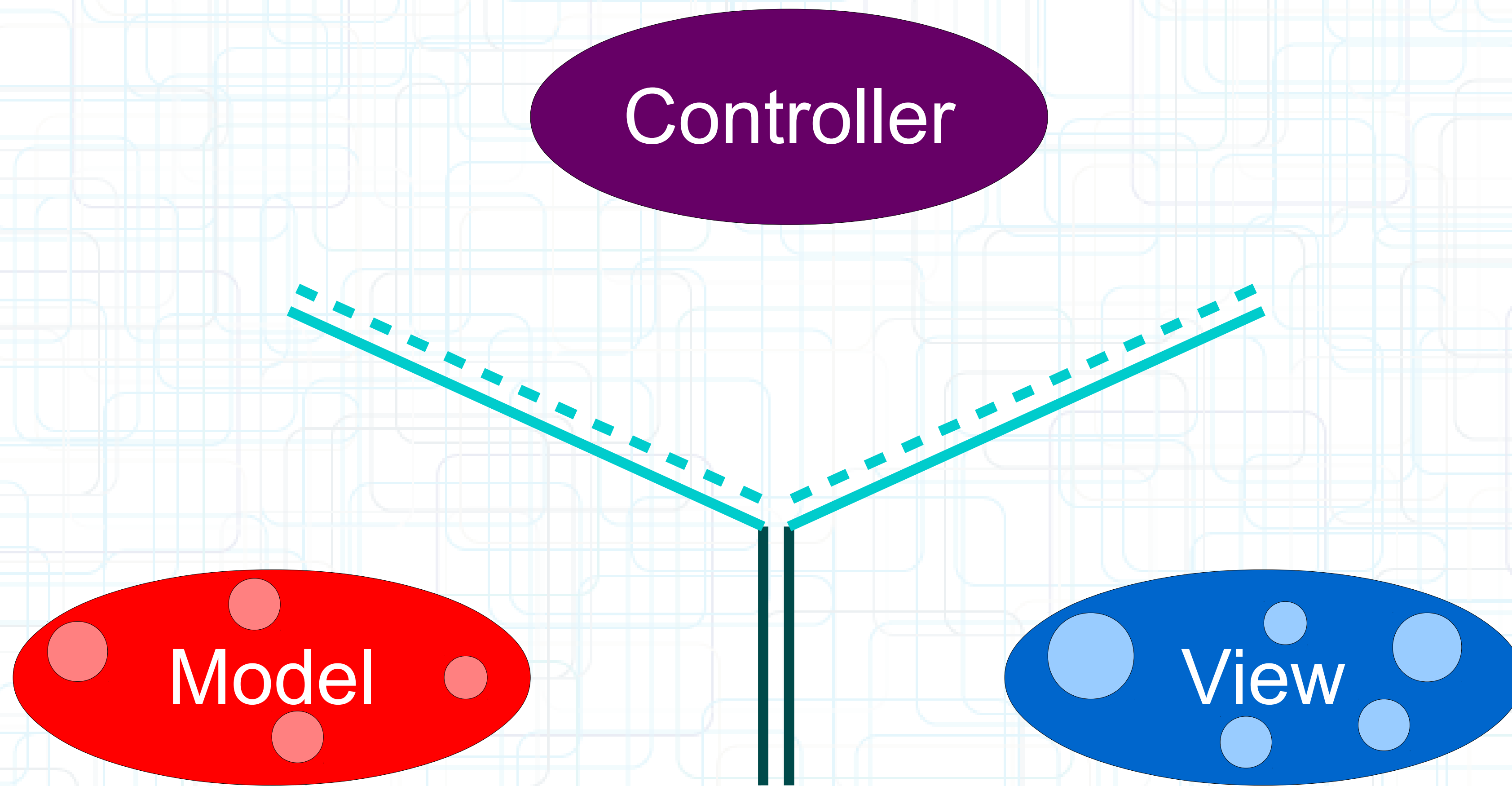
MVC Design Model

Model

View

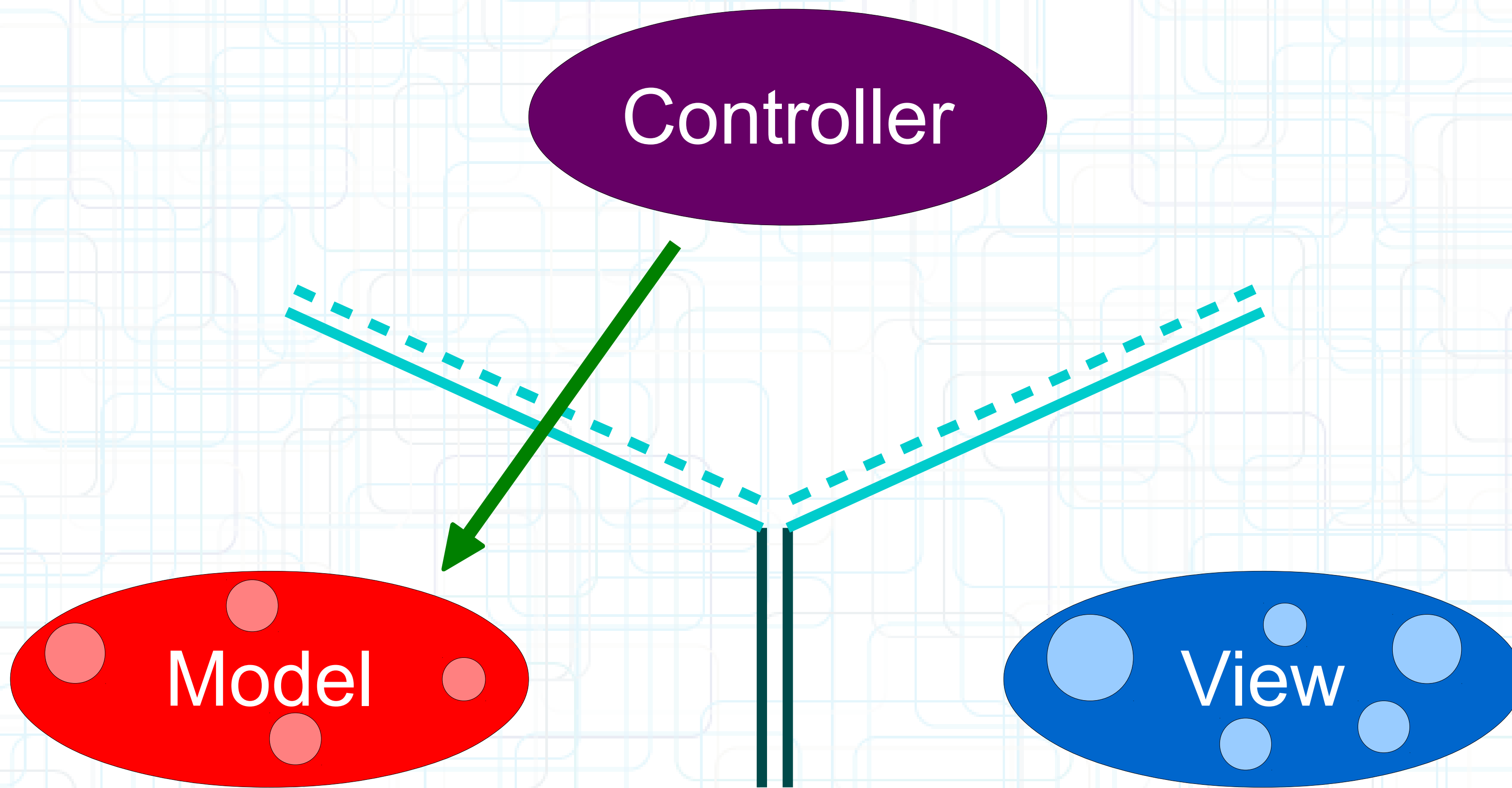
- View = How your application is displayed.

MVC Design Model



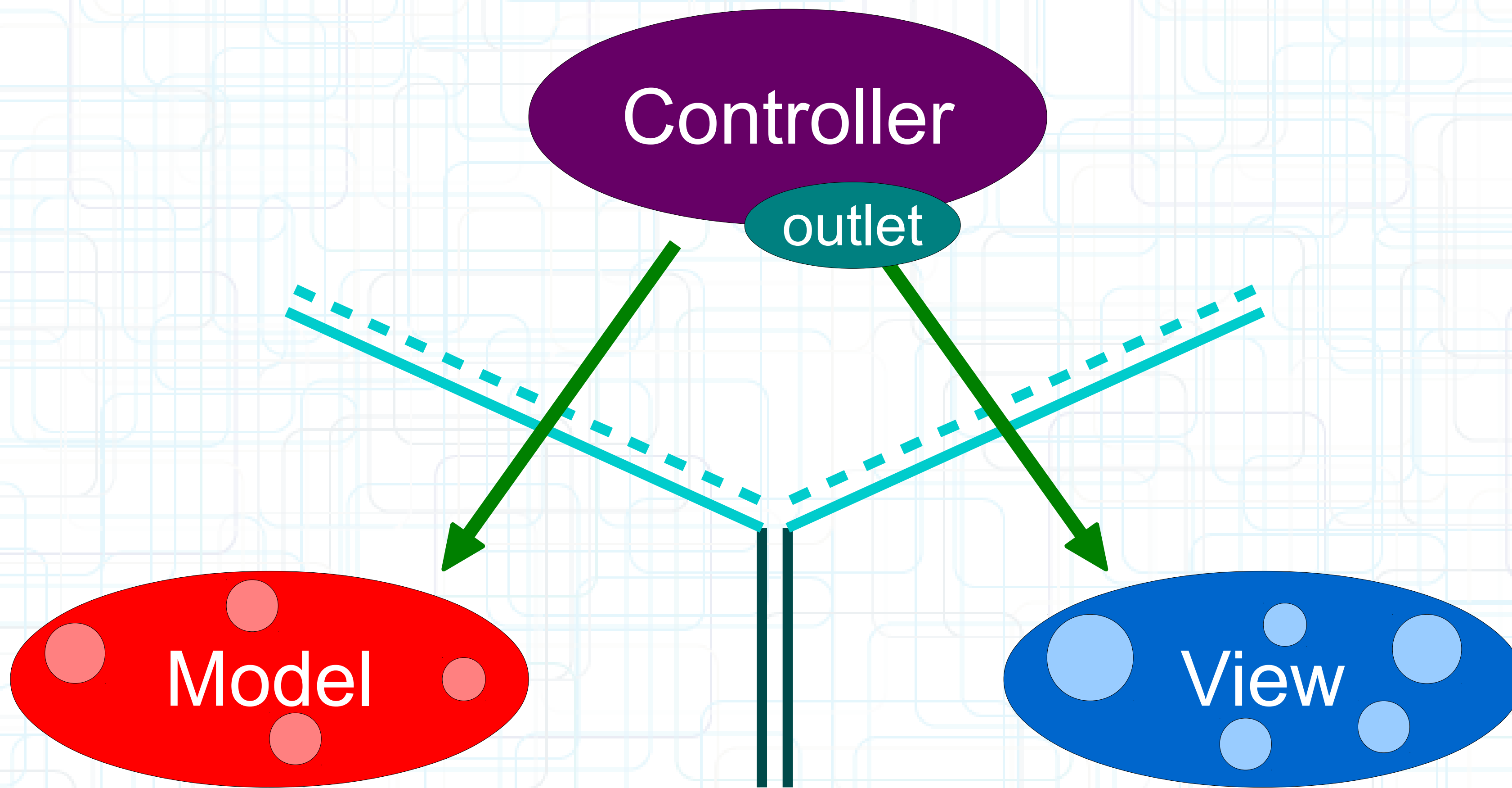
- It's all about managing communication between camps.

MVC Design Model



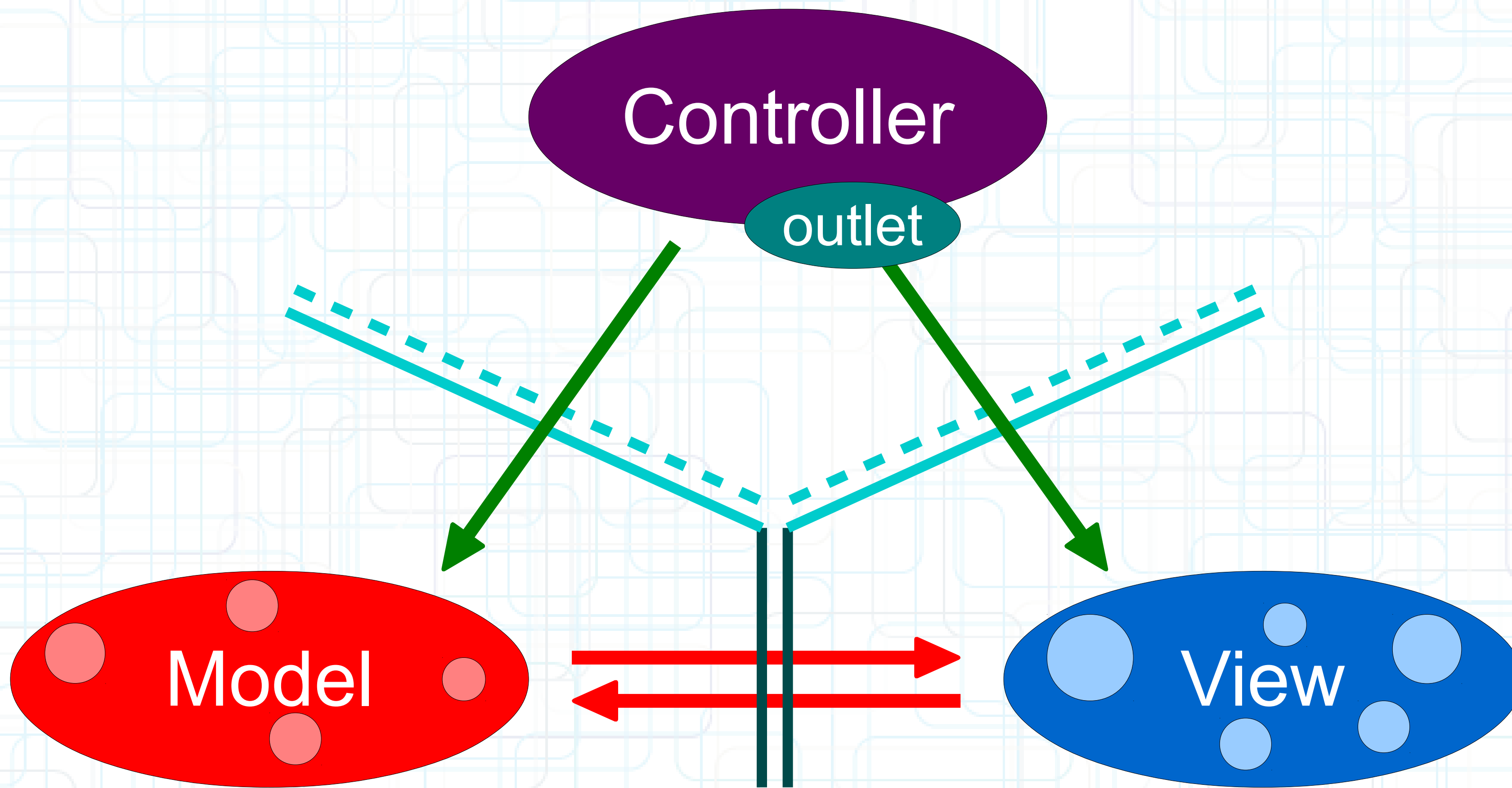
- Controllers can always talk directly to their Model.

MVC Design Model



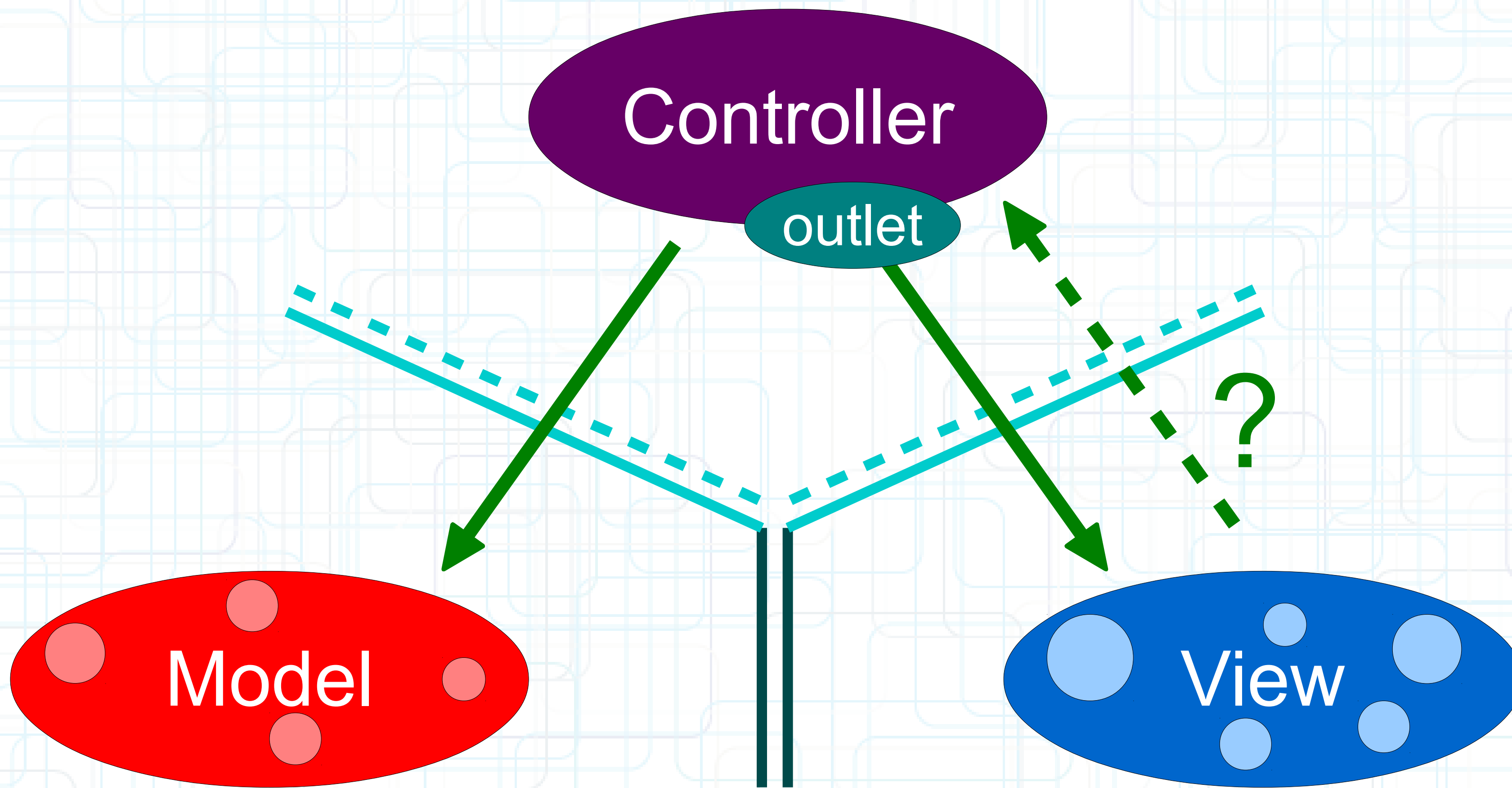
- Controllers can always talk directly to their View.

MVC Design Model



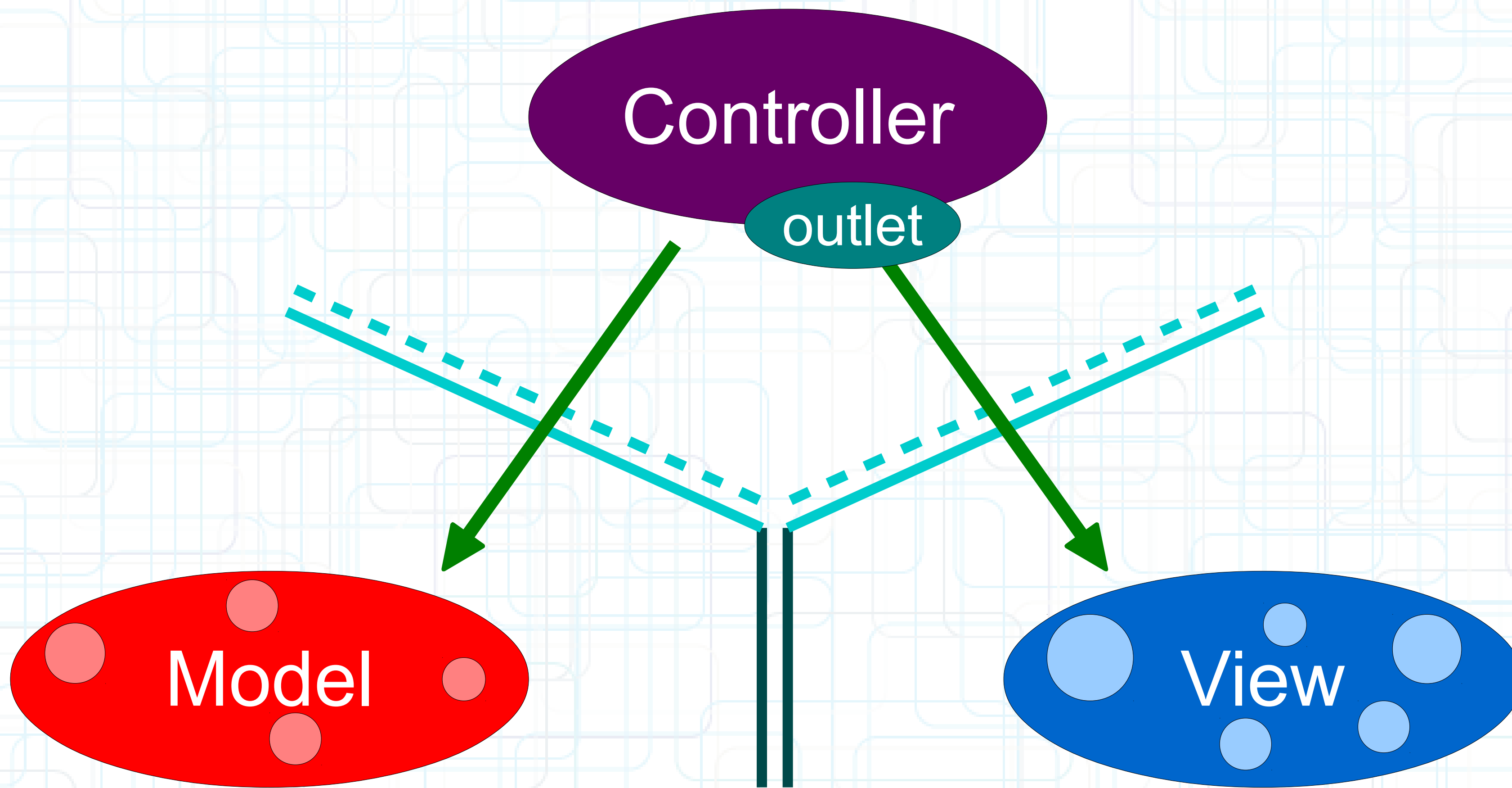
- The Model and View should never speak to each other.

MVC Design Model



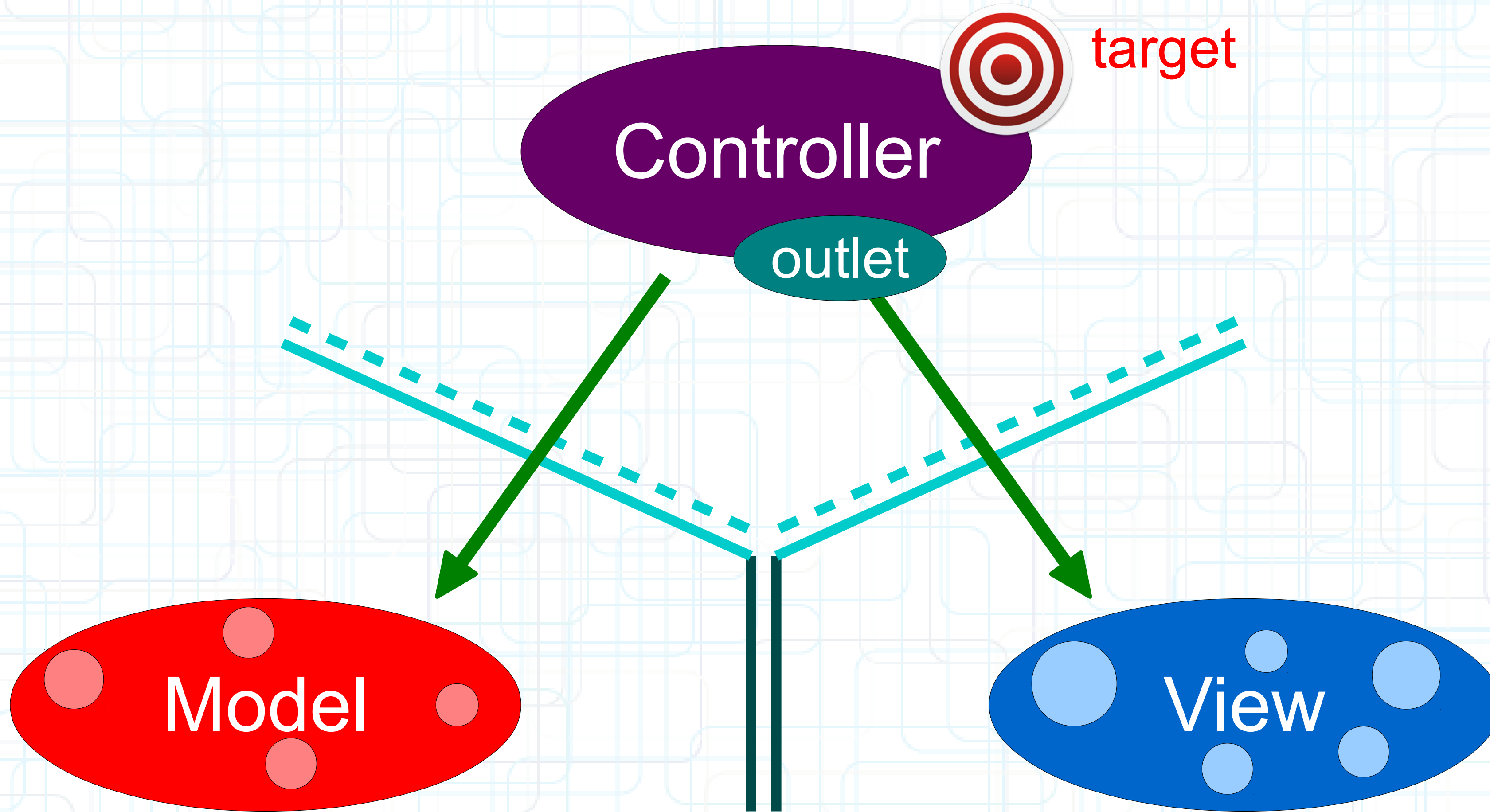
- Can the View speak to its Controller?

MVC Design Model



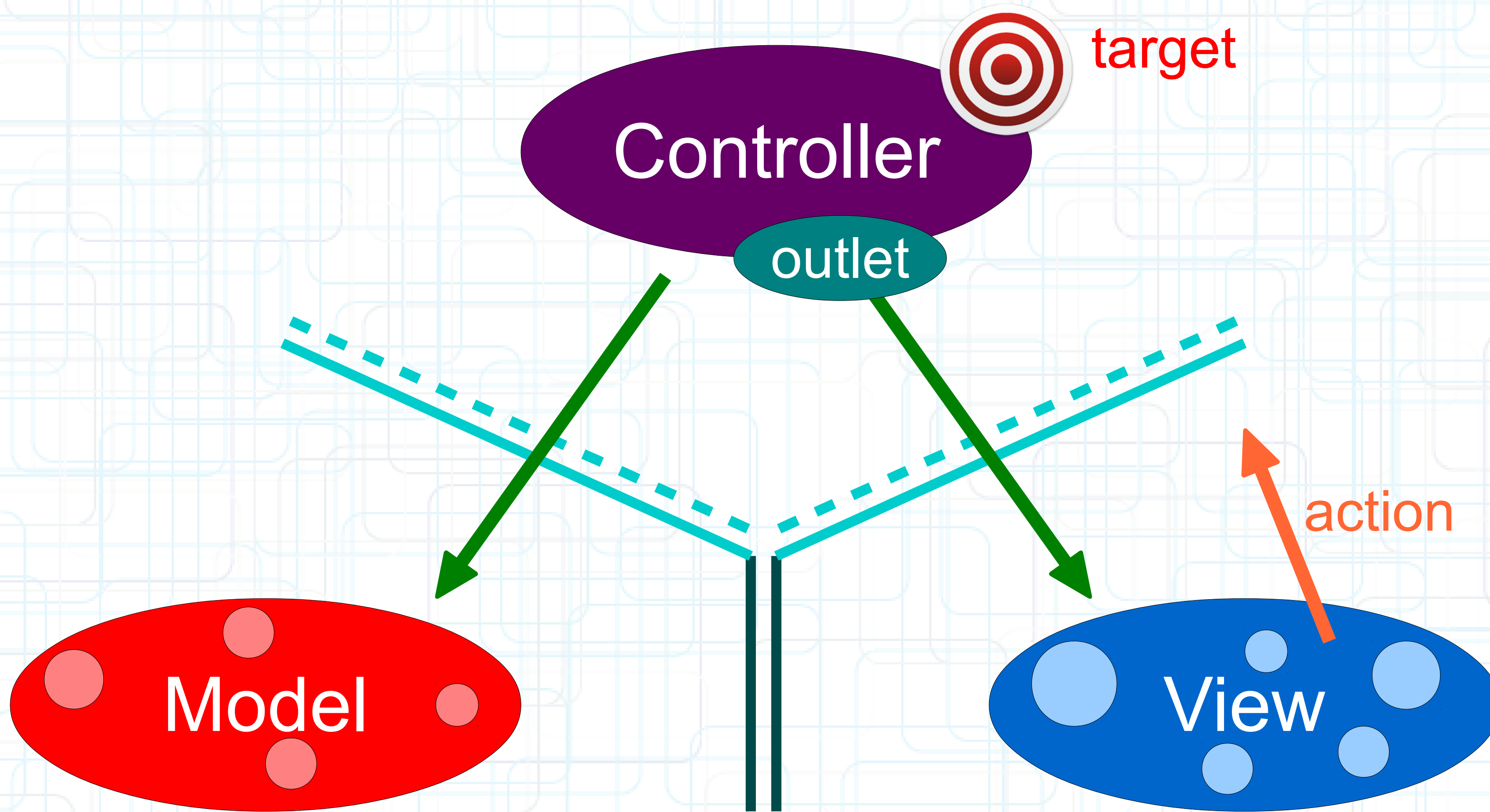
- Sort of. Communication is blind and structured.

MVC Design Model



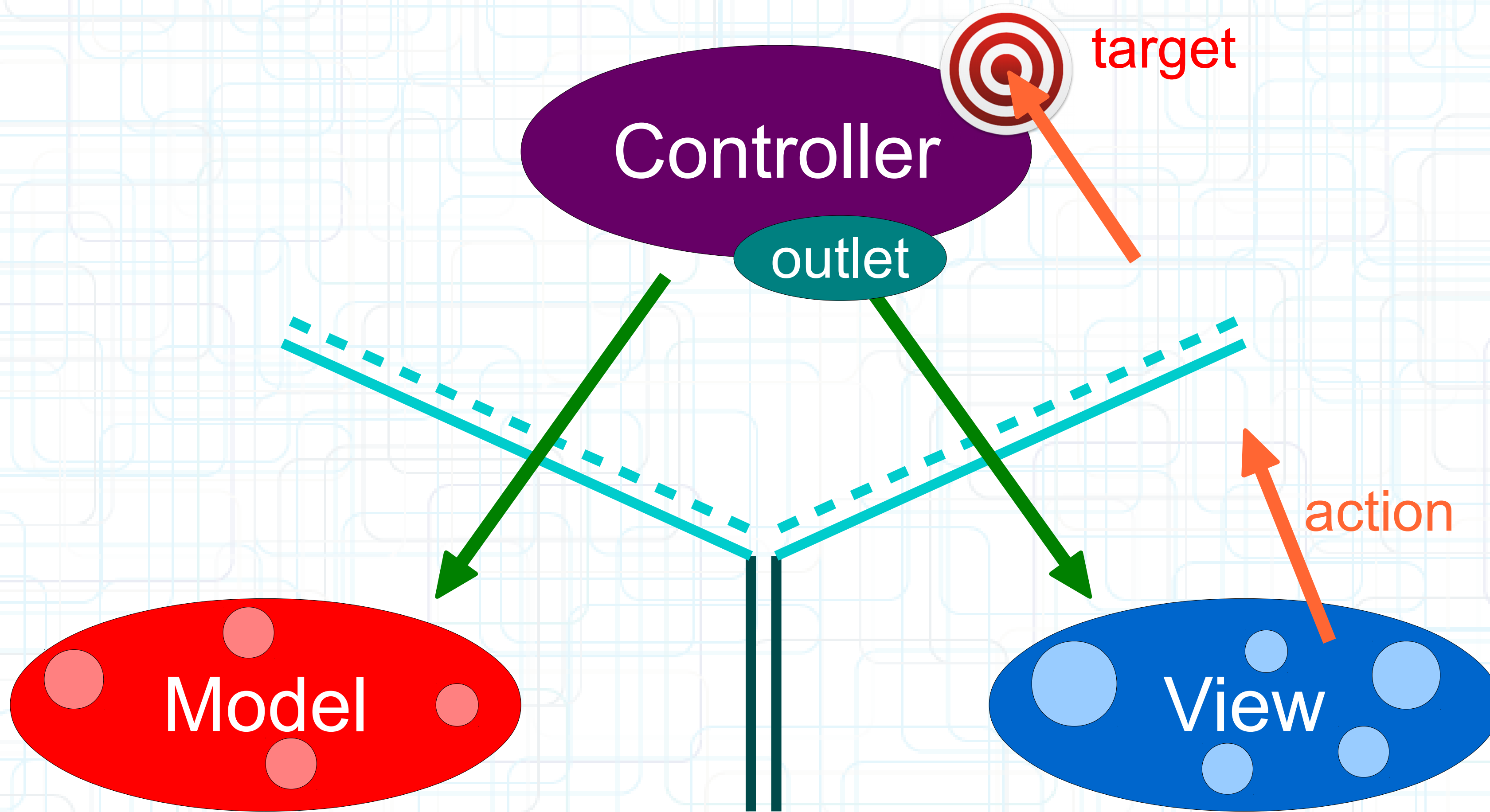
- The Controller can drop a target on itself.

MVC Design Model



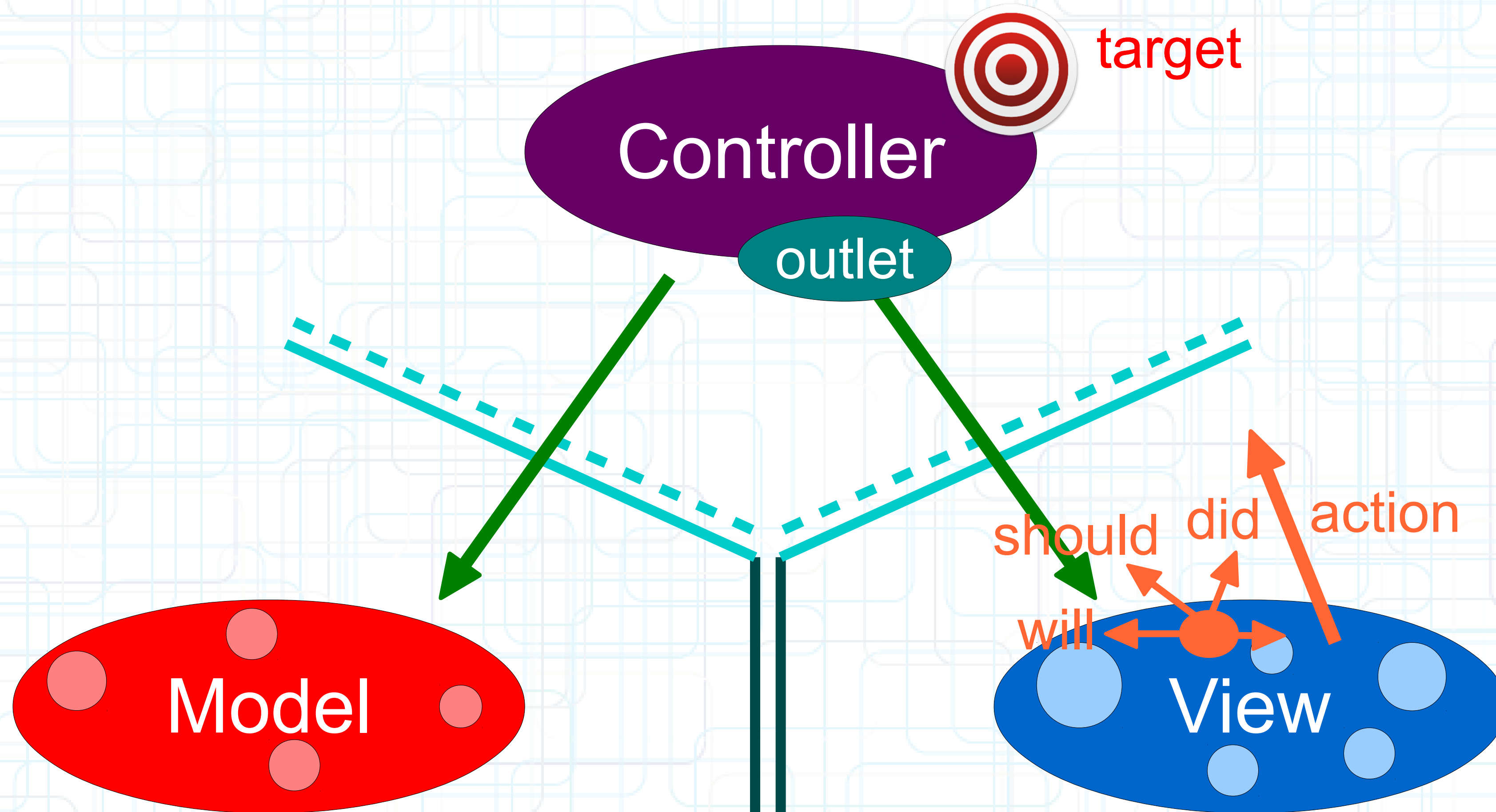
- Then hand out an action to the View.

MVC Design Model



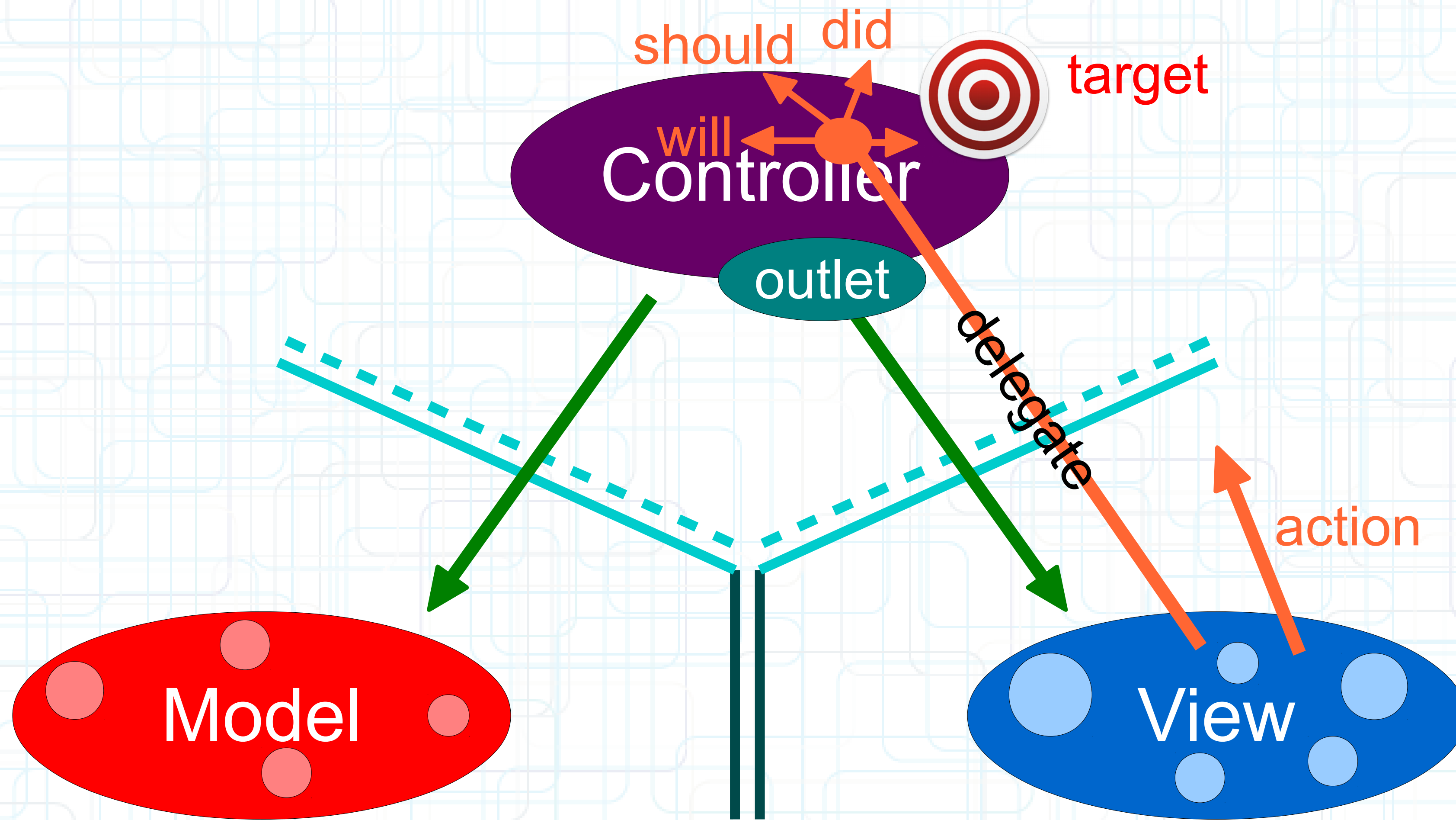
- The View sends the action when things happen in the UI.

MVC Design Model



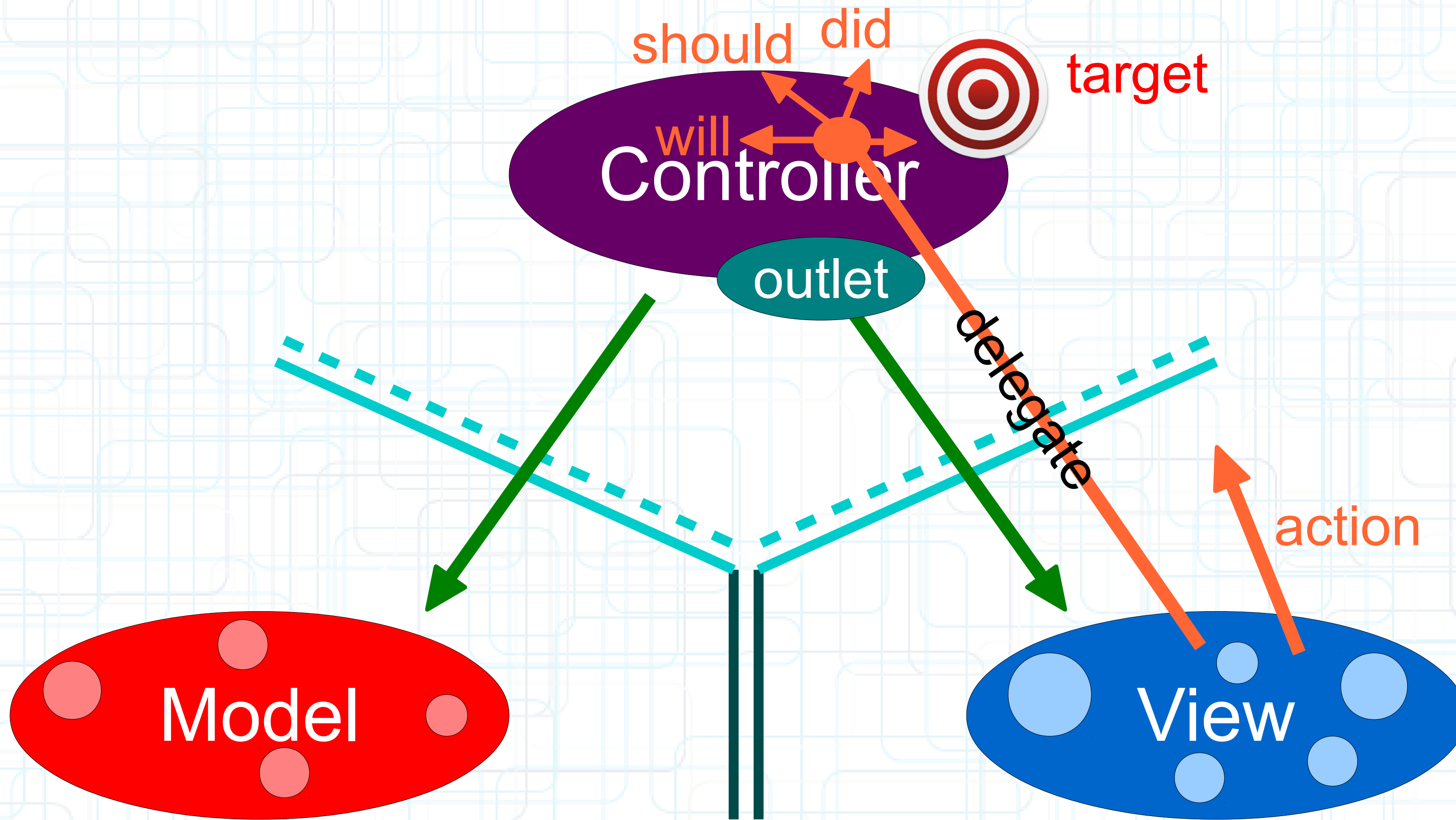
- Sometimes the View needs to synchronize with the Controller.

MVC Design Model



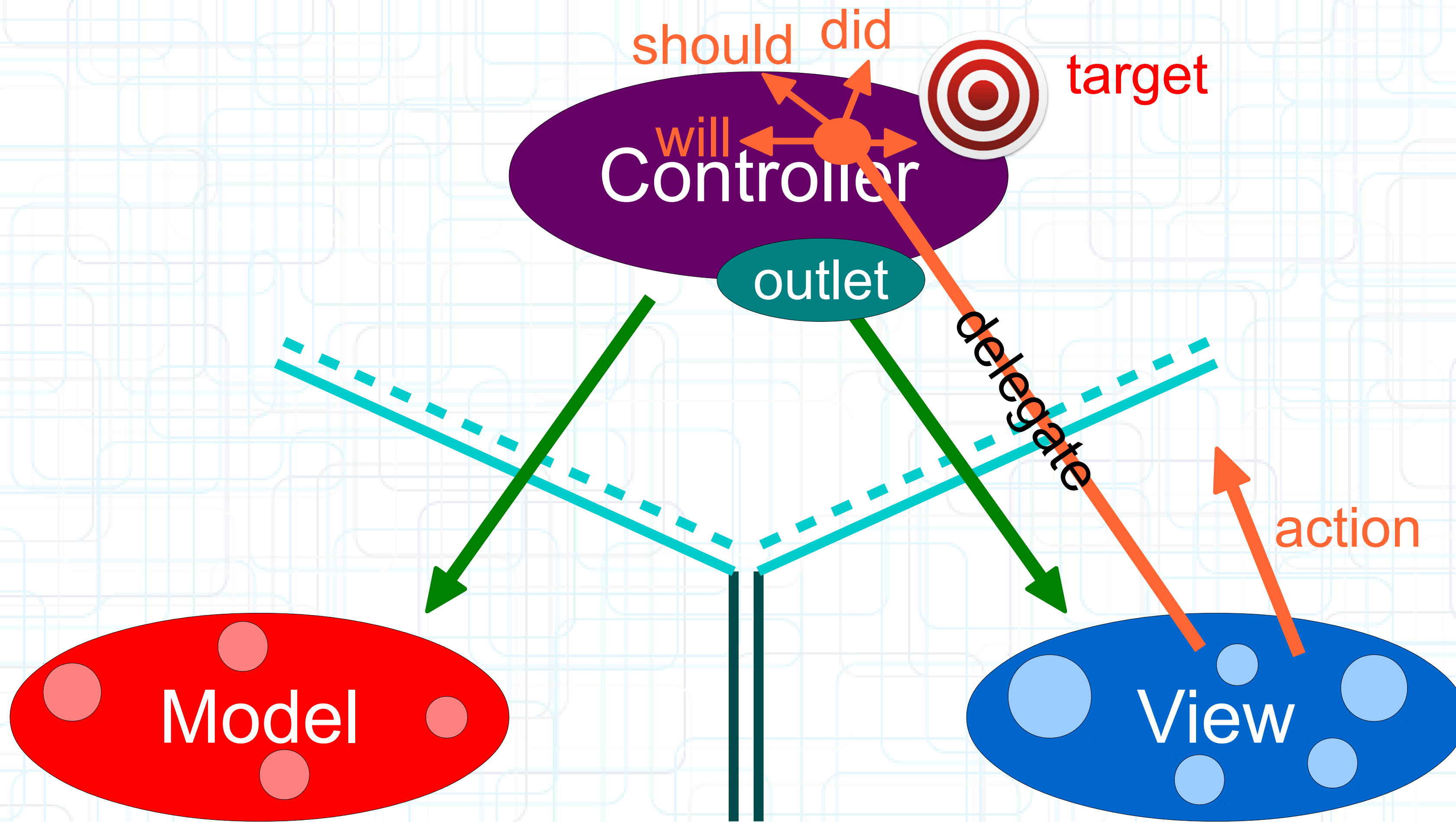
- The Controller sets itself as the View's delegate.

MVC Design Model



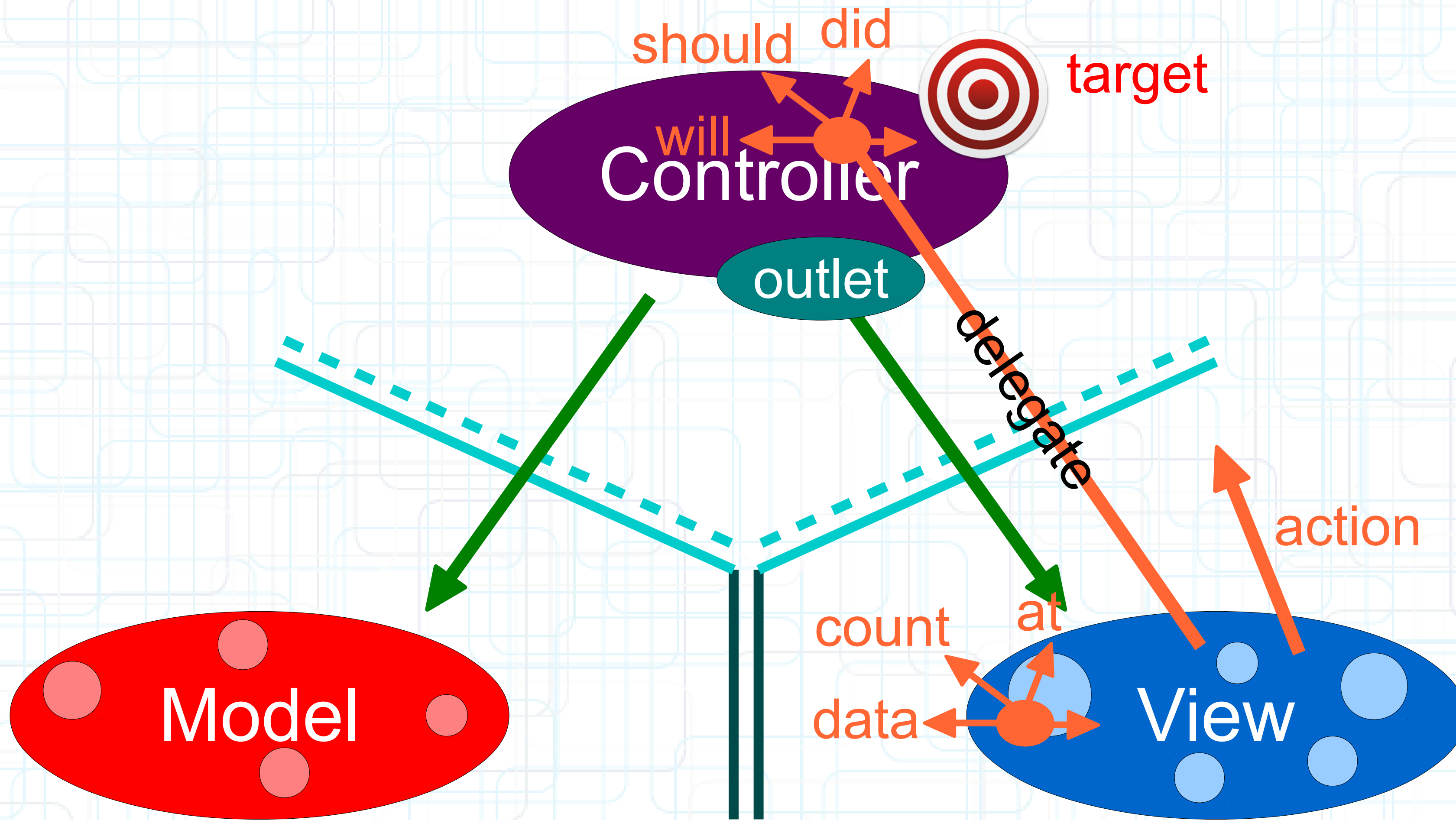
- The delegate is set via a protocol (it's blind to the View class).

MVC Design Model



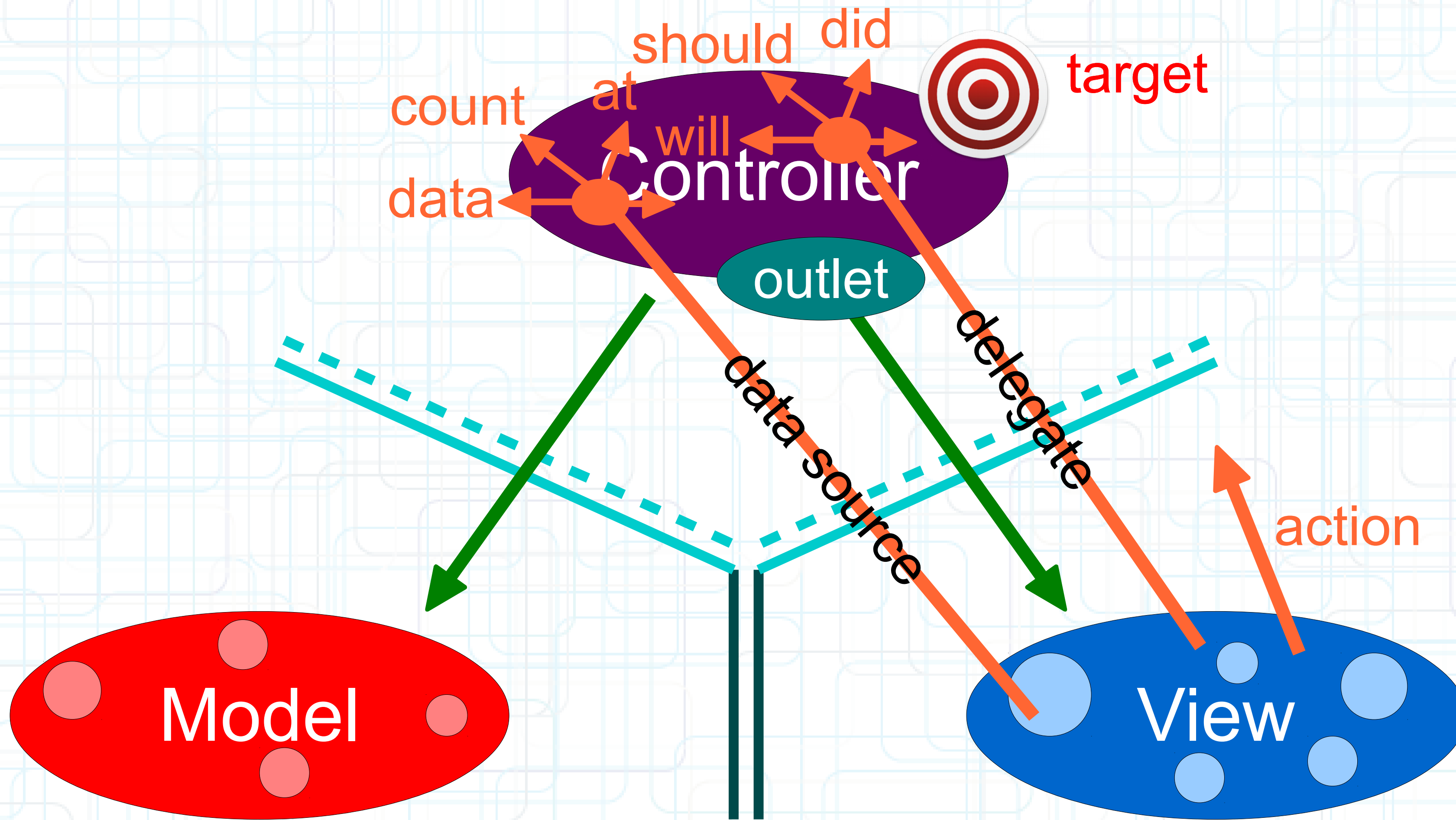
- Views do not own the data they display.

MVC Design Model



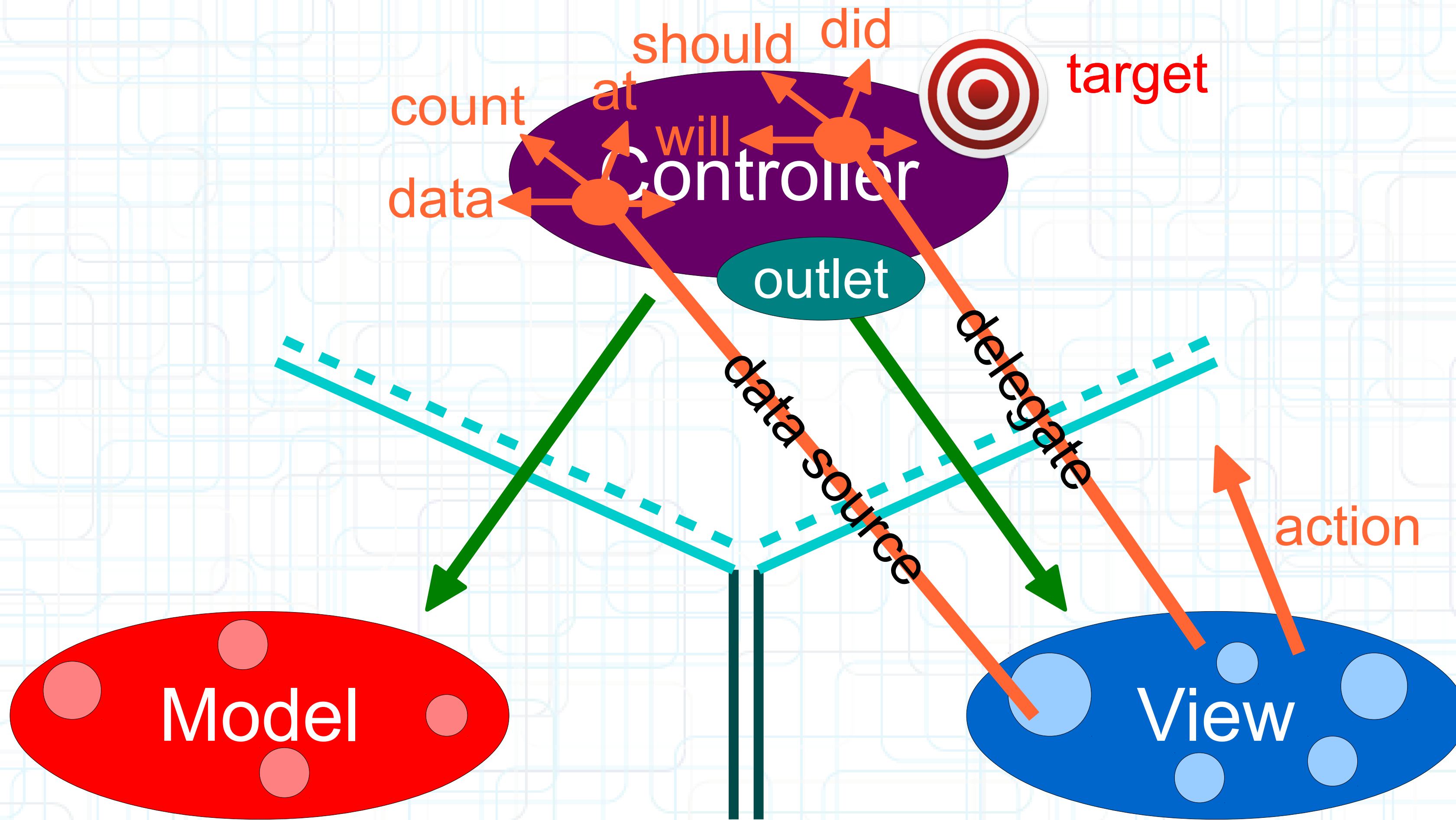
- If needed, they have a protocol to acquire the data.

MVC Design Model



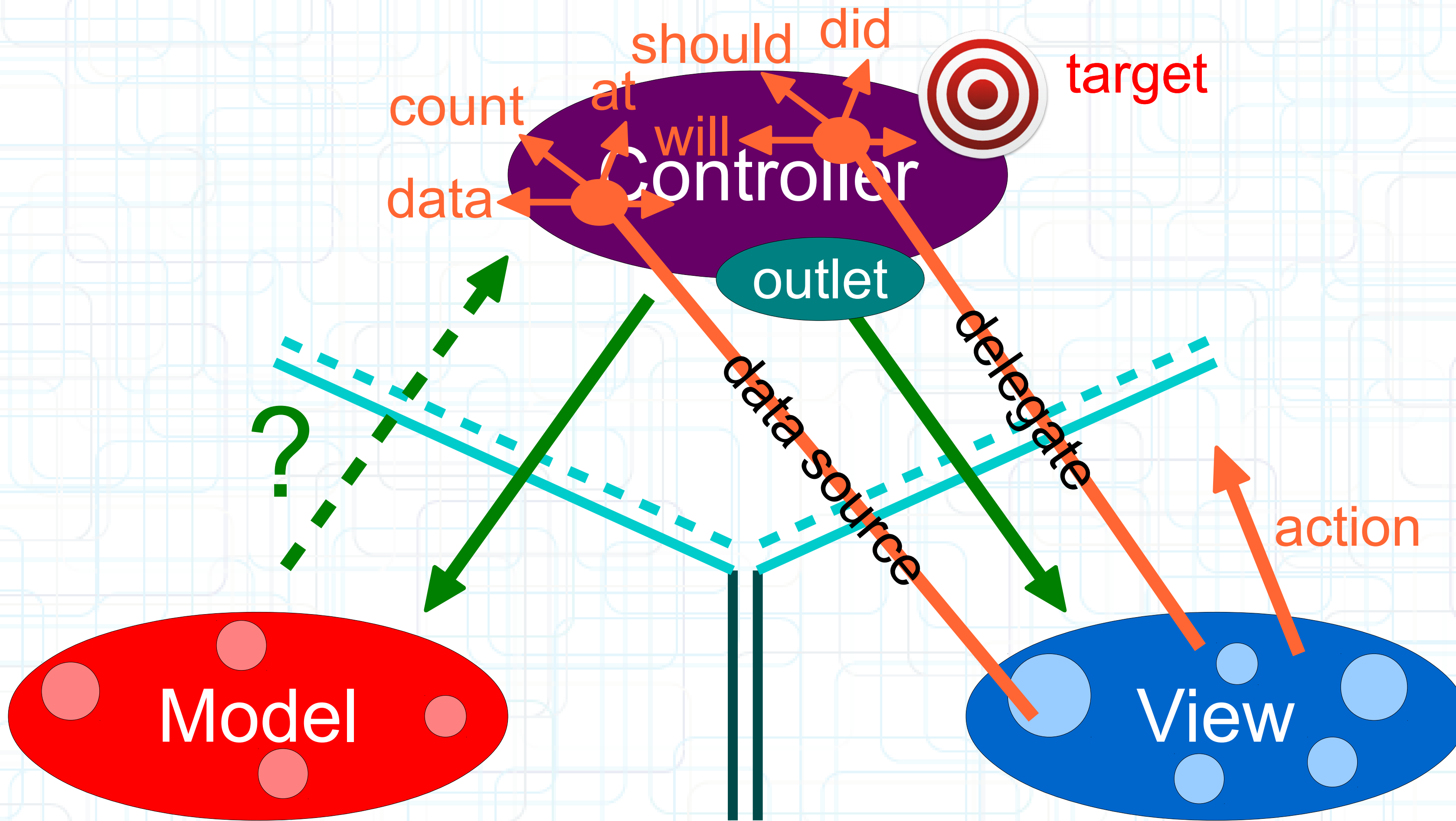
- Controllers are almost always that data source (not the Model).

MVC Design Model



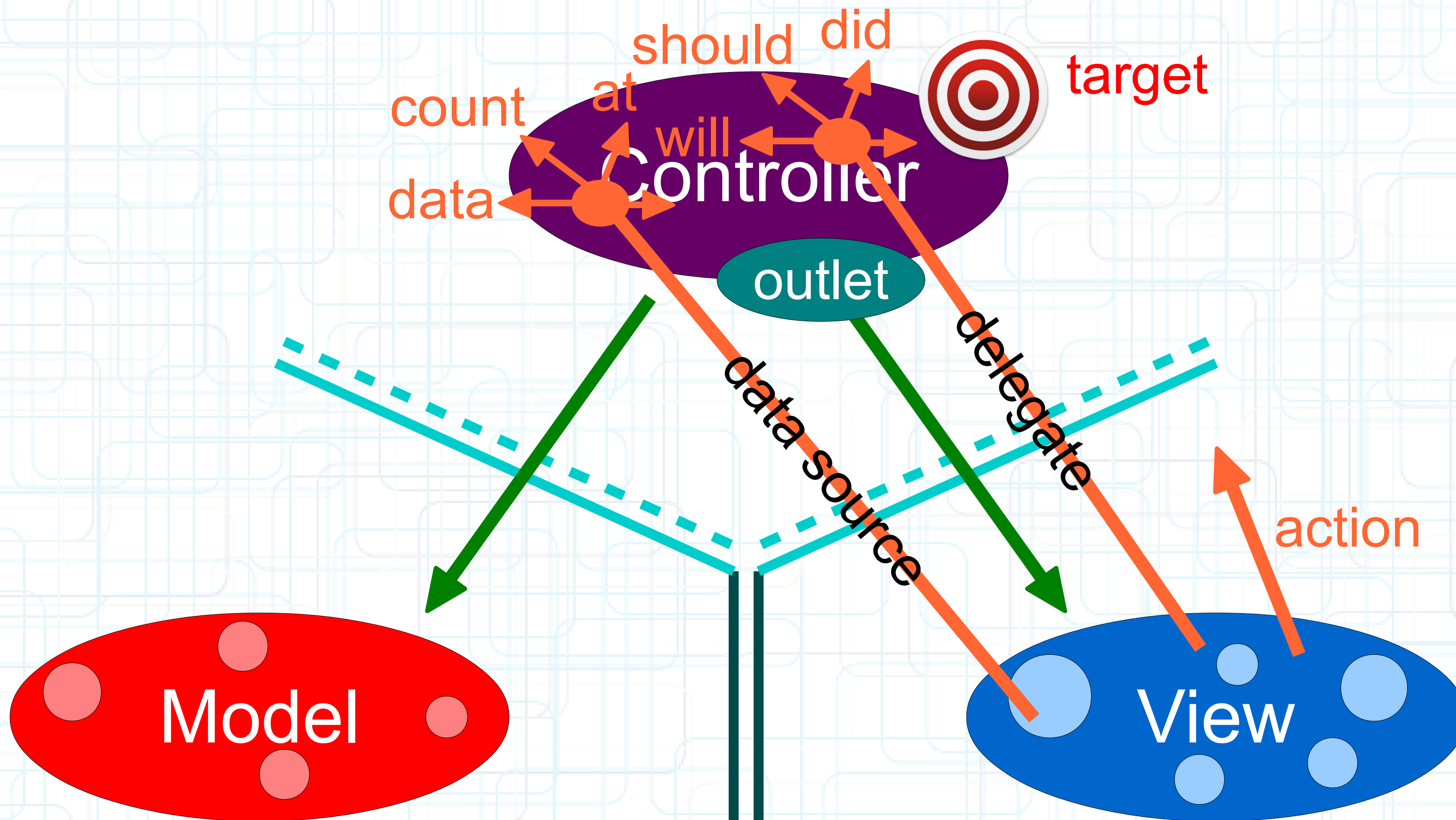
- Controllers interpret/format Model information for the View.

MVC Design Model



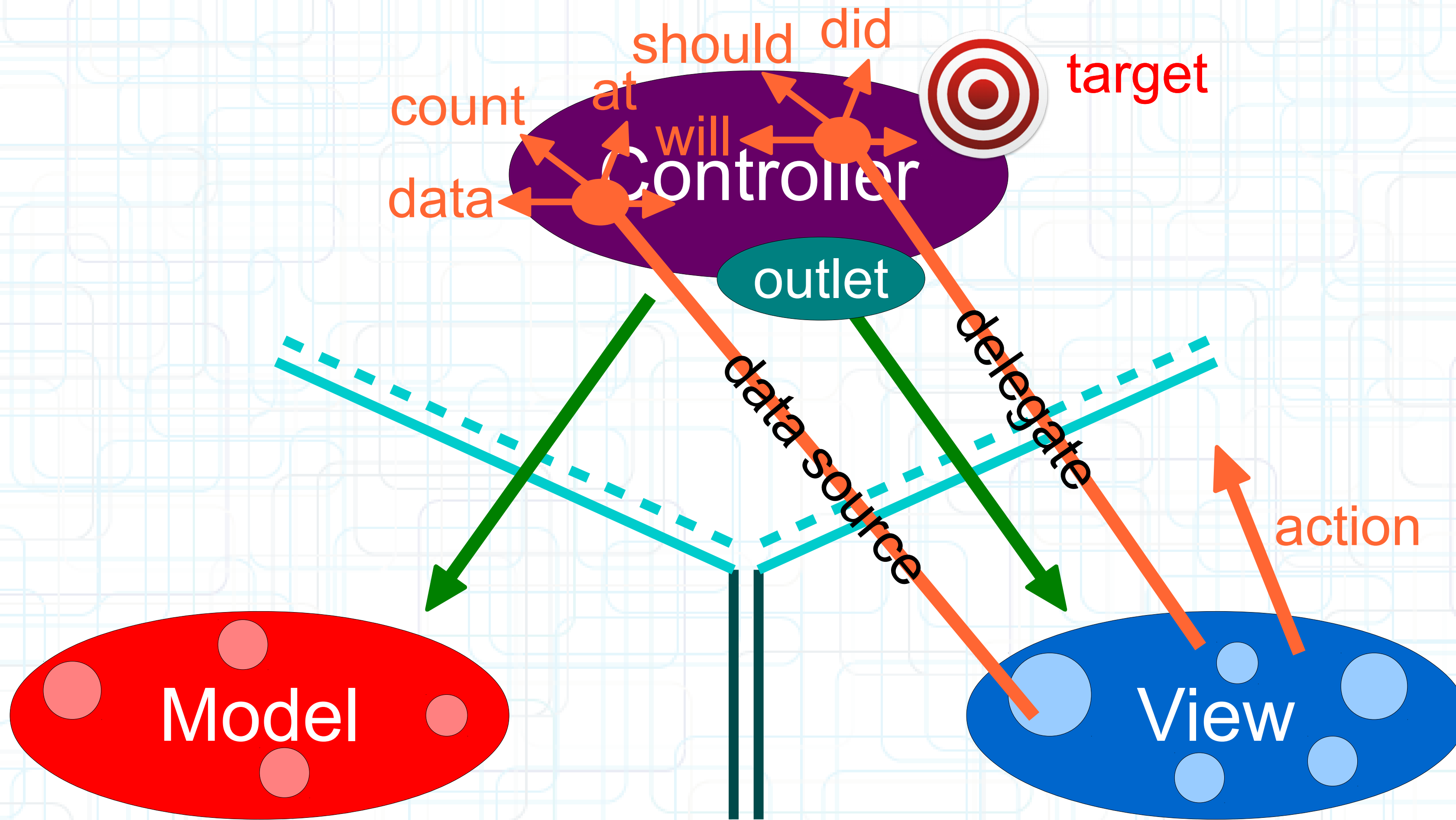
- Can the Model talk directly to the Controller?

MVC Design Model



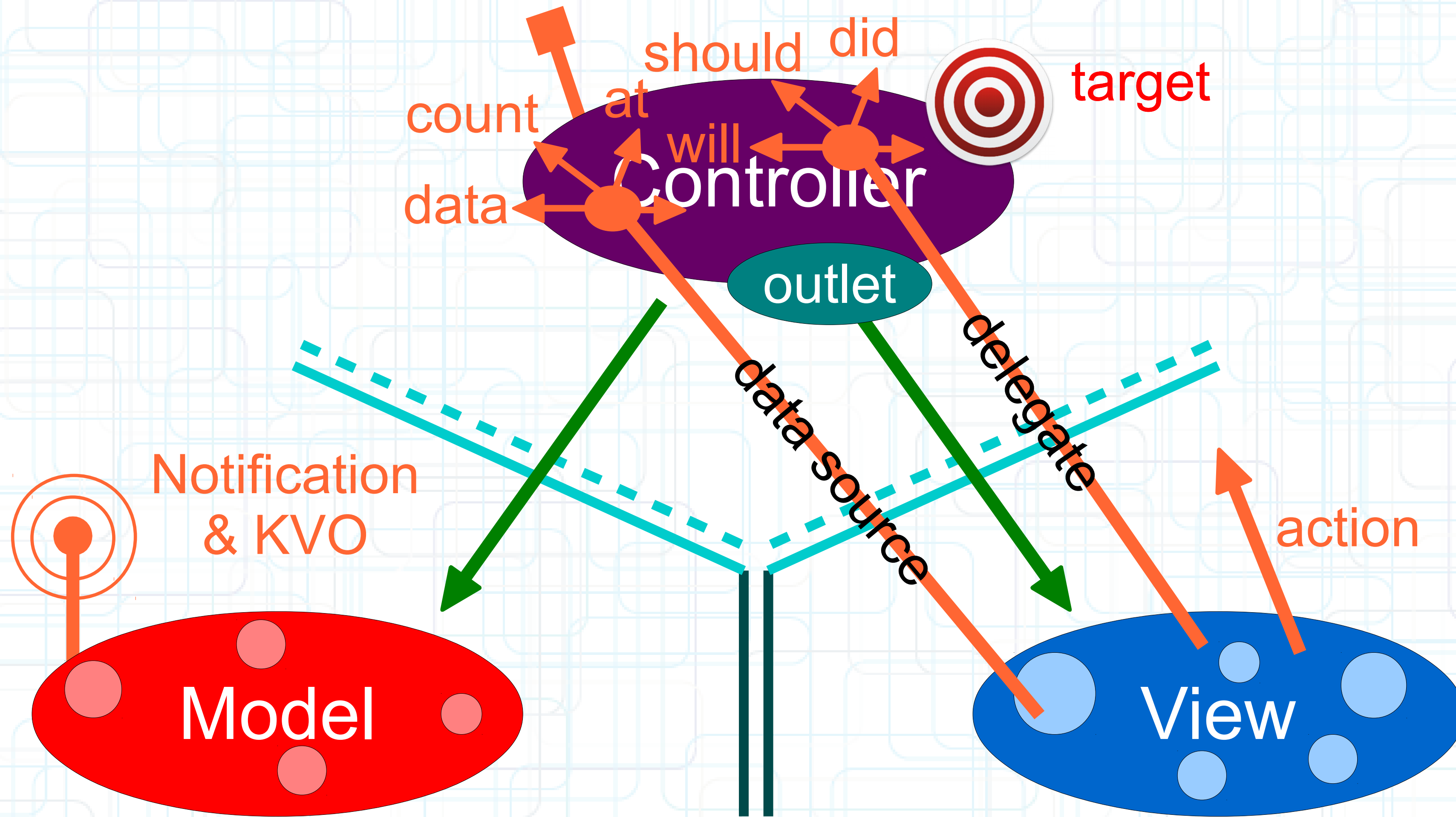
- No. The Model is (should be) UI independent.

MVC Design Model



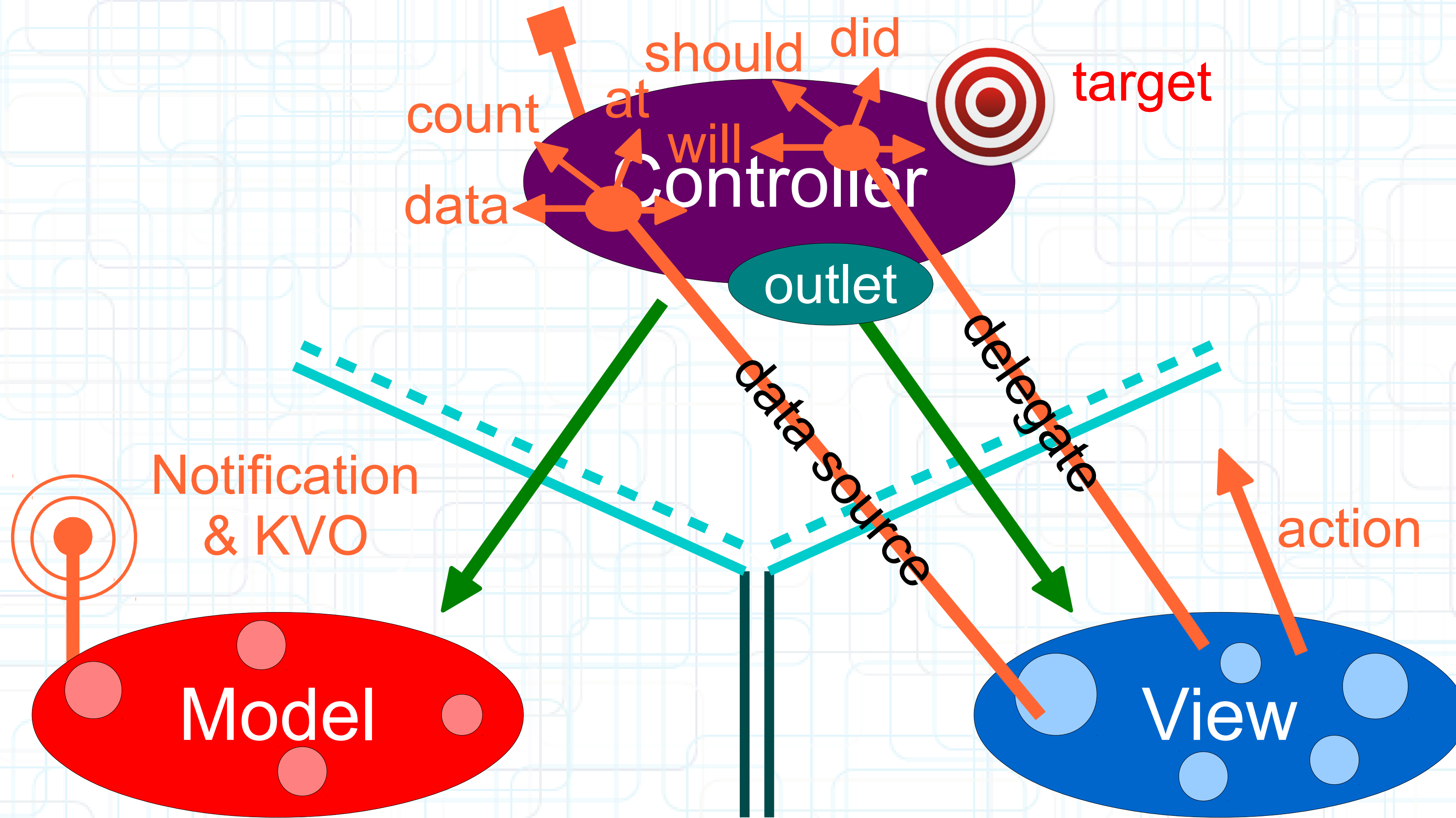
- But what if the Model has information to update or something?

MVC Design Model



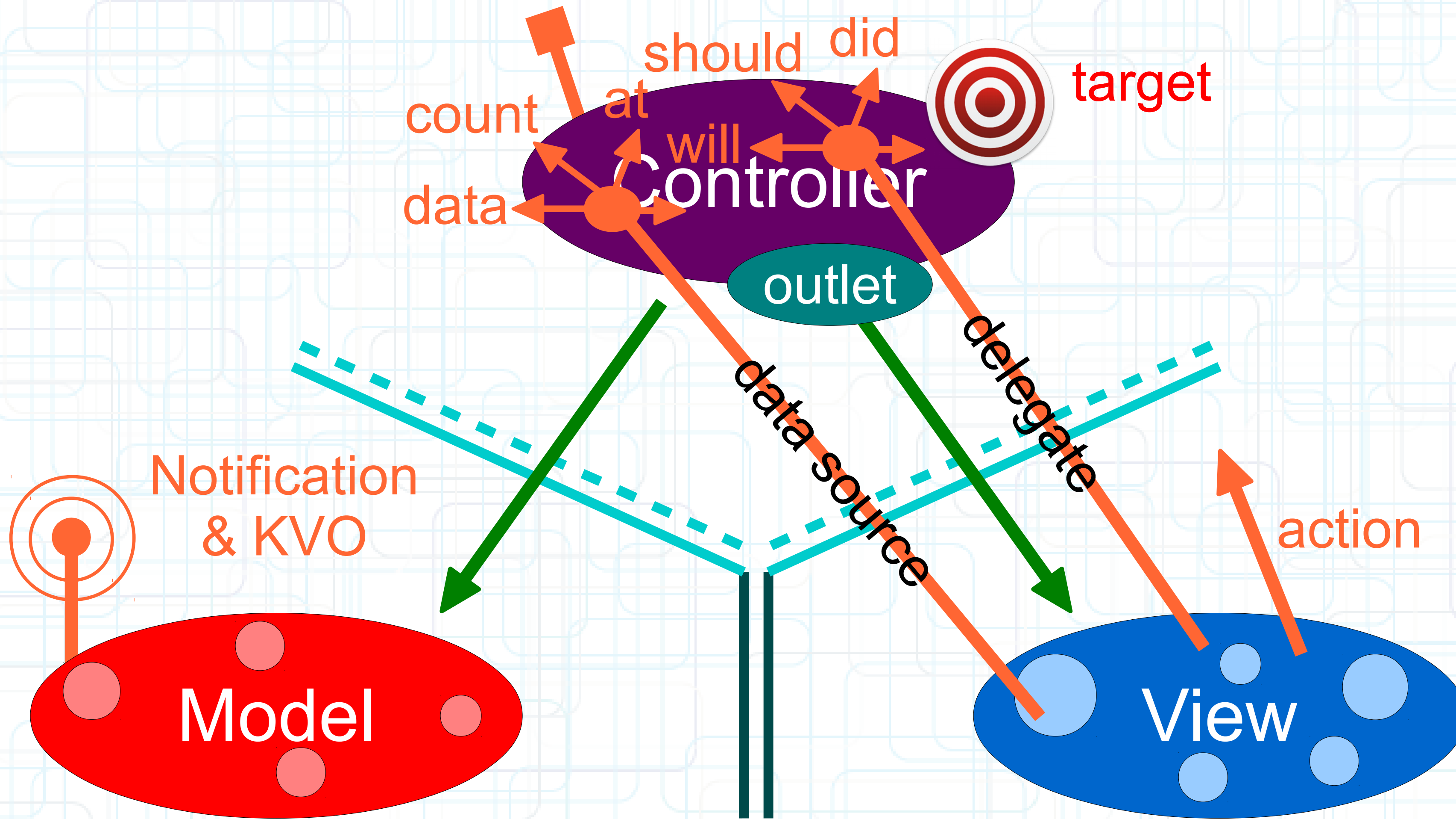
- It uses a “radio station” - broadcast mechanism.

MVC Design Model



- Controllers (or other Models) “tune in” to interesting stuff.

MVC Design Model



- Now combine MVC groups to make complicated programs.

Introduction to Swift

- The Swift language is a simple and concise computer language designed to enable sophisticated object oriented programming.
- Swift is a modern OOP language that provides syntax for defining classes and methods, as well as other constructs that promote dynamic extension of classes.
- If you have programmed with object-oriented languages before, you can learn the basic syntax of Swift from the following slides.

Introduction to Swift

- The traditional object-oriented concepts, such as encapsulation, inheritance, and polymorphism, are all present in Swift.

- These important aspects that are going to be discussed in detail in the next lecture.

- We are now going to discuss the basic concepts.

Introduction to Swift

We will talk about:

- Code Organization
- Classes
- Strong Typing
- Constants and Variables
- Basic Types, Tuples
- Optionals
- Functions Declaration and Calling

Code Organization

- As many other OOP languages, (most of) the Swift code is organized in **classes**, **structures** and **enumerations**.
- Additionally, you can use (recommended) extensions and protocols.
- Adding **extensions** is an extremely powerful mechanism that allows you to add new functionality to existing classes, structures or enumeration types without having to subclass. That means you can add your own methods to Cocoa classes such as `UIView` and `UIImage`!
- **Protocols** are used for delegation, a mechanism in which one object acts on behalf of, or in coordination with, another object.

Code Organization

- It usually recommended to define a single class in a *.swift file.
- However, you can define multiple classes, extensions and protocols in a single file. These can also be encapsulated in other classes. Depends on what is needed!
- Unlike Objective-C, you do not have to define header files and source files to separate public declarations from the implementation.

Code Organization

- When you want to include frameworks in your source code, you typically use an `import` directive.
- This is like `#include`, except that it makes sure that the same file is never included more than once.
- Not necessary to import your own classes when you need to use them, just the frameworks.

Code Organization

- To add a single line comment use: `// my comment`
- To add a multiple line comment use: `/* my very long comment */`
- Use these special comments comment your code (will be surfaced in the Xcode source navigator):

```
// MARK: - (dash inserts a line)

// TODO:

// FIXME:
```

- Always use MARK. If you don't use the source navigator to browse through your classes' methods, you're doing it wrong!

Code Organization

C Bicycle

- P *style*
- P *gearing*
- P *handlebar*
- P *frameSize*
- P *numberOfTrips*
- P *distanceTravelled*
- M init(style:gearing:handlebar:frameSize:)
- M travel(distance:)

Printable

C Bicycle

- P *description*

FIXME: Use a distance formatter

TODO: Allow bikes to be named?

Classes in Swift

- Classes in Swift provide the basic construct for encapsulating some data with the actions that operate on that data.
- An object is a runtime instance of a class, and contains its own in-memory copy of the instance variables declared by that class and pointers to the methods of the class.
- The definition and the implementation of a class in Swift are specified in a single file, although extensions are allowed in different files.
- The class declaration (in a `.swift` file) includes instance variables, methods associated with the class and their implementation.

Classes in Objective-C

- Here is an example where `Point` inherits from the Cocoa's base class.

- The class declaration begins with the `class` compiler directive.

```
3   import Foundation
4
5   class Point : AnyObject
6   {
7       private var x: Float
8       private var y: Float
9
10      init(x: Float, y: Float)
11      {
12          self.x = x
13          self.y = y
14      }
15
16      func quadrant() -> String
17      {
18          var q = "Unknown"
19
20          if x >= 0 && y >= 0
21          {
22              q = "I"
23          }
24          else if x < 0 && y >= 0
25          {
26              q = "II"
27          }
28          else if x < 0 && y < 0
29          {
30              q = "III"
31          }
32          else if x >= 0 && y < 0
33          {
34              q = "IV"
35          }
36          return q
37      }
38  }
39
40  let A = Point(x:5,y:-3)
41  print(A.quadrant())
```

```
IV
```

Declaring Constants and Variables

- Constants and variables must be declared before they are used. You declare constants with the `let` keyword and variables with the `var` keyword:

```
let secondsPerMinute = 60
var secondsSinceMidnight = 54311
secondsPerMinute = 100 // ERROR!
```

- You can declare multiple constants or multiple variables on a single line, separated by commas:

```
var x = 0.0, y = 0.0, z = 0.0
```

- You can provide a type annotation when you declare a constant or variable:

```
var red, green, blue: Double
```

Constants and Variables

- Constant and variable names can contain almost any character, including Unicode characters:

```
let π = 3.14159
let 🐶🐮 = "dogcow"
```

- You can print the current value of a constant or variable with the `print` function:

```
print(🐶🐮)
```

- Swift uses string interpolation to include the name of a constant or variable as a placeholder in a longer string:

```
print("One minute has \(secondsPerMinute) seconds.")
```

Strong Typing

- Swift is a type-safe language that supports strong typing for variables containing objects.
- When the type cannot be implicitly inferred, strongly typed variables must include the class name in the variable type declaration.
- If you want to use generally-typed variables, you can declare them as `AnyObject`.

Strong Typing

- The following example shows equivalent instance declarations of strongly typed variables:

```
// long form initializer method call
let twenty = Int.init(20)
// uses initializer from Double type
let twenty = Int.init(20.0)
// shorthand for initializer method
let twenty = Int(20)
// uses the initializer from UInt type
let twenty = Int(UInt(20))
// type annotation syntax
let twenty: Int = 20
// inferred type syntax
let twenty = 20
```

Integers

- Swift provides signed and unsigned integers in 8, 16, 32, and 64 bit forms:

  `UInt8, UInt16, UInt32, UInt64, UInt`

  `Int8, Int16, Int32, Int64, Int`

- Unless you need to work with a specific size of integer, always use `Int` or `UInt` for integer values.
- You can access the minimum and maximum values of each integer type with its min and max properties:

```
let minValue = UInt8.min
// minValue is equal to 0
let maxValue = UInt8.max
// maxValue is equal to 255
```

Floating-Point Numbers

- Swift provides two signed floating-point number types:

  `Double` represents a 64-bit floating-point number.

  `Float` represents a 32-bit floating-point number.

- Conversions between integer and floating-point numeric types must be made explicit:

```
let three = 3
let pointOneFour = 0.14
let pi = Double(three) + pointOneFour
let integerPi = Int(pi)
```

Numeric Literals

- All of these integer literals have a decimal value of 17:

```
let decimalInteger = 17
let binaryInteger = 0b10001
// 17 in binary notation
let octalInteger = 0o21
// 17 in octal notation
let hexadecimalInteger = 0x11
// 17 in hexadecimal notation
```

- For decimal numbers with an exponent:

```
let x = 1.24e2 // means 124.0
let y = 1.24e-2 // means 0.0124
```

Type Aliases

- Type aliases define an alternative name for an existing type:

```
typealias AudioSample = UInt16
```

- Once you define a type alias, you can use the alias anywhere you might use the original name:

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

Booleans

- Swift has a basic Boolean type, called `Bool`:

```
let orangesAreOrange = true
let bananasAreBlue = false
```

- Boolean values are particularly useful when you work with conditional statements such as the if statement:

```
if bananasAreBlue {
    print("Maybe on Mars!")
} else {
    print("Actually, no!")
}
```

Booleans

- Swift's type safety prevents non-Boolean values from being substituted for `Bool`. The following example reports a compile-time error:

```
let i = 1
if i {
    // compiler error
}
```

- However, the alternative example below is valid:

```
if i == 1 {
    // compiles successfully
}
```

Tuples

- Tuples group multiple values into a single compound value. The values within a tuple can be of any different type. This tuple describes an HTTP status code:

```
let http404Error = (404, "Not Found")
```

- You can decompose a tuple’s contents into separate constants or variables, which you then access as usual:

```
let (code, message) = http404Error
print("The status code is \(code)")
// Prints "The status code is 404"
```

- You may ignore parts of the tuple with an underscore:

```
let (_, message) = http404Error
print("The status message is \(message)")
```

Tuples

- Alternatively, access the individual element values in a tuple using index numbers starting at zero:

```
print("Status code is \(http404Error.0)")
```

- You can name the individual elements in a tuple when the tuple is defined:

```
let status = (code: 200, message: "OK")
```

- If you name the elements in a tuple, you can use the element names to access the values of those elements:

```
print("Status code is \(status.code)")
// Prints "The status code is 200"
```

Tuples

- You can compare tuples that have the same number of values, as long as each of the values in the tuple can be compared. For example, both `Int` and `String` can be compared, which means tuples of the type `(Int, String)` can be compared. In contrast, `Bool` can't be compared, which means tuples that contain a Boolean value can't be compared:

```
(3, "apple") < (3, "bird")
/* true because 3 is equal to 3, and "apple" is less than "bird" */


(4, "dog") == (4, "dog")
/* true because 4 is equal to 4, and "dog" is equal to "dog" */
```

Optionals

- You use optionals in situations where a value may be absent (`nil`). If there is a value, you can unwrap the optional to access that value:

```
let possibleNumber = "123"
let x = Int(possibleNumber)
// x inferred as Int? or "optional Int"
```

- **Forced unwrapping** means adding an exclamation mark (!) to the end of the optional's name:

```
if x != nil {
    print("x is \(x!)")
}
```

Optionals

- You use **optional binding** to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable:

```
if let x = Int(possibleNumber) {
    print("String converted to \(x)")
} else {
    print("\String was not converted")
}
```

- You can include more optional bindings and Boolean conditions in a single `if` statement:

```
if let x = Int("7"), let y = Int("42") {
    print("\(x) < \(y)")
}
```

Optionals

- If you define an optional variable without providing a default value, the variable is automatically set to `nil` for you:

```
var text: String?
// text is automatically set to nil
```

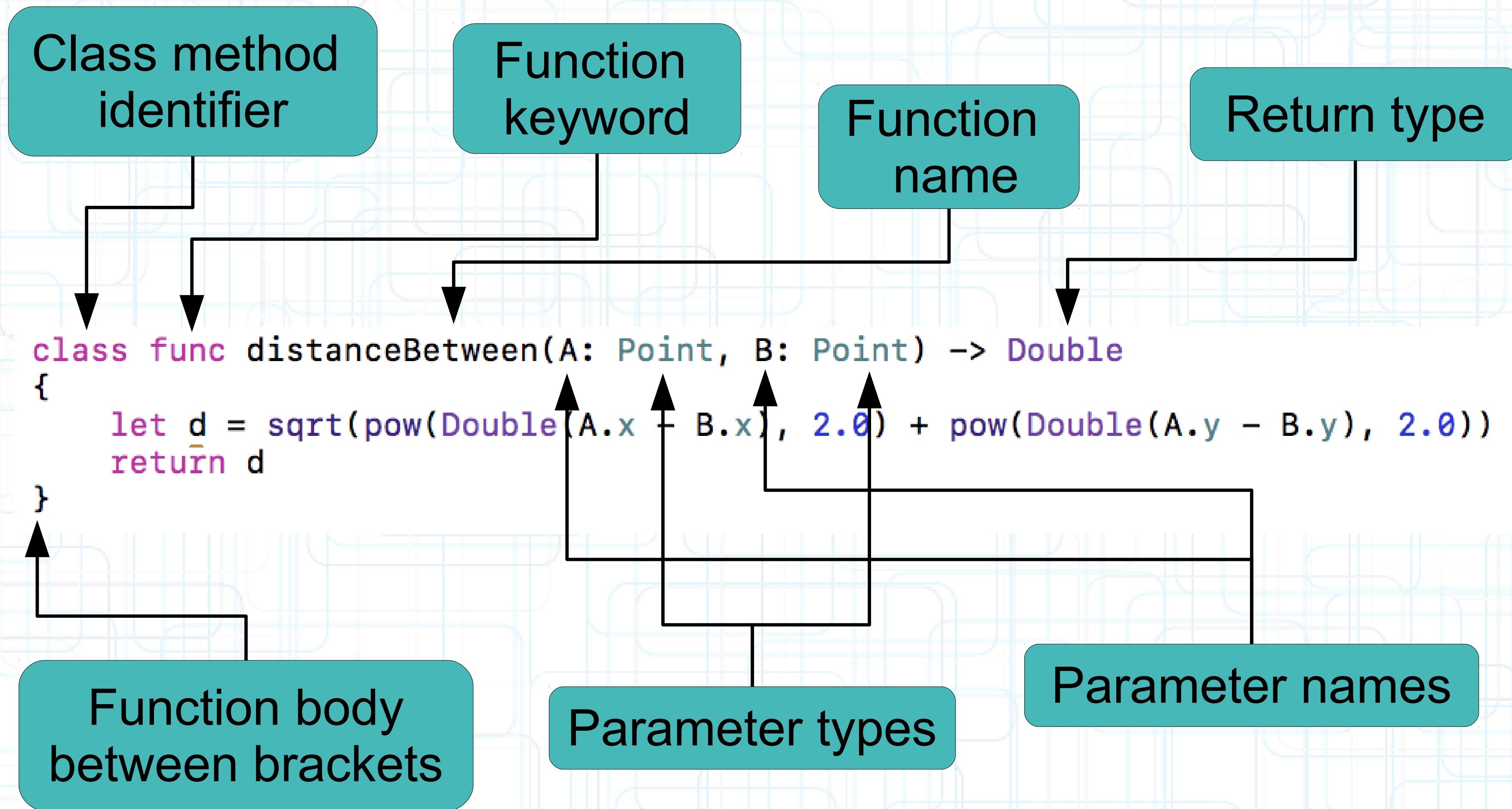
- Sometimes it is clear from a program's structure that an optional will always have a value, after that value is first set. These kinds of optionals are declared as implicitly unwrapped optionals:

```
let a: String! = "Implicitly unwrapped."
let b: String = a
```

- If an implicitly unwrapped optional is `nil` and you try to access its wrapped value, you'll trigger a runtime error!

Functions Declaration

- A class in Swift can declare two types of methods: instance methods and class methods.



Functions and Calling

- The declaration is preceded by the `func` keyword.
- The function's name is then specified:

  `distanceBetween`
- The input parameters are given in parentheses.
- The return type comes after `->`.
- When you want to call a method, you must specify the input parameter labels:

```
let A = Point(x:5, y:-3)
print(A.quadrant())

let B = Point(x:1, y:4)

let d = Point.distanceBetween(A: B, B: A)
```

Functions and Calling

- Swift lets you nest messages. Thus, if you had another object called `myObject` that had methods for accessing an array object and an object to be appended, you could do it in a single line of code:

```
myObject.someArray.append(anObject)
```

- Here is another example:

```
53  print("The distance between A and B is \(Point.distanceBetween(A: B, B: A))")
```

```
The distance between A and B is 8.06225774829855
```

Next Time

A Tour of Swift:

- Strings and Characters
- Collection Types
- Control Flow
- Functions and Closures
- Classes, Structures, Enumerations
- Properties and Methods