Developing Applications for iOS

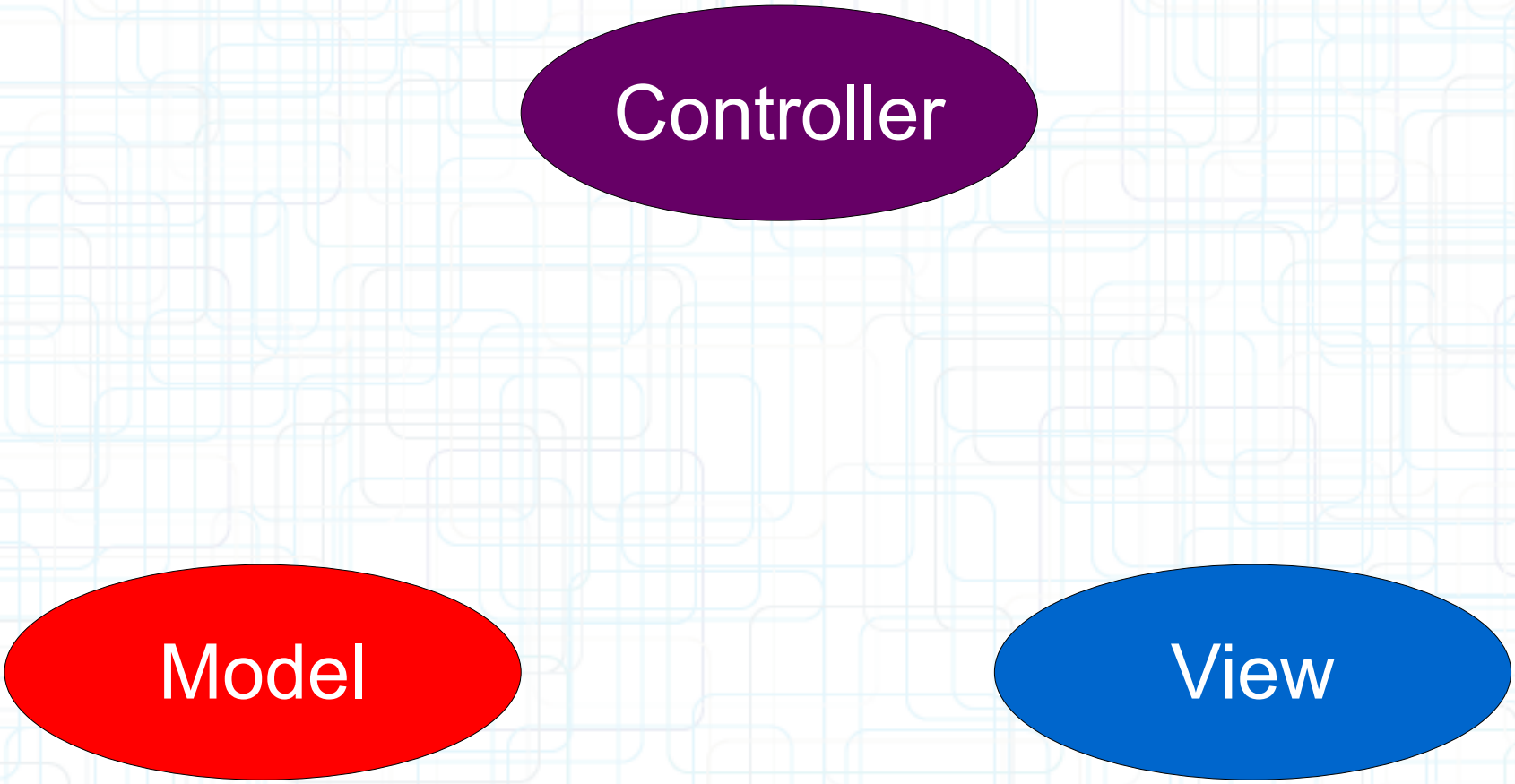
Lecture 2:
MVC Design Concept

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

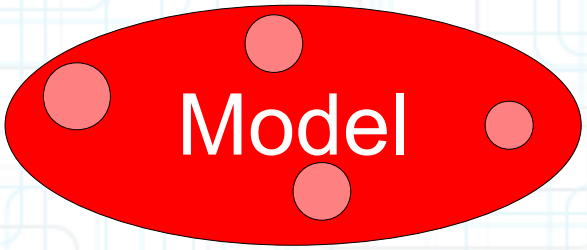
Content

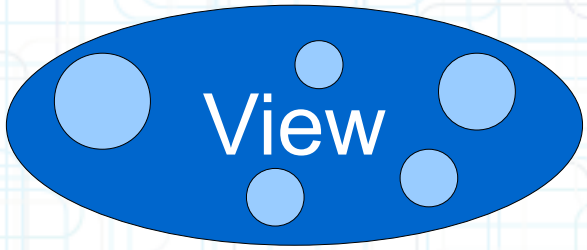
- MVC Design Concept
- Introduction to Objective-C
- Objective-C Example

MVC Design Model



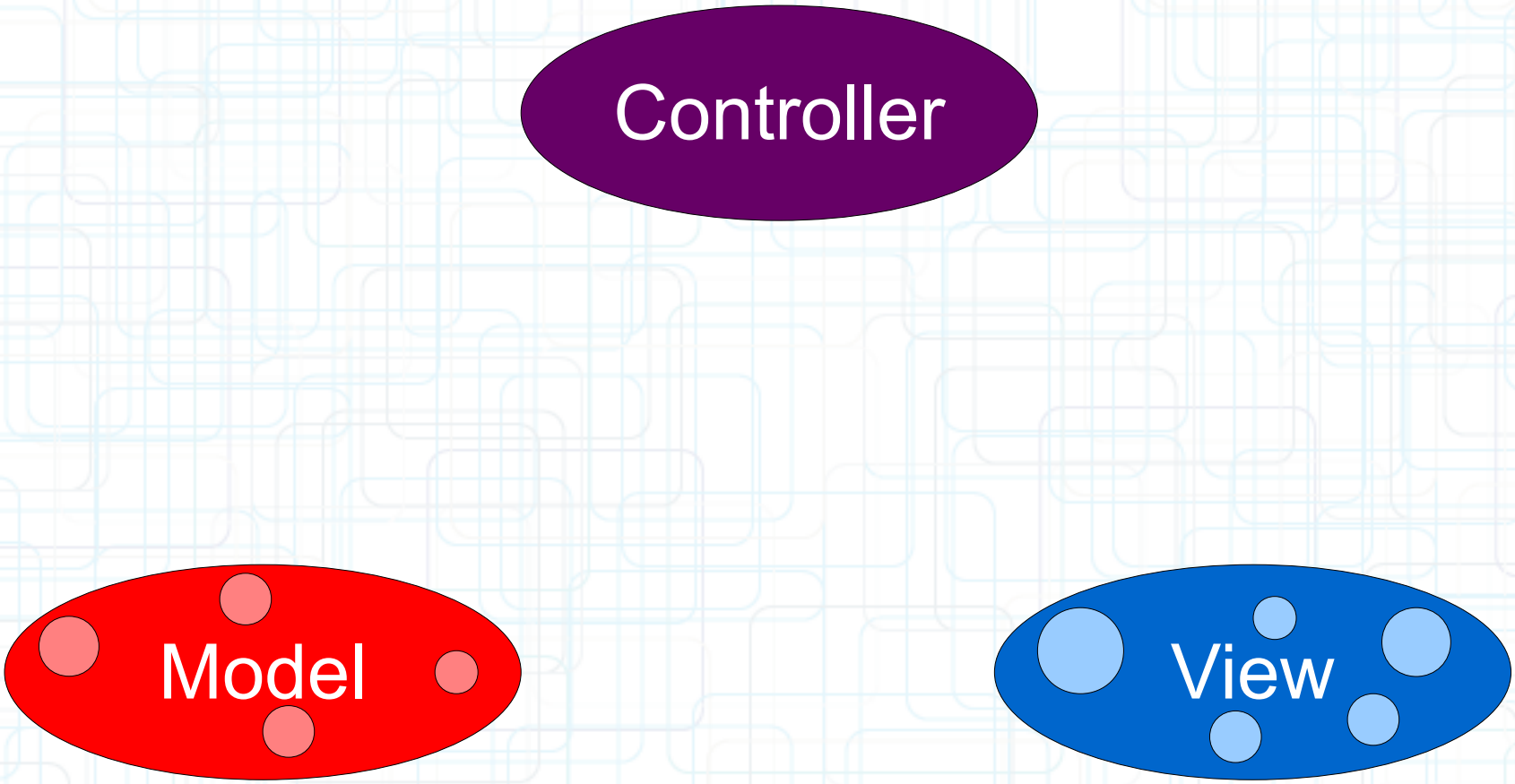
MVC Design Model





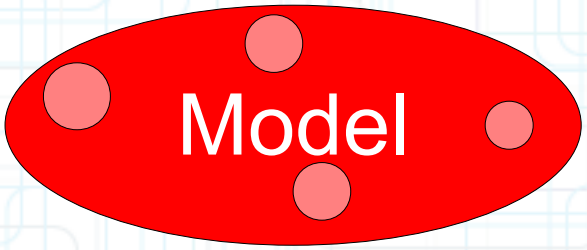
- Divide objects in your program into 3 camps.

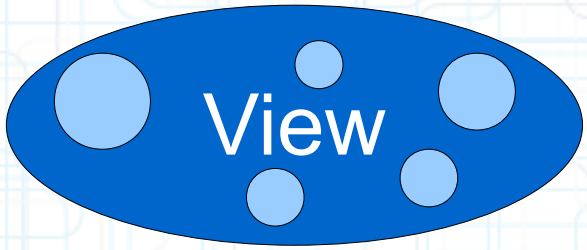
MVC Design Model



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

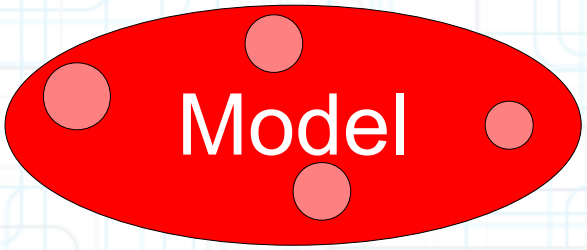
MVC Design Model

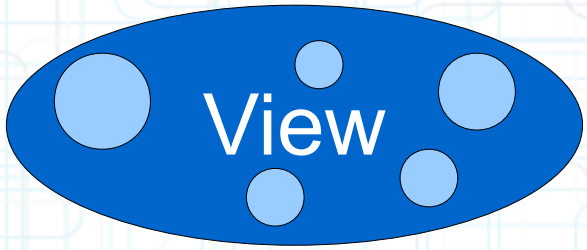




- Controller = How your Model is presented to the user (UI logic)

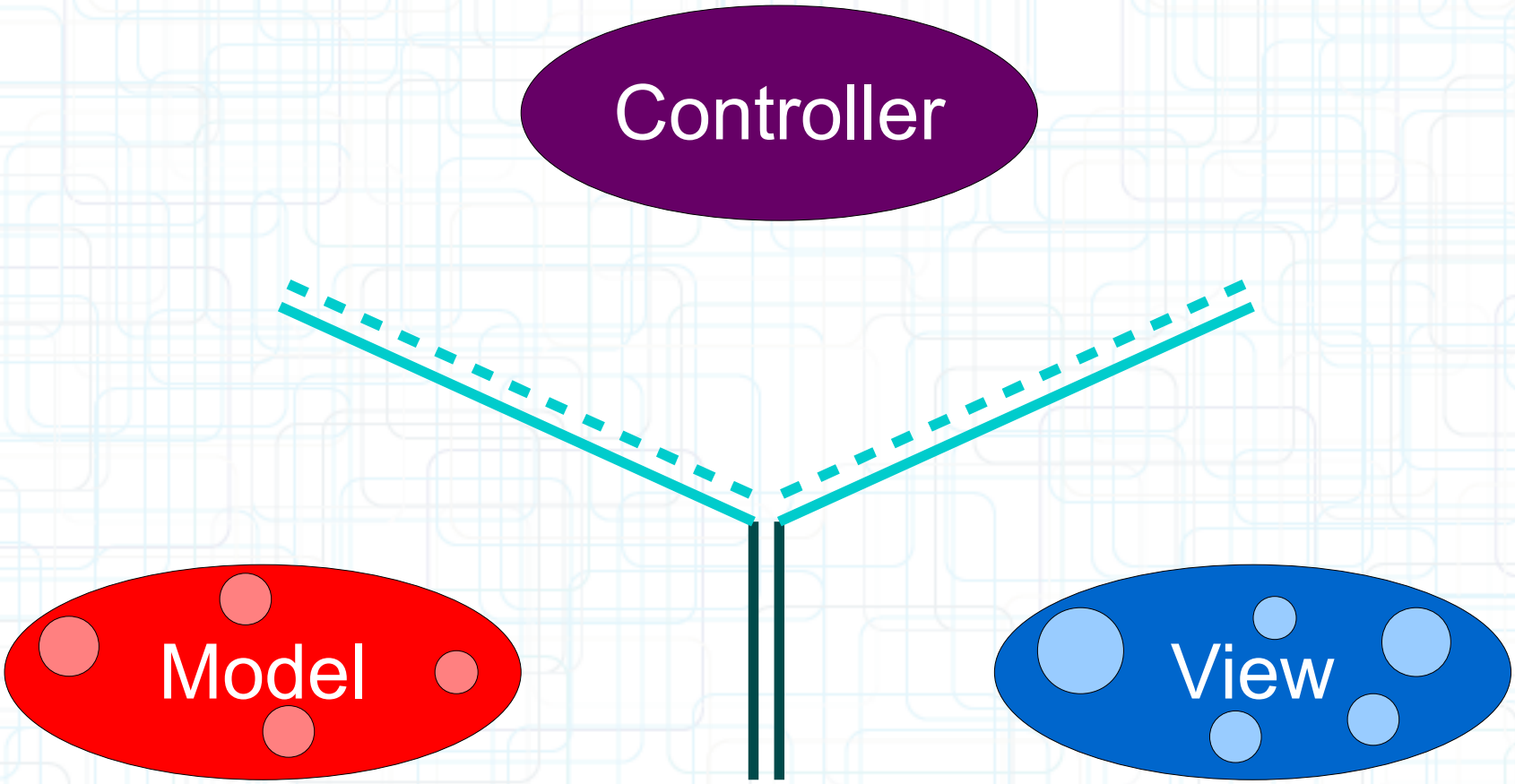
MVC Design Model





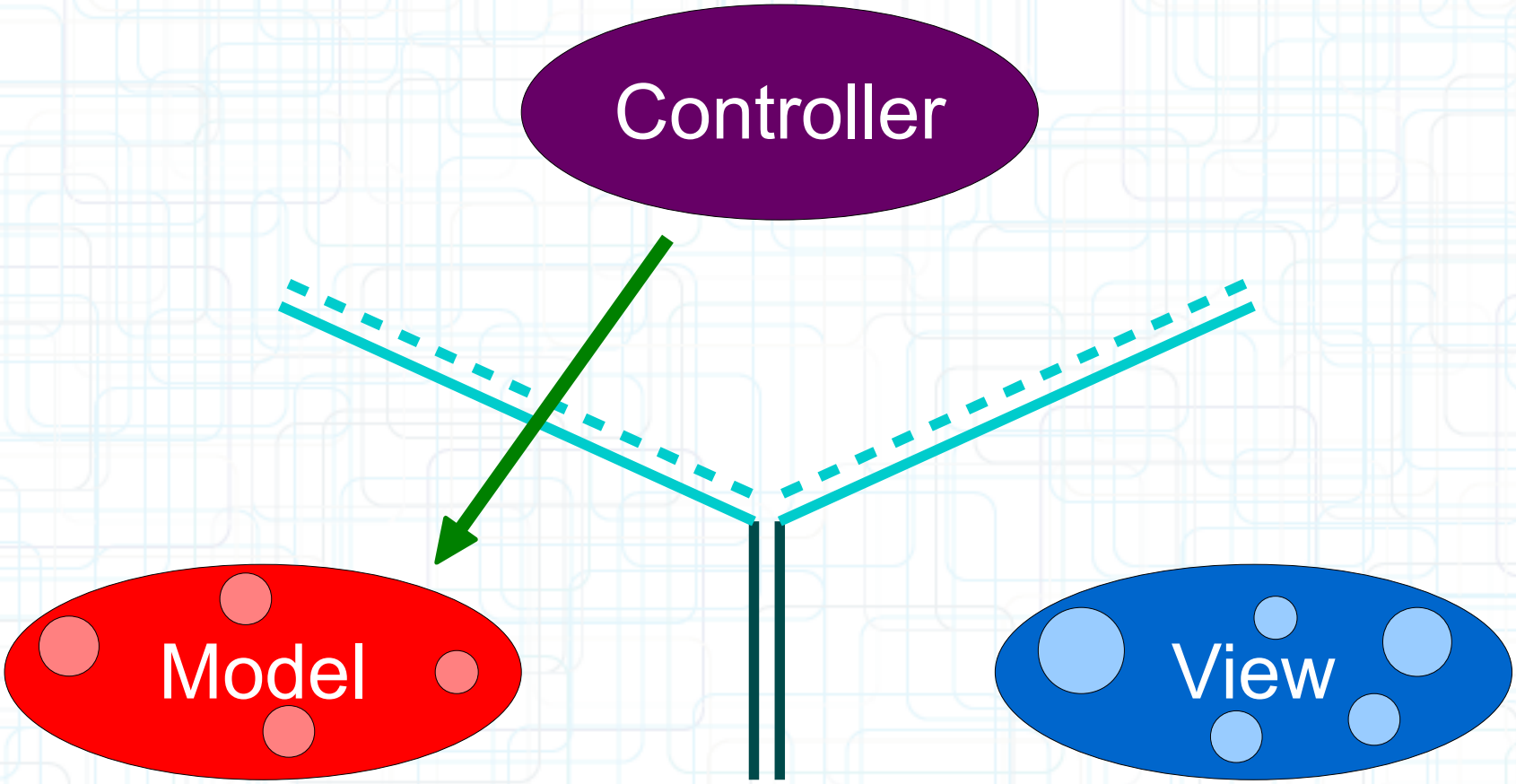
- View = How your application is displayed.

MVC Design Model



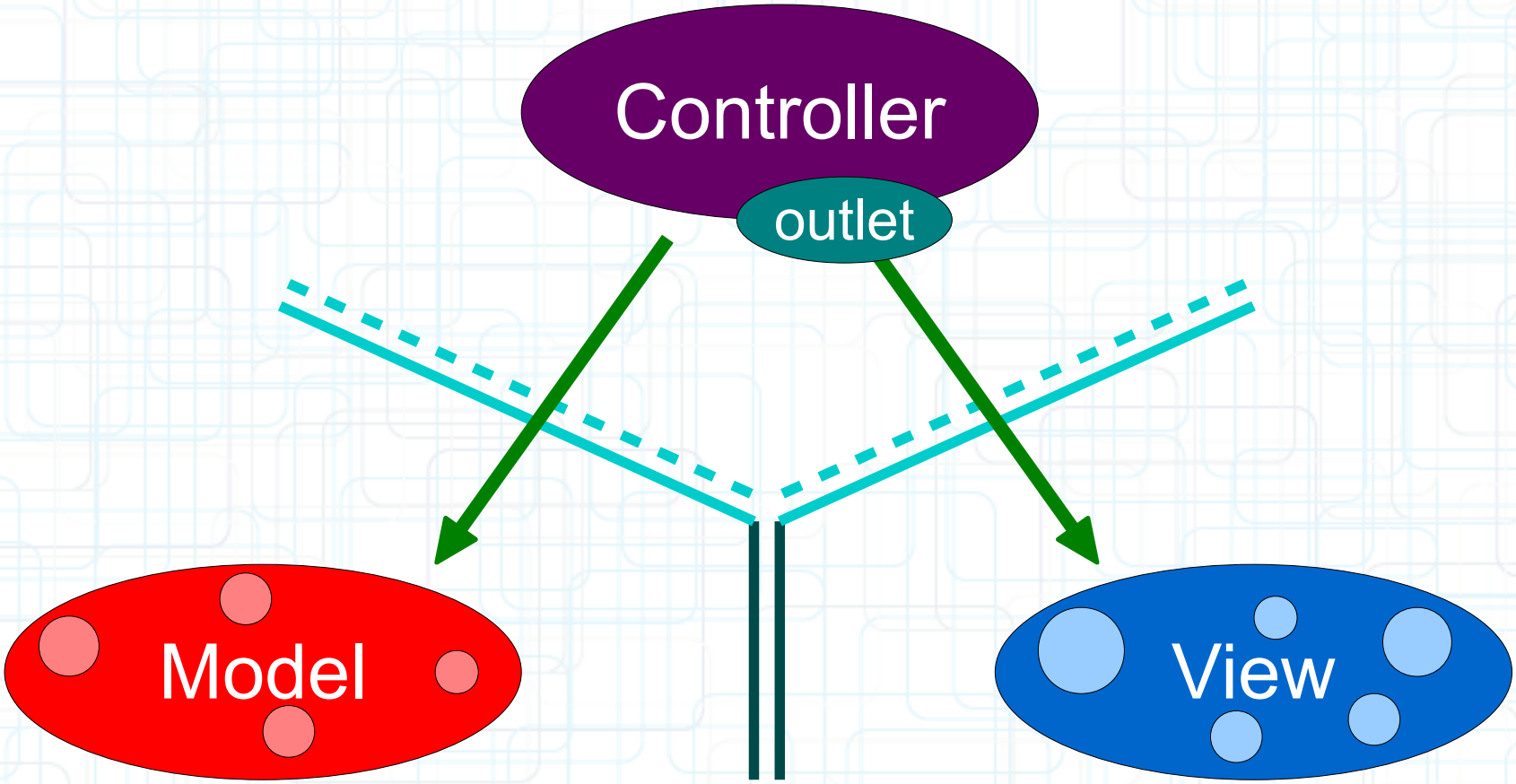
- It's all about managing communication between camps.

MVC Design Model



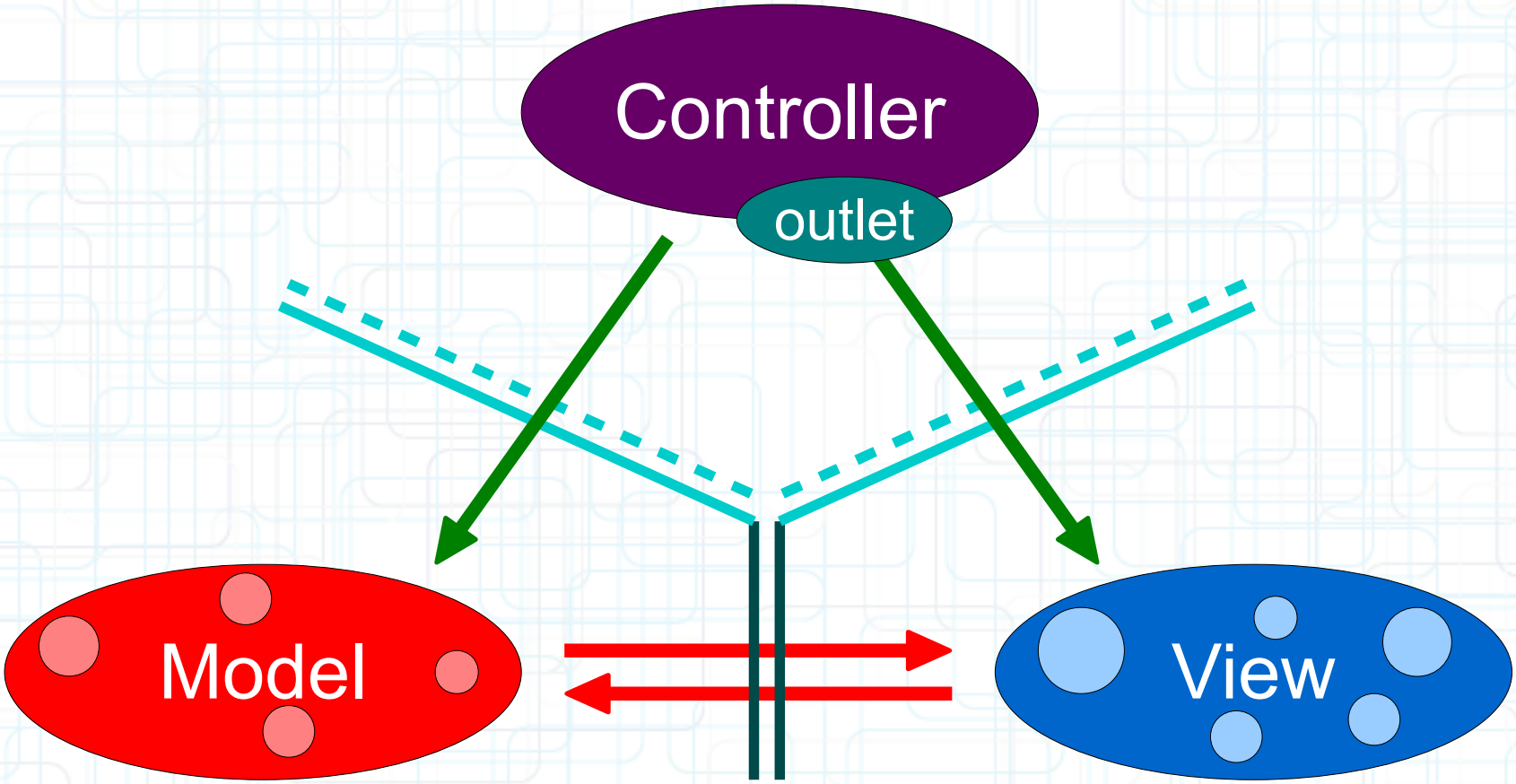
- Controllers can always talk directly to their Model.

MVC Design Model



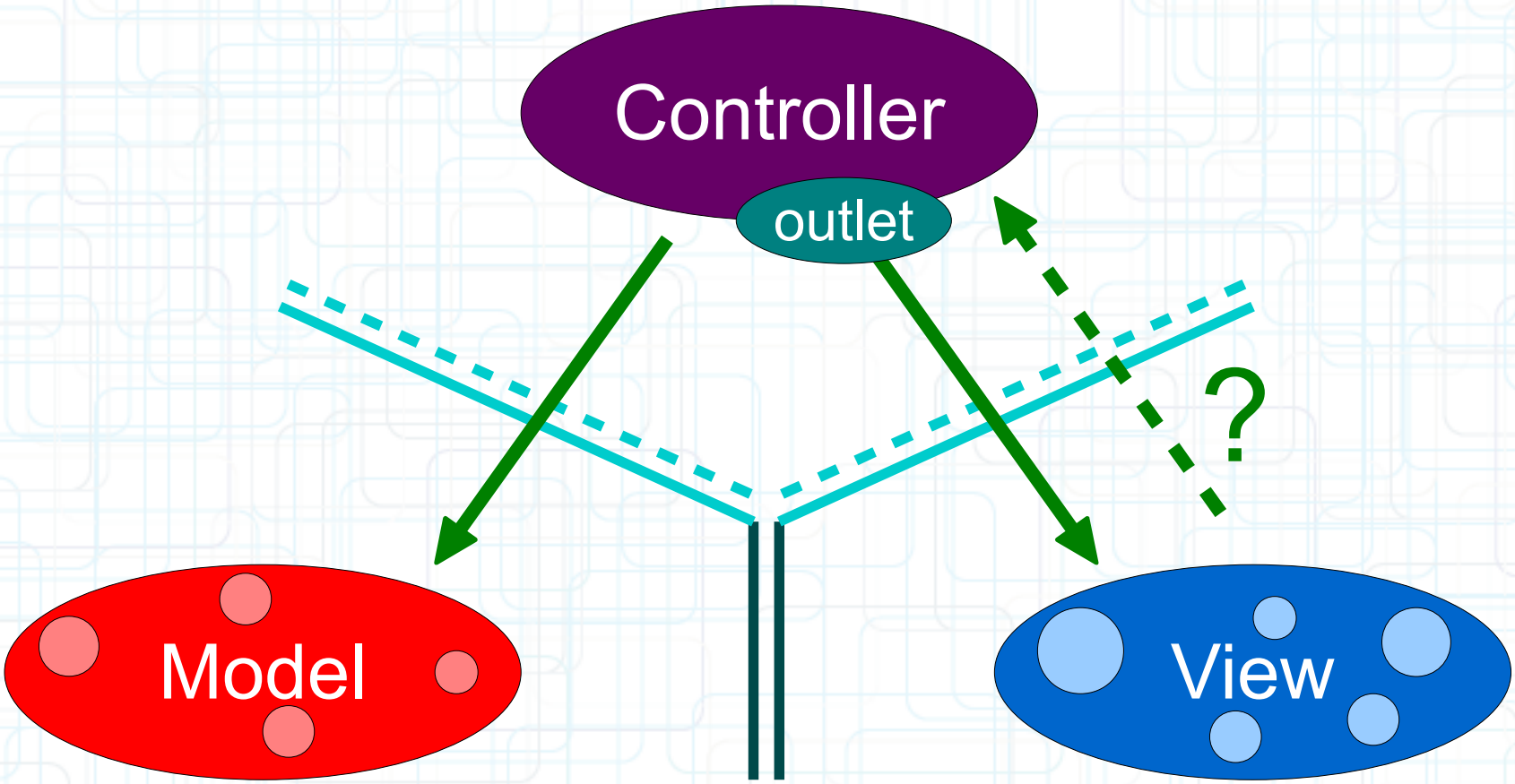
- Controllers can always talk directly to their View.

MVC Design Model



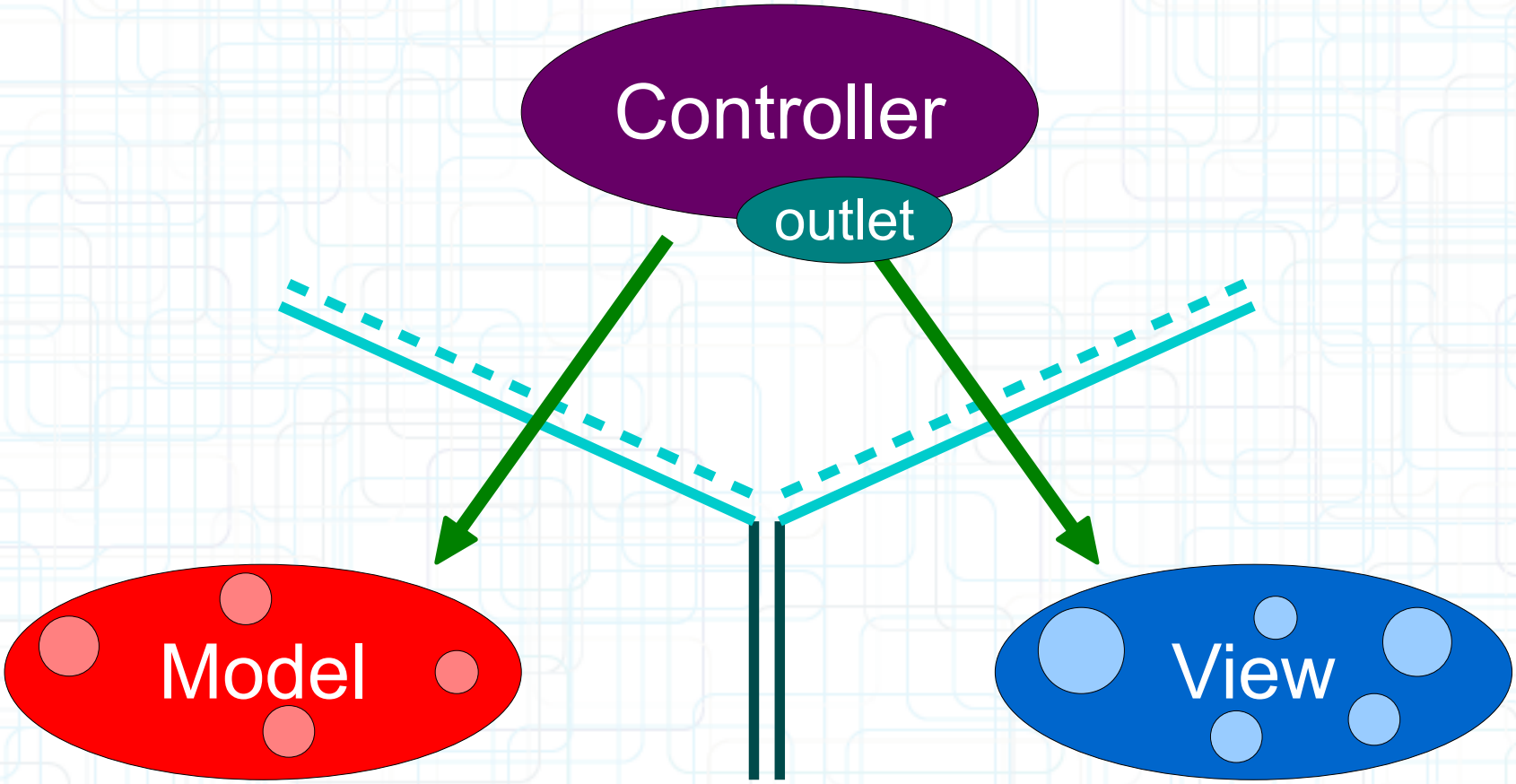
- The Model and View should never speak to each other.

MVC Design Model



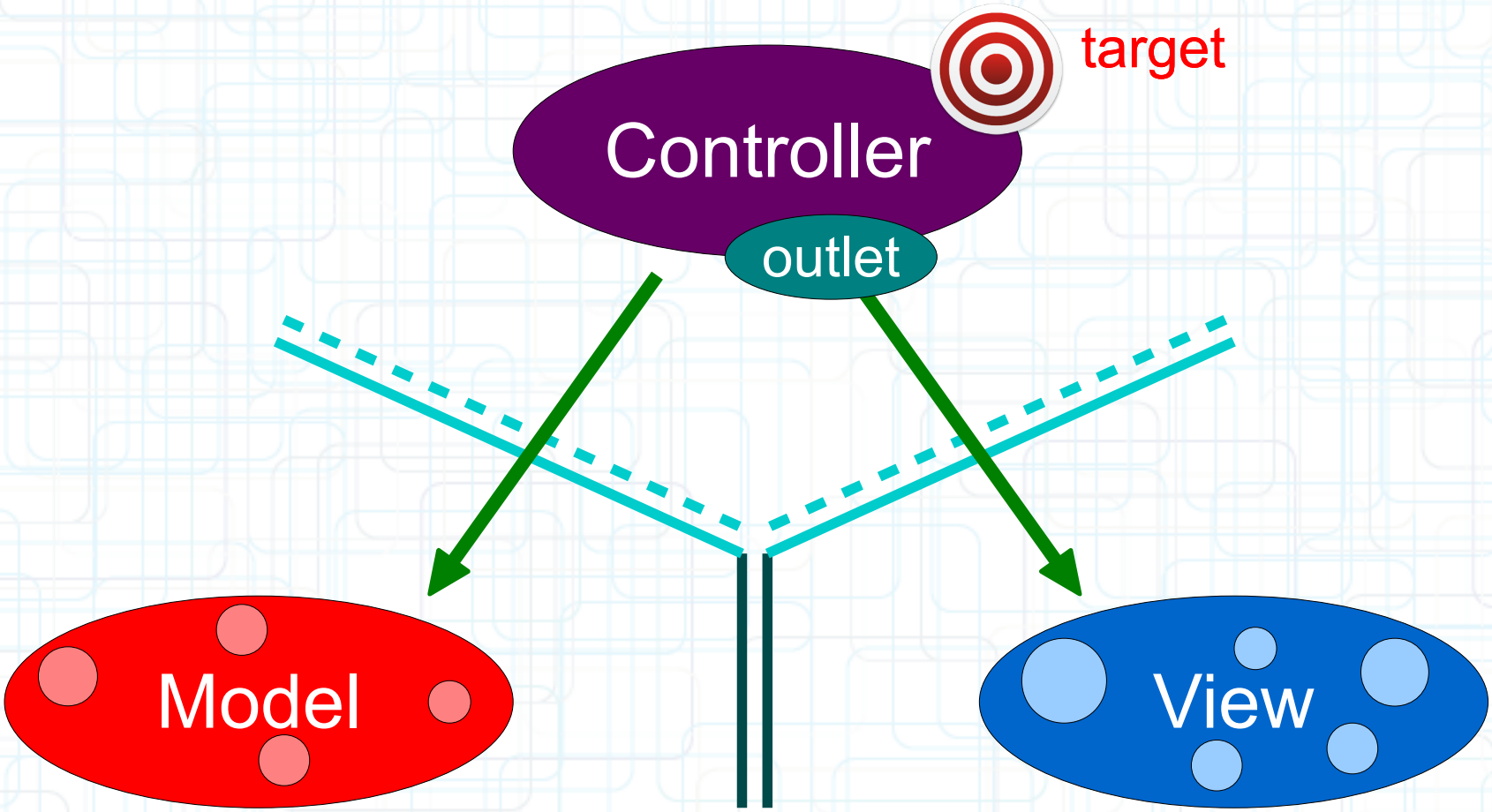
- Can the View speak to its Controller?

MVC Design Model



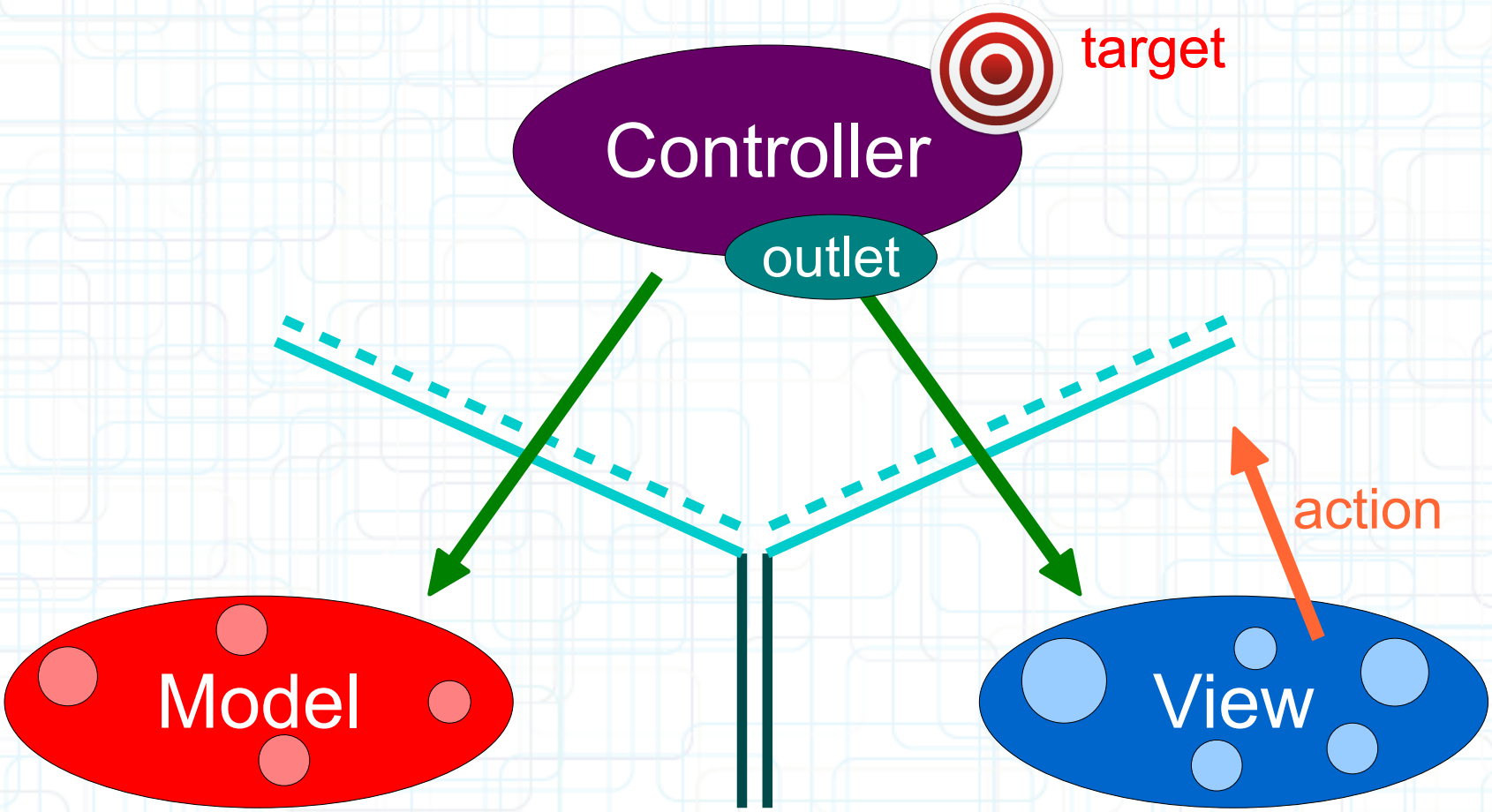
- Sort of. Communication is blind and structured.

MVC Design Model



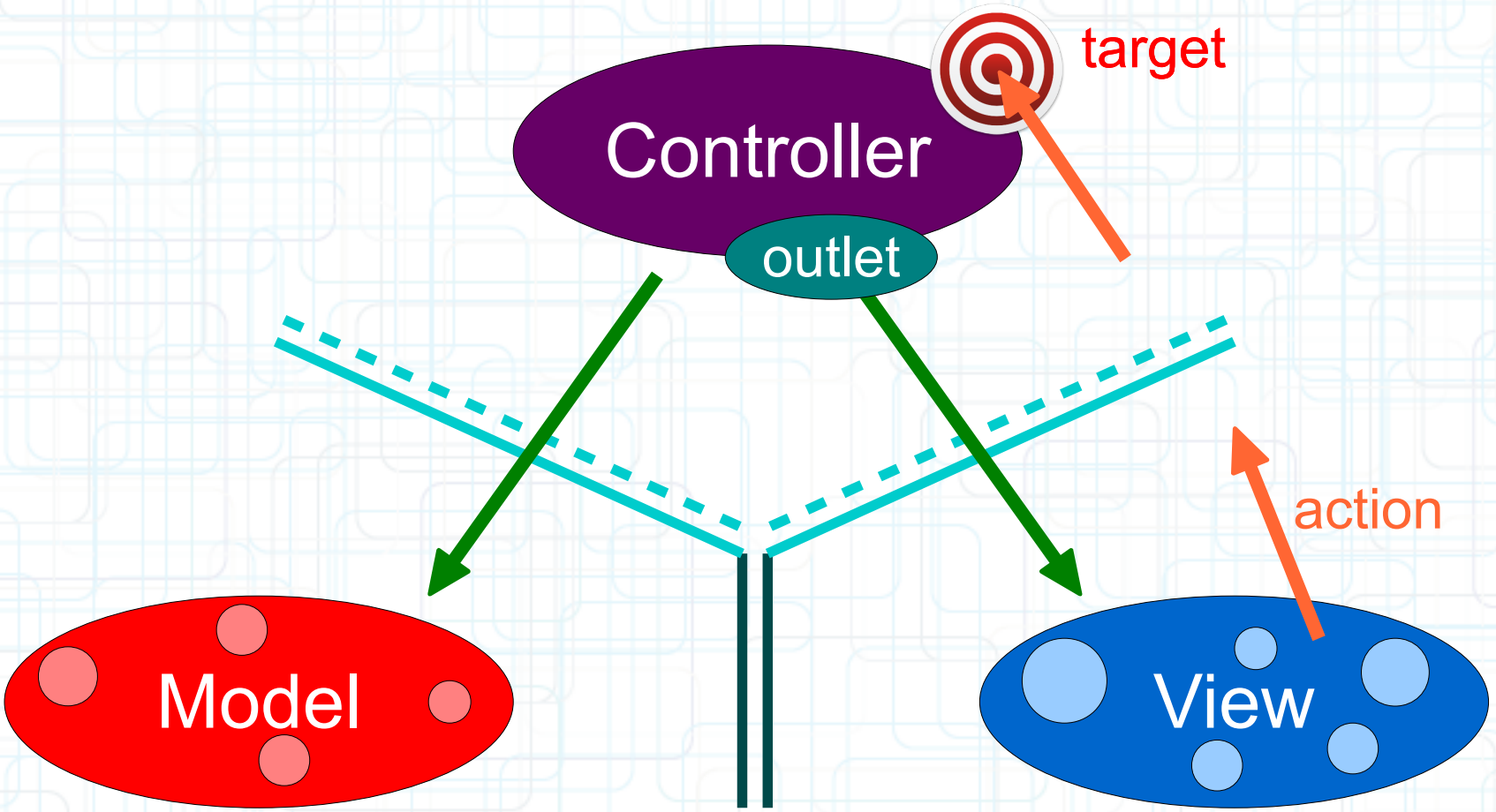
- The Controller can drop a target on itself.

MVC Design Model



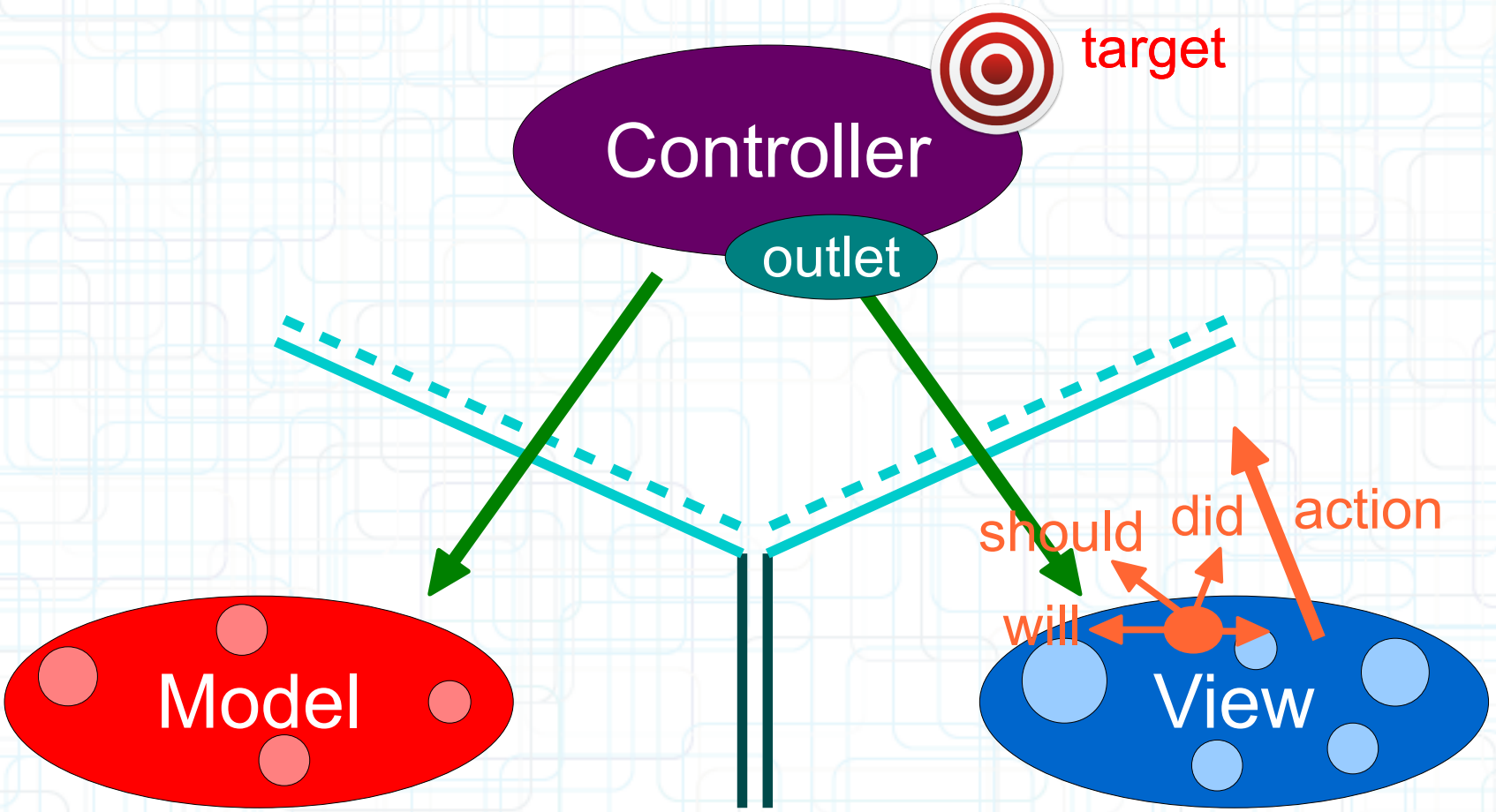
- Then hand out an action to the View.

MVC Design Model



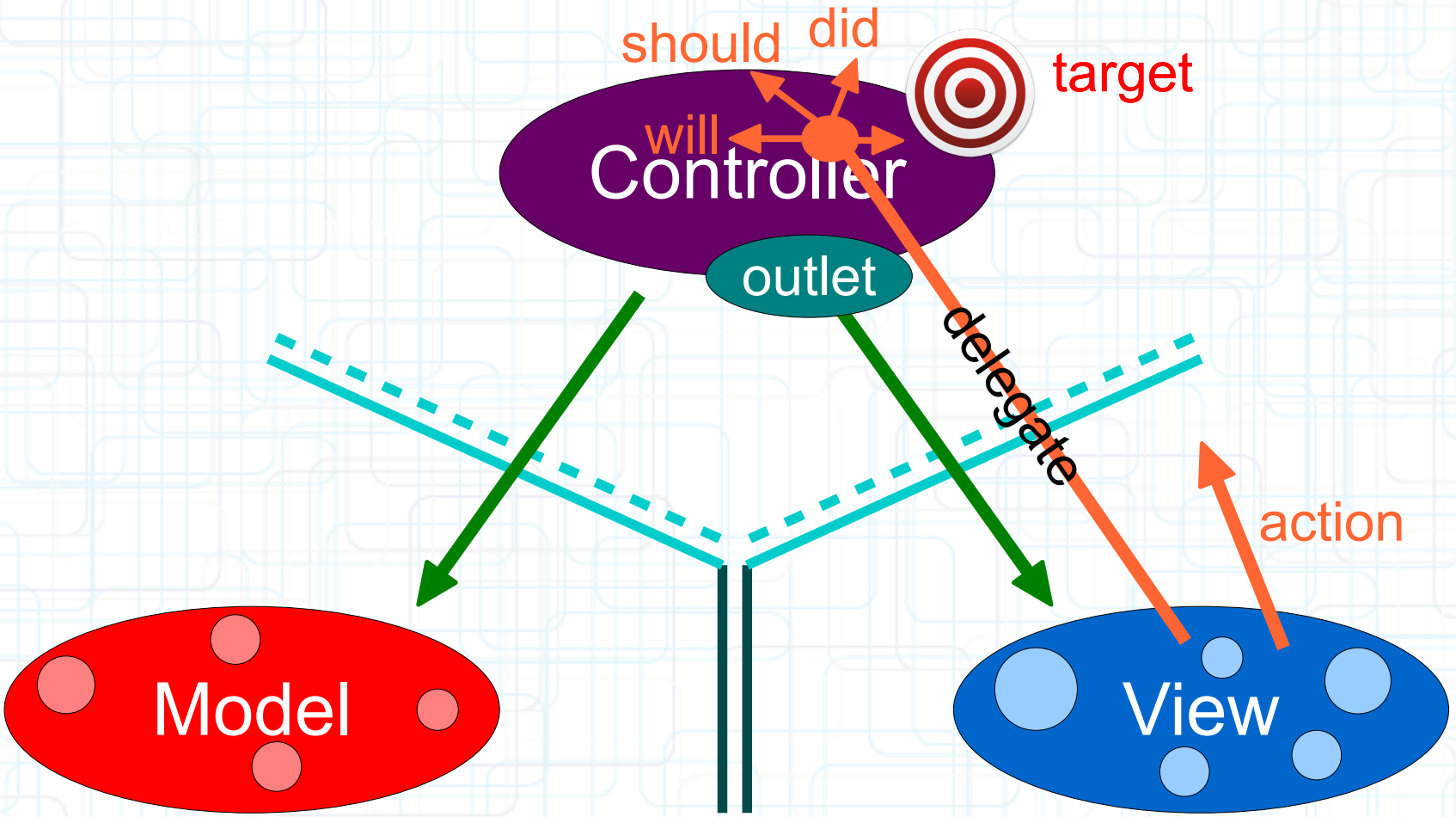
- The View sends the action when things happen in the UI.

MVC Design Model



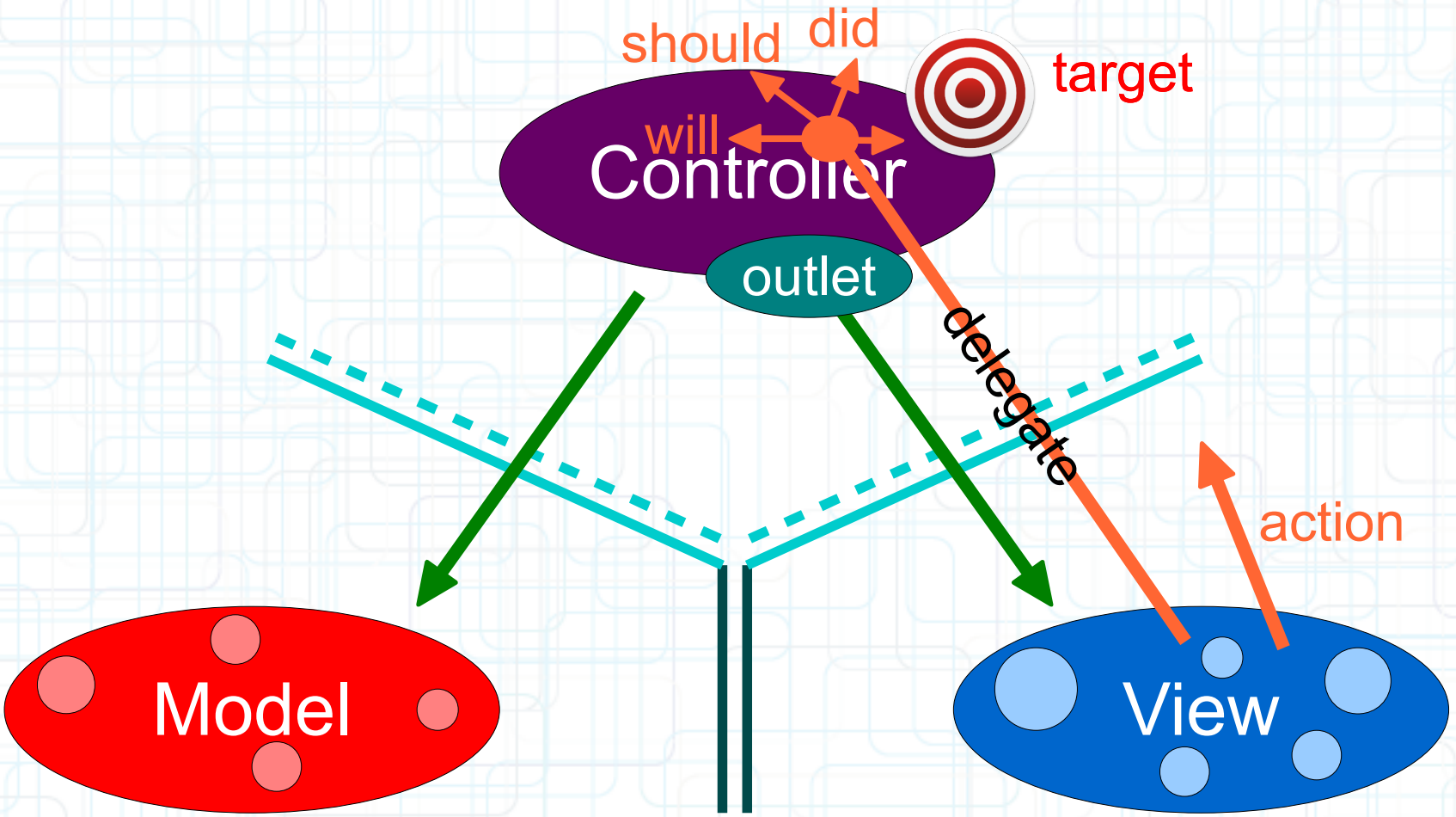
- Sometimes the View needs to synchronize with the Controller.

MVC Design Model



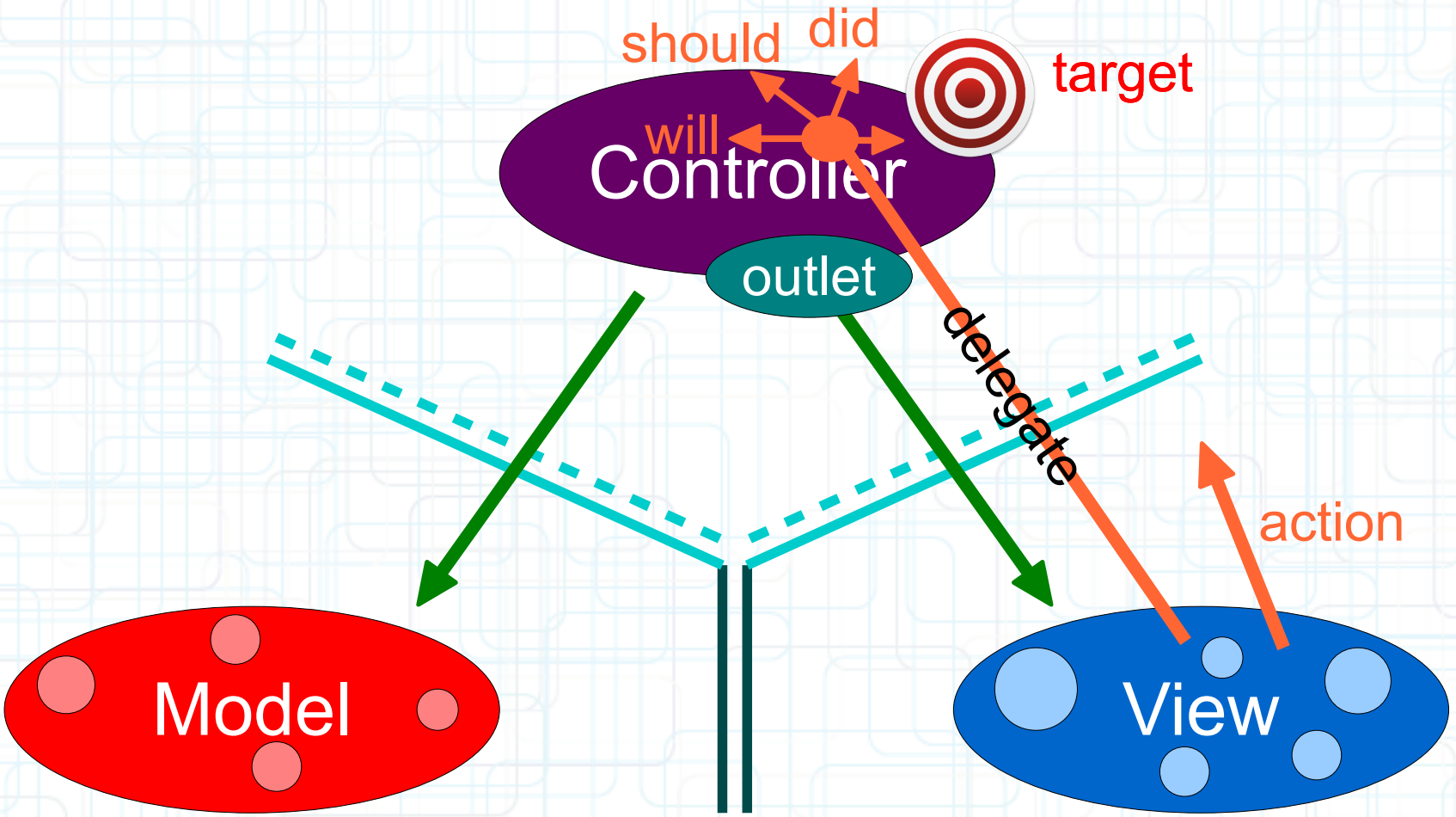
- The Controller sets itself as the View's delegate.

MVC Design Model



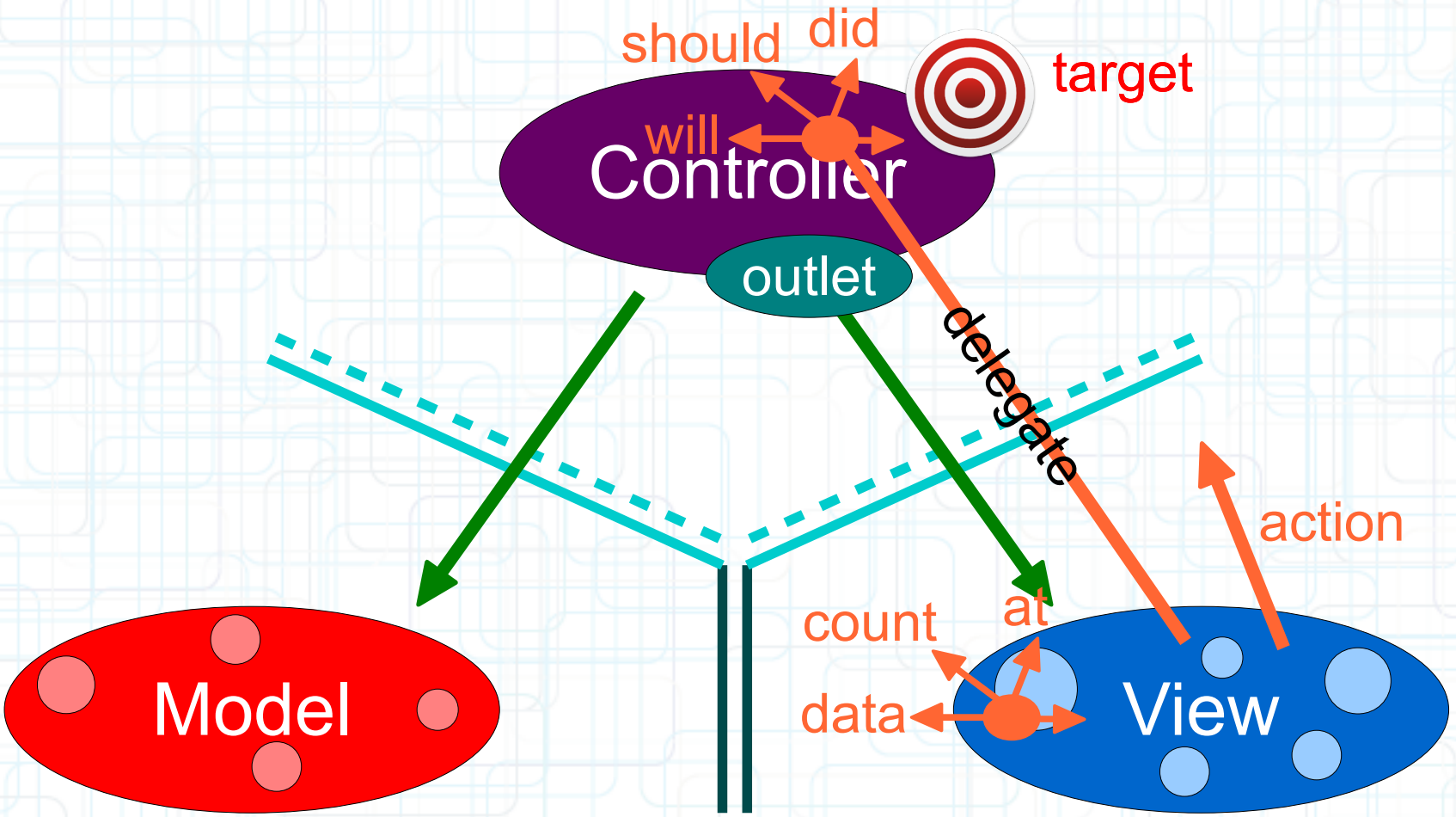
- The delegate is set via a protocol (it's blind to the View class).

MVC Design Model



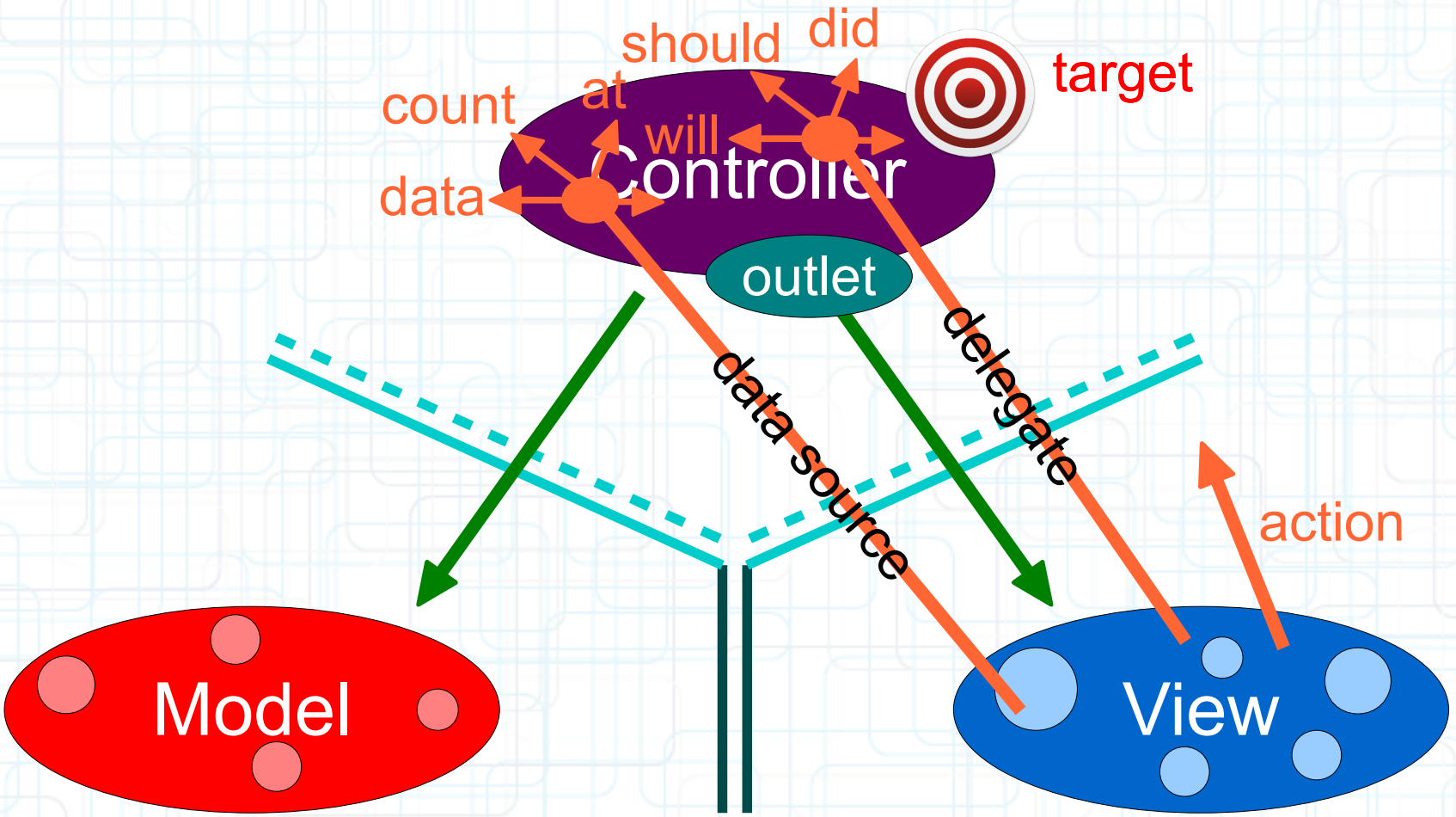
- Views do not own the data they display.

MVC Design Model



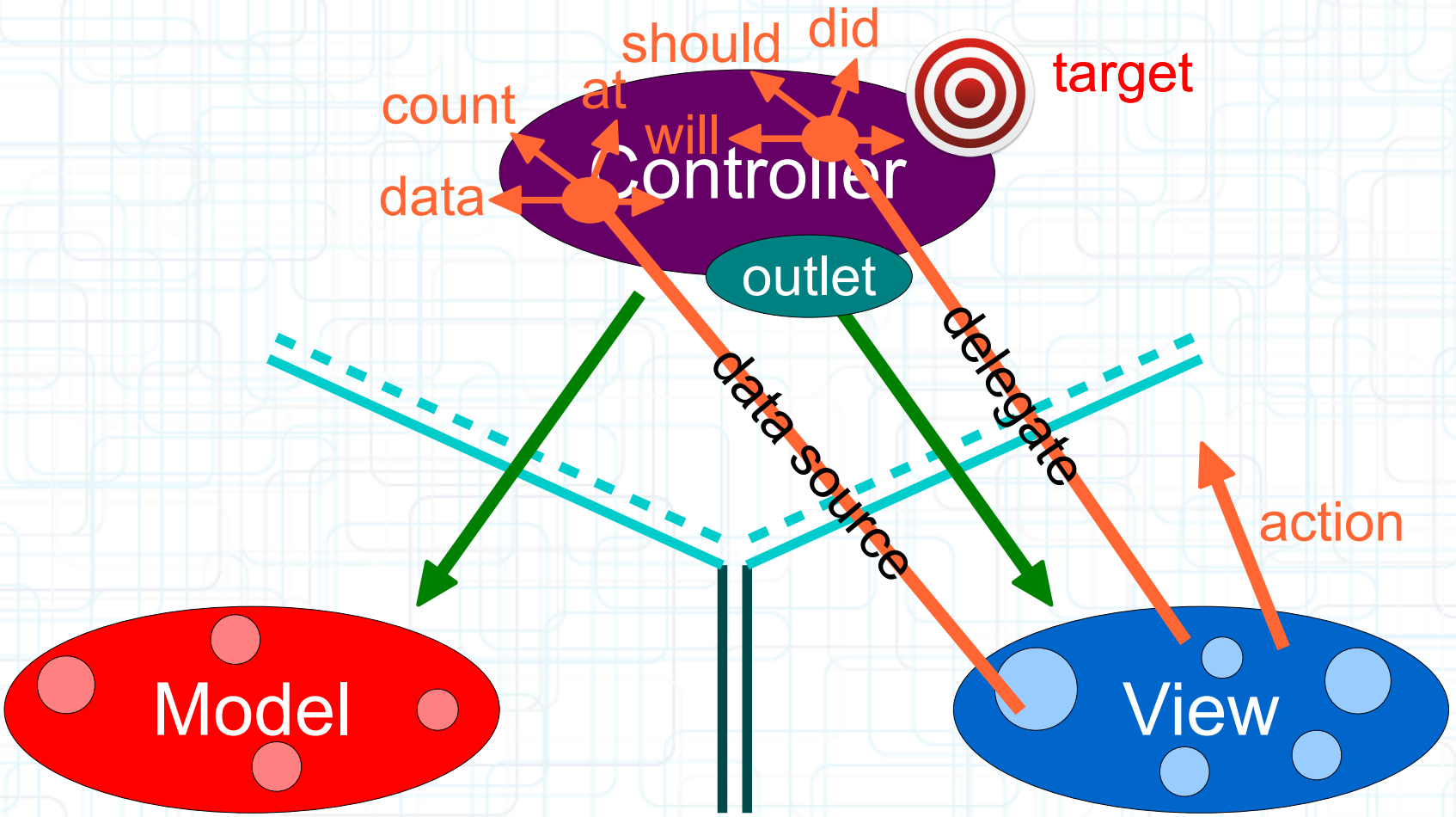
- If needed, they have a protocol to acquire the data.

MVC Design Model



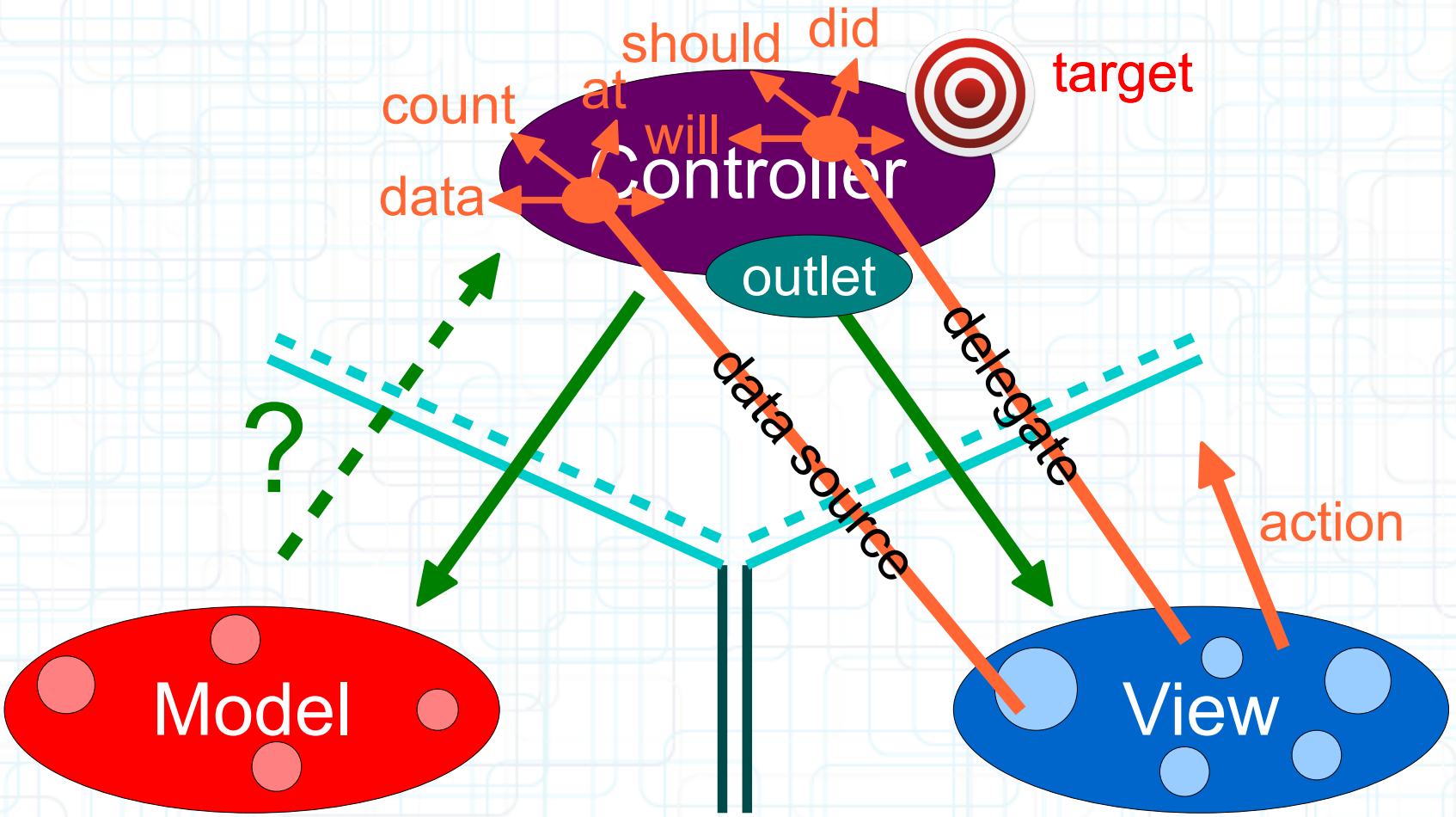
- Controllers are almost always that data source (not the Model).

MVC Design Model



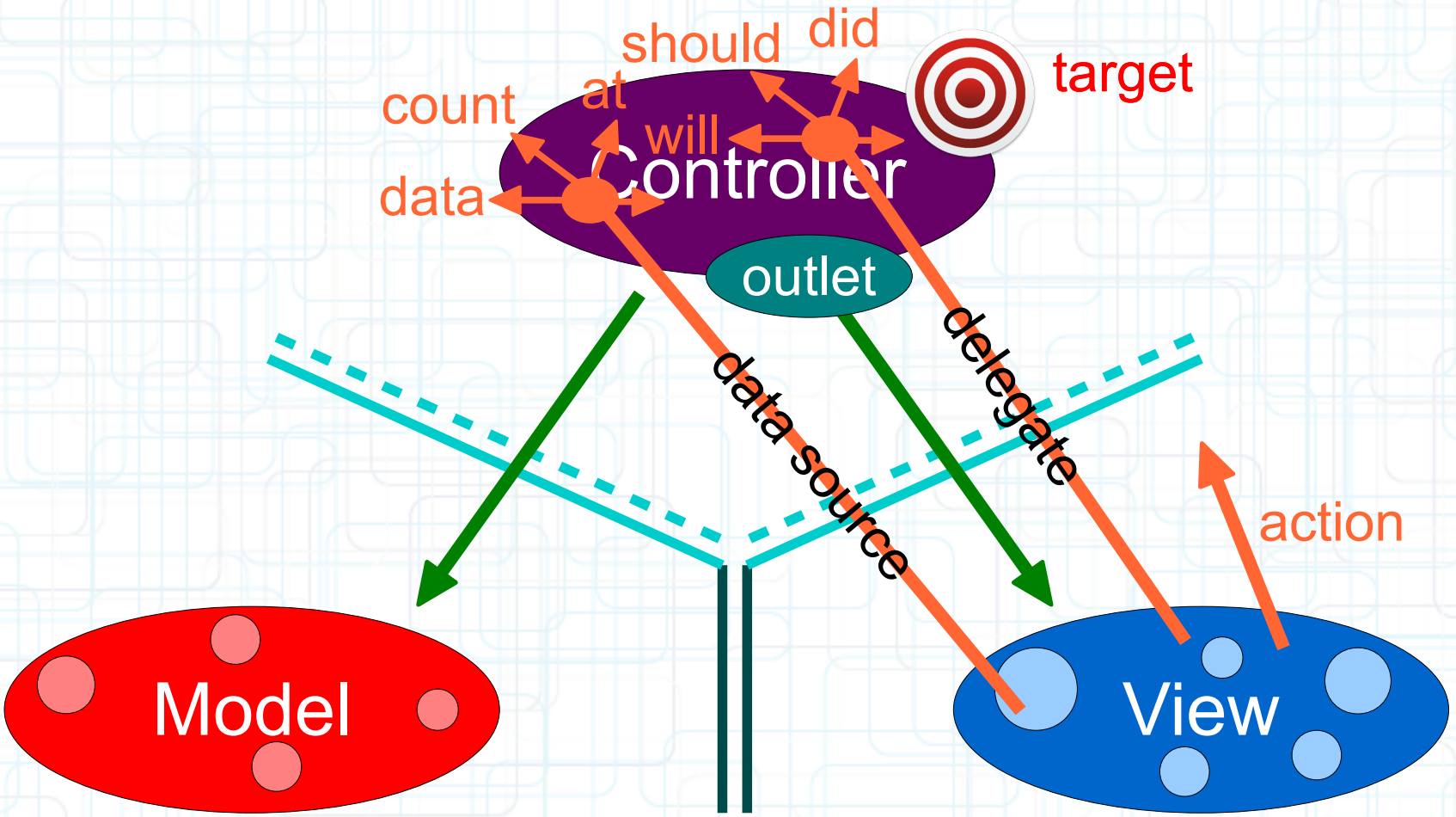
- Controllers interpret/format Model information for the View.

MVC Design Model



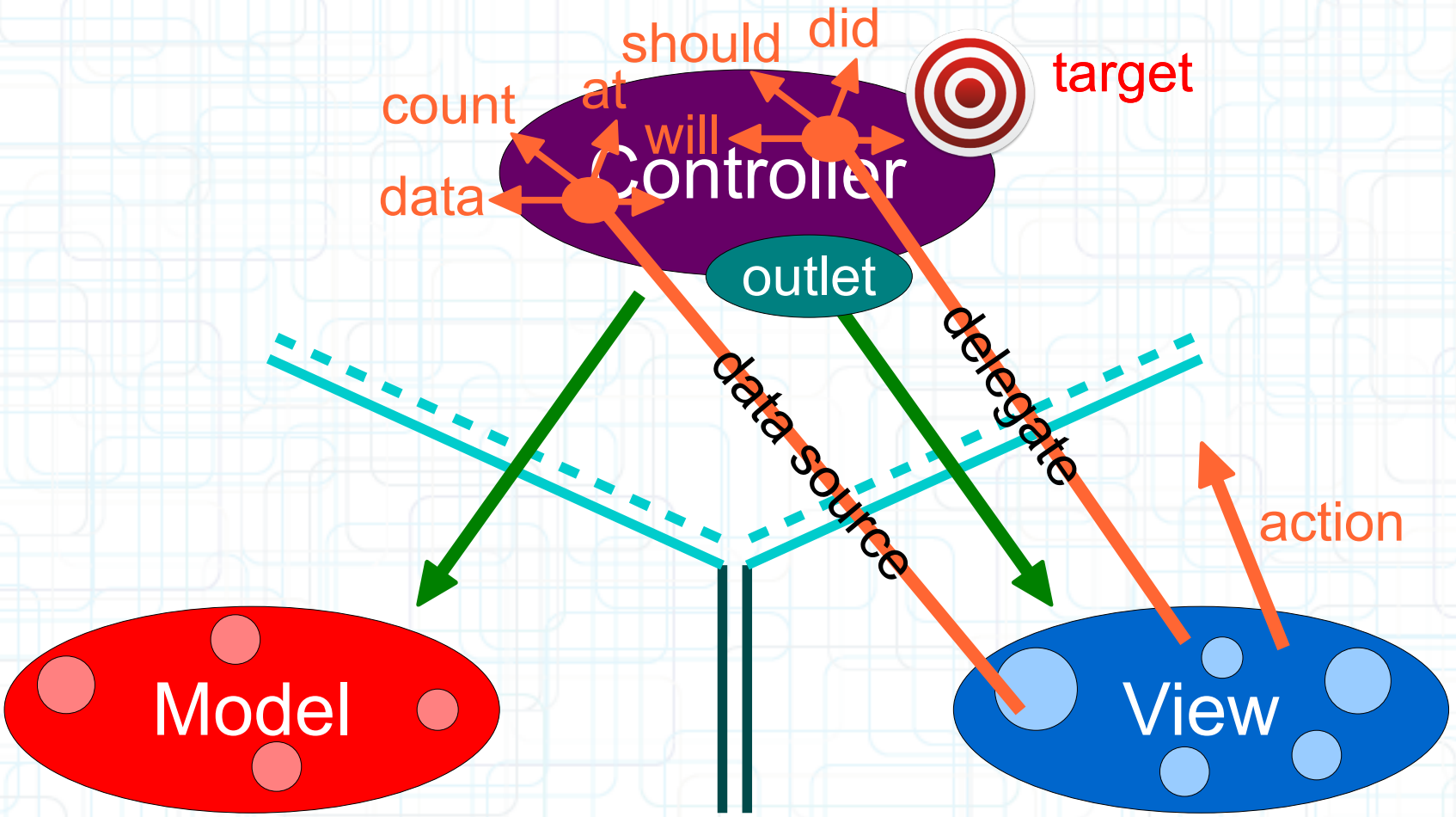
- Can the Model talk directly to the Controller?

MVC Design Model



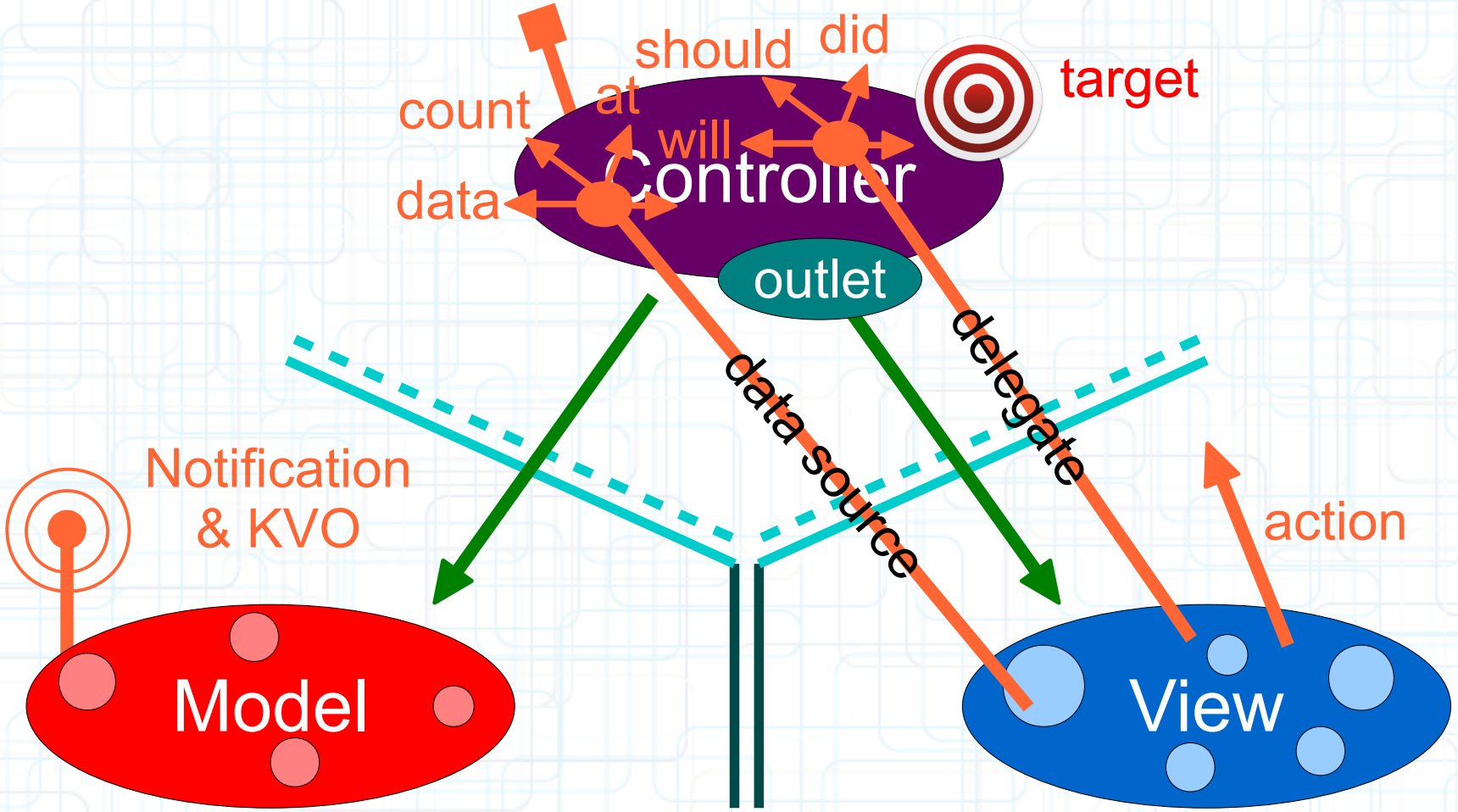
- No. The Model is (should be) UI independent.

MVC Design Model



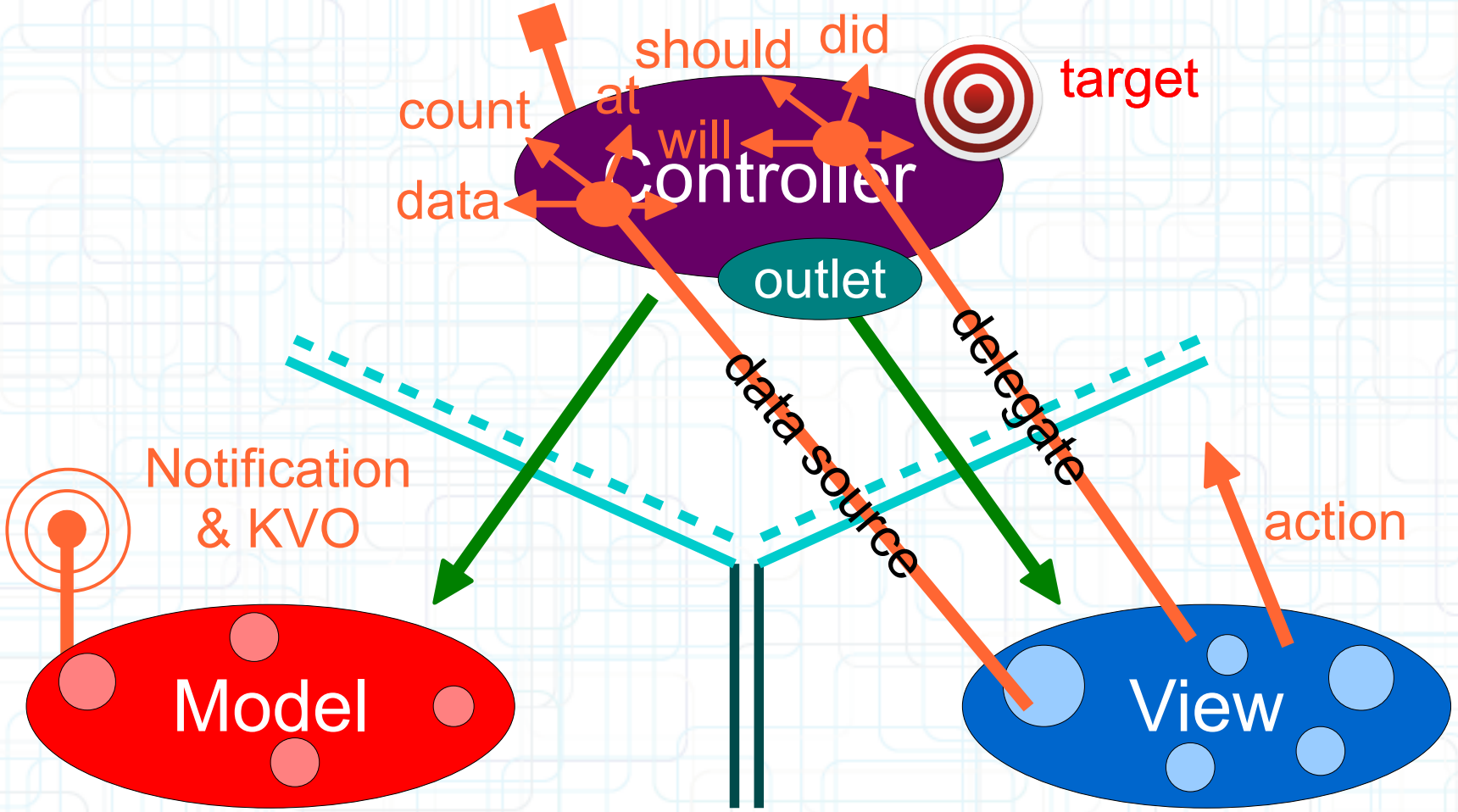
- But what if the Model has information to update or something?

MVC Design Model



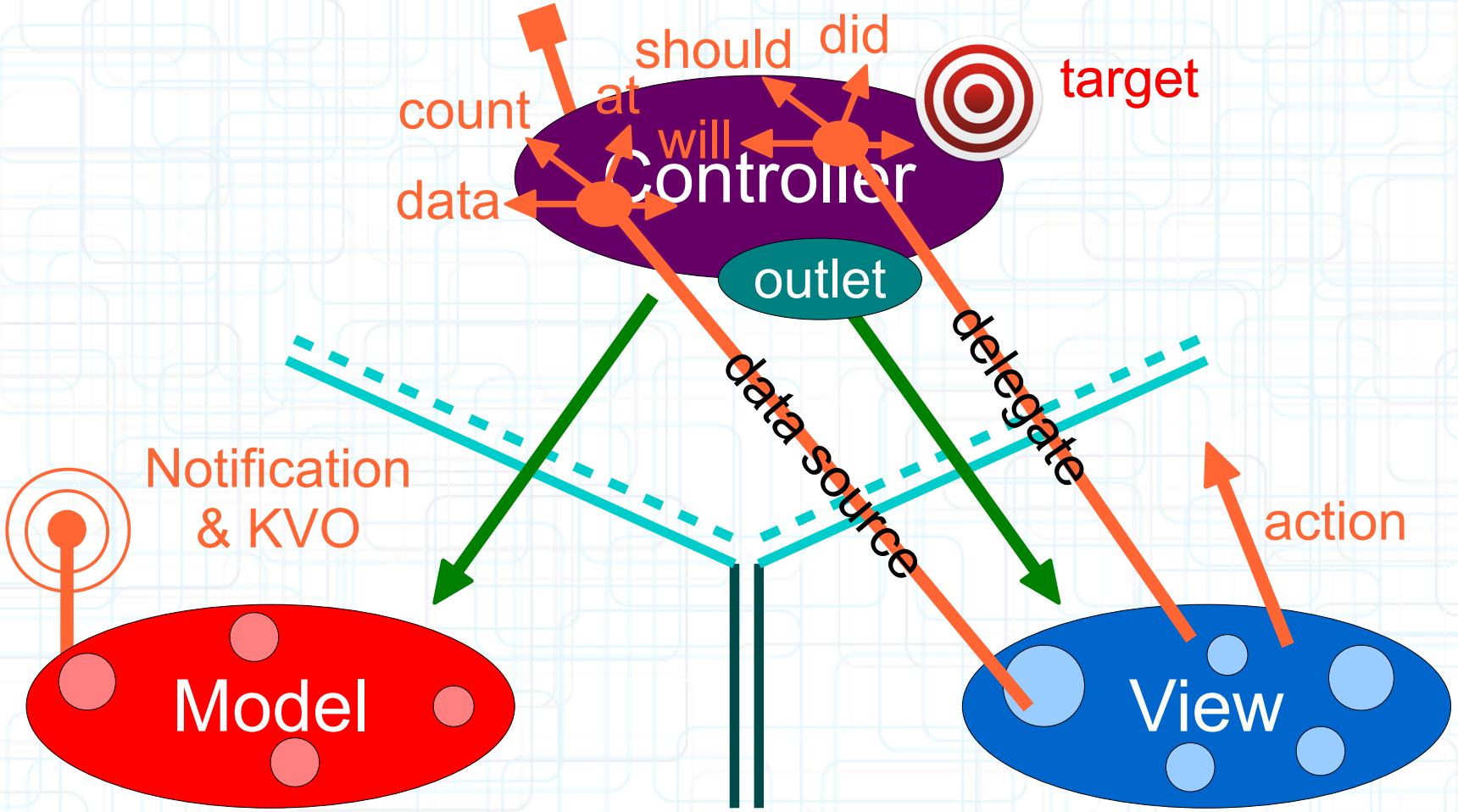
- It uses a “radio station” - broadcast mechanism.

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 Objective-C

- The Objective-C language is a simple computer language designed to enable sophisticated object oriented programming.
- Objective-C extends the standard ANSI C language by providing syntax for defining classes, and methods, as well as other constructs that promote dynamic extension of classes.
- If you are familiar with C and have programmed with object-oriented languages before, you can learn the basic syntax of Objective-C from the following slides.

Introduction to Objective-C

- Many of the traditional object-oriented concepts, such as encapsulation, inheritance, and polymorphism, are all present in Objective-C.

- There are a few important differences that are going to be discussed later.

Introduction to Objective-C

We will talk about:

- Code Organization
- Classes
- Weak Typing vs Strong Typing
- Methods and Messaging
- Properties
- Public and Private Methods

Code Organization

- As with C code, you define header files and source files to separate public declarations from the implementation details of your code.
- Objective-C files use the file extensions listed here:

Extension	Source type
.h	Header files. Header files contain class, type, function, and constant declarations.
.m	Source files. This is the typical extension used for source files and can contain both Objective-C and C code.
.mm	Source files. A source file with this extension can contain C++ code in addition to Objective-C and C code. This extension should be used only if you actually refer to C++ classes or features from your Objective-C code.

Code Organization

- When you want to include header files in your source code, you typically use a `#import` directive.
- This is like `#include`, except that it makes sure that the same file is never included more than once.
- The Objective-C samples and documentation all prefer the use of `#import`, and your own code should too.

Classes in Objective-C

- Classes in Objective-C 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 specification of a class in Objective-C requires two distinct pieces: the interface and the implementation.
- The interface (usually in a `.h` file) contains the class declaration and defines the instance variables and methods associated with the class.
- The implementation (usually in a `.m` file) contains the actual code for the methods of the class.

Classes in Objective-C

- Here is an example where `MyClass` inherits from Cocoa's base class. The class declaration begins with the `@interface` compiler directive.

```
/*  MyClass.h
    Created by Radu Ionescu apple on 10/9/11.
    Copyright 2011 FMI. All rights reserved. */

#import <Foundation/Foundation.h>

@interface MyClass : NSObject
{
    // Member variable declarations go here:
    int count;
    id data;
    NSString *name;
}

// Method declarations go here:
- (id)initWithName:(NSString *)aName;
+ (MyClass *)createMyClassWithName:(NSString *)aName;

@end
```

Weak Typing vs Strong Typing

- Objective-C supports both strong and weak typing for variables containing objects.
- Strongly typed variables include the class name in the variable type declaration.
- Weakly typed variables use the type `id` for the object instead. Weakly typed variables are used frequently for things such as collection classes, where the exact type of the objects in a collection may be unknown.
- Weakly typed variables provide tremendous flexibility and allow for much greater dynamism in Objective-C programs.

Weak Typing vs Strong Typing

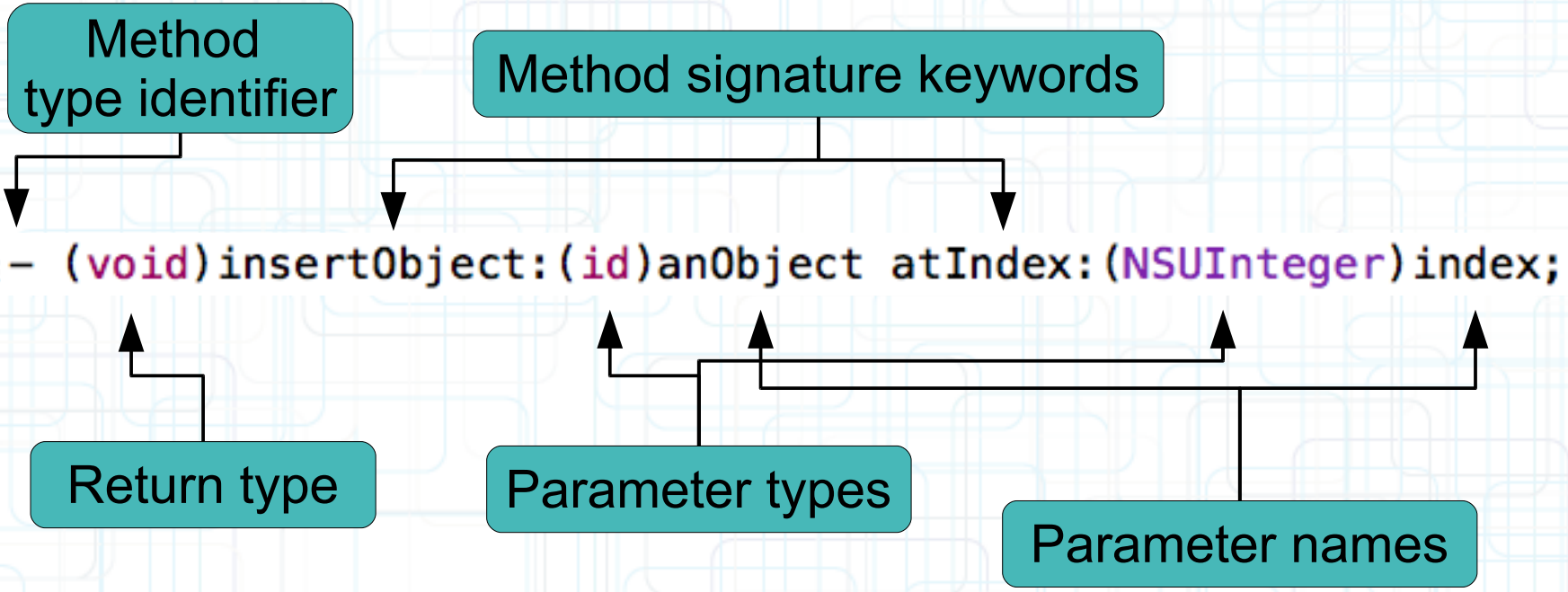
- The following example shows strongly and weakly typed variable declarations:

```
MyClass *myObject1;  // Strong typing
id       myObject2;  // Weak typing
```

- In Objective-C, object references are pointers. The `id` type implies a pointer.

Methods and Messaging

- A class in Objective-C can declare two types of methods: instance methods and class methods.
- The declaration of a method consists of the method type identifier, a return type, one or more signature keywords, and the parameter type and name information:



Methods and Messaging

- The declaration preceded by a minus (`-`) sign indicates that this is an instance method.
- The method's actual name is a concatenation of all of the signature keywords, including colon characters:

```
insertObject:atIndex:
```

- When you want to call a method, you do so by **messaging** an object.
- A message is the method signature, along with the parameter information the method needs.
- All messages you send to an object are dispatched dynamically, thus facilitating the polymorphic behavior of Objective-C classes.

Methods and Messaging

- To send the `insertObject:atIndex:` message to an object in the `myArray` variable, you would use the following syntax:

```
[myArray insertObject:anObject atIndex:0];
```

- Objective-C lets you nest messages. Thus, if you had another object called `myAppObject` that had methods for accessing the array object and the object to insert into the array, you could rewrite the preceding example as:

```
[[myAppObject theArray]
   insertObject:[myAppObject someObject]
         atIndex:0];
```

Dot Syntax

- Objective-C also provides a dot syntax for invoking **accessor methods**. Accessor methods get and set the state of an object, and typically take the form:

```
-(type)propertyName
-(void)setPropertyName:(type)
```

- Using dot syntax, you could rewrite the previous example as:

```
[myAppObject.theArray
    insertObject:myAppObject.someObject
         atIndex:0];
```

- You can also use dot syntax for assignment:

```
myAppObject.theArray = aNewArray;
```

Methods and Messaging

- Now we can add the `MyClass` implementation:

```
#import "MyClass.h"

@implementation MyClass

- (id)initWithName:(NSString *)aName
{
    self = [super init];
    if (self)
    {
        name = [aName copy];
    }
    return self;
}

+ (MyClass *)createMyClassWithName: (NSString *)aName
{
    return [[[self alloc] initWithName:aName] autorelease];
}

@end
```

Plane.h

Objective-C Example

```
#import “Vehicle.h”

@interface Plane : Vehicle

@end
```

Superclass header file.
This is often <UIKit/UIKit.h>

Class name

Superclass

Plane.h

Objective-C Example

```
#import “Vehicle.h”

@interface Plane : Vehicle

// declaration of public methods




@end
```

Plane.m

Objective-C Example

```
#import “Plane.h”

@implementation Plane

@end
```

Import our own header file.

Note, superclass **not** specified here.

Plane.m

Objective-C Example

```
#import “Plane.h”




@implementation Plane
//implementation of public and private methods









@end
```

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)


@end

@implementation Plane
//implementation of public and private methods









@end
```

The () are mandatory.

No superclass here either.

Plane.h

Objective-C Example

```
#import “Vehicle.h”
#import “Airport.h”

@interface Plane : Vehicle

// declaration of public methods


- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km;


@end
```

We need to import Airport.h for method declaration below to work.

The full name of this method is `flyToAirport:atAltitude:`

It doesn't return any value.

Lining up the colons makes things look nice.

It takes two arguments. Note how each is preceded by its own keyword.

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)


@end

@implementation Plane
//implementation of public and private methods









- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km
{
   //put the code to land the plane here



}

@end
```

No semicolon here.

Plane.h

Objective-C Example

```
#import “Vehicle.h”
#import “Airport.h”

@interface Plane : Vehicle

// declaration of public methods




- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km;
```

Now let's add the possibility to set/get the plane's speed.

```
@end
```

Plane.h

Objective-C Example

```
#import “Vehicle.h”
#import “Airport.h”

@interface Plane : Vehicle

// declaration of public methods




- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km;

- (void)setCruiseSpeed:(double)aSpeed;
- (double)cruiseSpeed;









@end
```

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods



- (void)setCruiseSpeed:(double)aSpeed
{
    ???
}

- (double)cruiseSpeed
{
    ???
}

- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

How do we implement these methods?

Plane.h

Objective-C Example

```
#import “Vehicle.h”
#import “Airport.h”

@interface Plane : Vehicle

// declaration of public methods
```

We have to declare something to hold the speed value.

```
- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km;

- (void)setCruiseSpeed:(double)aSpeed;
- (double)cruiseSpeed;

@end
```

Plane.h

Objective-C Example

```
#import “Vehicle.h”
#import “Airport.h”

@interface Plane : Vehicle

// declaration of public methods

@property (nonatomic) double cruiseSpeed;



- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km;

- (void)setCruiseSpeed:(double)aSpeed;
- (double)cruiseSpeed;



@end
```

This property essentially declares the two “cruiseSpeed” methods below.

nonatomic means its setter and getter are not thread-safe. That’s no problem if this is UI code because all UI code happens on the main thread of the application.

Plane.h

Objective-C Example

```
#import “Vehicle.h”
#import “Airport.h”

@interface Plane : Vehicle

// declaration of public methods

@property (nonatomic) double cruiseSpeed;



- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km;
```

We never declare both the `@property` and its setter and getter in the header file (just the `@property`).

```
@end
```

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods



- (void)setCruiseSpeed:(double)aSpeed
{
    ???
}

- (double)cruiseSpeed
{
    ???
}

- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

But how do we implement the accessors and how do we store the value?

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;

- (void)setCruiseSpeed:(double)aSpeed
{
    ???
}

- (double)cruiseSpeed
{
    ???
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

We almost always use `@synthesize` to create the implementation of the setter and getter for a `@property`. It both creates the setter and getter methods AND creates some storage to hold the value.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;

- (void)setCruiseSpeed:(double)aSpeed
{
    ???
}

- (double)cruiseSpeed
{
    ???
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

This is the name of the storage location to use.

We almost always use `@synthesize` to create the implementation of the setter and getter for a `@property`. It both creates the setter and getter methods AND creates some storage to hold the value.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;

- (void)setCruiseSpeed:(double)aSpeed
{
    ???
}

- (double)cruiseSpeed
{
    ???
}

- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

This is the name of the storage location to use.

If we don't use "=" here, @synthesize uses the name of the property (which is not recommended).

"_" then the name of the property is a common naming convention.

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;

- (void)setCruiseSpeed:(double)aSpeed
{
    _cruiseSpeed = aSpeed;
}

- (double)cruiseSpeed
{
    return _cruiseSpeed;
}

- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

This is what the methods created by `@synthesize` would look like.

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
```

Most of the time, you can let @synthesize do all the work of creating setters and getters.

```
- (void)flyToAirport:(Airport *)destination
          atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)

@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;


- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}


- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

However, we can create our own if there is any special work to do when setting or getting.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;


- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}




- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

Here is another @property. This one is private (because it's in our .m file).

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;


- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}


- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

All objects are always allocated on the heap. So we **always** access them through a pointer.

It’s a pointer to an object (of class Airport). It’s strong which means that the memory used by this object will stay around for as long as we need it.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

This creates the setter and getter for our new @property.

@synthesize does NOT create storage for the object this pointer points to. It just allocates room for the pointer.

We'll talk about how to allocate and initialize the objects themselves later.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
}

@end
```

Now let's take a look at some example coding. This is just to get a feel for Objective-C syntax.

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
   if (destination == [self nearestAirport])


}

@end
```

The “square brackets” syntax is used to send messages.

We are calling the `nearestAirport`'s getter on ourself here.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
   if (destination == [self nearestAirport])
      [[self nearestAirport] landPlane:self
                           fromAltitude:km];

}

@end
```

Square brackets inside square brackets.

Here's another example of sending a message that has 2 arguments. It is being sent to an instance of Airport.

Plane.m

Objective-C Example

```
#import "Plane.h"

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
   if (destination == [self nearestAirport])
      [self.nearestAirport  landPlane:self
                         fromAltitude:km];

}

@end
```

This is identical to `[self nearestAirport]`.

Calling getters and setters is such an important task, it has its own syntax: dot notation.

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
   if (destination == self.nearestAirport)
      [self.nearestAirport  landPlane:self
                          fromAltitude:km];

}

@end
```

We can use dot notation here too.

Plane.m

Objective-C Example

```
#import “Plane.h”

@interface Plane()
//declaration of private methods (as needed)
@property (nonatomic, strong) Airport *nearestAirport;
@end

@implementation Plane
//implementation of public and private methods

@synthesize cruiseSpeed = _cruiseSpeed;
@synthesize nearestAirport = _nearestAirport;

- (void)setCruiseSpeed:(double)aSpeed
{
    if (aSpeed > 0) _cruiseSpeed = aSpeed;
}

- (void)flyToAirport:(Airport *)destination
         atAltitude:(double)km
{
   //put the code to land the plane here
   if (destination == self.nearestAirport)
      [self.nearestAirport  landPlane:self
                           fromAltitude:km];
   //continue flight to destination
}

@end
```

Next Time

Objective-C in Depth:

- More on Dot Notation
- Instance Methods and Class Methods
- Object Typing
- Introspection
- Foundation Framework