

Stanford CS193p

Developing Applications for iOS
Fall 2011



Today

- ⦿ What is this class all about?

- Description

- Prerequisites

- Homework / Final Project

- ⦿ iOS Overview

- What's in iOS?

- ⦿ MVC

- Object-Oriented Design Concept

- ⦿ Objective C

- New language!

- Basic concepts only for today.

What will I learn in this course?

⦿ How to build cool apps

Easy to build even very complex applications

Result lives in your pocket!

Very easy to distribute your application through the AppStore

Vibrant development community

⦿ Real-life Object-Oriented Programming

The heart of Cocoa Touch is 100% object-oriented

Application of MVC design model

Many computer science concepts applied in a commercial development platform:

Databases, Graphics, Multimedia, Multithreading, Animation, Networking, and much, much more!

Numerous students have gone on to sell products on the AppStore

Prerequisites

⦿ Most Important Prereq!

Object-Oriented Programming
CS106A&B required, CS107 recommended

⦿ Object-Oriented Terms

Class (description/template for an object)
Instance (manifestation of a class)
Message (sent to object to make it act)
Method (code invoked by a Message)
Instance Variable (object-specific storage)
Superclass/Subclass (Inheritance)
Protocol (non-class-specific methods)

⦿ You should know these terms!

If you are not very comfortable with all of these, this might not be the class for you

⦿ Programming Experience

This is an upper-level CS course.
If you have never written a program where you had to design and implement more than a handful of classes, this will be a big step up in difficulty for you.

Assignments

• Weekly Homework

7 weekly assignments

Assigned Thursday after lecture

Due the following Wednesday at 11:59pm

Individual work only

Homework graded ✓, ✓+ and ✓- based on
Required Tasks and Evaluation criteria

Lots of extra credit available, bank it

Only 3 “free” late days per quarter

#1 fail: falling behind on homework

• Final Project

3 weeks to work on it

But weighted like 4 weeks of homework

Proposal requires instructor approval

Some teams of 2 might be allowed

Keynote presentation required (3 mins or so)



Core OS

OSX Kernel	Power Management
Mach 3.0	Keychain Access
BSD	Certificates
Sockets	File System
Security	Bonjour



Core Services

Collections	Core Location
Address Book	Net Services
Networking	Threading
File Access	Preferences
SQLite	URL Utilities



Media

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



Cocoa Touch

Multi-Touch

Core Motion

View Hierarchy

Localization

Controls

Alerts

Web View

Map Kit

Image Picker

Camera

Platform Components

- Tools



Xcode 4



Instruments

- Language

[display setTextColor:[UIColor blackColor]];

- Frameworks



Foundation

Core Data



UIKit

Core Motion

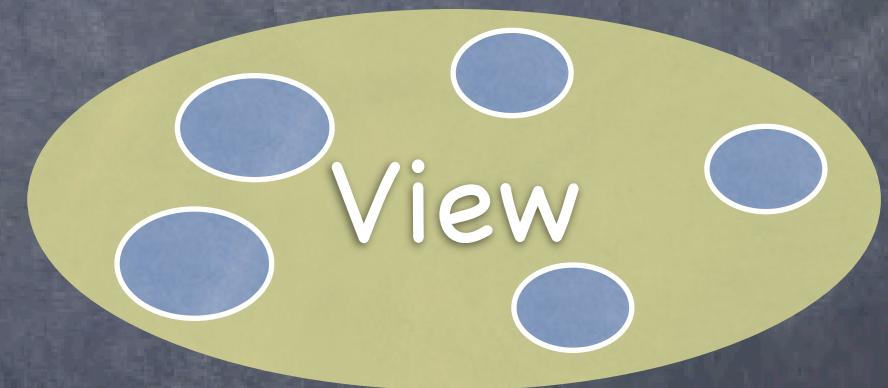
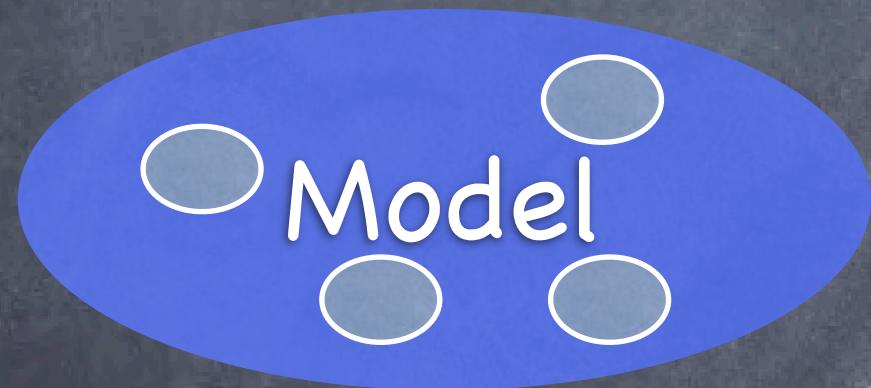
Map Kit

- Design Strategies

MVC

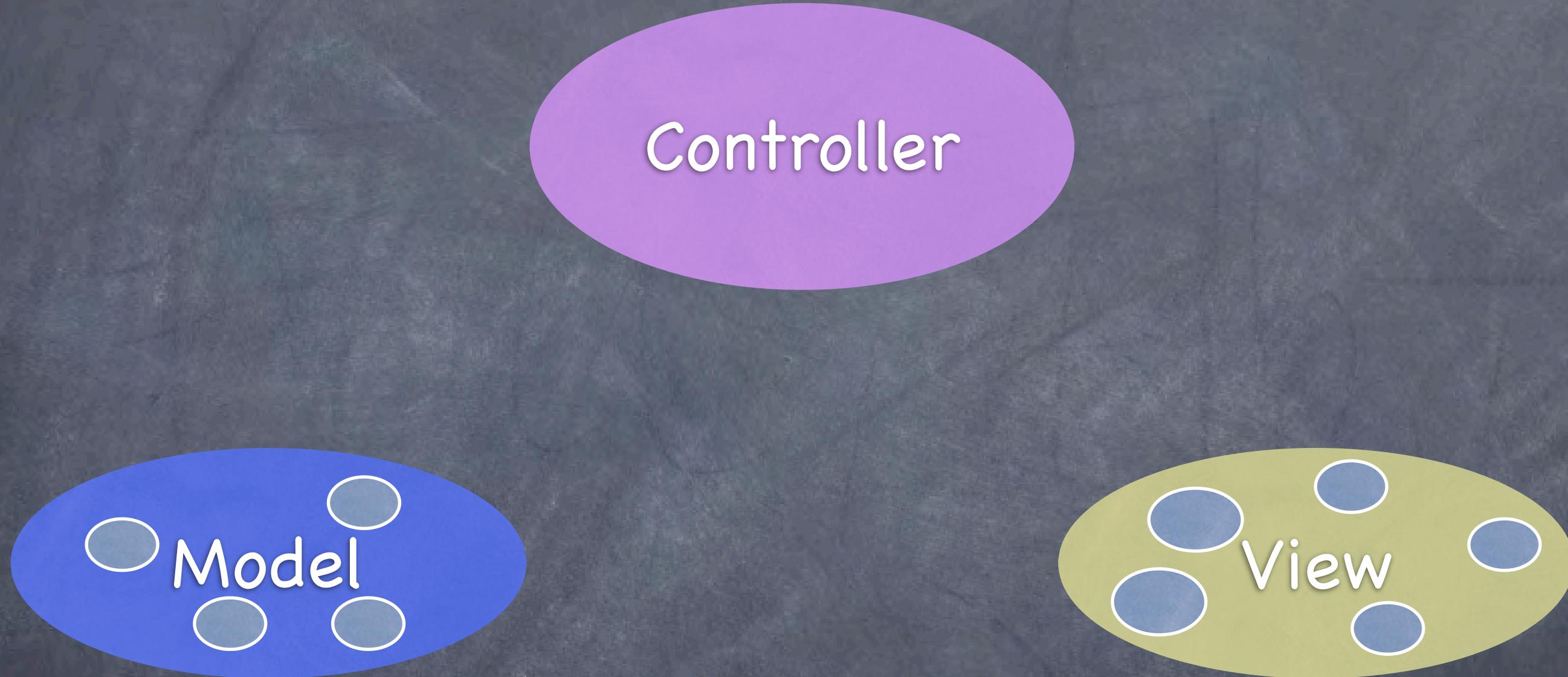
MVC

Controller



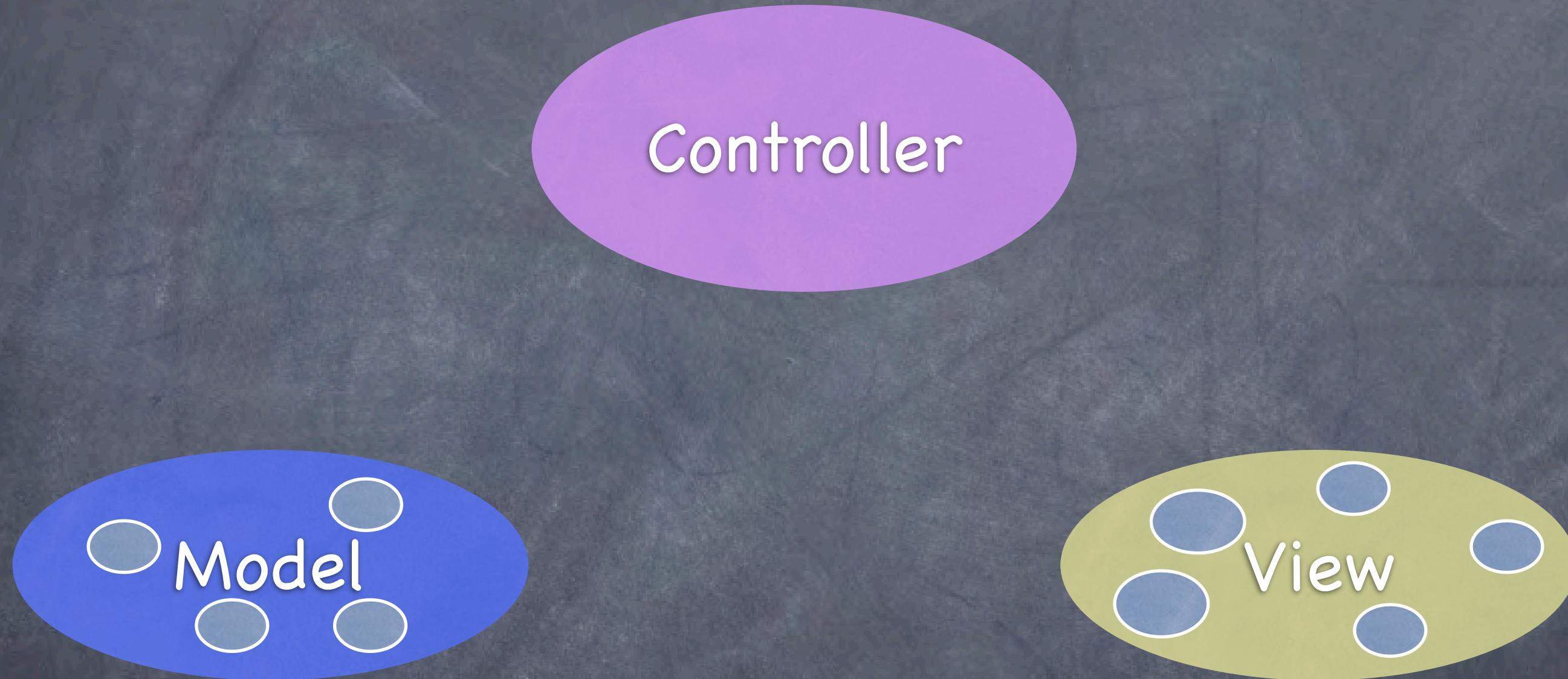
Divide objects in your program into 3 “camps.”

MVC



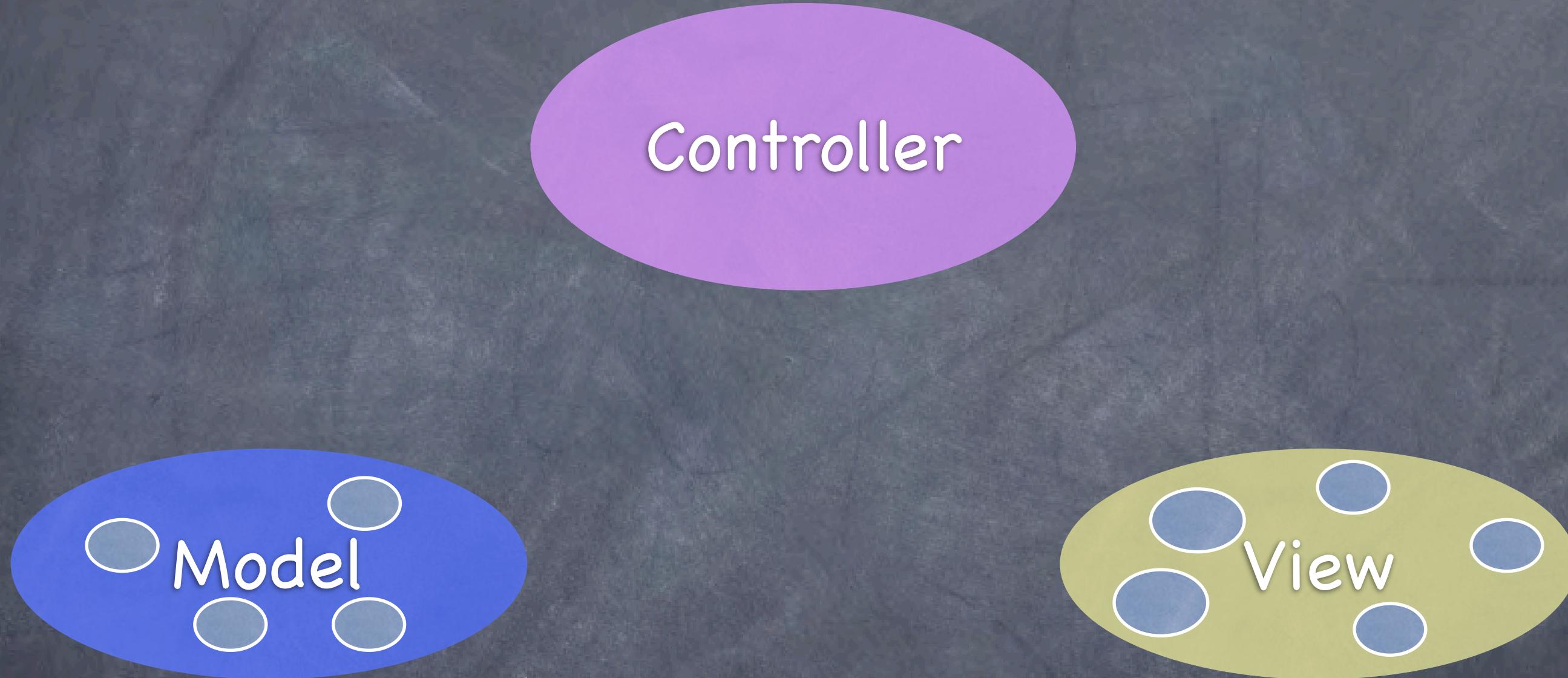
Model = What your application is (but not how it is displayed)

MVC



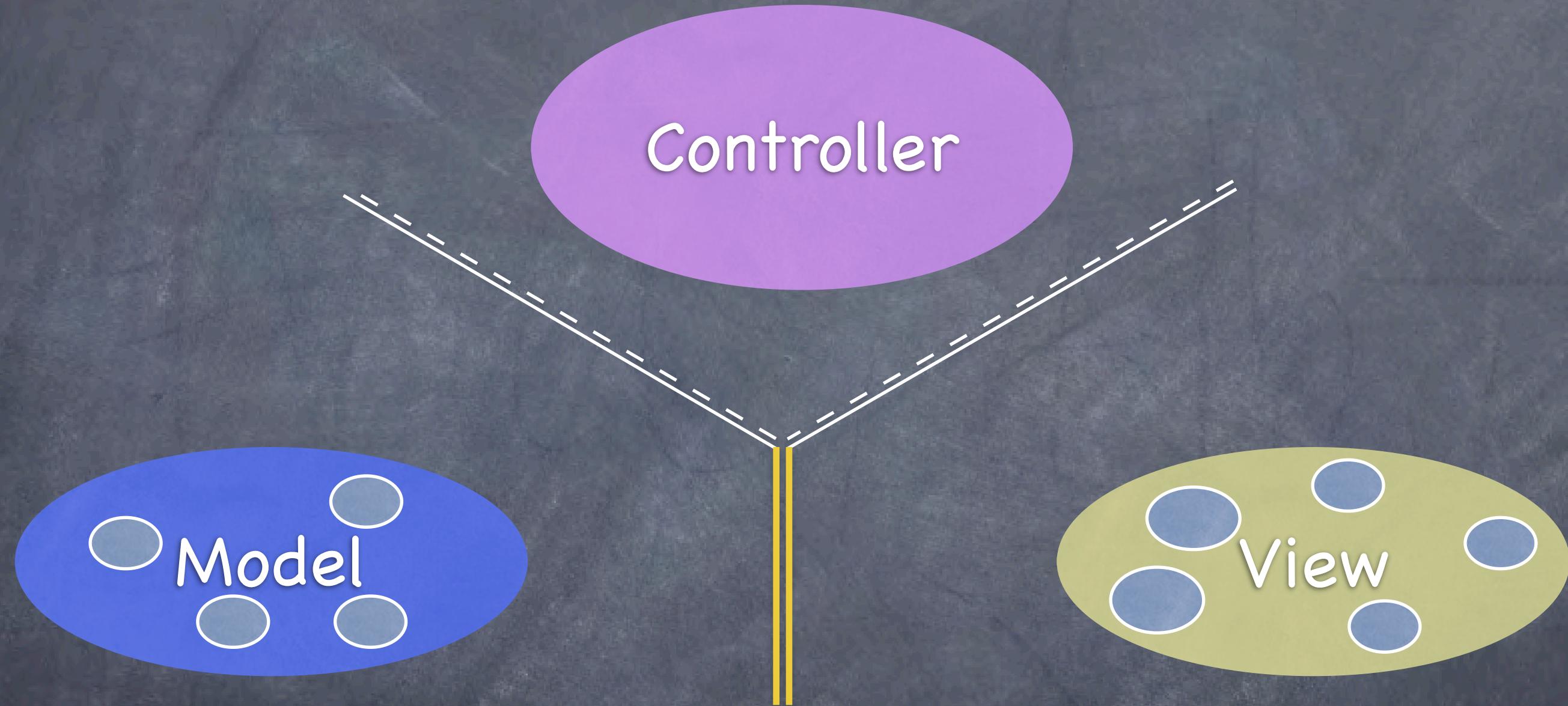
Controller = How your Model is presented to the user (UI logic)

MVC



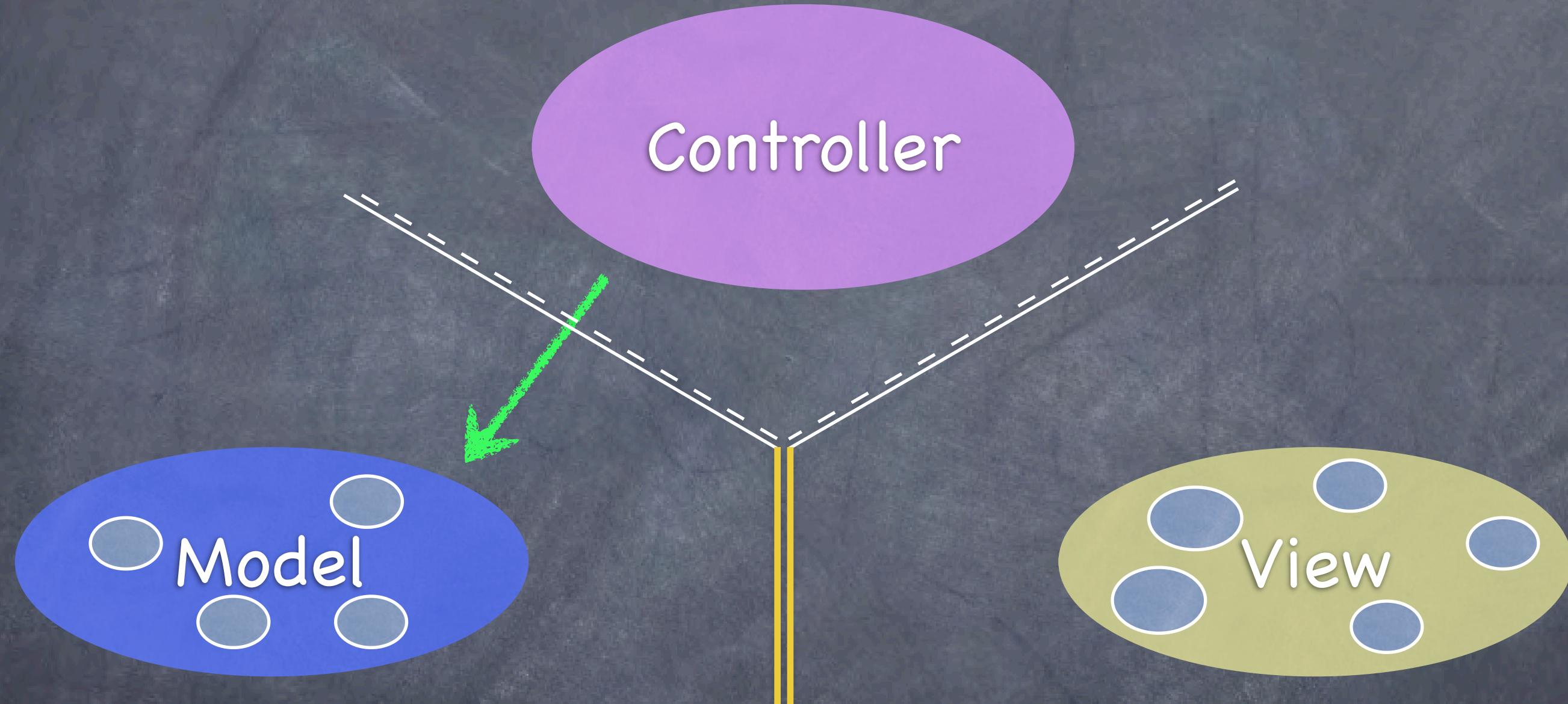
View = Your Controller's minions

MVC



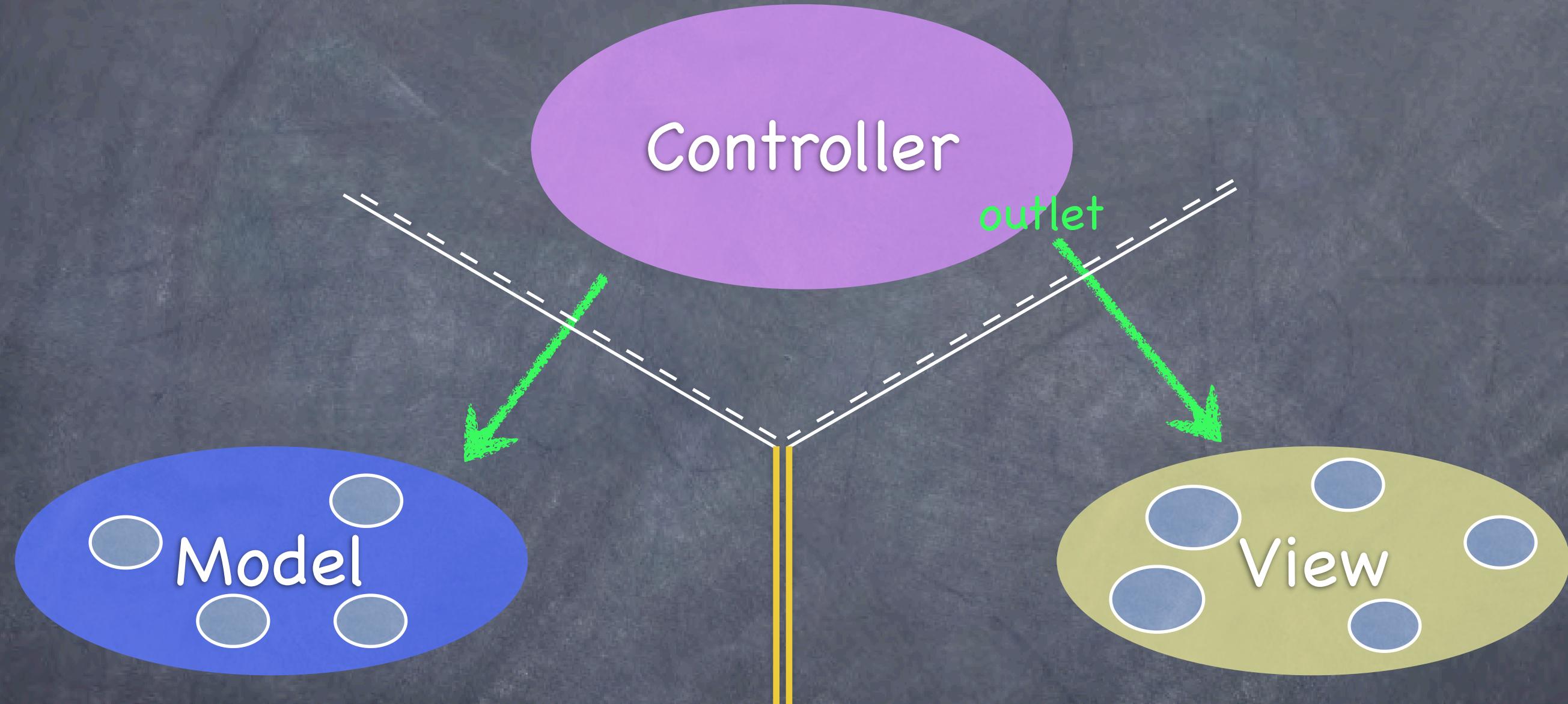
It's all about managing communication between camps

MVC



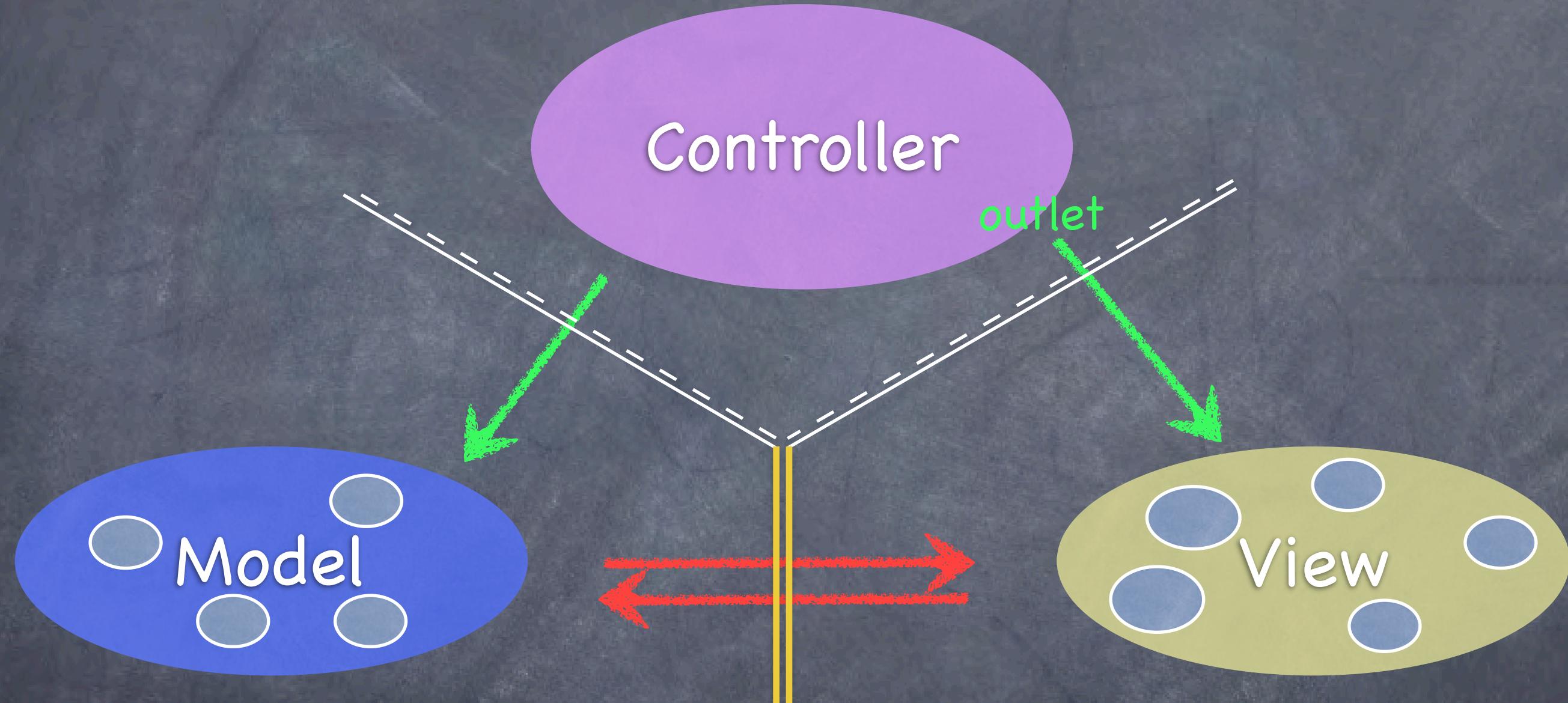
Controllers can always talk directly to their Model.

MVC



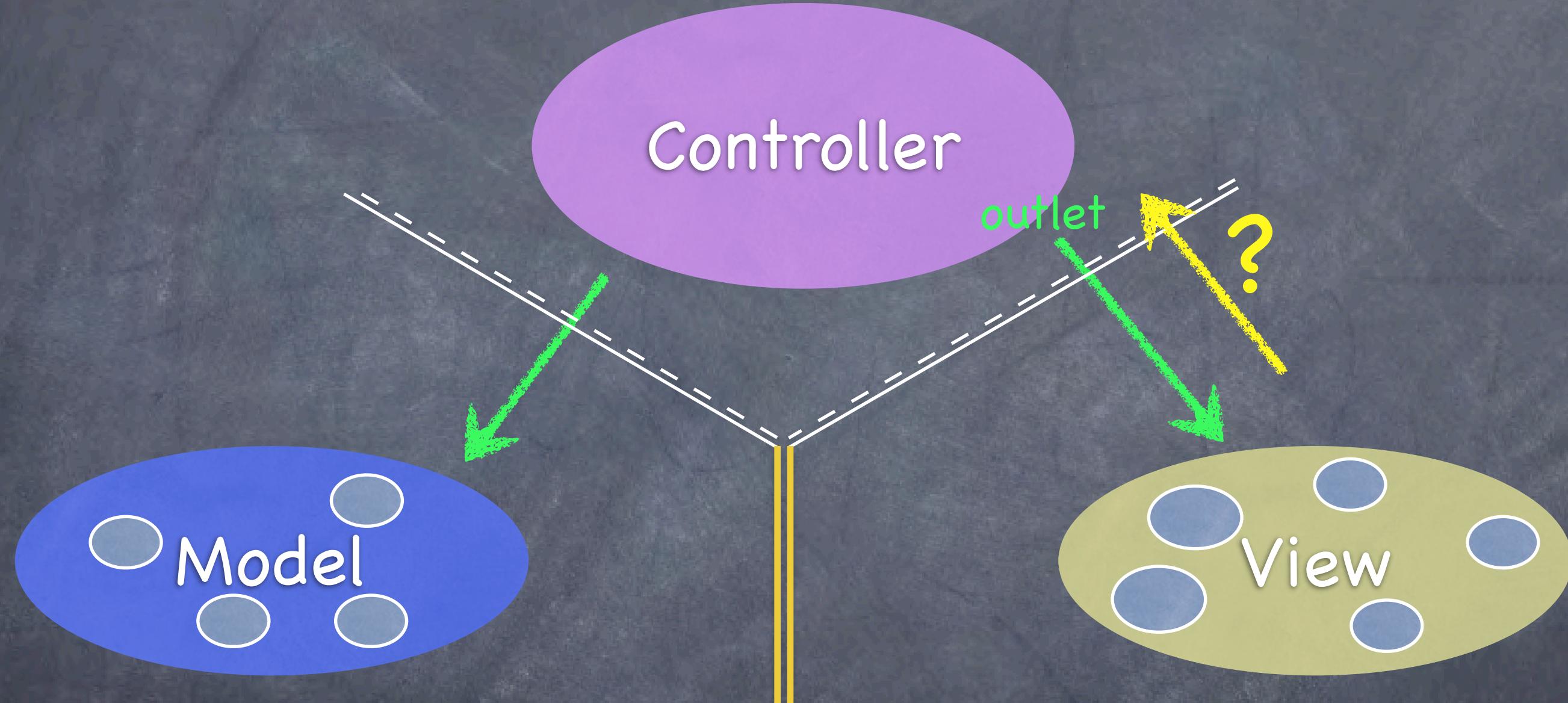
Controllers can also talk directly to their View.

MVC



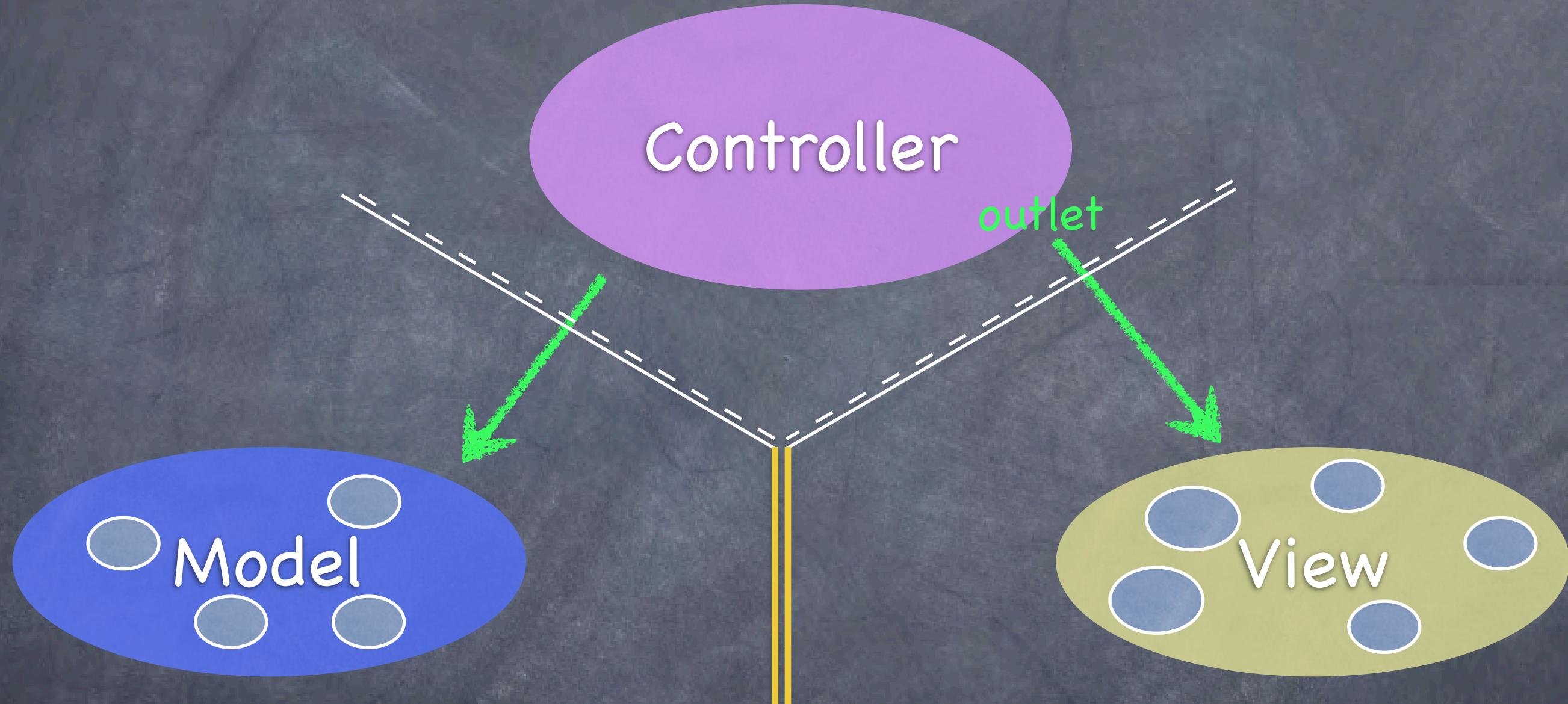
The Model and View should never speak to each other.

MVC



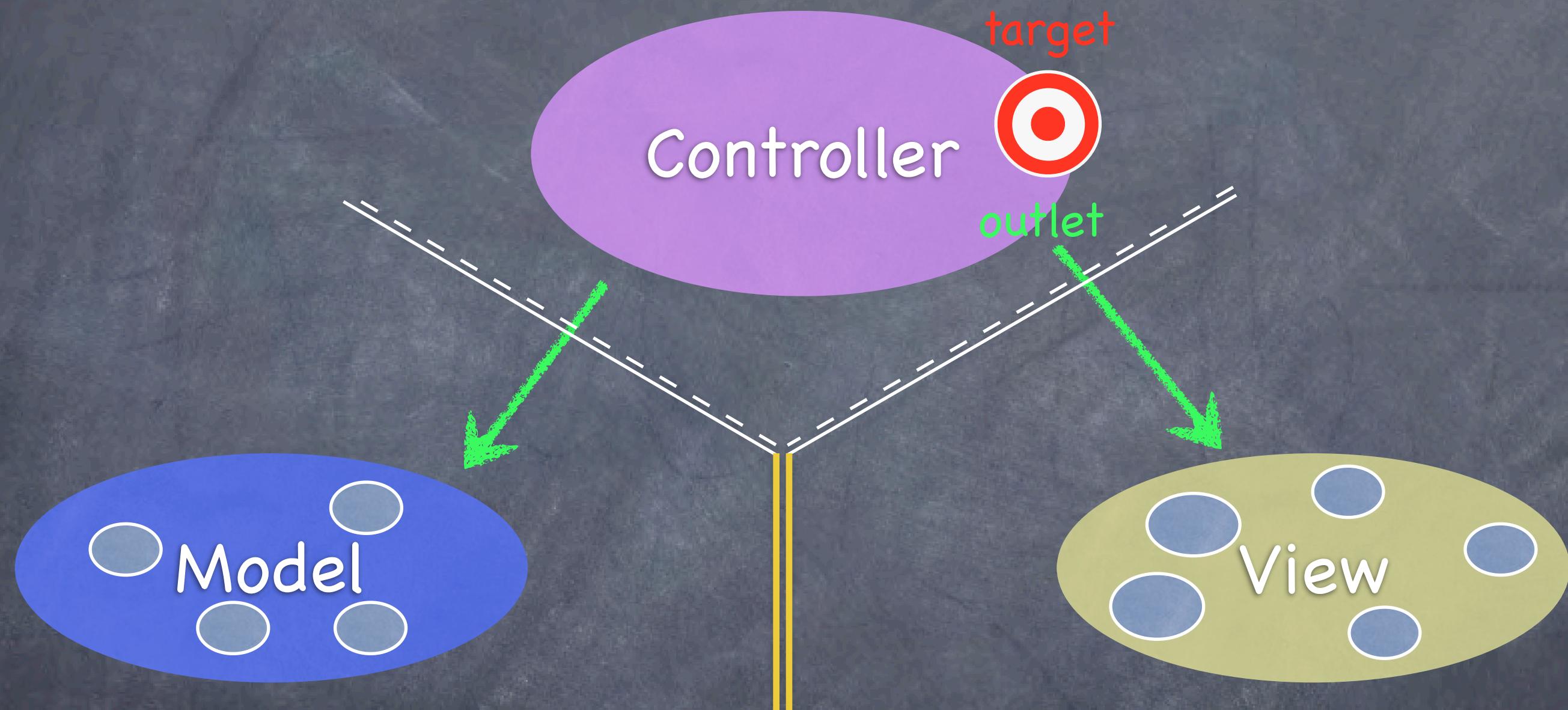
Can the View speak to its Controller?

MVC



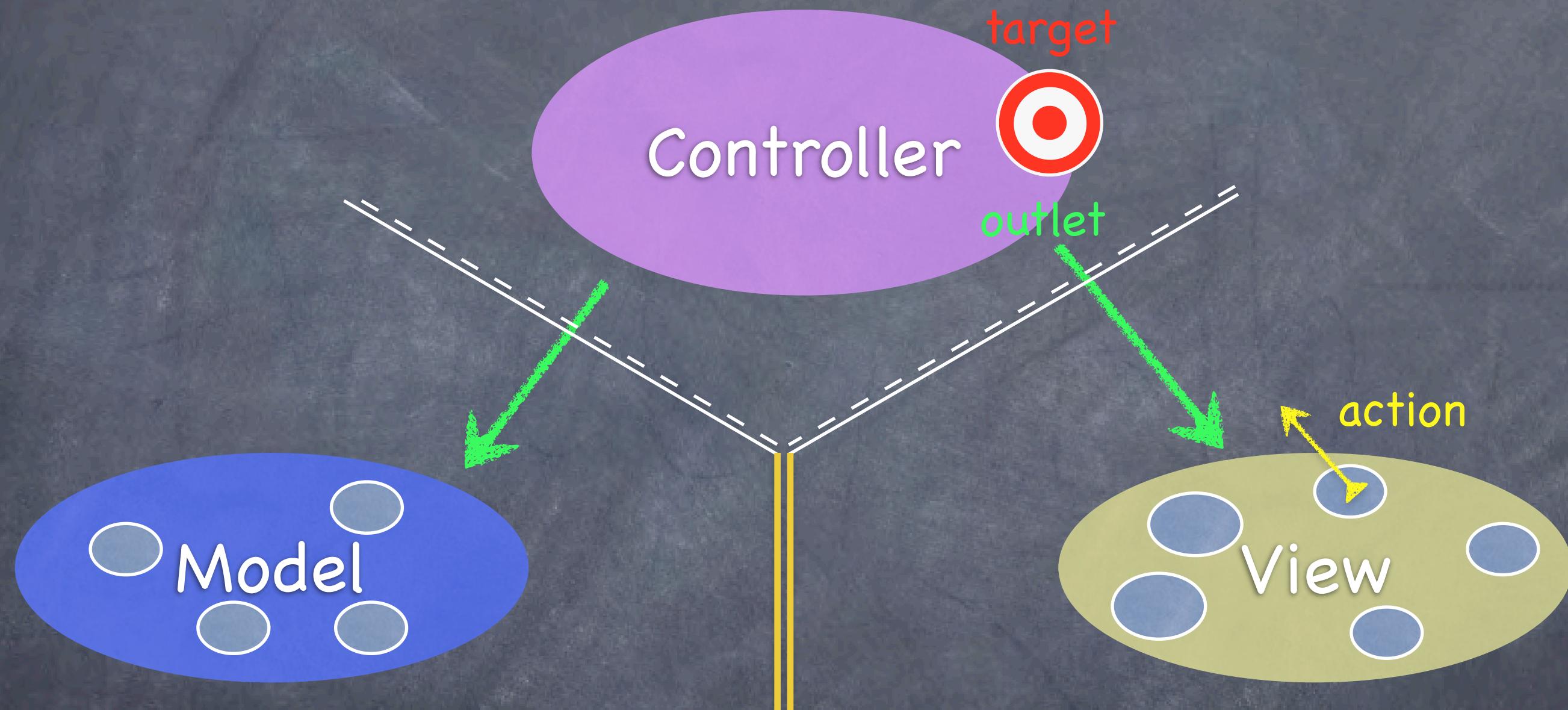
Sort of. Communication is “blind” and structured.

MVC



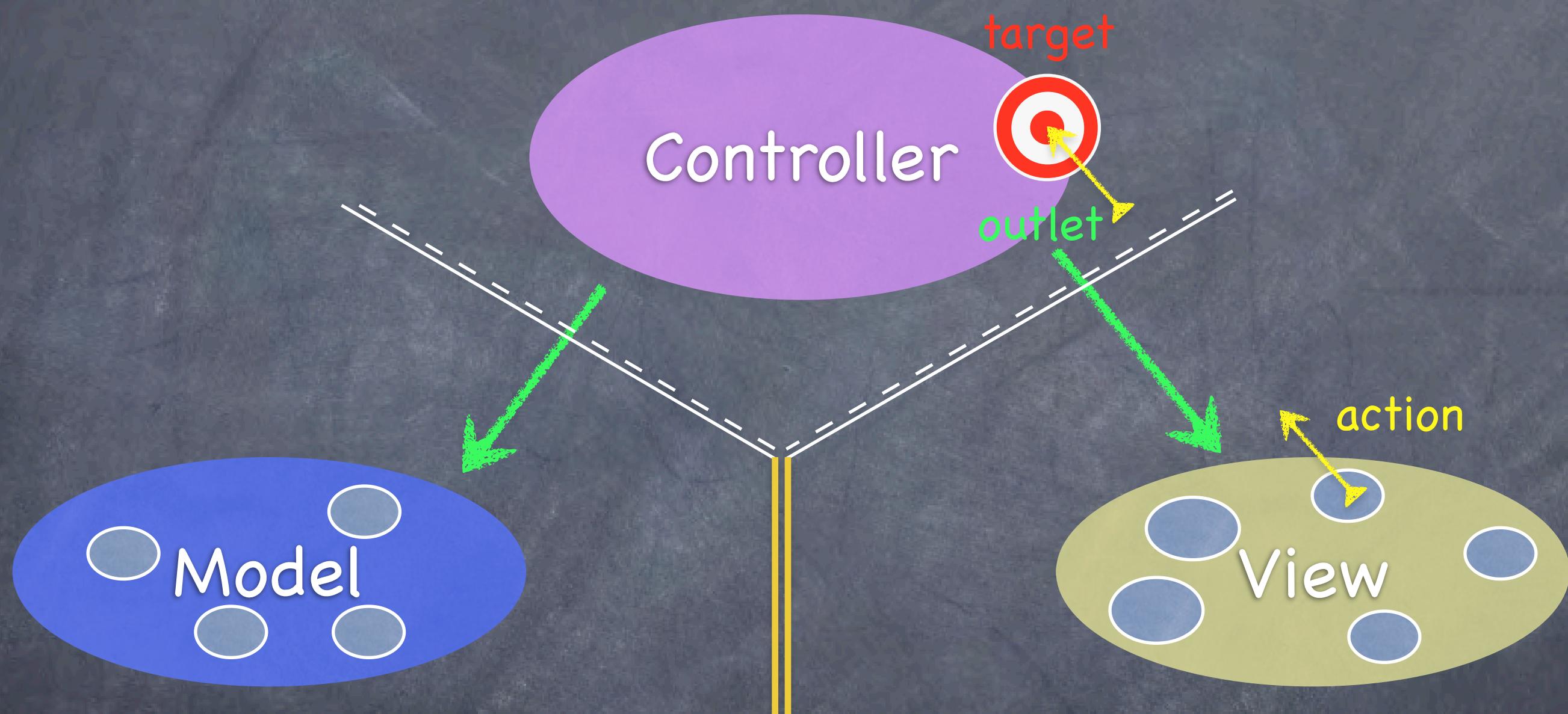
The Controller can drop a target on itself.

MVC



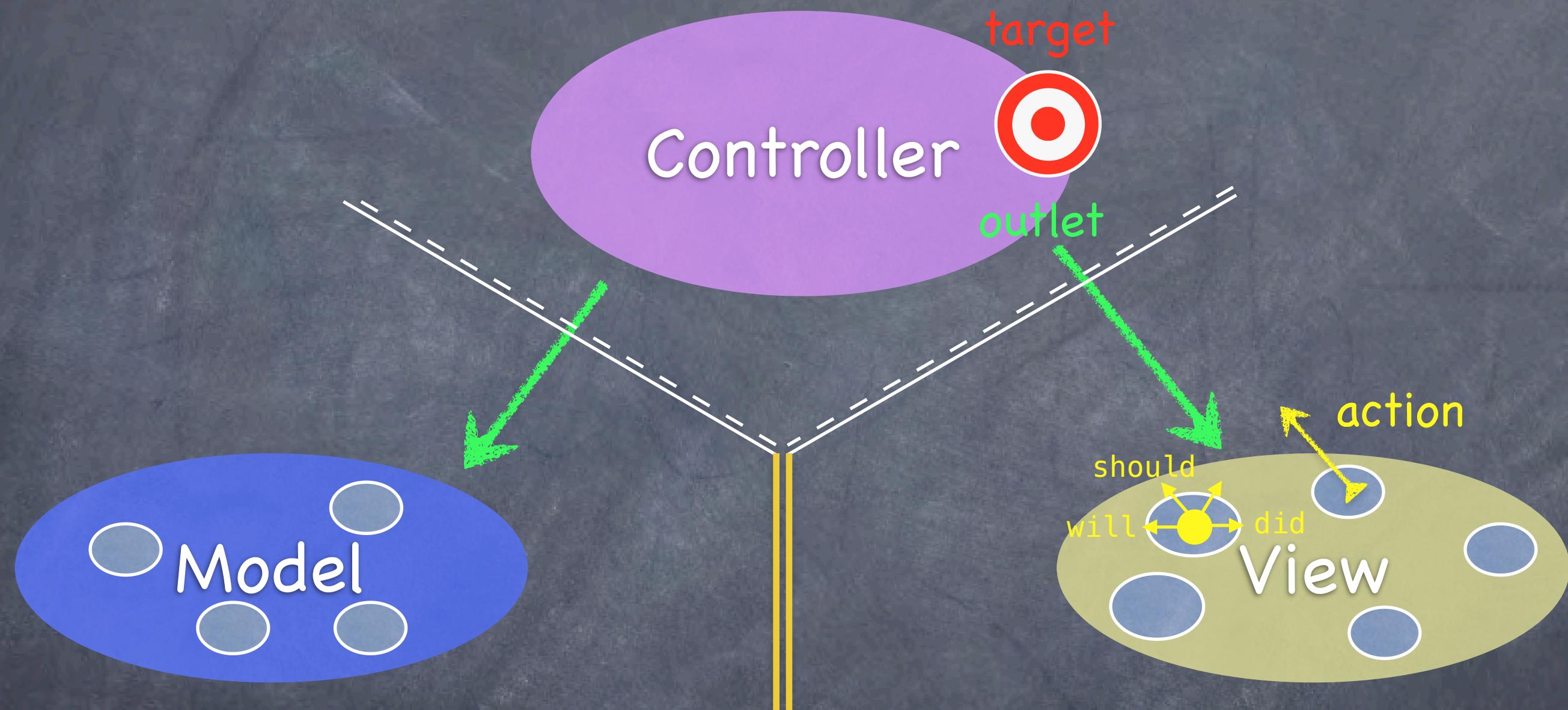
Then hand out an **action** to the View.

MVC



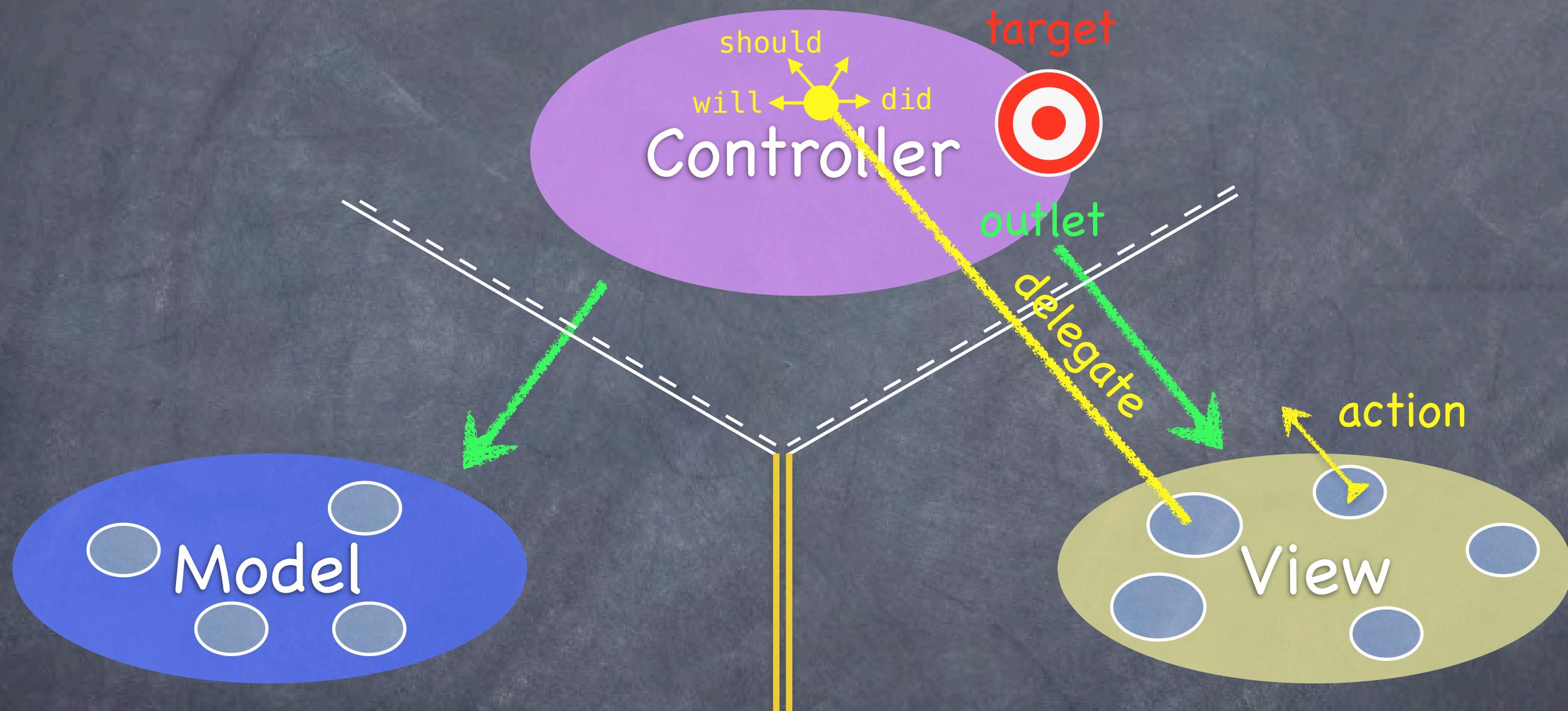
The View sends the **action** when things happen in the UI.

MVC



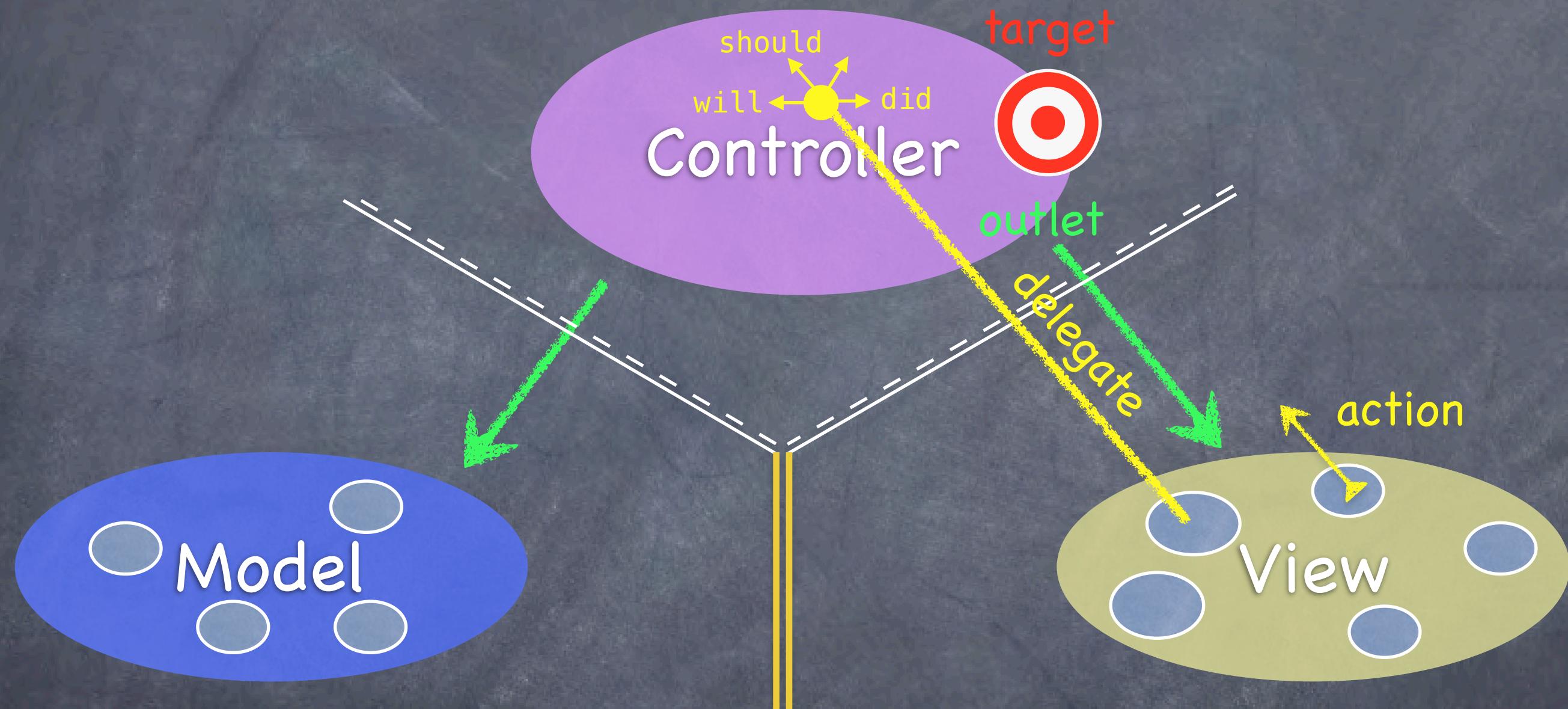
Sometimes the **View** needs to synchronize with the **Controller**.

MVC



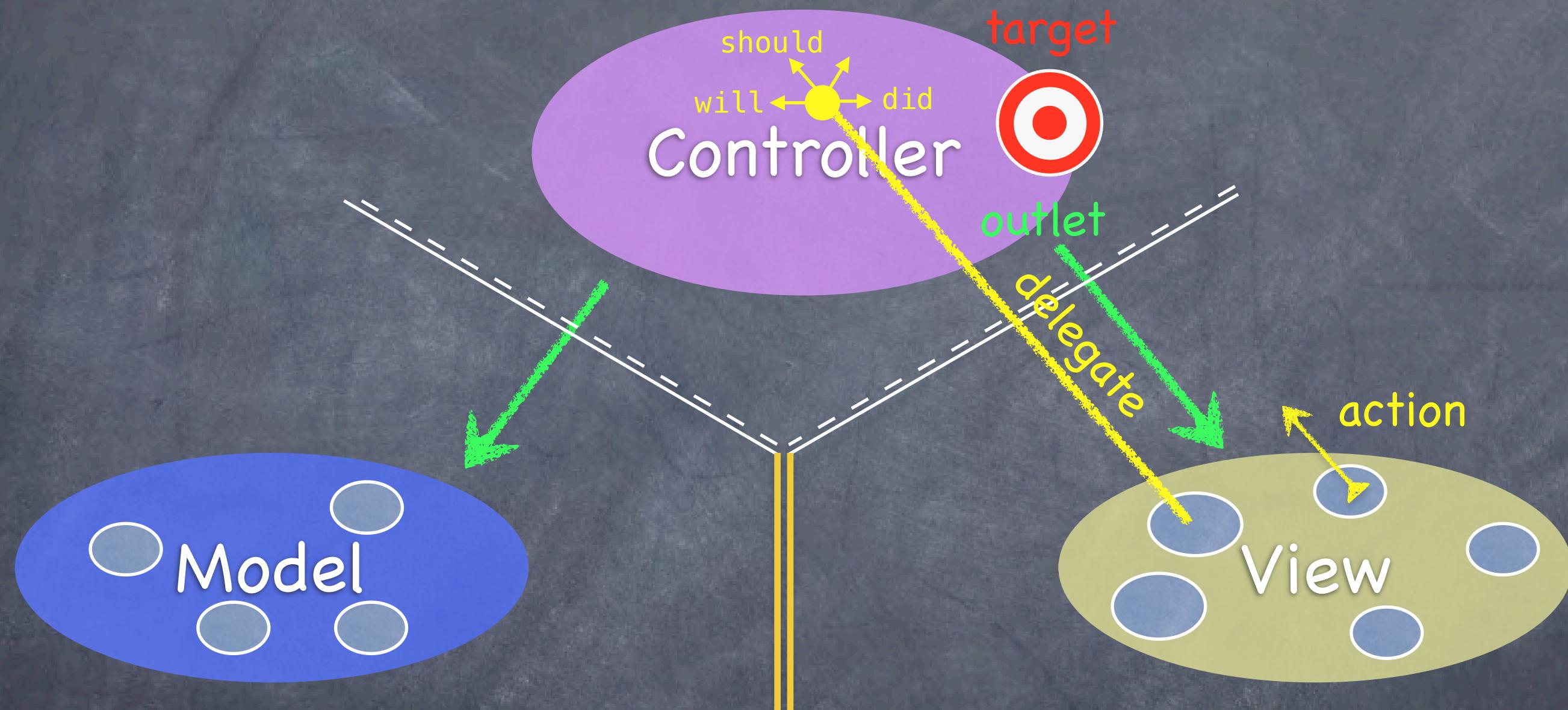
The Controller sets itself as the View's delegate.

MVC



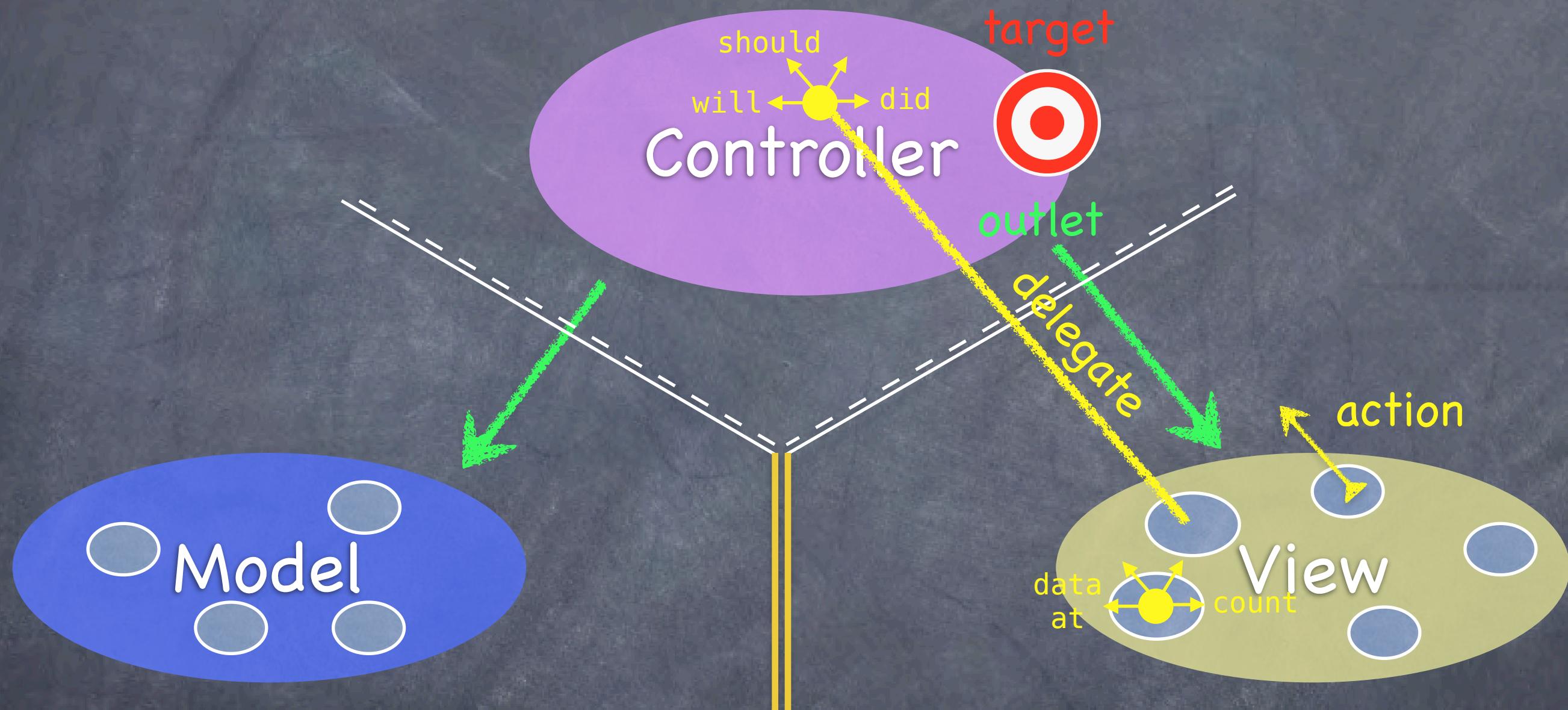
The **delegate** is set via a protocol (i.e. it's “blind” to class).

MVC



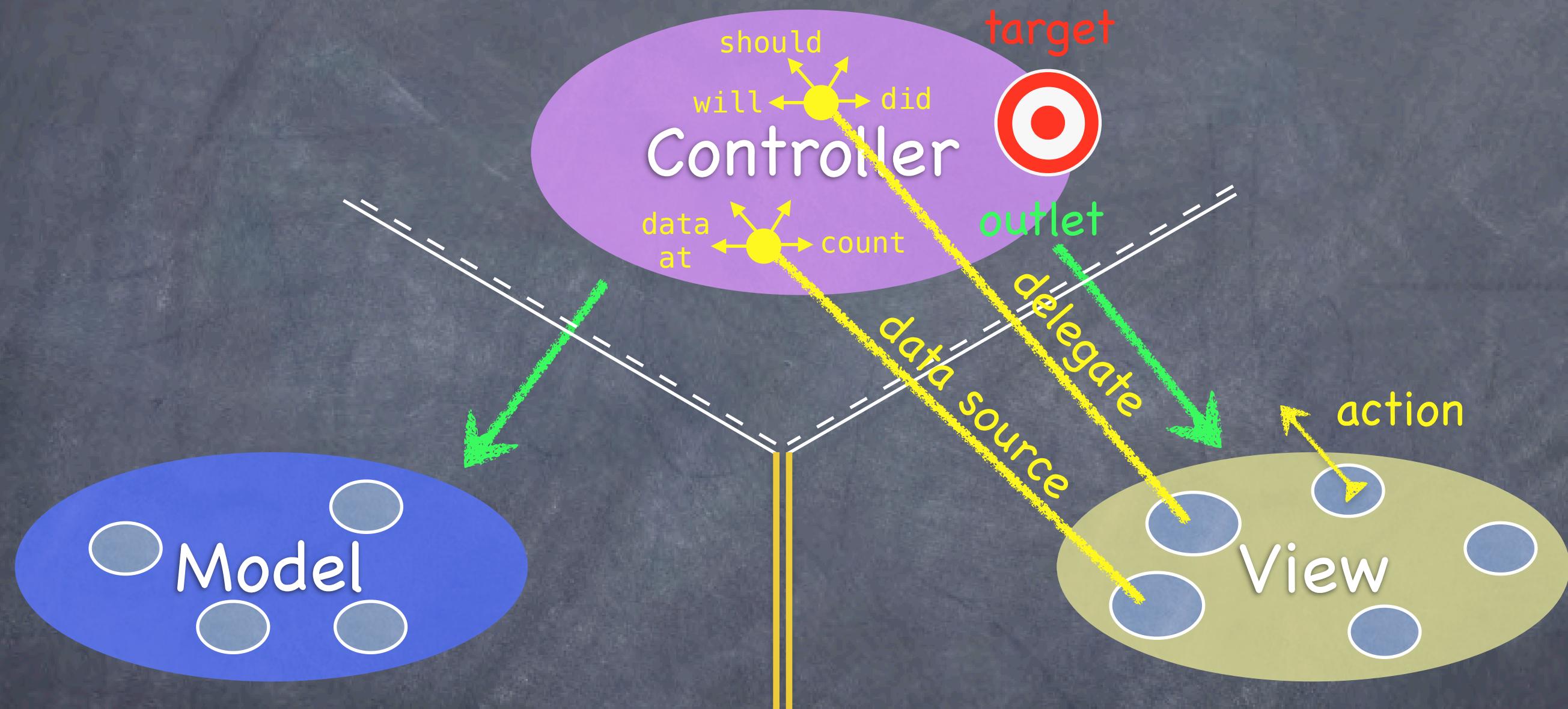
Views do not own the data they display.

MVC



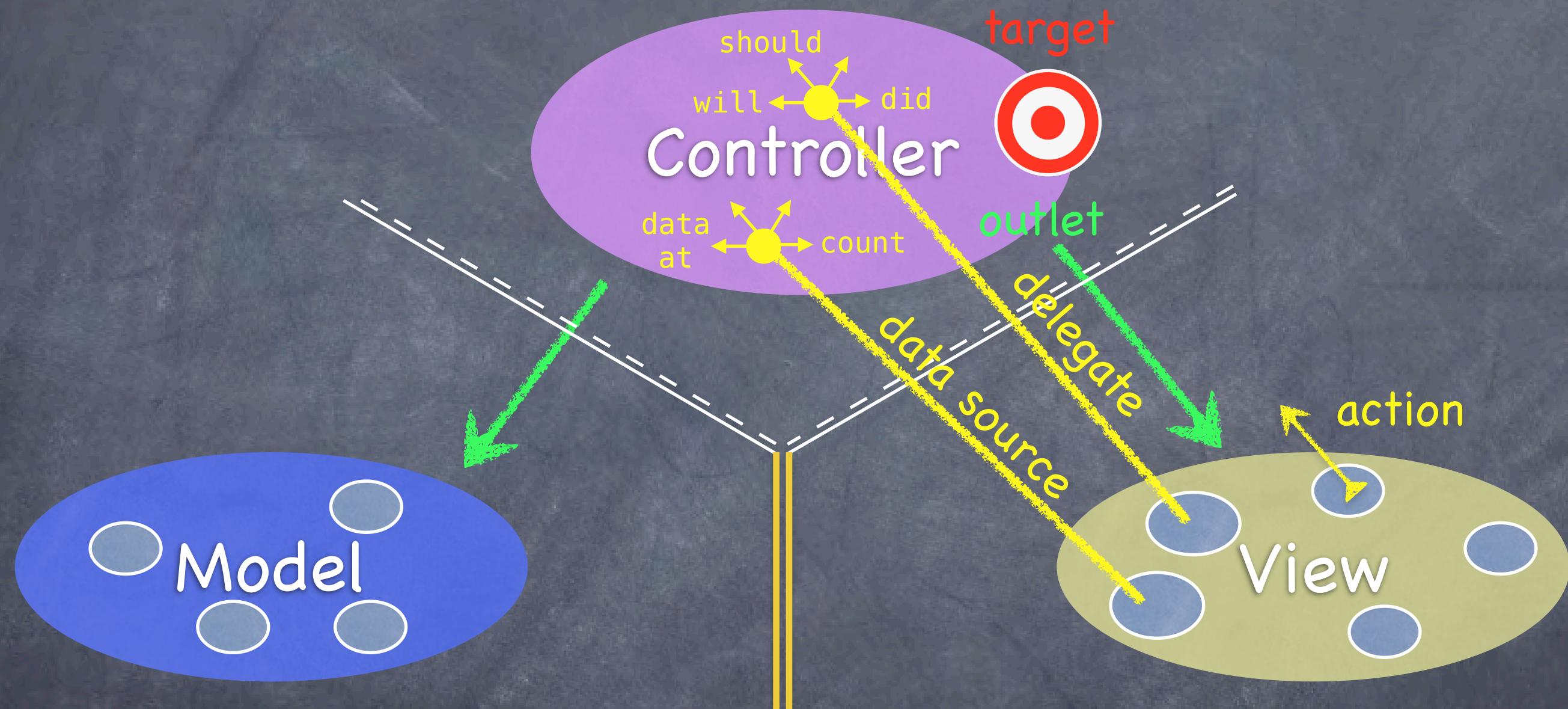
So, if needed, they have a protocol to acquire it.

MVC



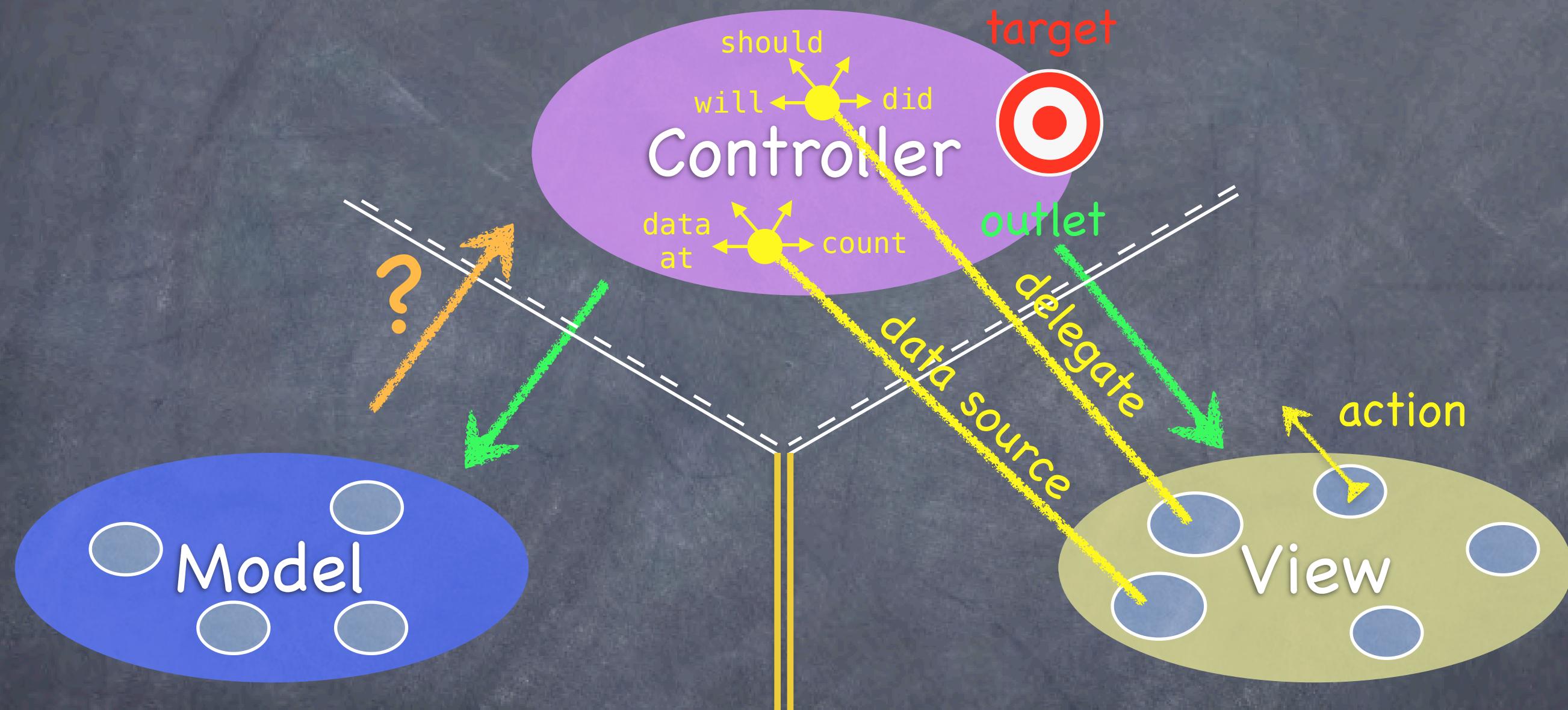
Controllers are almost always that data source (not Model!).

MVC



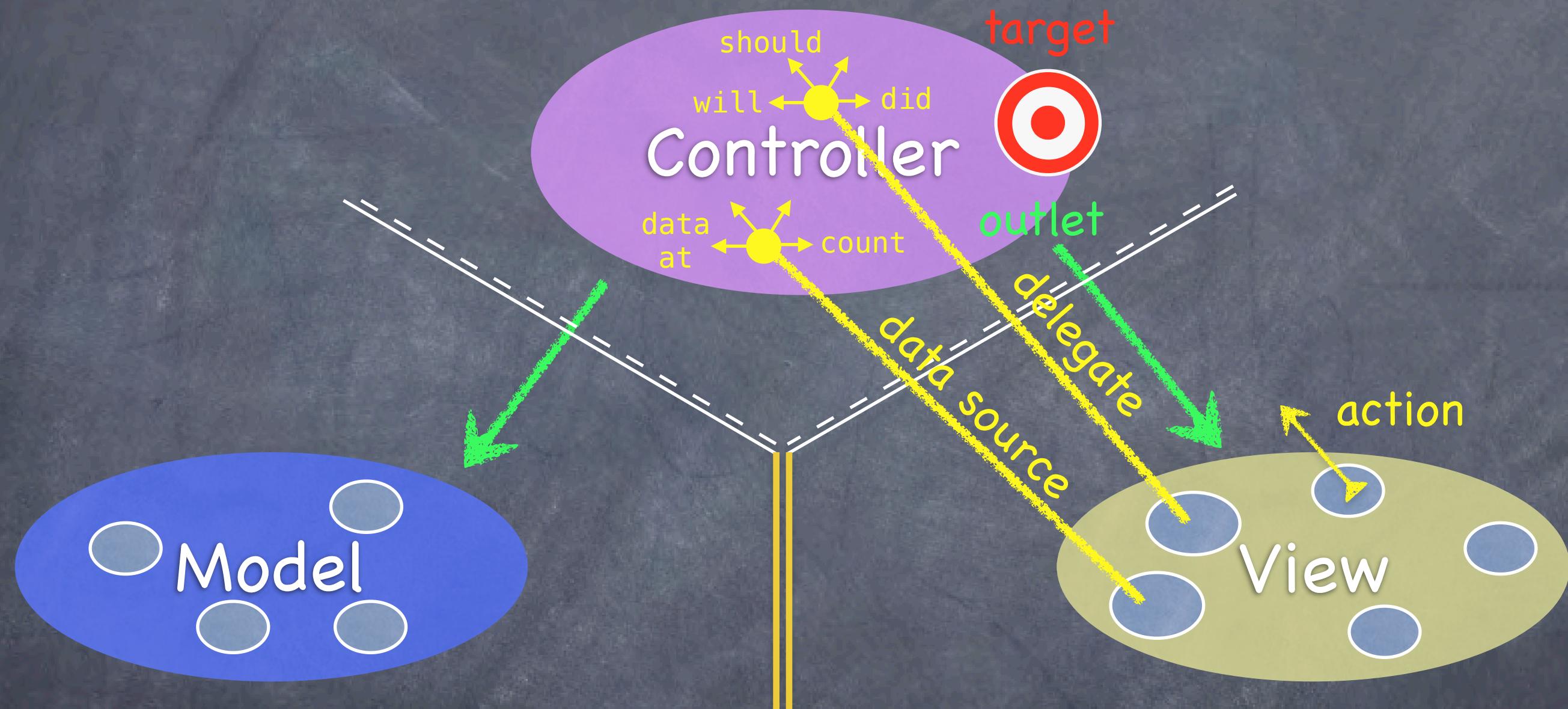
Controllers interpret/format Model information for the View.

MVC



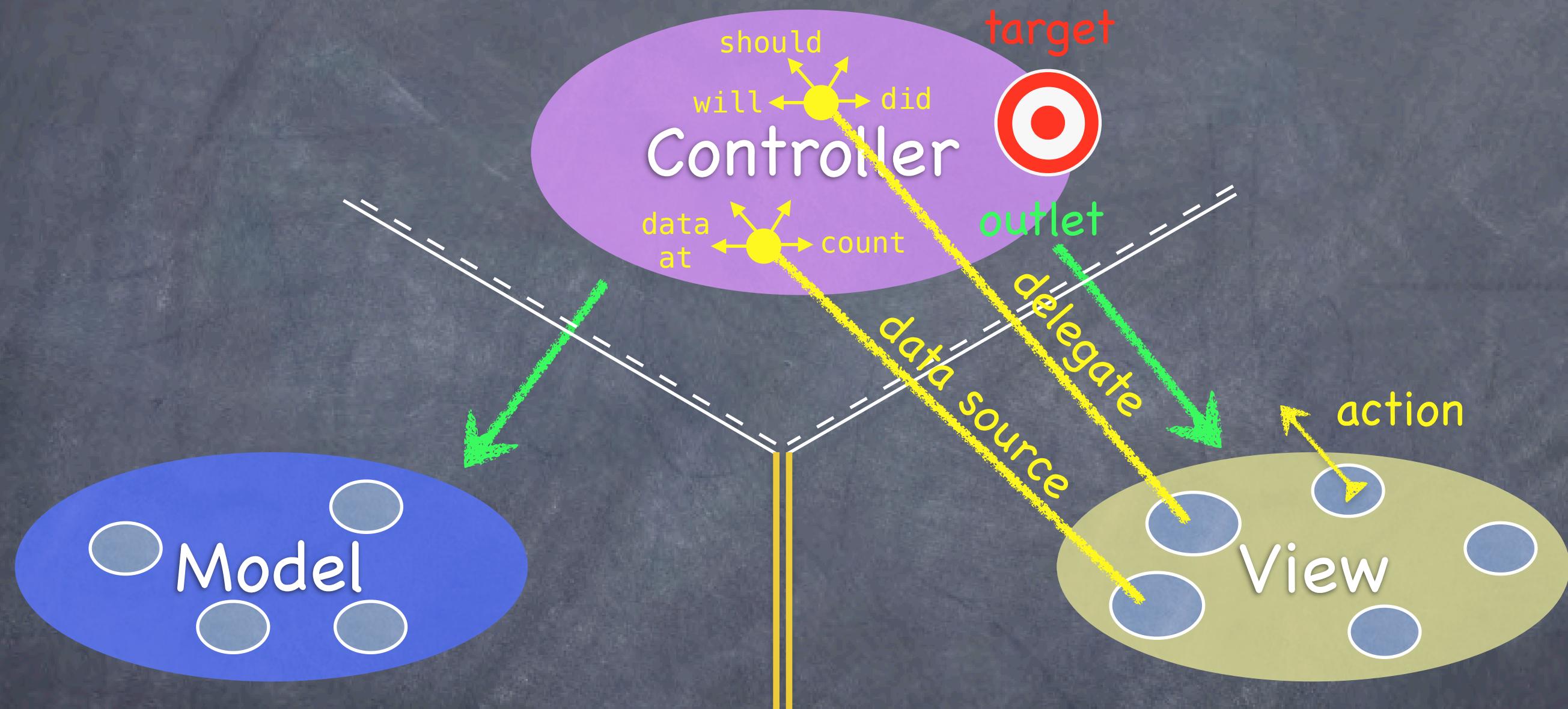
Can the Model talk directly to the Controller?

MVC



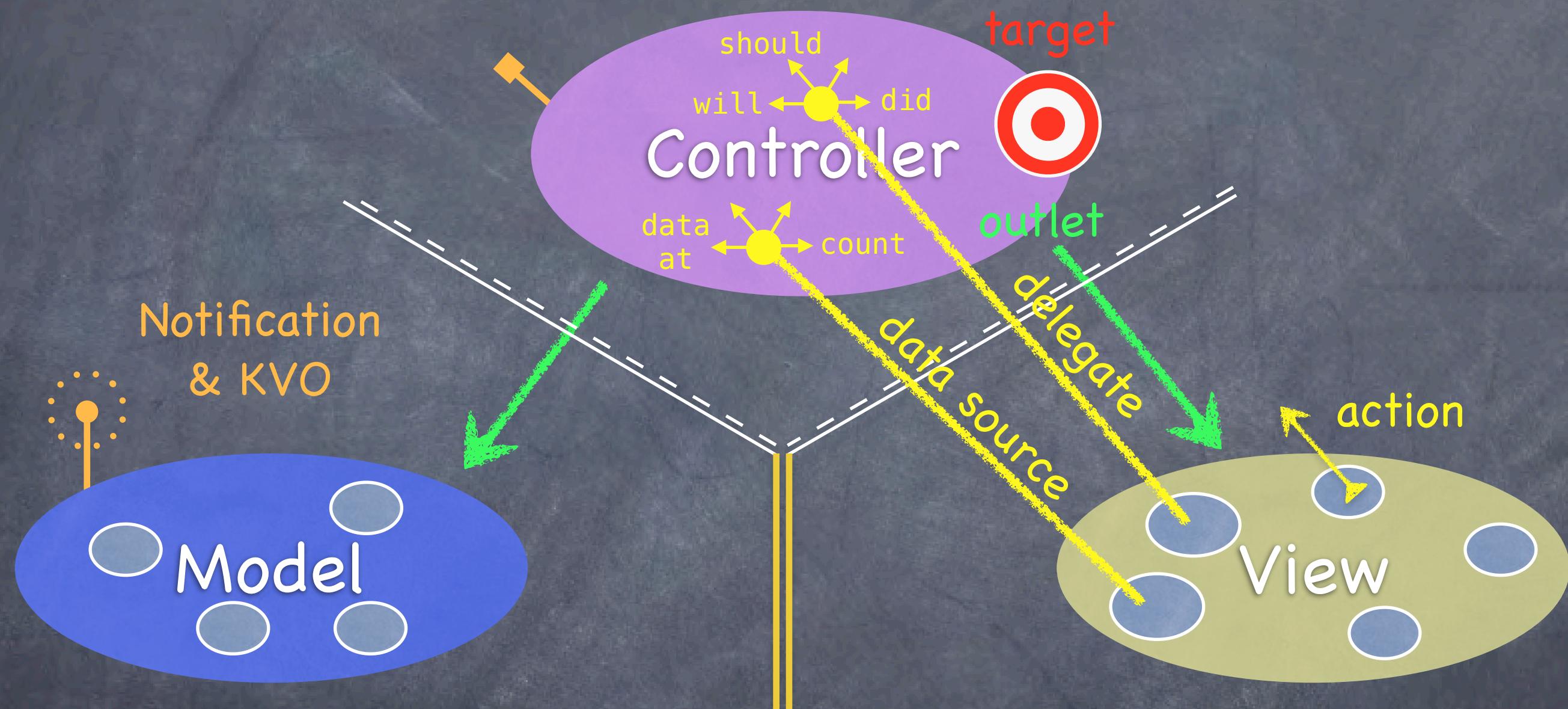
No. The Model is (should be) UI independent.

MVC



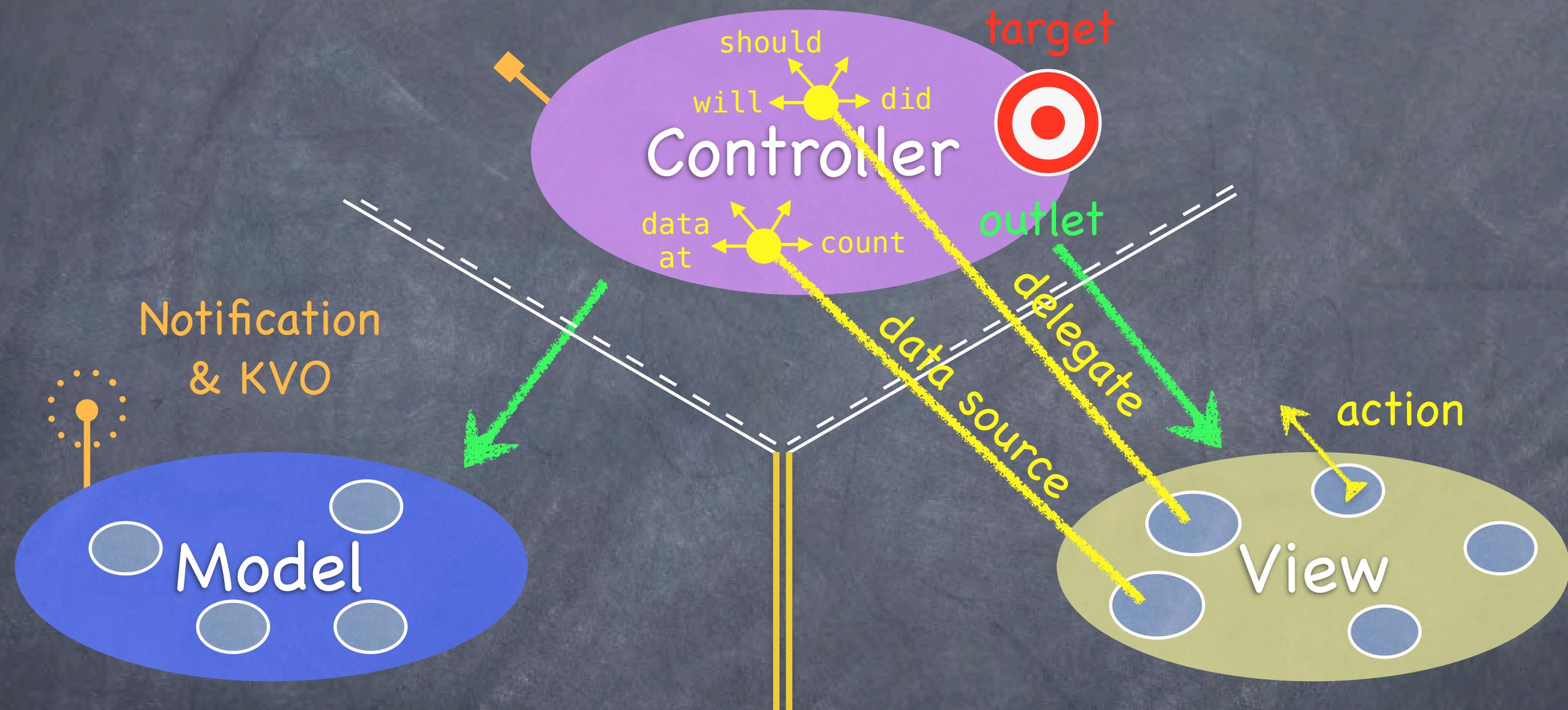
So what if the Model has information to update or something?

MVC



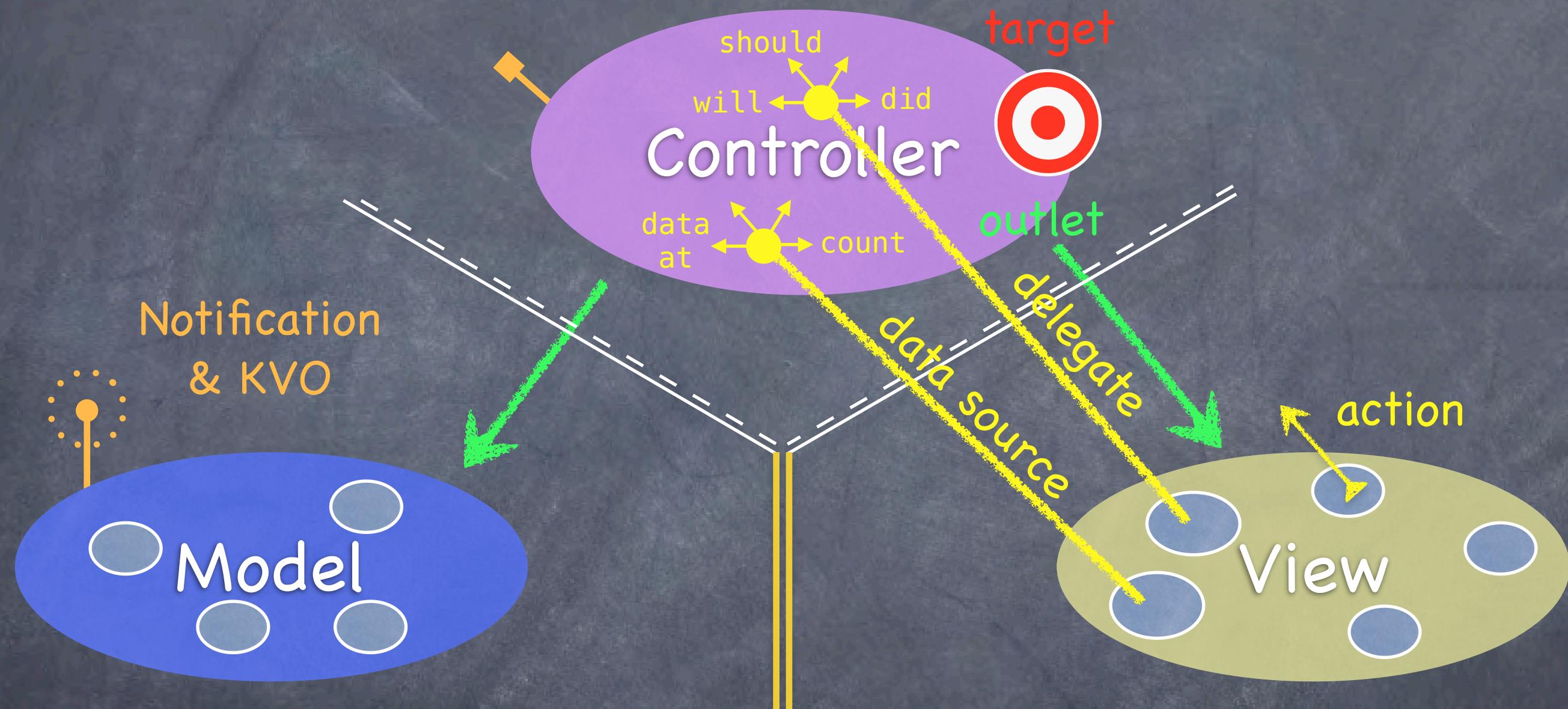
It uses a “radio station”-like broadcast mechanism.

MVC



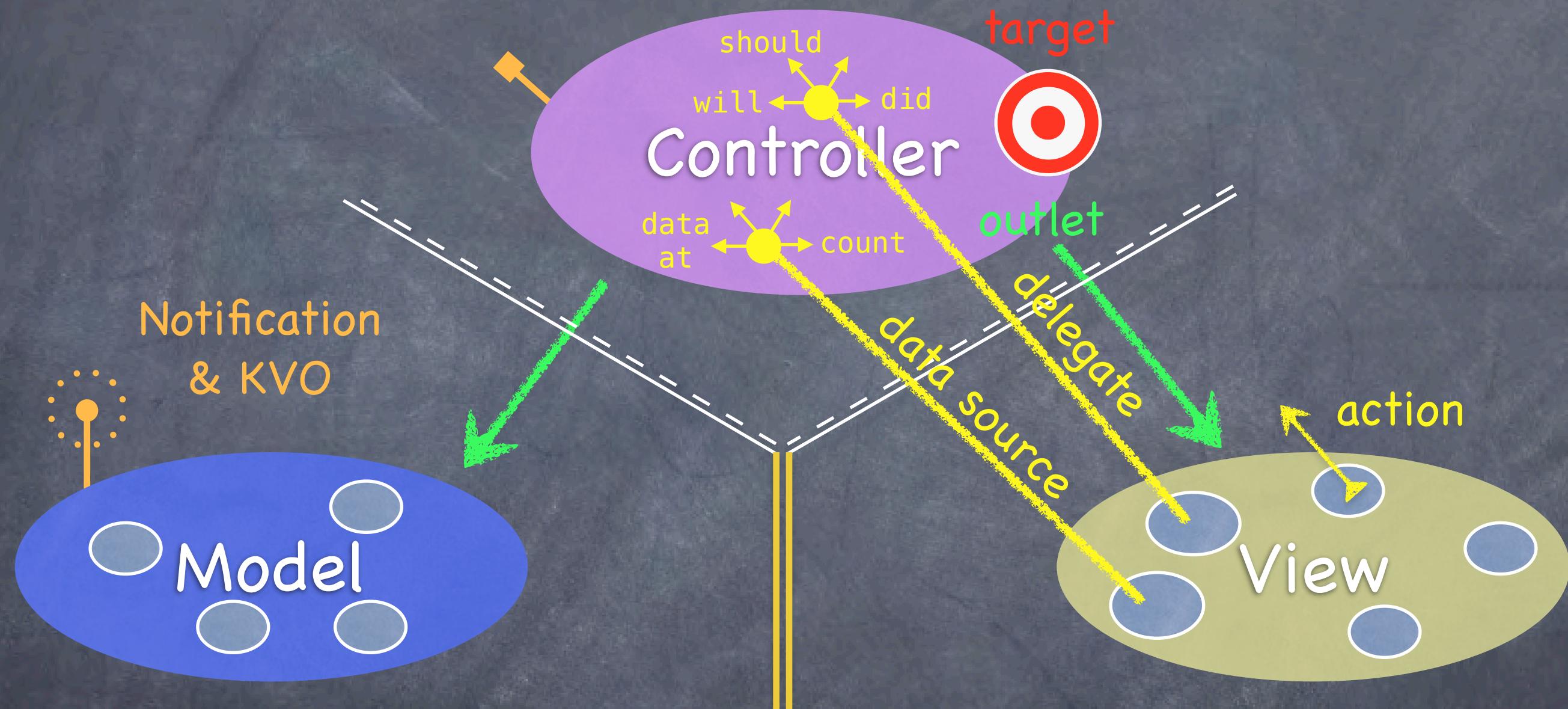
Controllers (or other Model) “tune in” to interesting stuff.

MVC



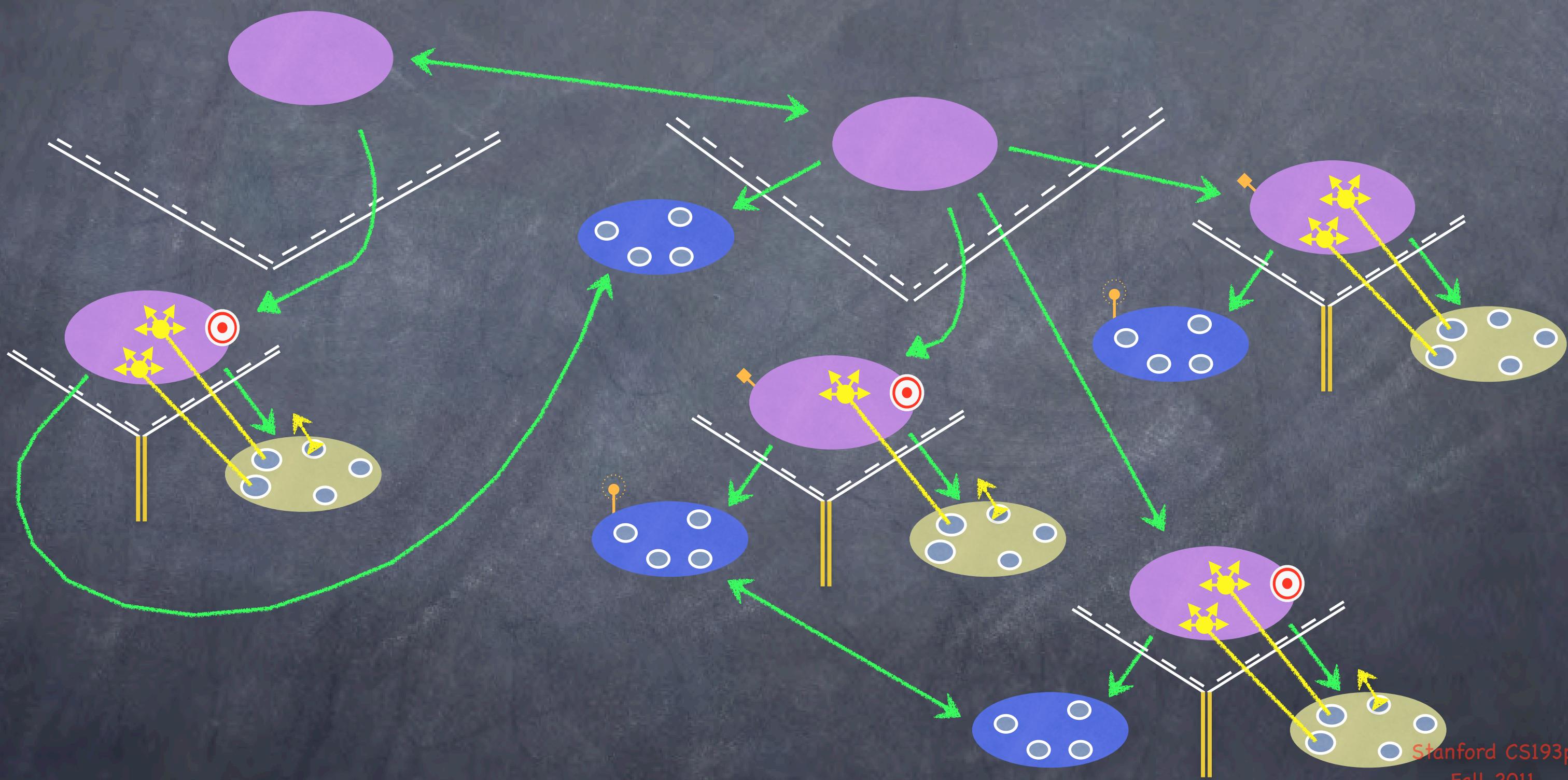
A View might “tune in,” but probably not to a Model’s “station.”

MVC

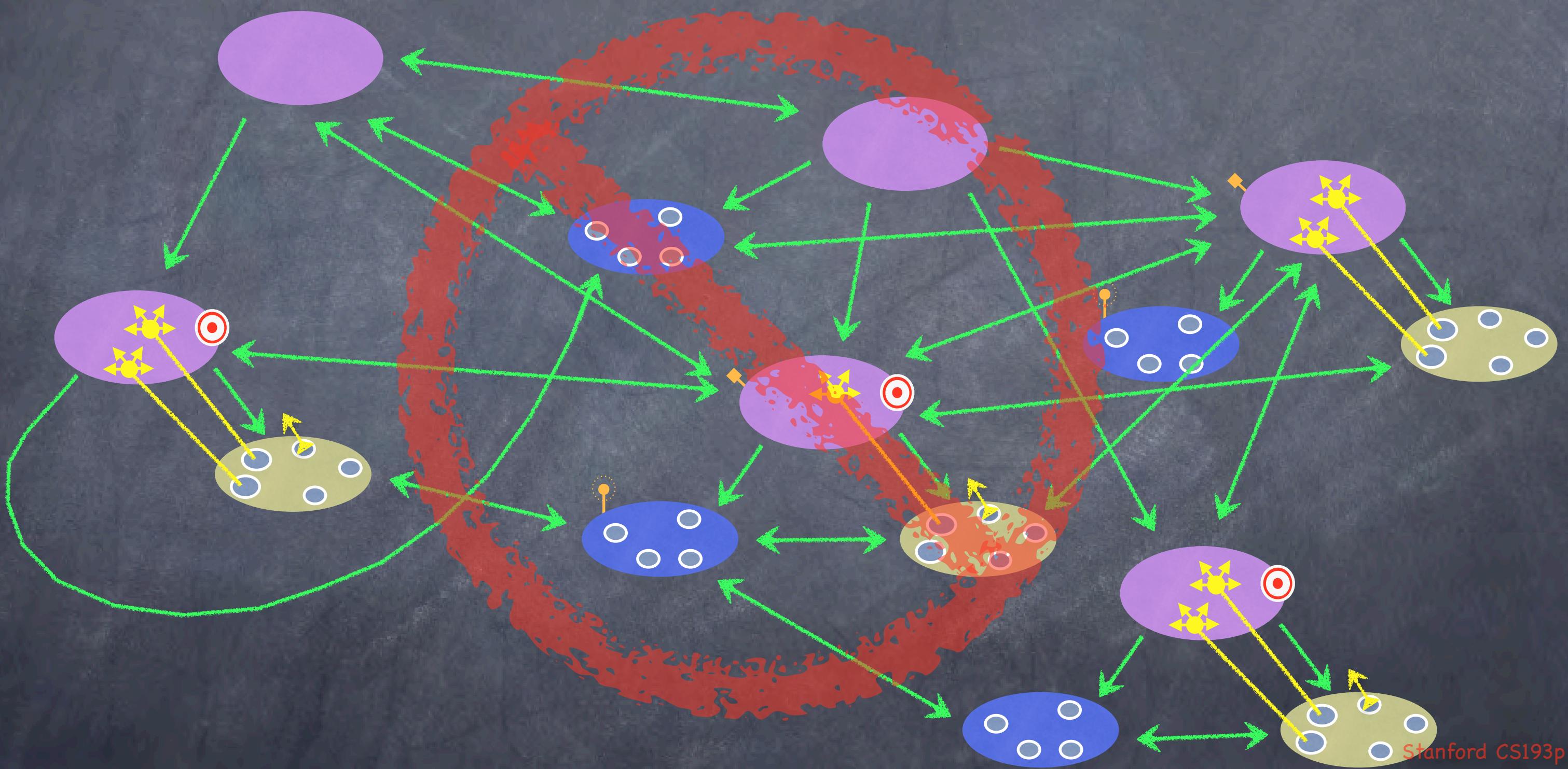


Now combine MVC groups to make complicated programs ...

MVCs working together



MVCs not working together



Objective-C

- ⦿ New language to learn!

- Strict superset of C

- Adds syntax for classes, methods, etc.

- A few things to “think differently” about (e.g. properties, dynamic binding)

- ⦿ Most important concept to understand today: Properties

- Usually we do not access instance variables directly in Objective-C.

- Instead, we use “properties.”

- A “property” is just the combination of a getter method and a setter method in a class.

- The getter has the name of the property (e.g. “myValue”)

- The setter’s name is “set” plus capitalized property name (e.g. “setMyValue:”)

- (To make this look nice, we always use a lowercase letter as the first letter of a property name.)

- We just call the setter to store the value we want and the getter to get it. Simple.

- ⦿ This is just your first glimpse of this language!

- We’ll go much more into the details next week.

- Don’t get too freaked out by the syntax at this point.

Objective-C

Spaceship.h

```
#import "Vehicle.h"
```

Class name

Superclass

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

```
@interface Spaceship : Vehicle
```

```
@end
```

Spaceship.m

```
#import "Spaceship.h"
```

```
@implementation Spaceship
```

```
@end
```

Importing our own header file.

Note, superclass not specified here.

Objective-C

Spaceship.h

```
#import "Vehicle.h"

@interface Spaceship : Vehicle

// declaration of public methods

@end
```

Spaceship.m

```
#import "Spaceship.h"

@implementation Spaceship

// implementation of public and private methods

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"

@interface Spaceship : Vehicle

// declaration of public methods
```

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods
```

Don't forget the () .

No superclass here either.

@end

@end

Objective-C

Spaceship.h

```
#import "Vehicle.h"  
#import "Planet.h"  
  
@interface Spaceship : Vehicle  
  
// declaration of public methods
```

The full name of this method is
orbitPlanet:atAltitude:

```
- (void)orbitPlanet:(Planet *)aPlanet  
atAltitude:(double)km;
```

Lining up the colons
makes things look nice.

It does not return any value.

```
#import "Spaceship.h"  
  
@interface Spaceship()  
// declaration of private methods (as needed)
```

```
@end
```

```
@implementation Spaceship
```

```
// implementation of public and private methods
```

It takes two arguments.

Note how each is preceded by its own keyword.

```
@end
```

Spaceship.m

```
@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods
```

No semicolon here.

```
- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;

- (void)setTopSpeed:(double)percentSpeedOfLight;
- (double)topSpeed;
```

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods
```

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km;
- (void)setTopSpeed:(double)percentSpeedOfLight;
- (double)topSpeed;

@end

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

- (void)setTopSpeed:(double)speed
{
    ???
}

- (double)topSpeed
{
    ???
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods
```

This @property
essentially declares
the two “topSpeed”
methods below.

```
@property (nonatomic) double topSpeed;
- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
- (void)setTopSpeed:(double)percentSpeedOfLight;
- (double)topSpeed;
```

`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.

@end

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship
// implementation of public and private methods

- (void)setTopSpeed:(double)speed
{
    ???
}

- (double)topSpeed
{
    ???
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

- (void)setTopSpeed:(double)speed
{
    ???
}

- (double)topSpeed
{
    ???
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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.

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

@synthesize topSpeed = _topSpeed;

- (void)setTopSpeed:(double)speed
{
    ???
}

- (double)topSpeed
{
    ???
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

This is the name of the storage location to use.

`_` (underbar) then the name of the property is a common naming convention.

If we don't use `=` here, `@synthesize` uses the name of the property (which is **bad** so always use `=`).

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

@synthesize topSpeed = _topSpeed;

- (void)setTopSpeed:(double)speed
{
    _topSpeed = speed;
}

- (double)topSpeed
{
    return _topSpeed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
```

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

```
- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle

// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

Spaceship.m

```
#import "Spaceship.h"

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

@end

@implementation Spaceship

// implementation of public and private methods

@synthesize topSpeed = _topSpeed;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

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

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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

@end

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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

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

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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 next week.

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
}
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

We're calling topSpeed's getter on ourself here.

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

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
    double speed = [self topSpeed];
    if (speed > MAX_RELATIVE) speed = MAX_RELATIVE;
}
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
- (void)setTopSpeed:(double)percentSpeedOfLight;
- (double)topSpeed;

@end
```

A reminder of what our getter declaration looks like.
Recall that these two declarations are accomplished with
the `@property` for `topSpeed` above.

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship

// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
    double speed = [self topSpeed];
    if (speed > MAX_RELATIVE) speed = MAX_RELATIVE;
}

@end
```

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

Here's another example of sending a message.

It looks like this method has 2 arguments:
a Planet to travel to and a speed to travel at.
It is being sent to an instance of Wormhole.

```
@end
```

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
    double speed = [self topSpeed];
    if (speed > MAX_RELATIVE) speed = MAX_RELATIVE;
    [[self nearestWormhole] travelToPlanet:aPlanet
                                    atSpeed:speed];
}
```

Square brackets inside square brackets.

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;
```

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

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
    double speed = self.topSpeed;
    if (speed > MAX_RELATIVE) speed = MAX_RELATIVE;
    [[self nearestWormhole] travelToPlanet:aPlanet
        atSpeed:speed];
}
```

This is identical to [self topSpeed].

Objective-C

Spaceship.h

```
#import "Vehicle.h"
#import "Planet.h"

@interface Spaceship : Vehicle
// declaration of public methods

@property (nonatomic) double topSpeed;

- (void)orbitPlanet:(Planet *)aPlanet
    atAltitude:(double)km;

@end
```

Spaceship.m

```
#import "Spaceship.h"

@interface Spaceship()
// declaration of private methods (as needed)
@property (nonatomic, strong) Wormhole *nearestWormhole;
@end

@implementation Spaceship
// implementation of public and private methods

@synthesize topSpeed = _topSpeed;
@synthesize nearestWormhole = _nearestWormhole;

- (void)setTopSpeed:(double)speed
{
    if ((speed < 1) && (speed > 0)) _topSpeed = speed;
}

- (void)orbitPlanet:(Planet *)aPlanet atAltitude:(double)km
{
    // put the code to orbit a planet here
    double speed = self.topSpeed;
    if (speed > MAX_RELATIVE) speed = MAX_RELATIVE;
    [self.nearestWormhole travelToPlanet:aPlanet
        atSpeed:speed];
}

@end
```

We can use dot notation here too.

Coming Up

⌚ Next Lecture

Overview of the Integrated Development Environment (IDE, i.e. Xcode 4)

Objective-C in action

Concrete example of MVC

Major demonstration of all of the above: RPN Calculator

(HOMEWORK: if you do not know what an RPN Calculator is, look it up on the internet.)

⌚ Friday

Very simple introduction to using the debugger.

Optional. You can figure it out on your own if you want (not too difficult).

⌚ Next Week

Objective-C language in depth

Foundation classes: arrays, dictionaries, strings, etc.

Dynamic vs. static typing

Protocols, categories and much, much more!