

# Stanford CS193p

## Developing Applications for iOS

### Fall 2013-14



# 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

- (Time Permitting)

- 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 or backpack!

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 (or X) required

CS107 or CS108 or CS110 required

(or equivalent for non-Stanford undergrad)

## ⦿ 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)

## ⦿ 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

6 weekly (approximately) assignments

Individual work only

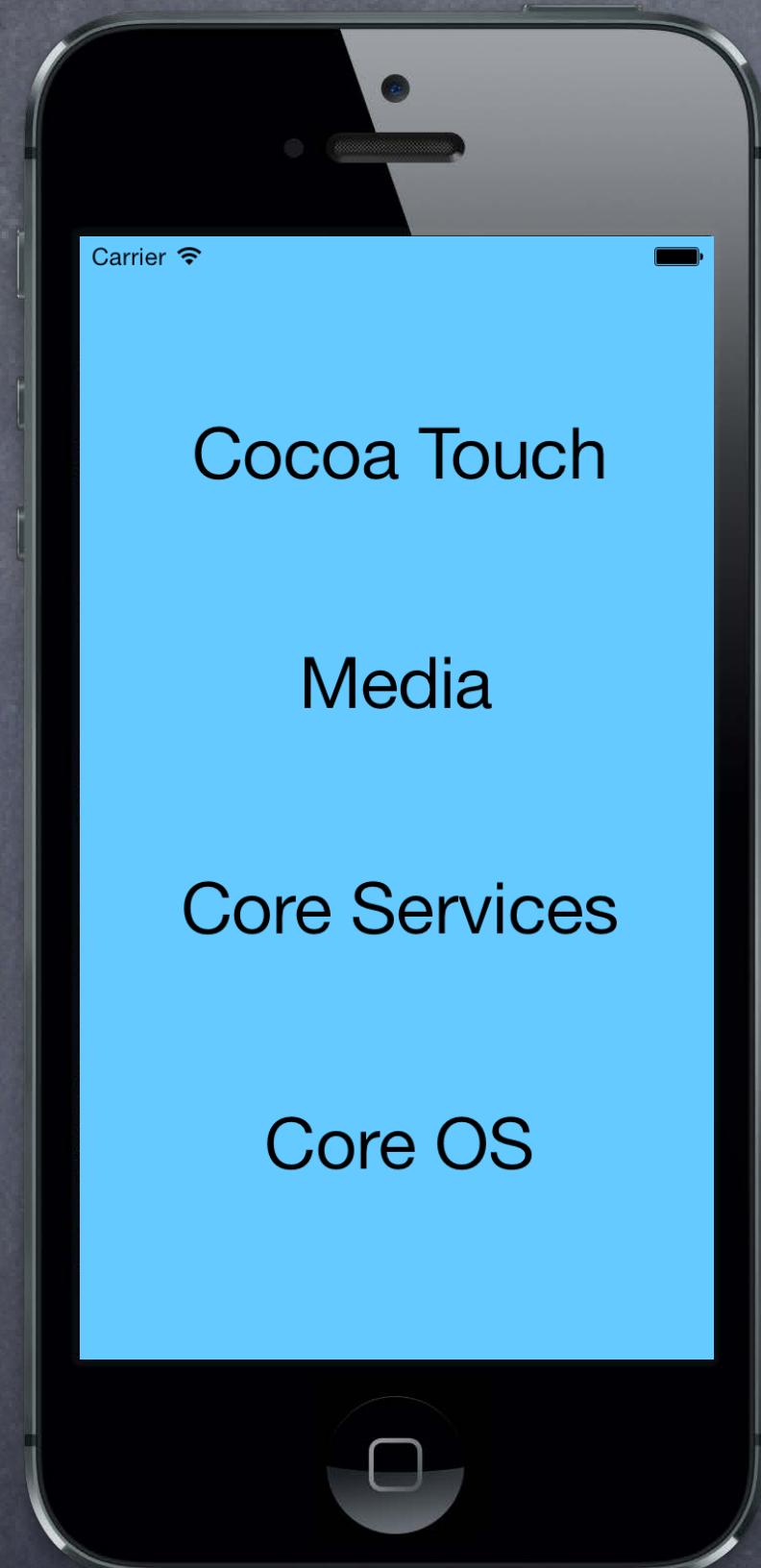
Required Tasks and Evaluation criteria

## ⌚ Final Project

3 weeks to work on it

Individual work only

Keynote presentation required (2 mins or so)



# What's in iOS?

Core OS

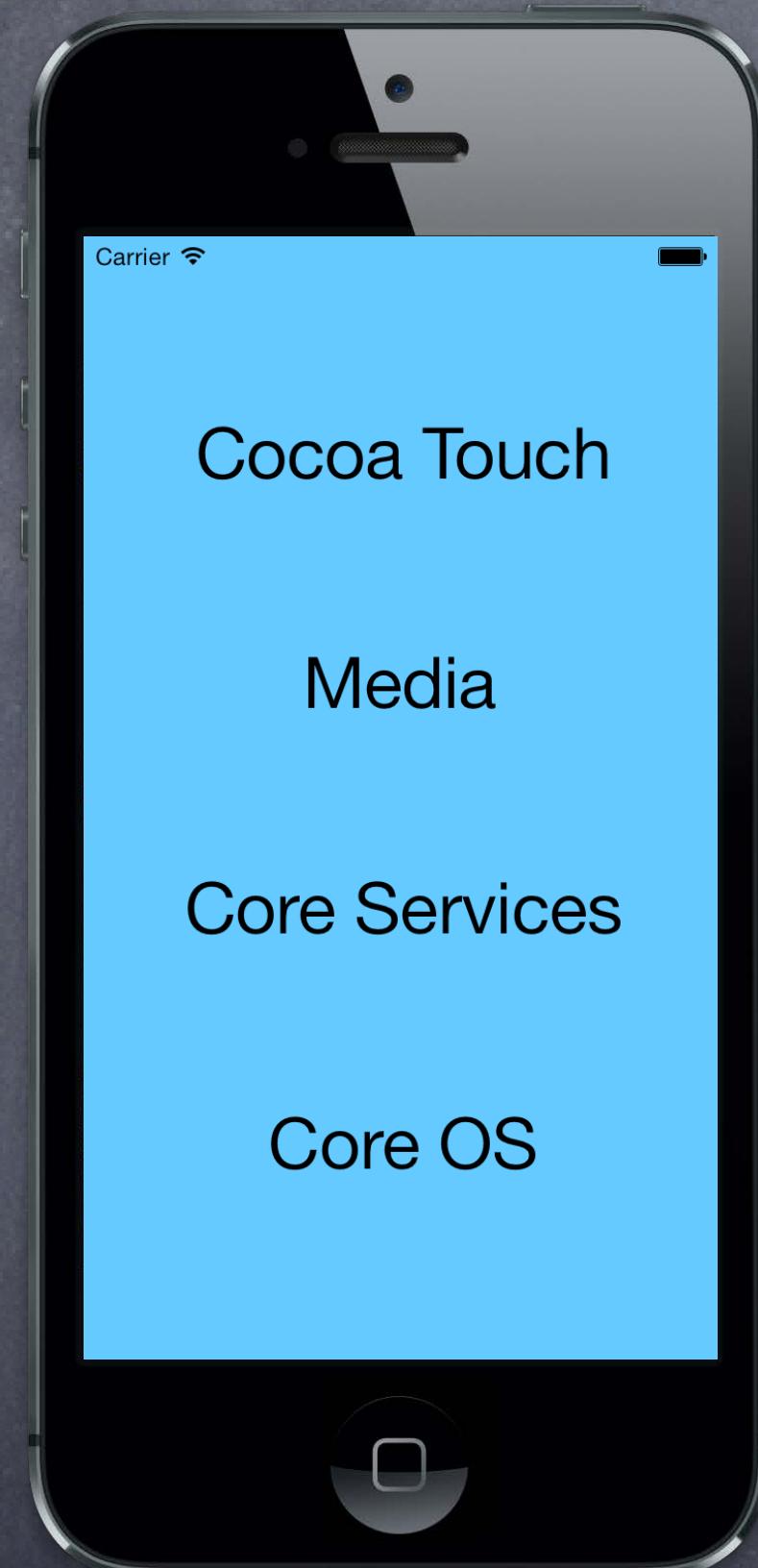
OSX Kernel Power Management

Mach 3.0 Keychain Access

BSD Certificates

Sockets File System

Security Bonjour



# What's in iOS?

## Core Services

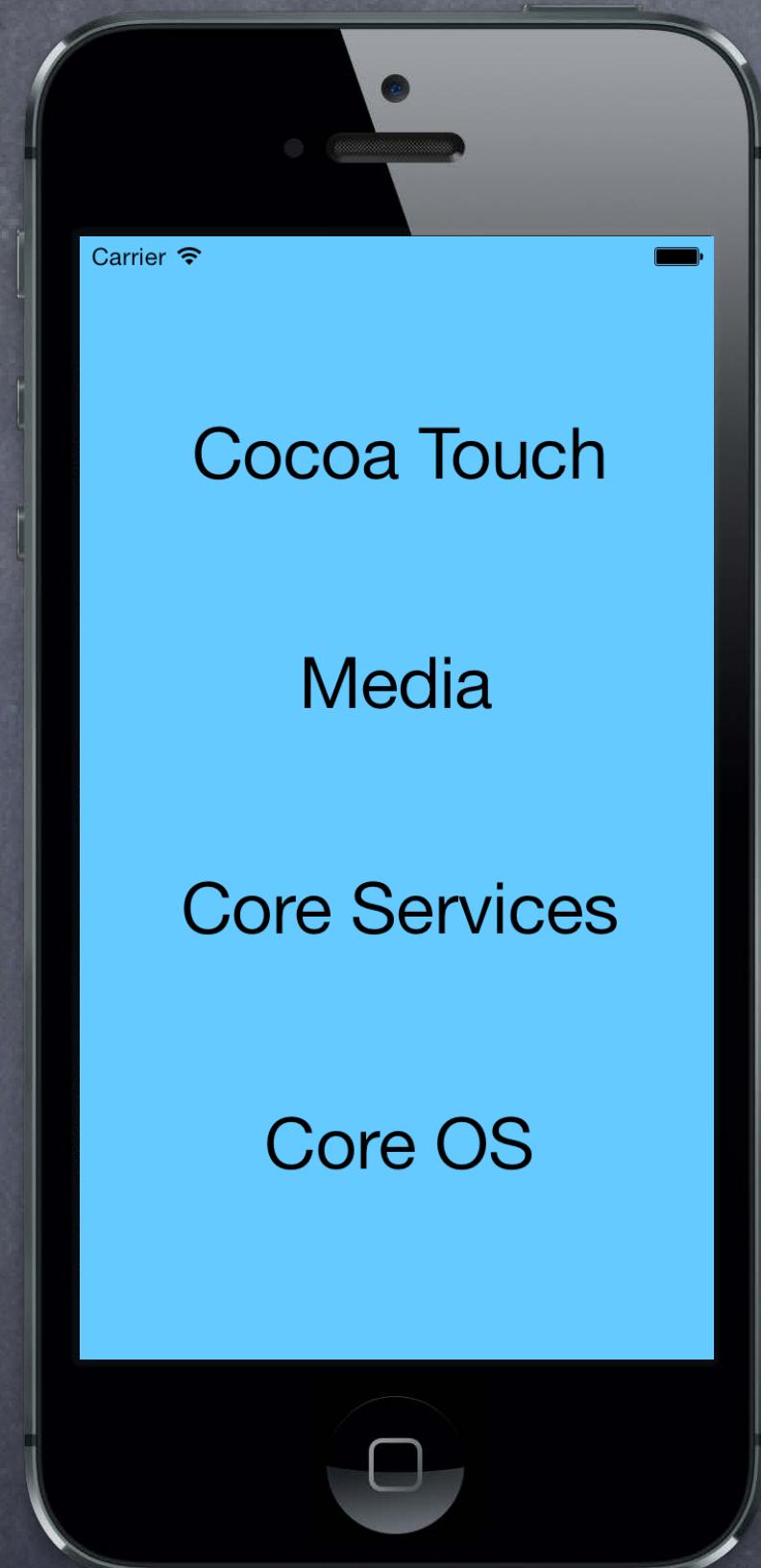
Collections      Core Location

Address Book      Net Services

Networking      Threading

File Access      Preferences

SQLite      URL Utilities



# What's in iOS?

## Media

Core Audio

JPEG, PNG, TIFF

OpenAL

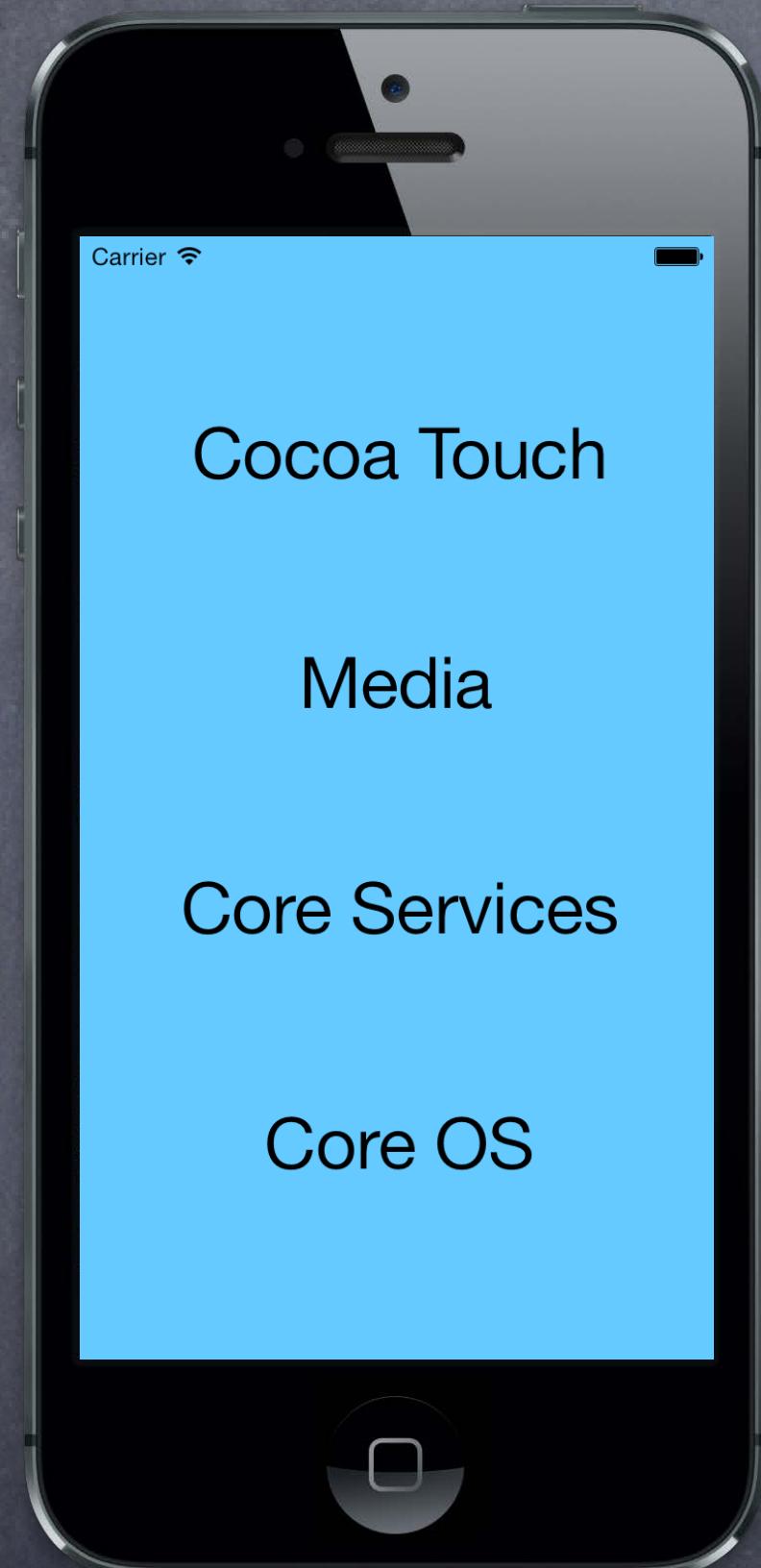
PDF

Audio Mixing

Quartz (2D)

Audio Recording      Core Animation

Video Playback      OpenGL ES



# What's in iOS?

Cocoa Touch

Multi-Touch

Core Motion

View Hierarchy

Localization

Controls

Alerts

Web View

Map Kit

Image Picker

Camera

# Platform Components

- Tools



Xcode 5



Instruments

- Language

[display setTextColor:[UIColor blackColor]];

- Frameworks



Foundation

Core Data



UIKit

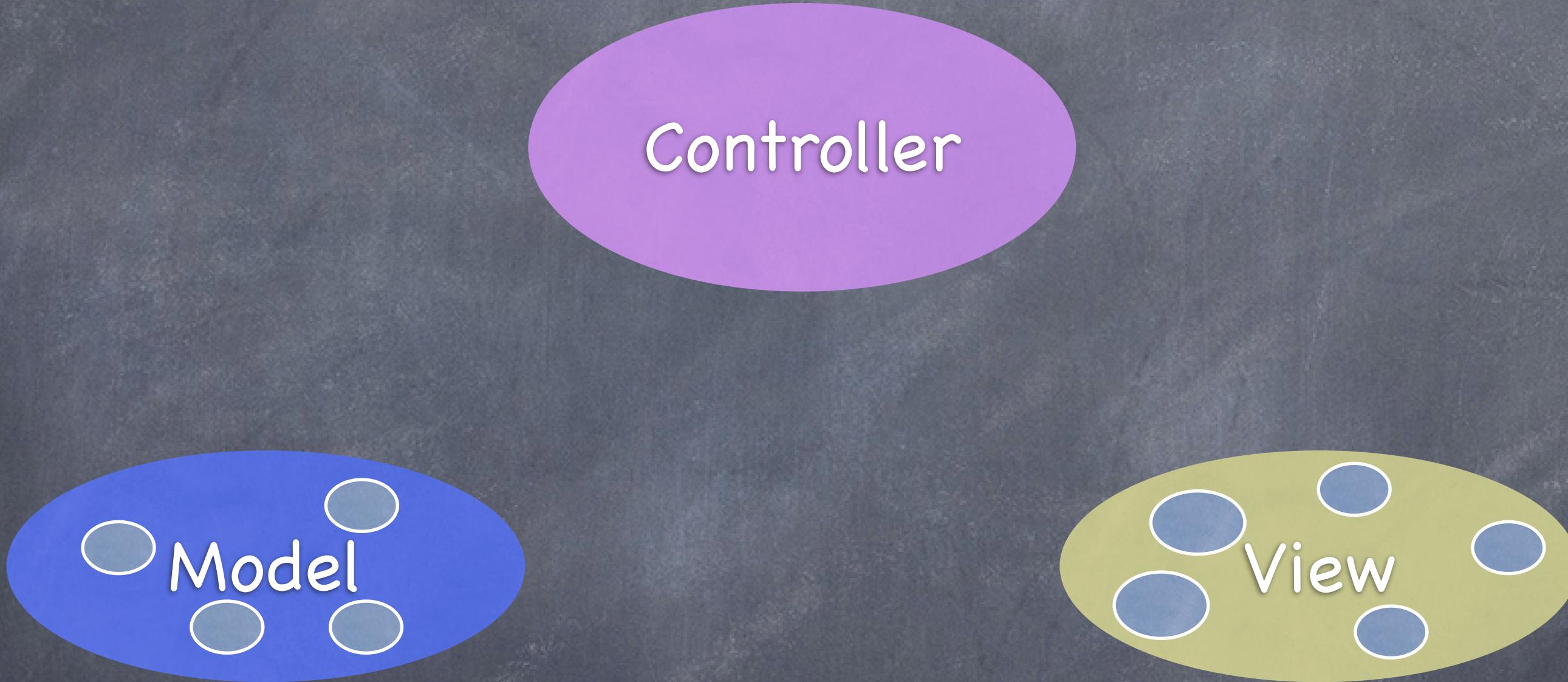
Core Motion

Map Kit

- Design Strategies

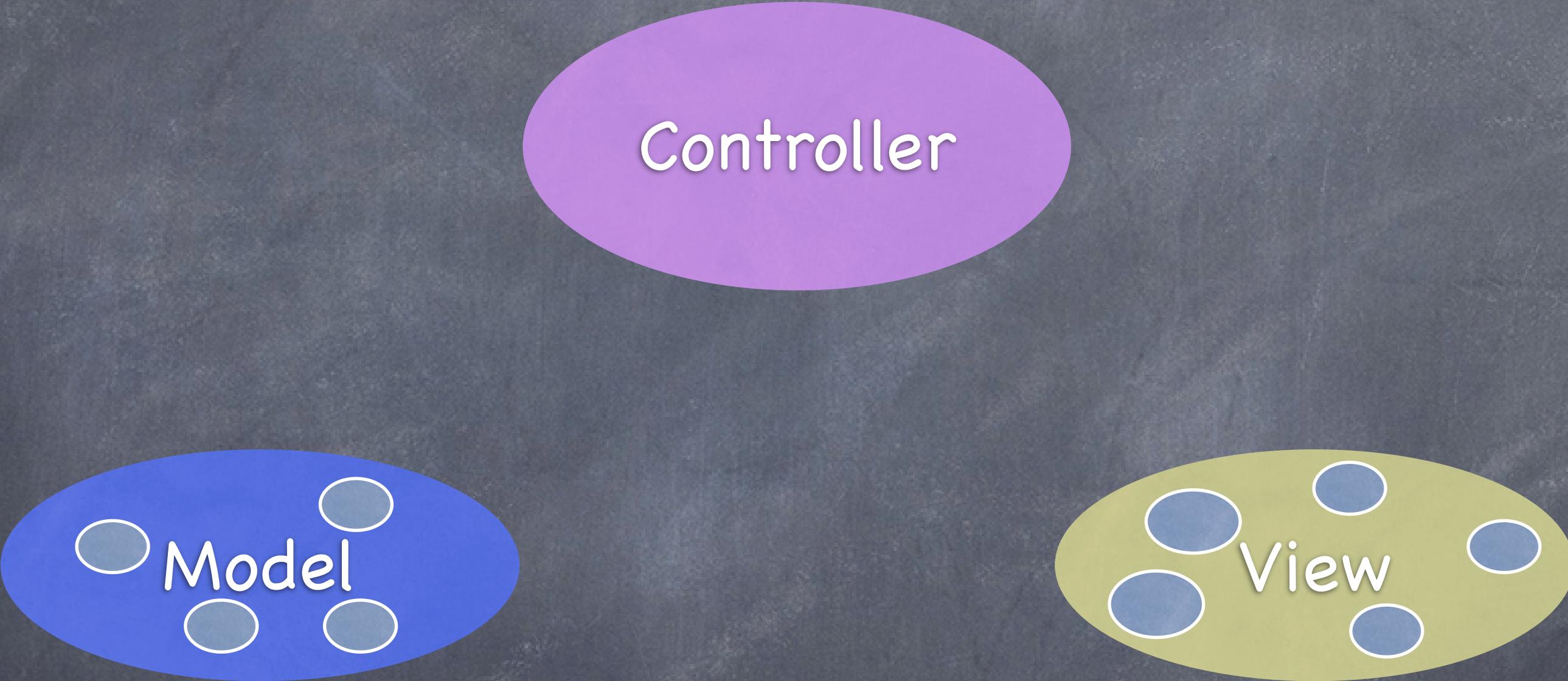
MVC

# MVC



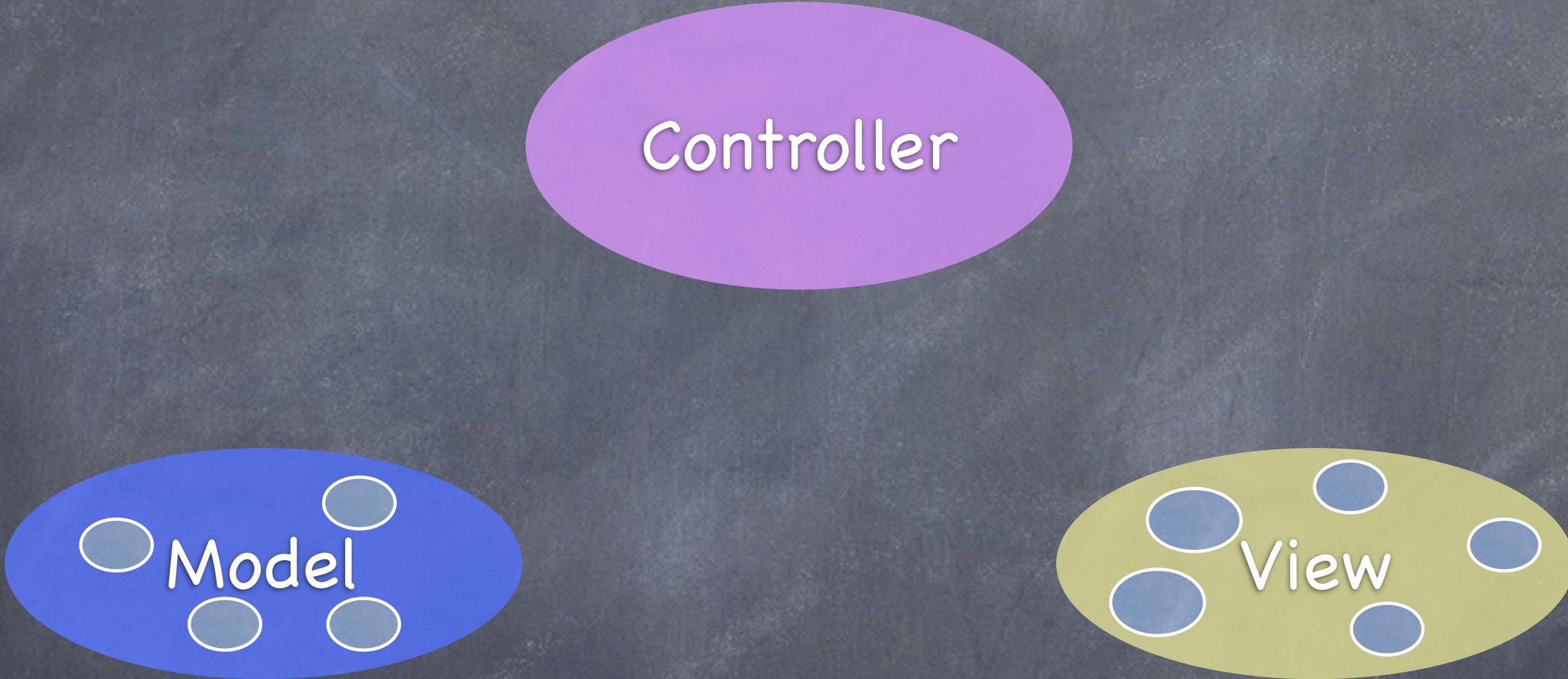
Divide objects in your program into 3 “camps.”

# MVC



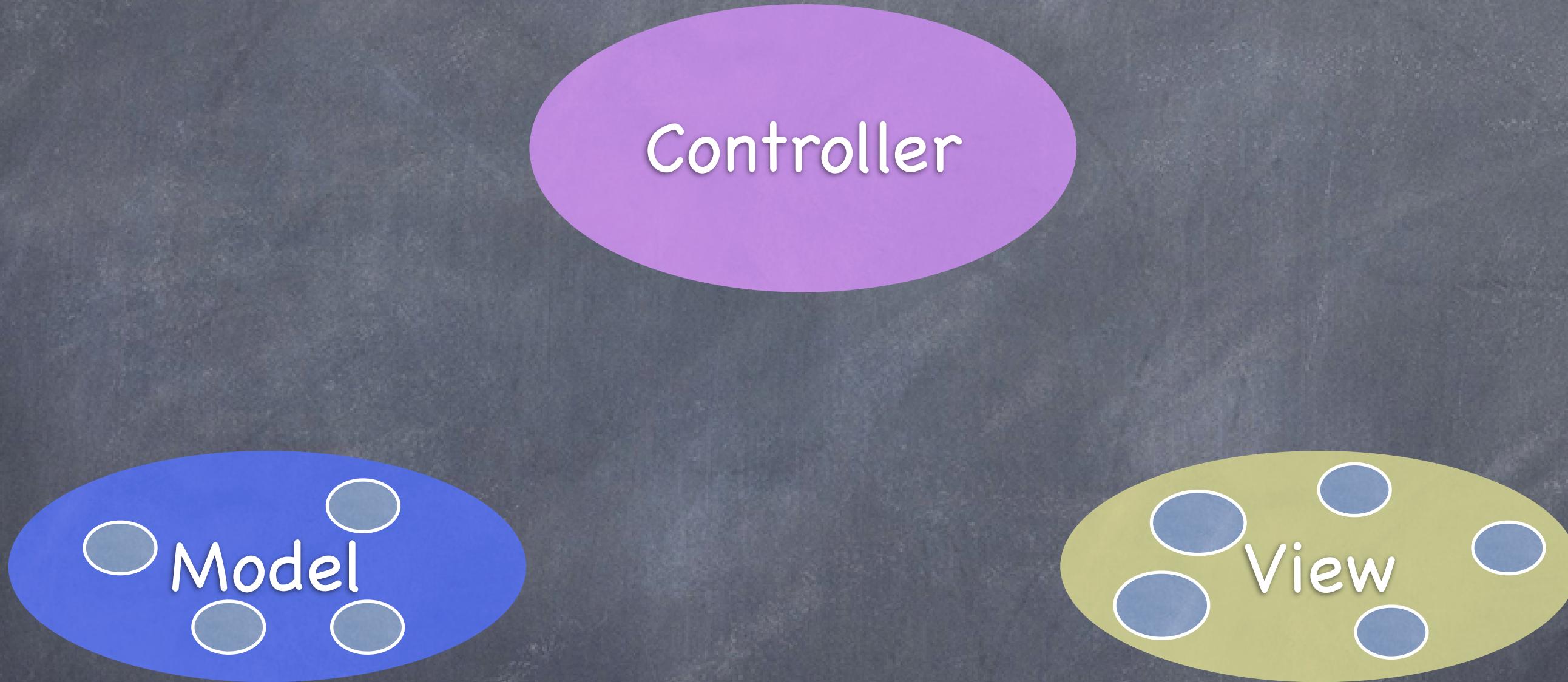
Model = What your application is (but not how it is displayed)

# MVC



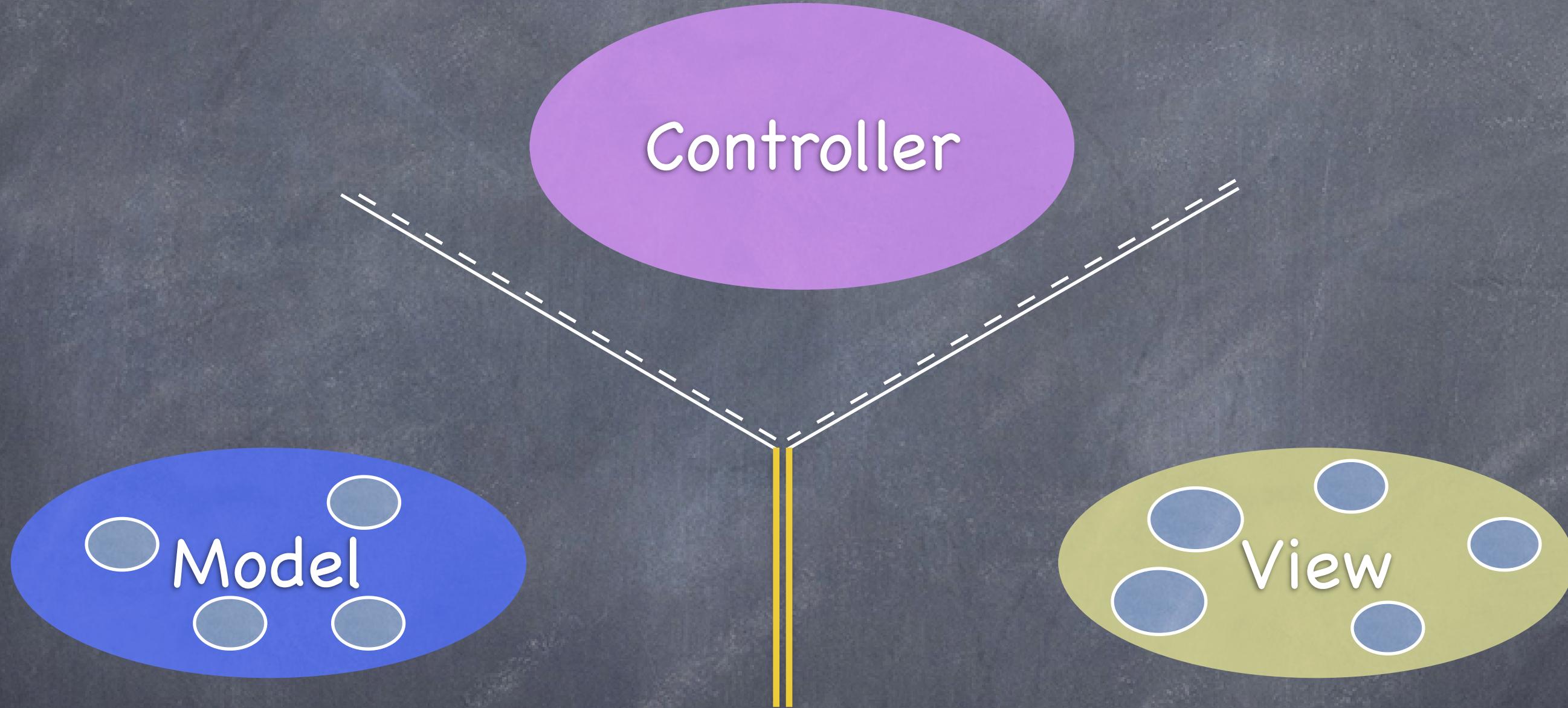
Controller = How your Model is presented to the user (UI logic)

# MVC



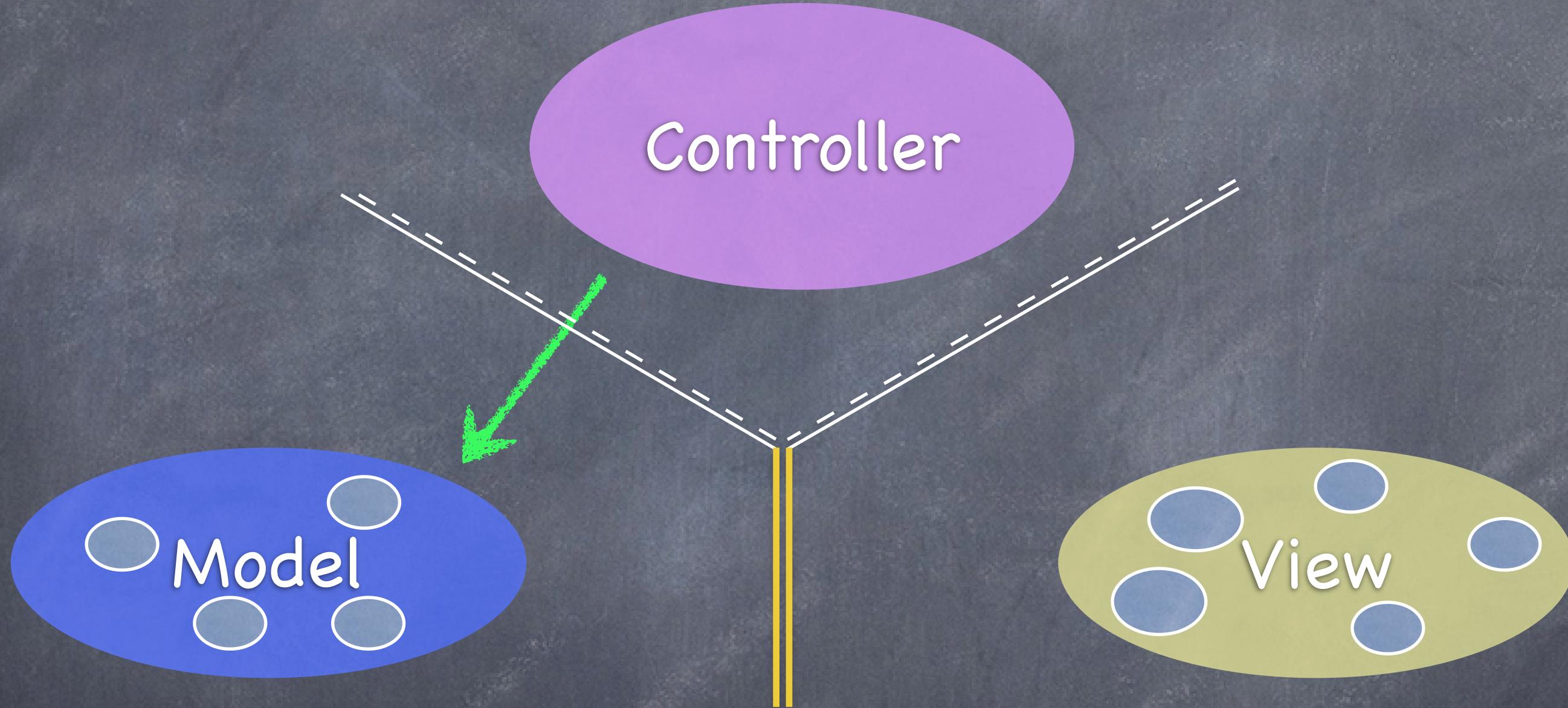
**View** = Your **Controller's** minions

# MVC



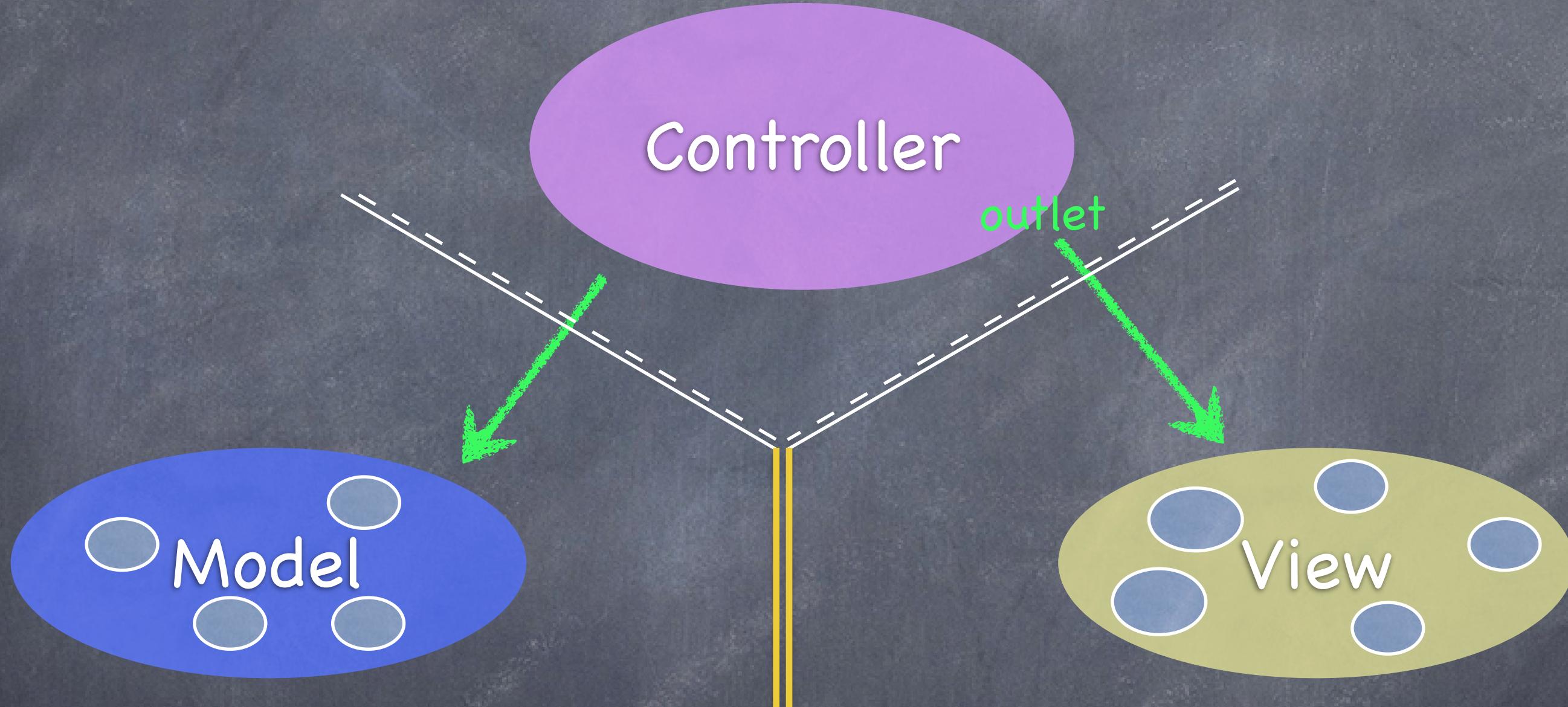
It's all about managing communication between camps

# MVC



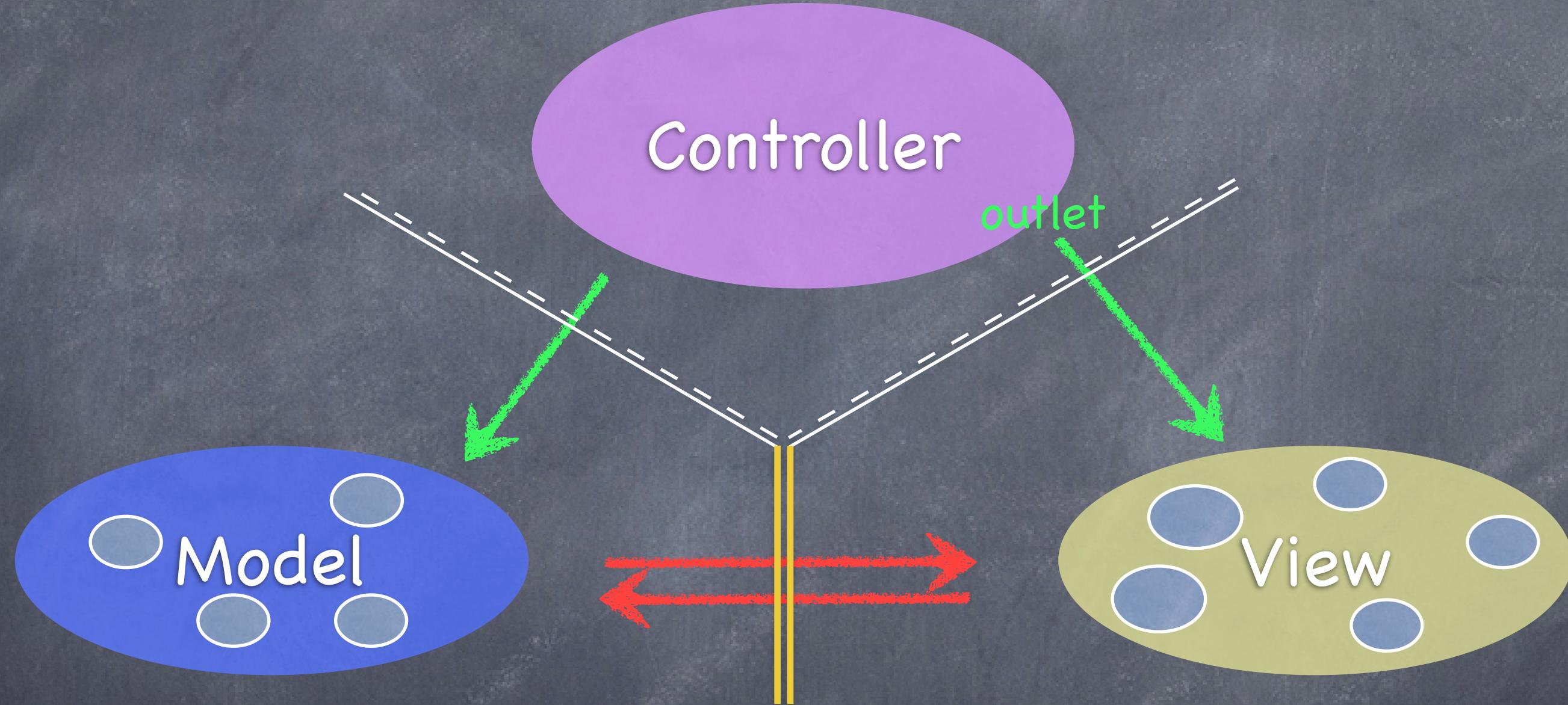
Controllers can always talk directly to their Model.

# MVC



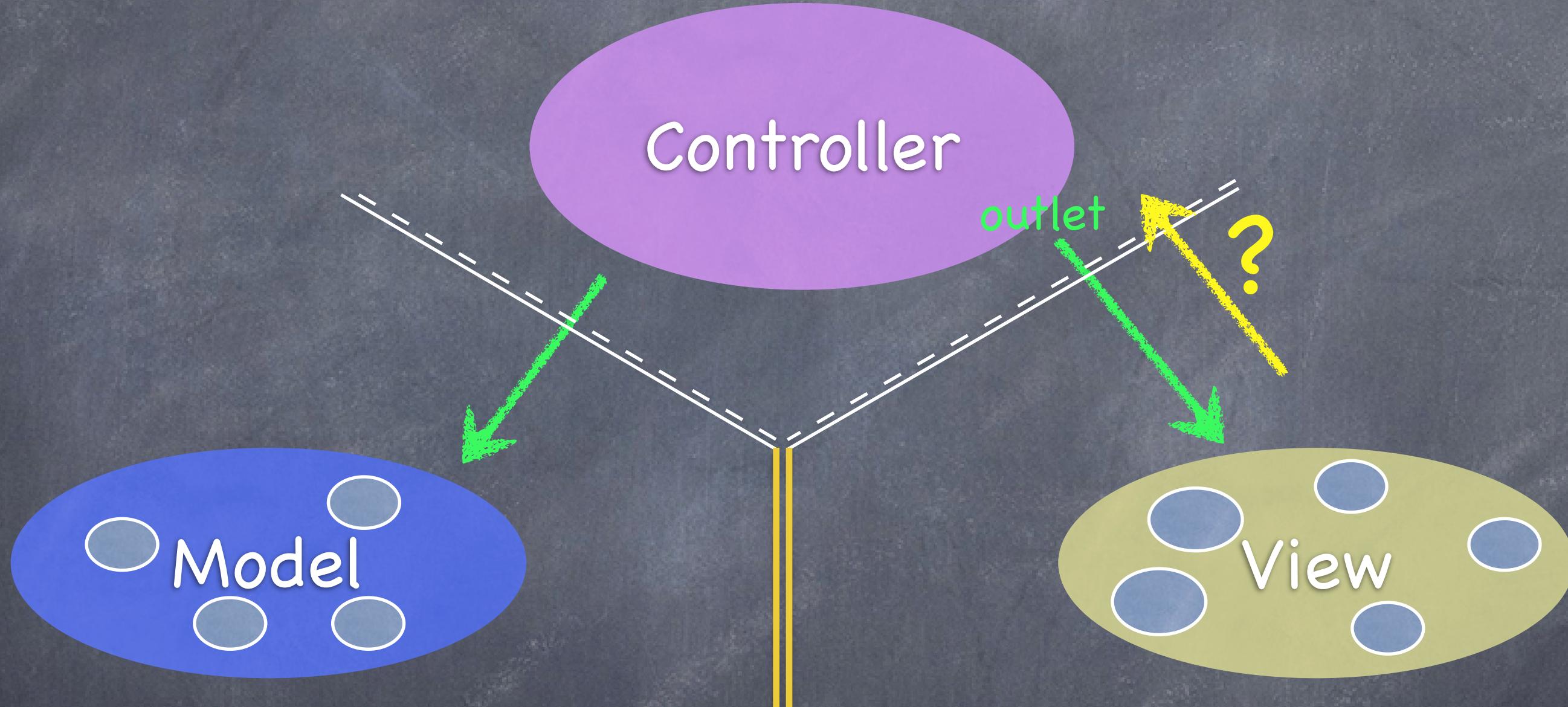
Controllers can also talk directly to their View.

# MVC



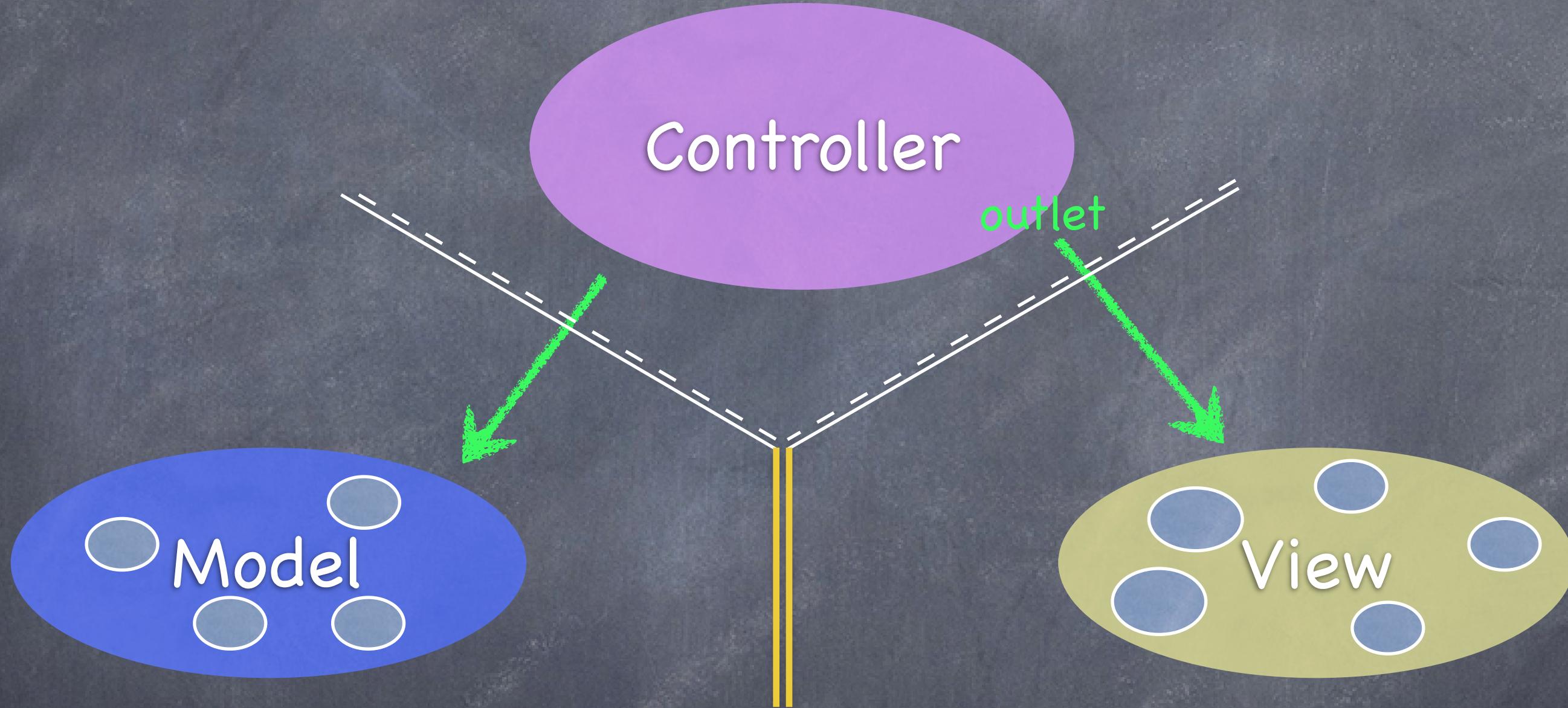
The Model and View should never speak to each other.

# MVC



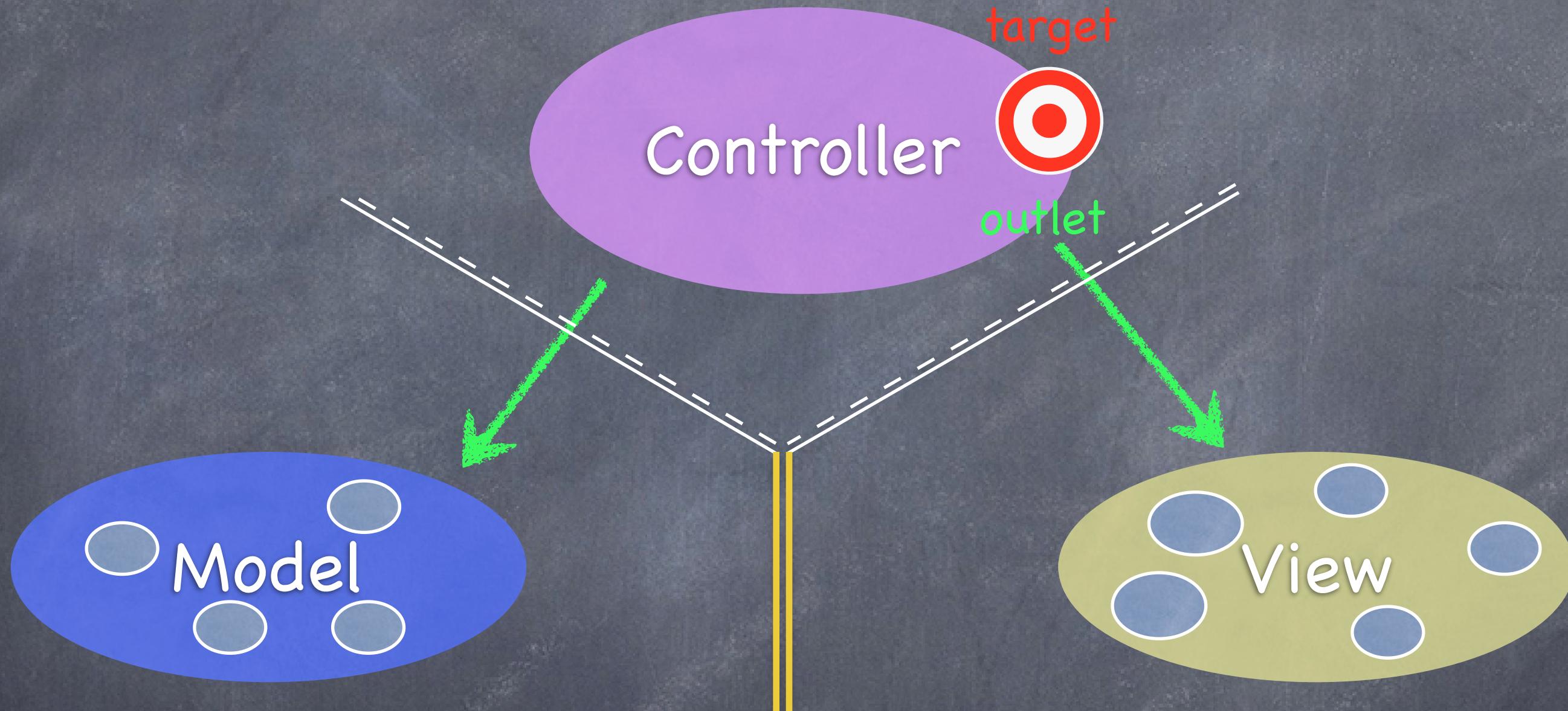
Can the **View** speak to its **Controller**?

# MVC



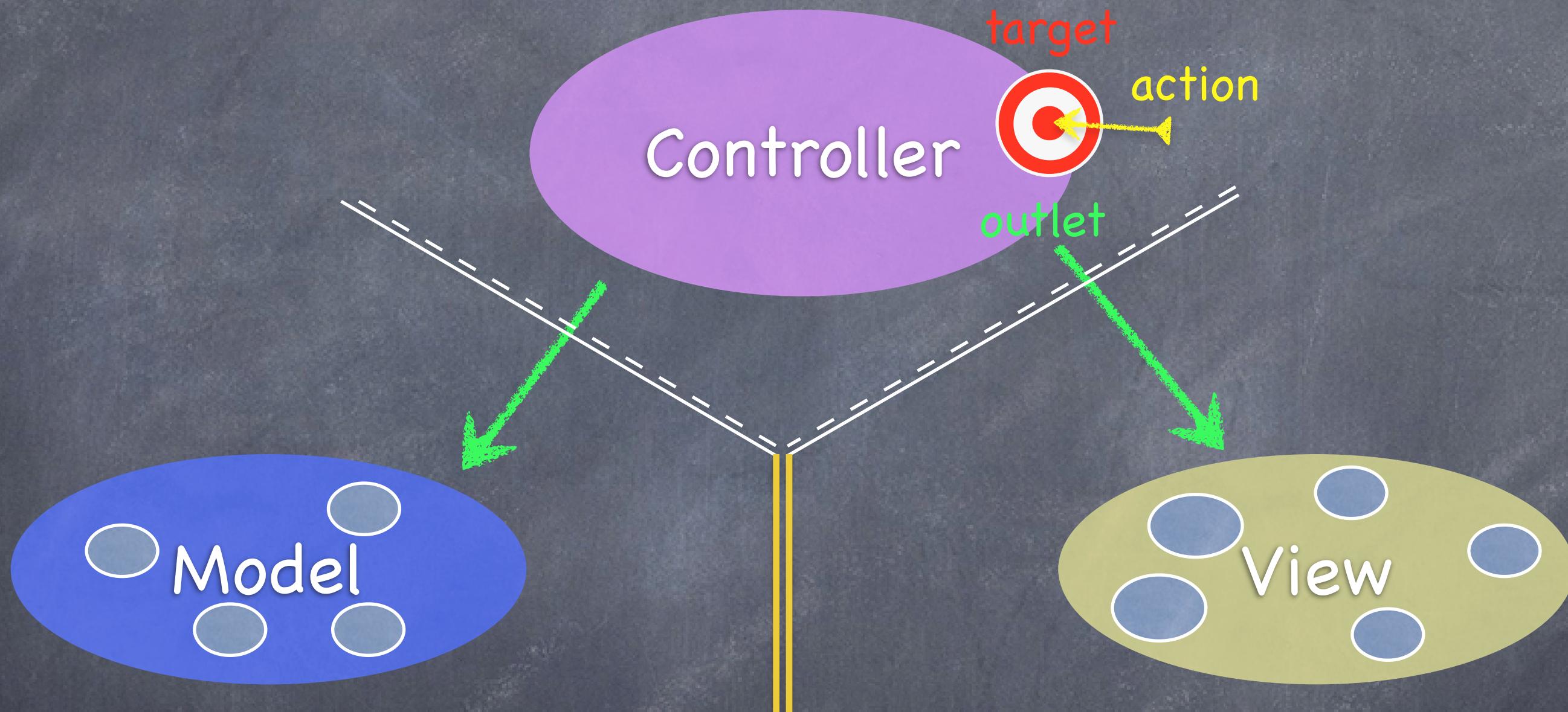
Sort of. Communication is “blind” and structured.

# MVC



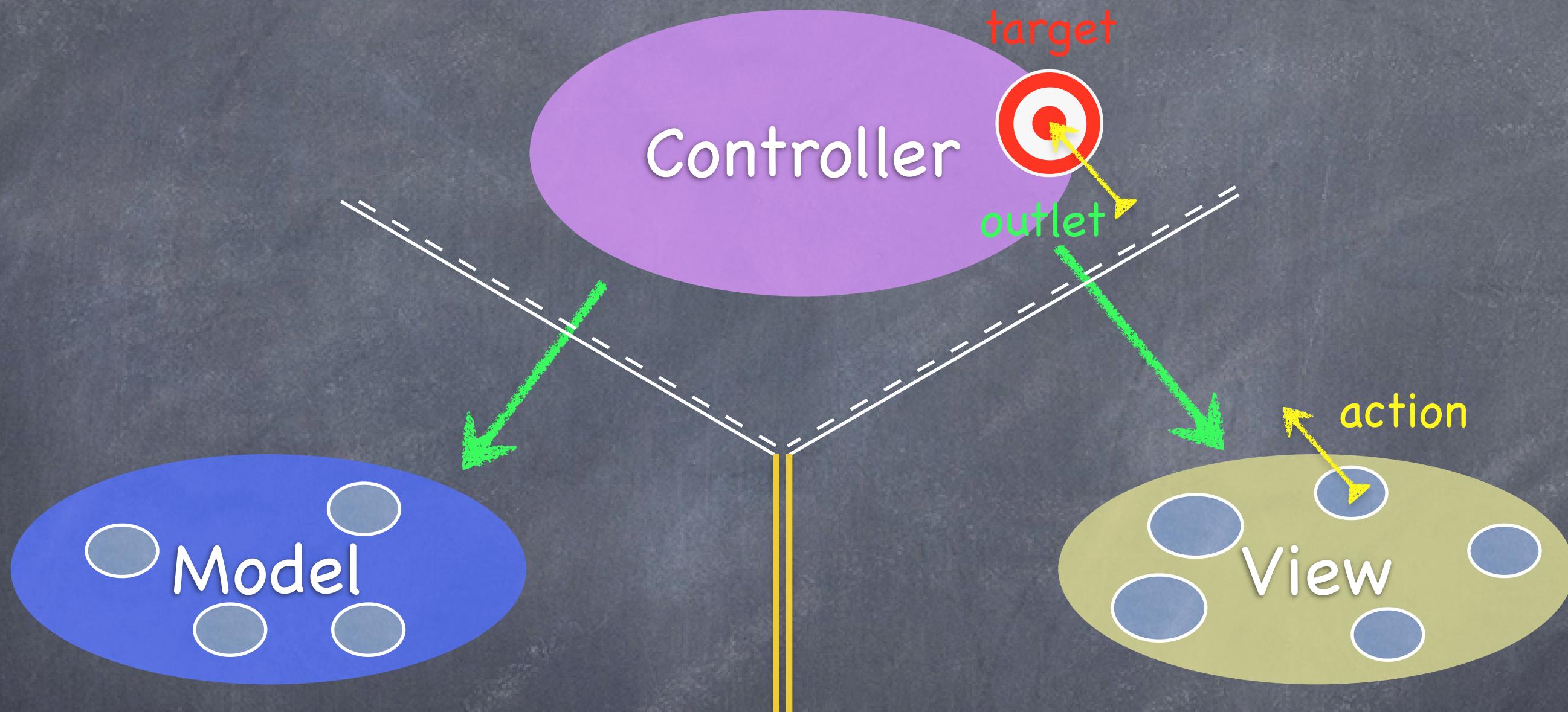
The Controller can drop a target on itself.

# MVC



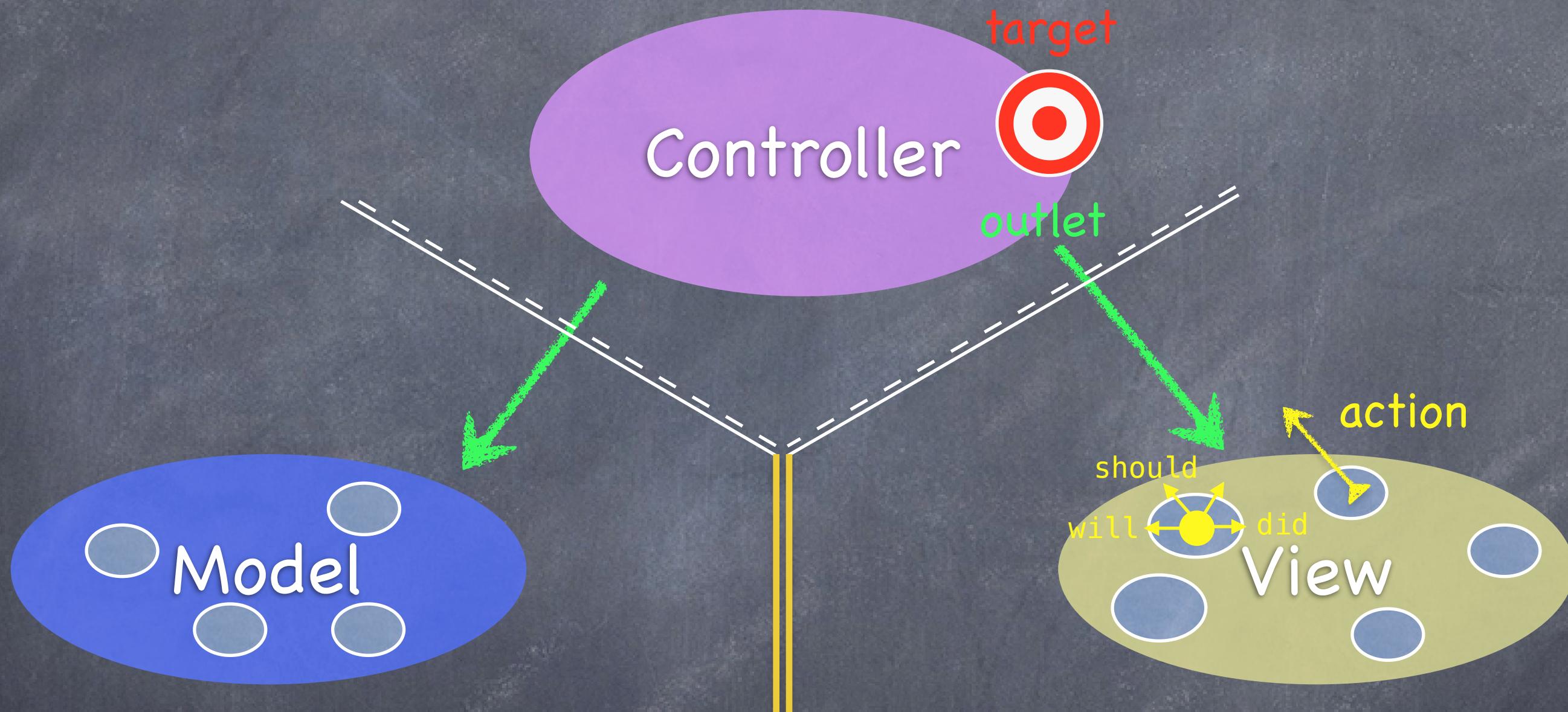
Then hand out an **action** to the View.

# MVC



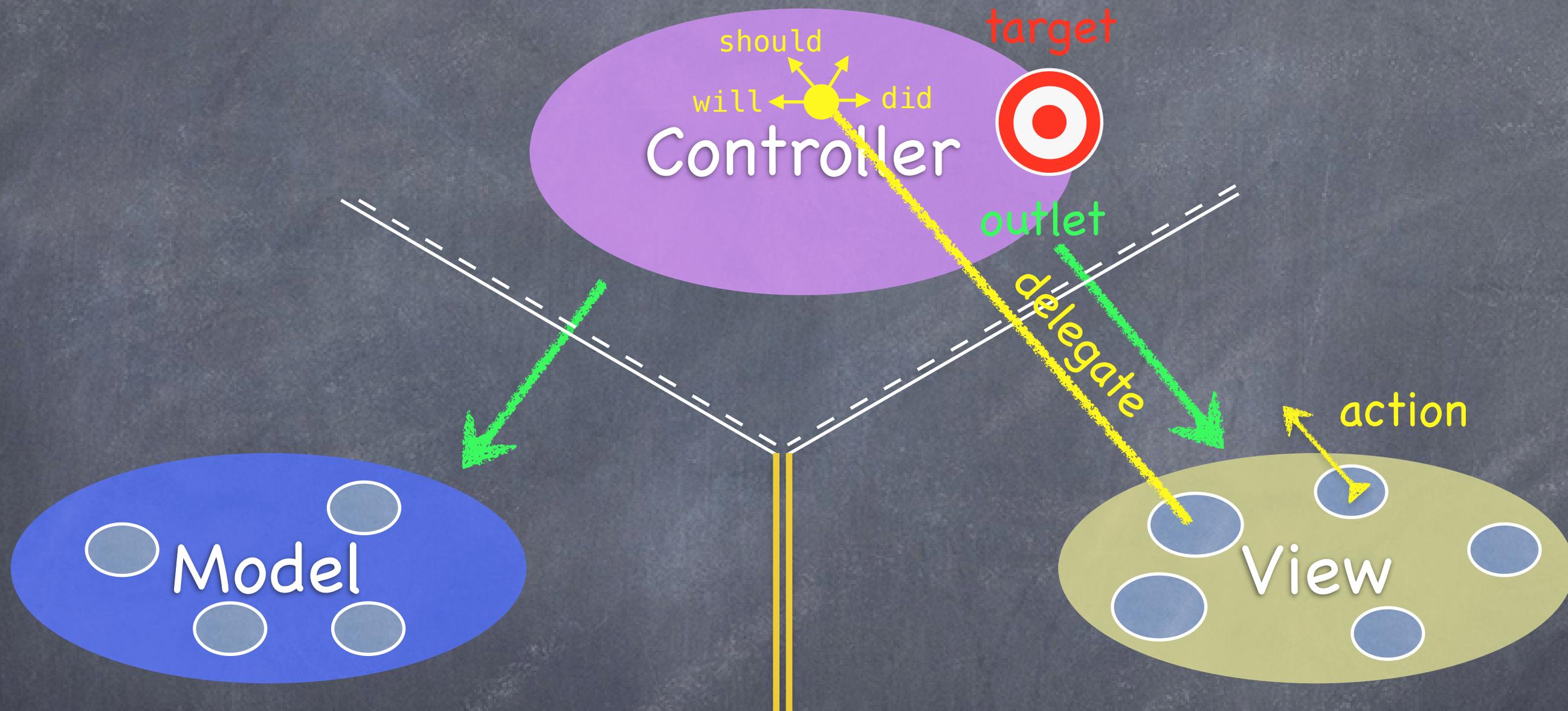
The View sends the **action** when things happen in the UI.

# MVC



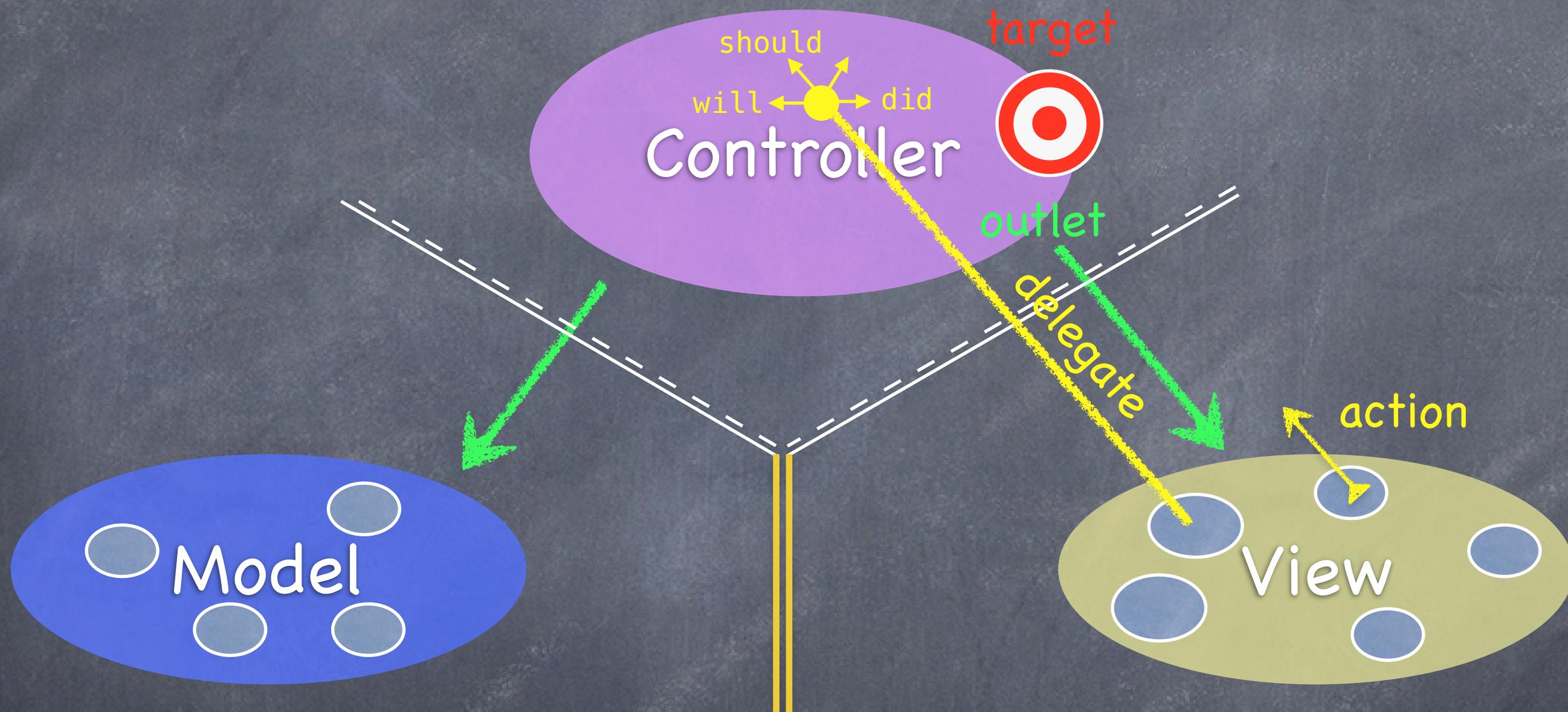
Sometimes the **View** needs to synchronize with the **Controller**.

# MVC



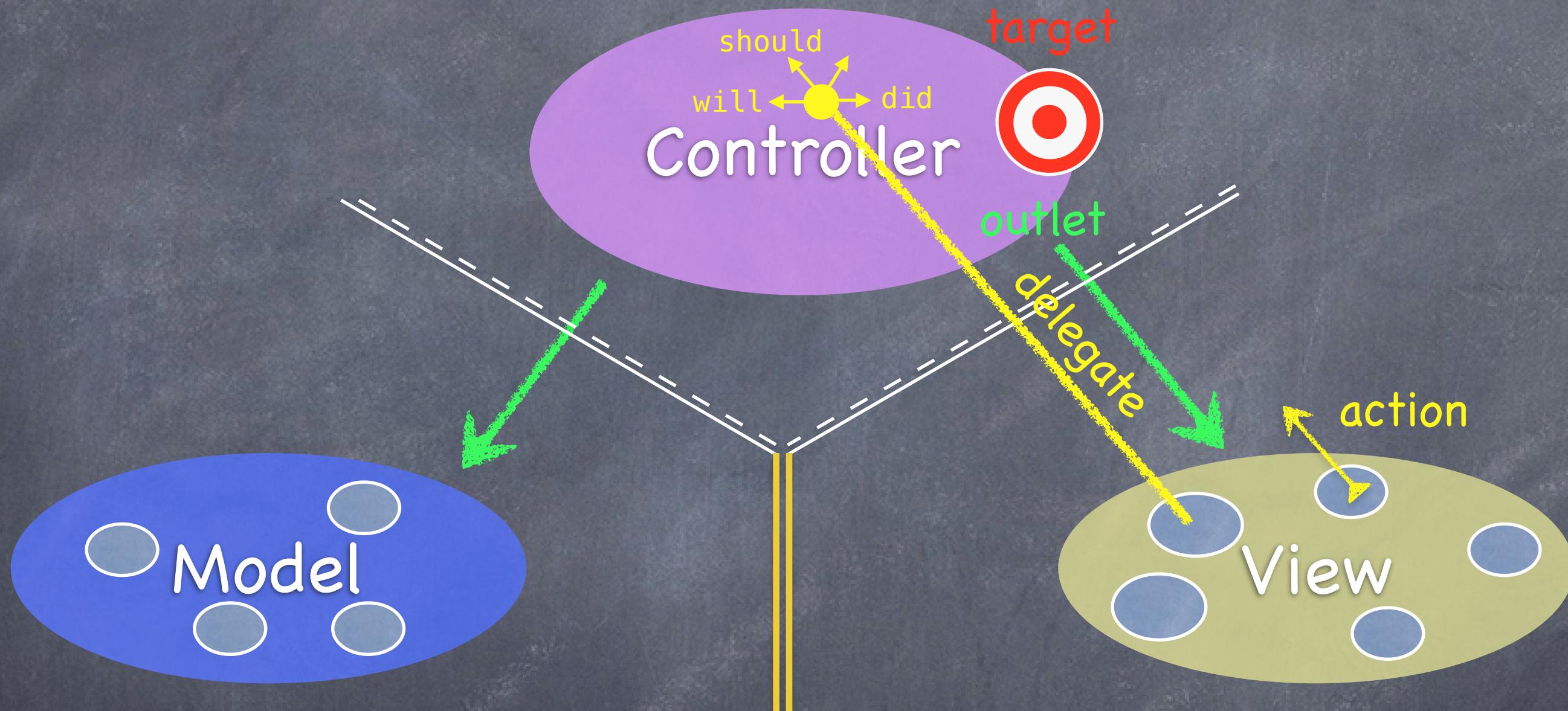
The Controller sets itself as the View's delegate.

# MVC



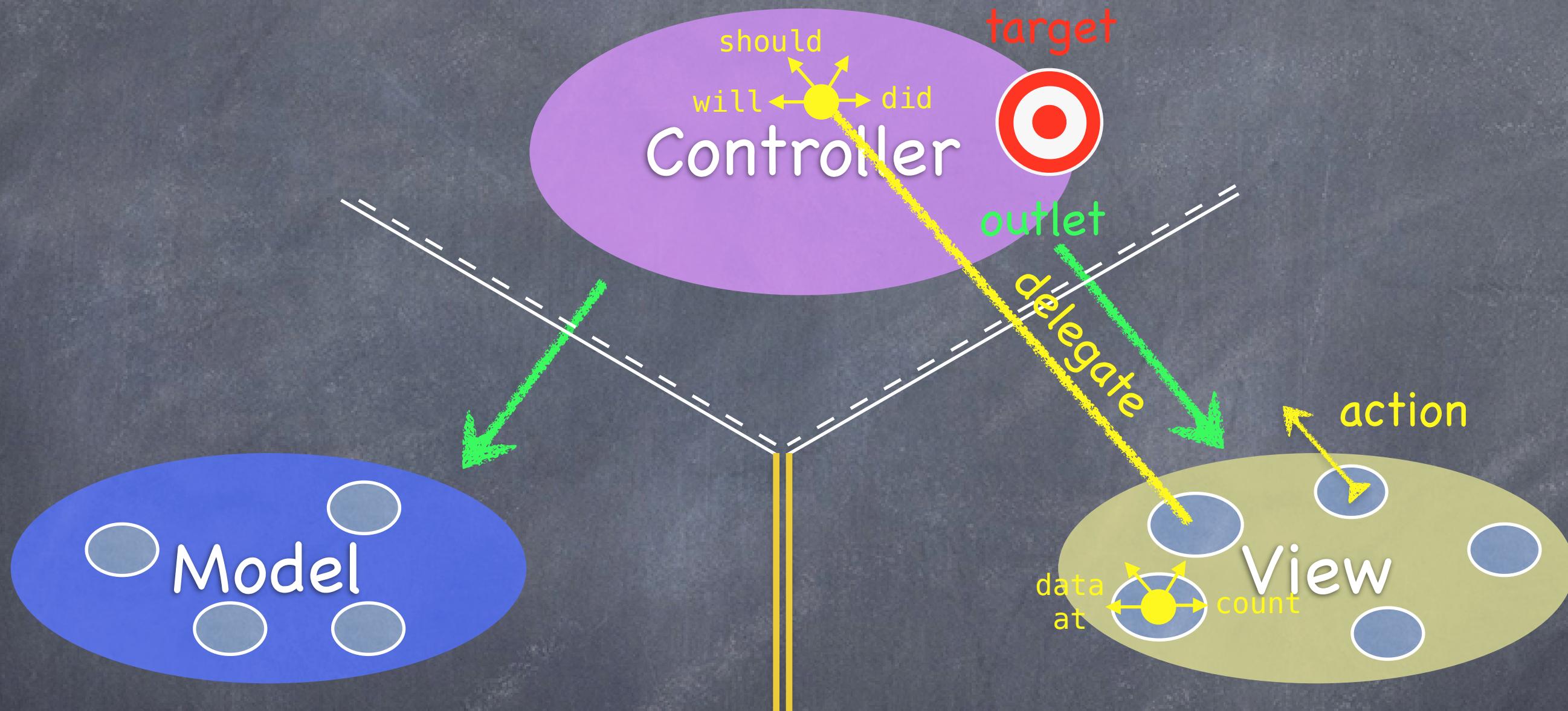
The **delegate** is set via a protocol (i.e. it's “blind” to class).

# MVC



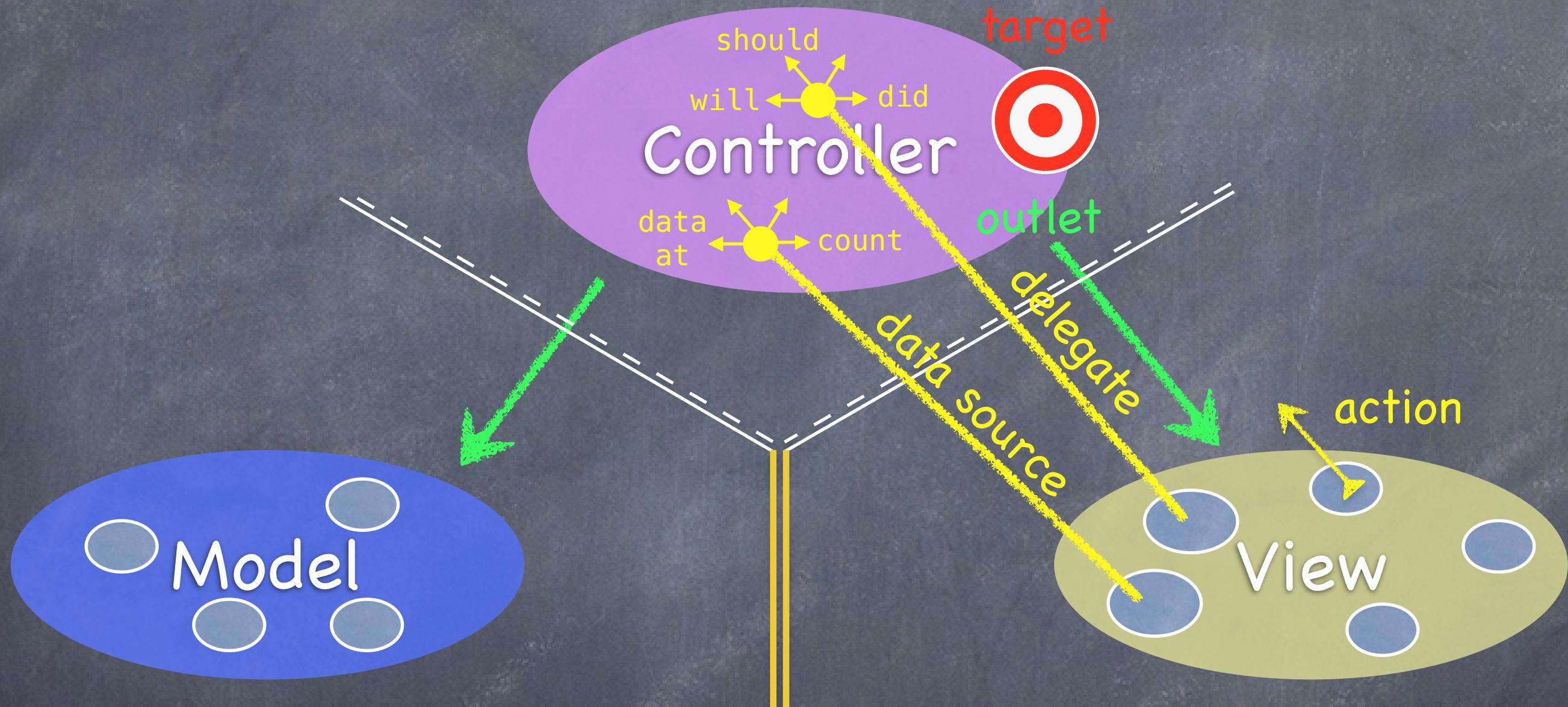
Views do not own the data they display.

# MVC



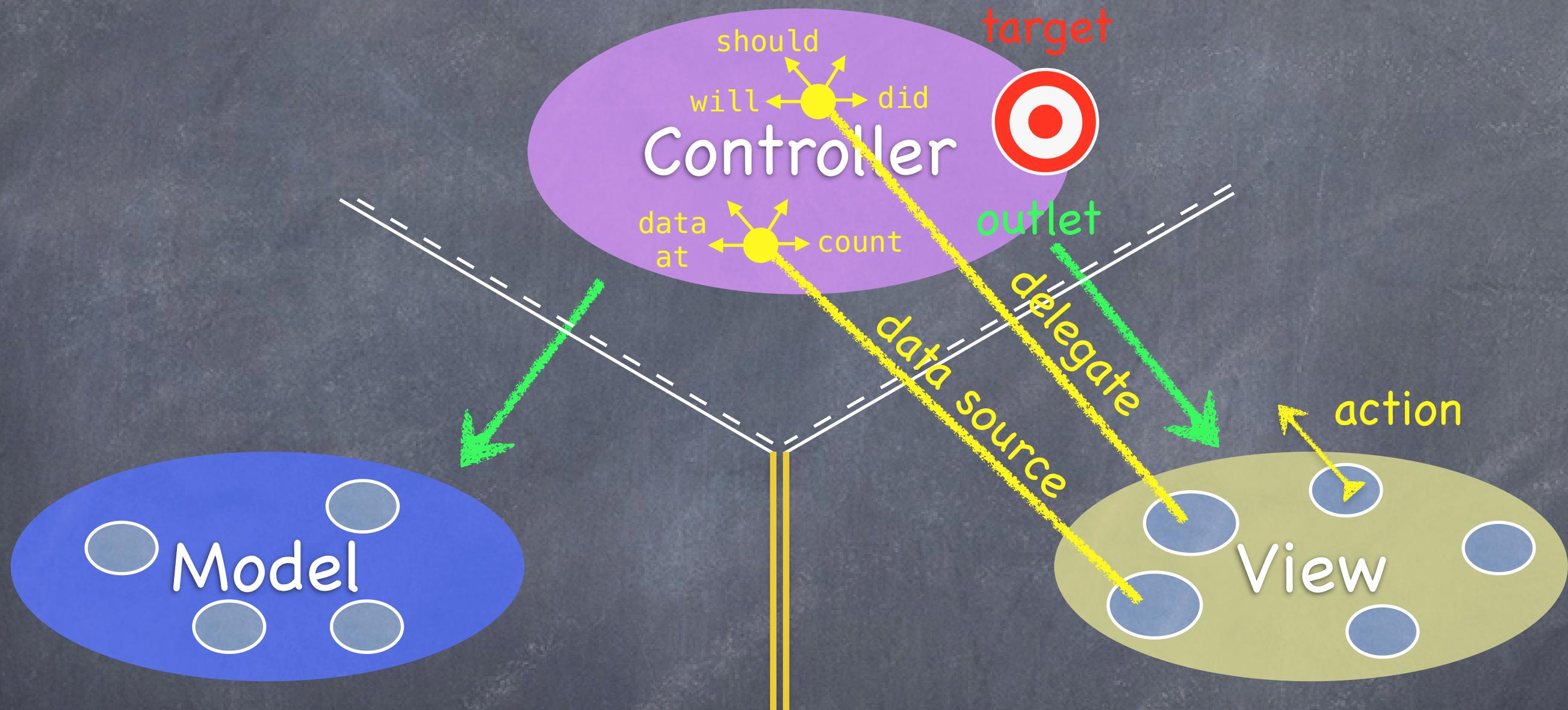
So, if needed, they have a protocol to acquire it.

# MVC



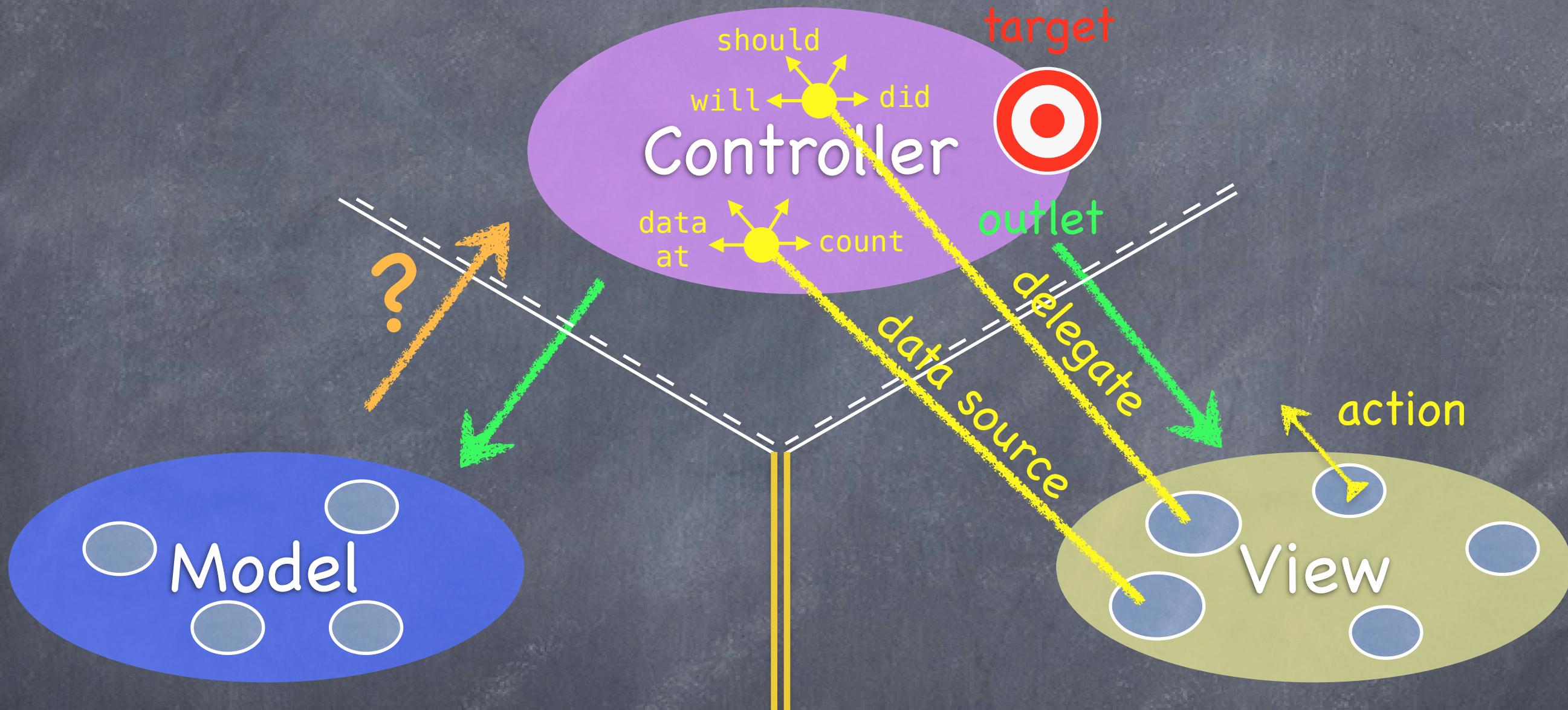
Controllers are almost always that data source (not Model!).

# MVC



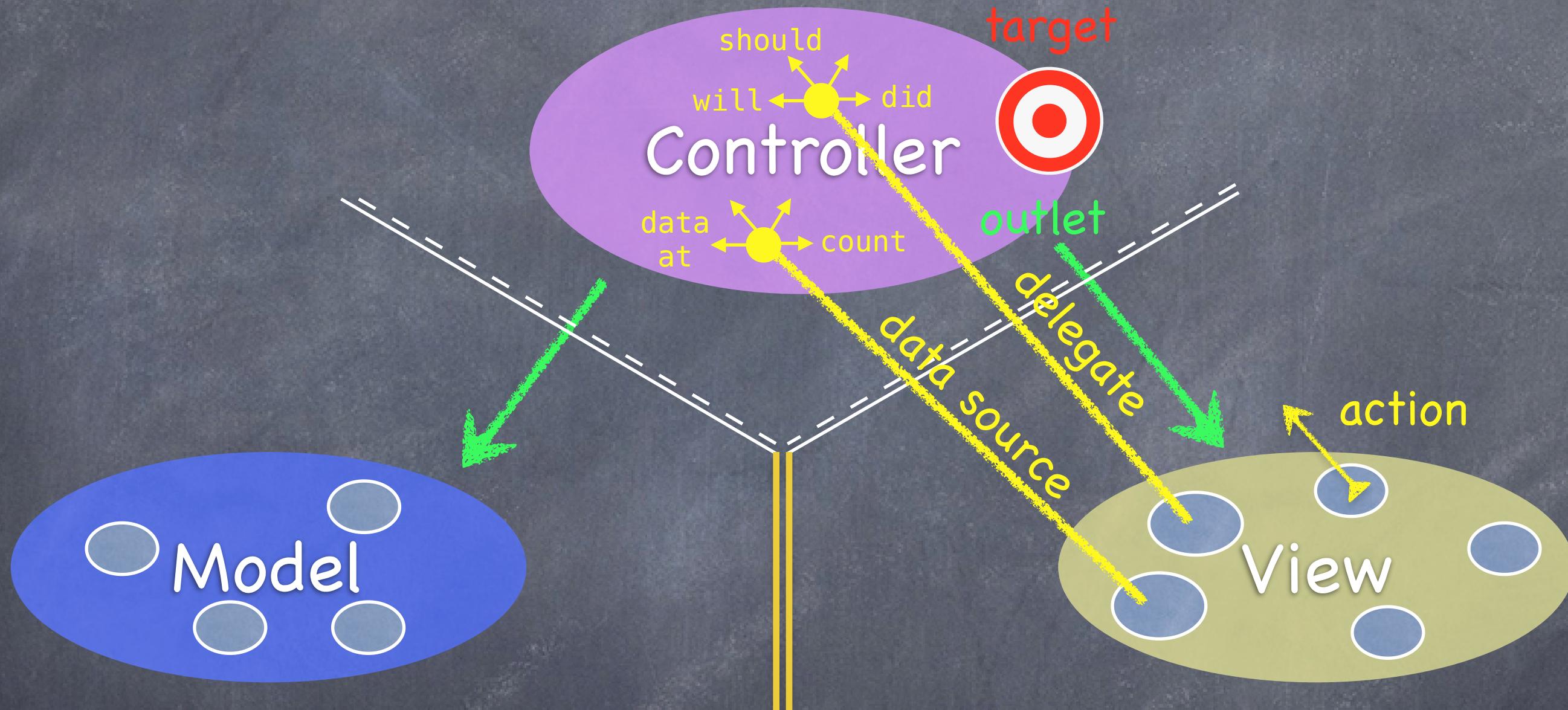
Controllers interpret/format Model information for the View.

# MVC



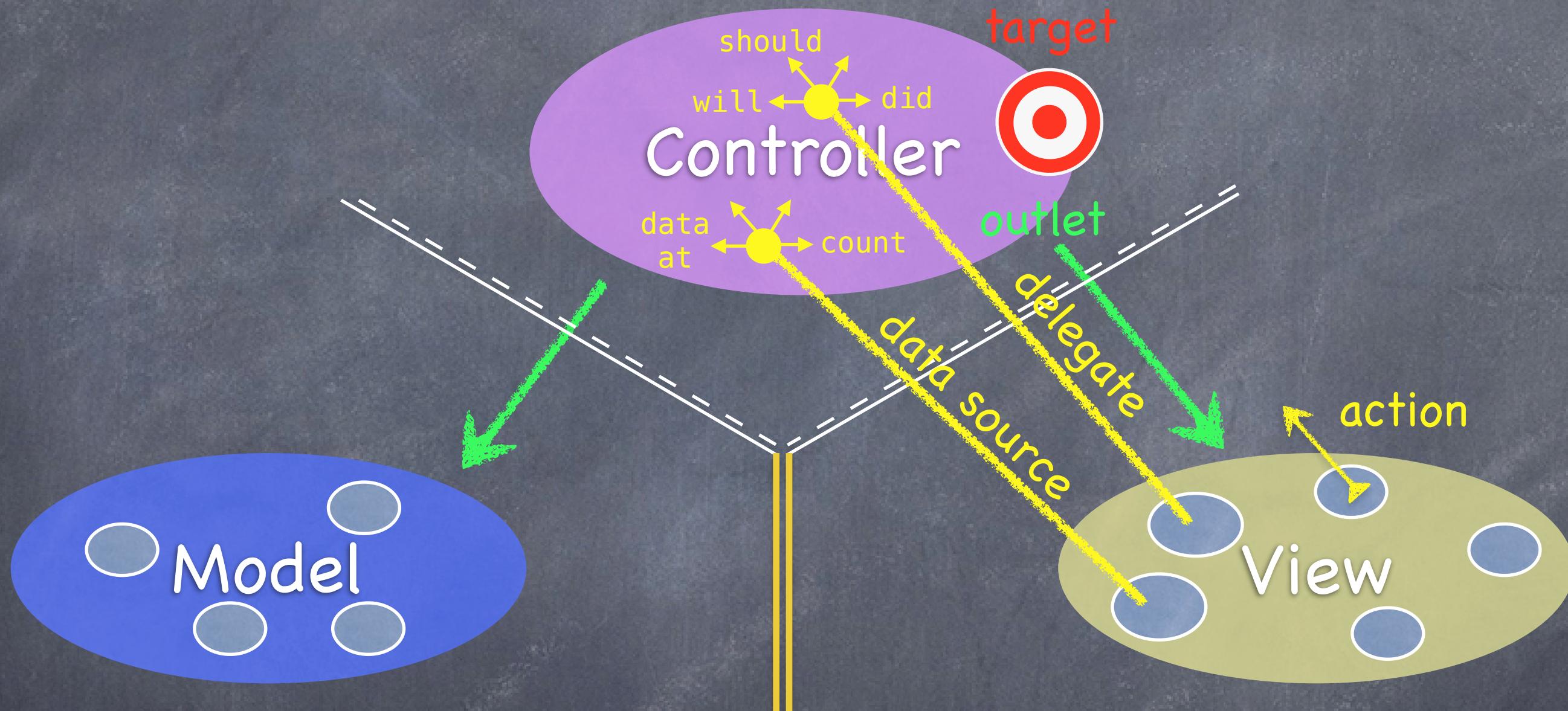
Can the Model talk directly to the Controller?

# MVC



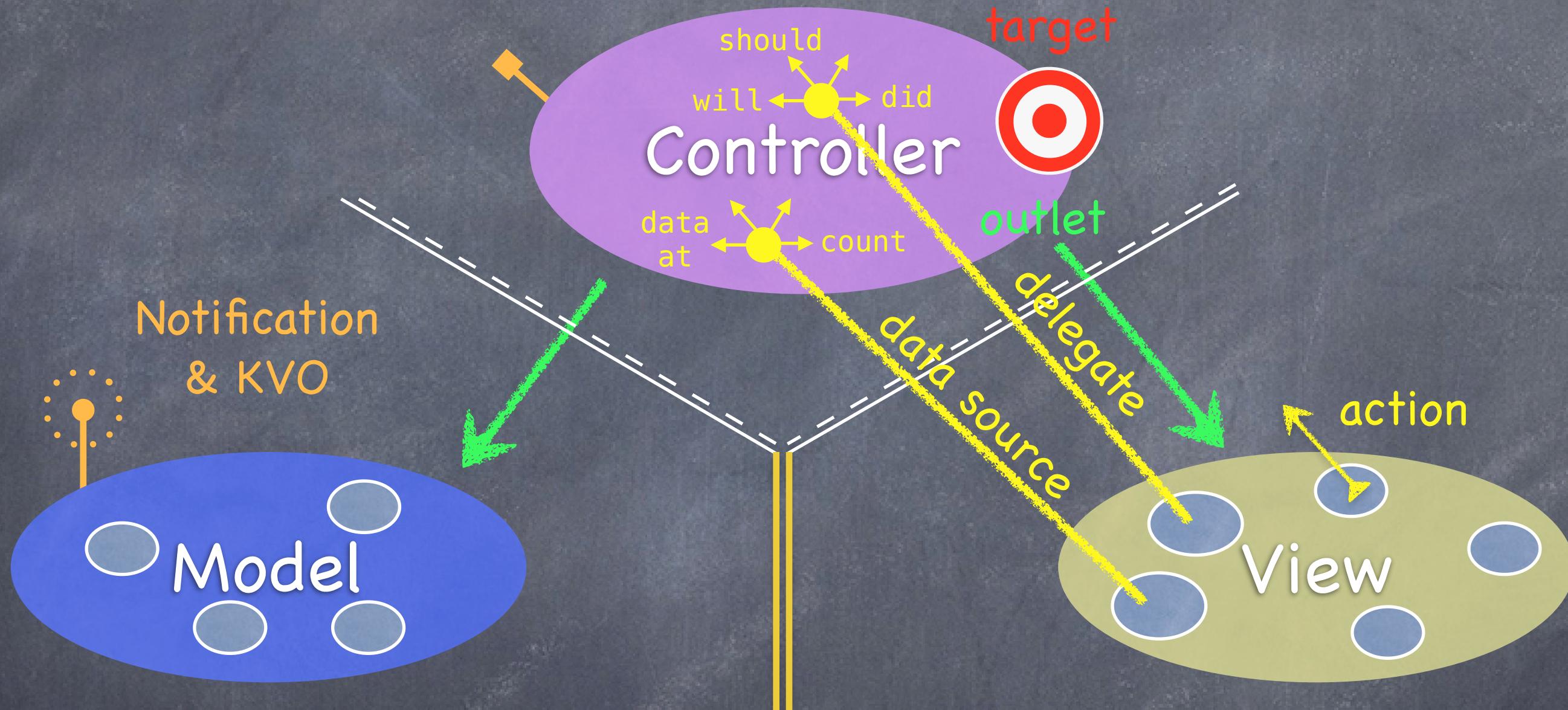
No. The Model is (should be) UI independent.

# MVC



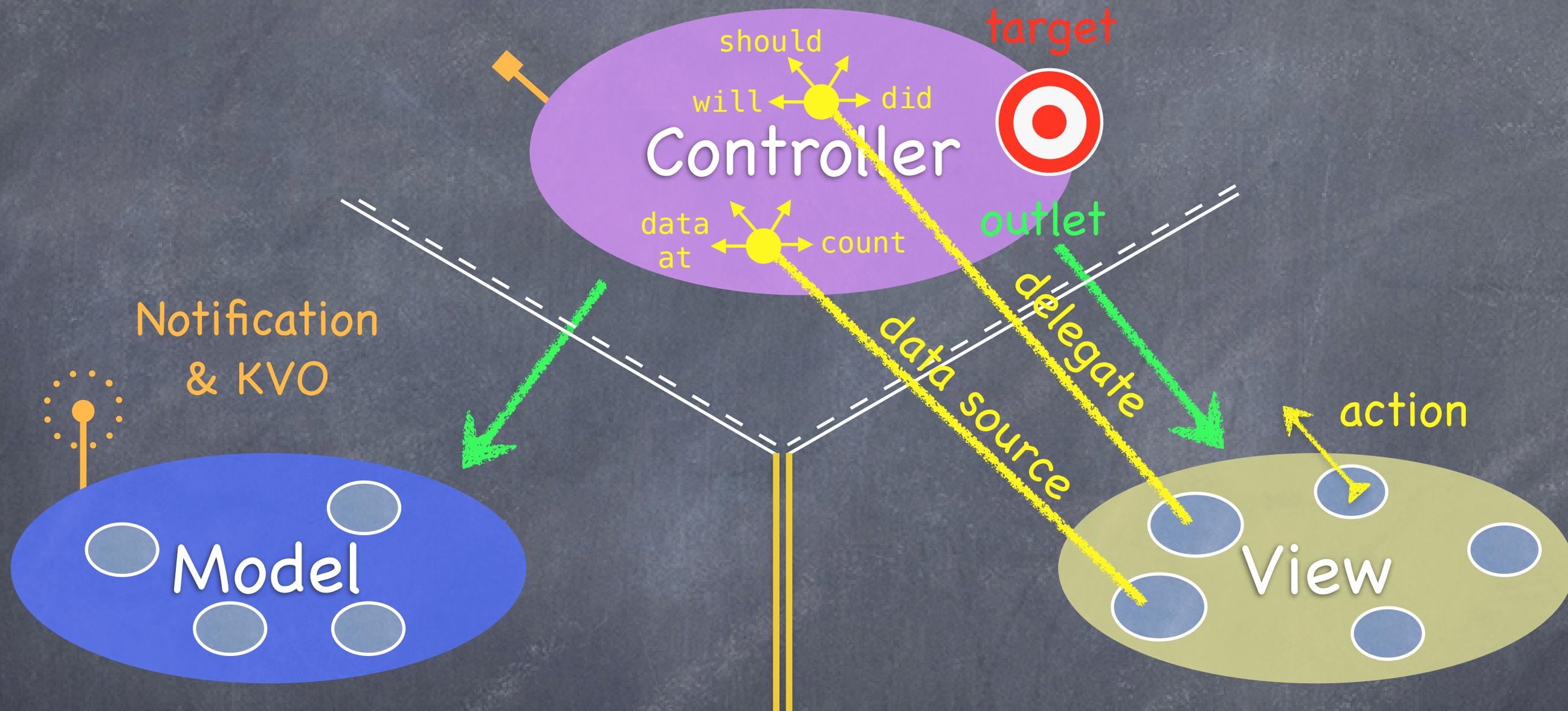
So what if the Model has information to update or something?

# MVC



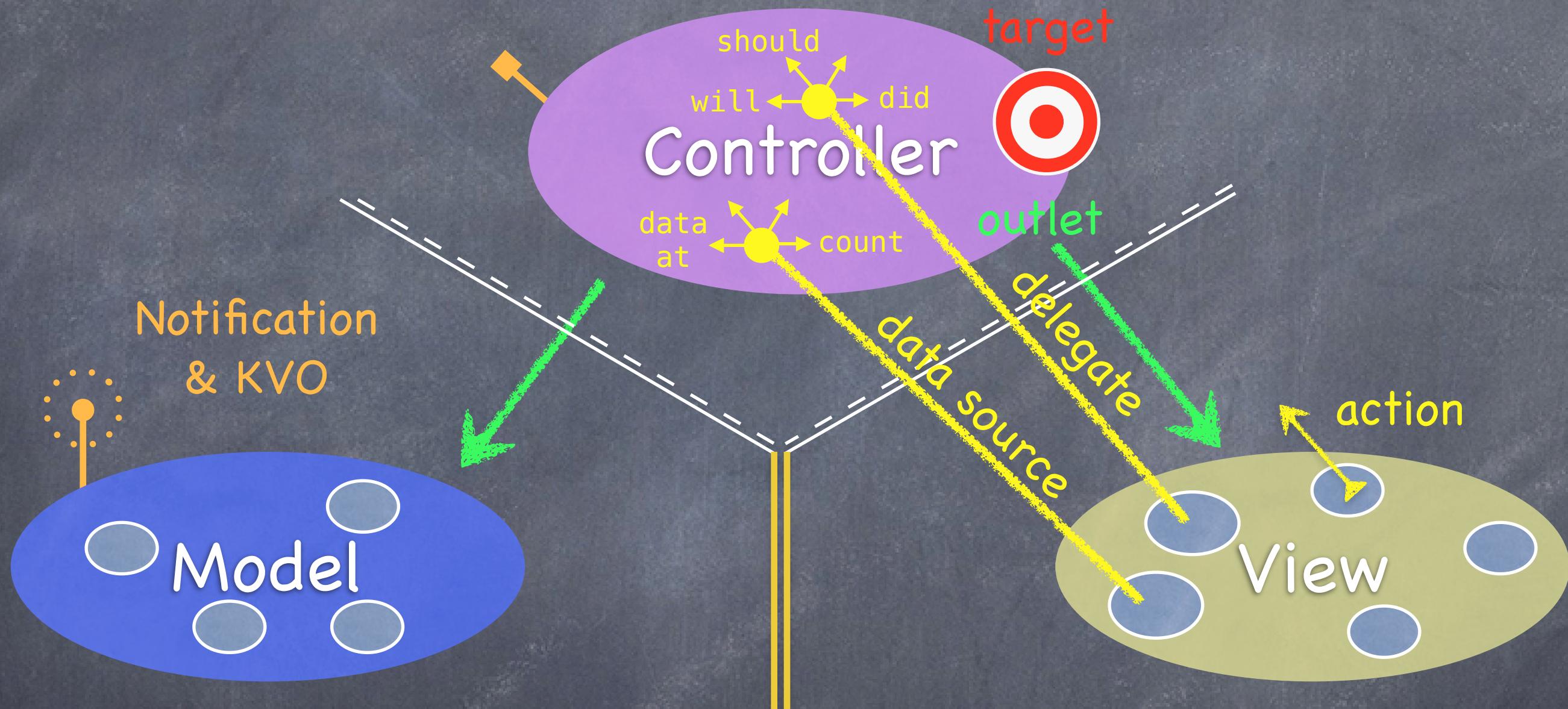
It uses a “radio station”-like broadcast mechanism.

# MVC



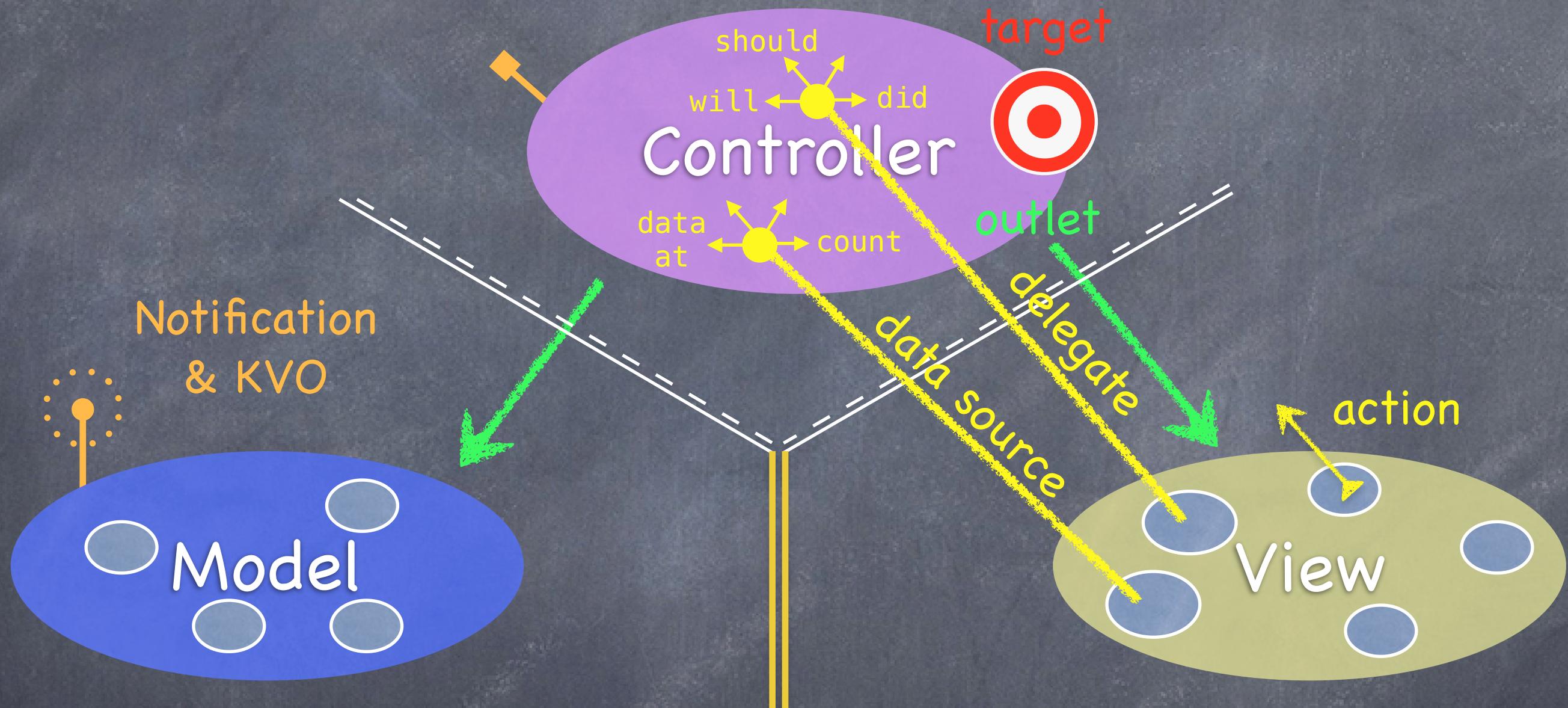
Controllers (or other Model) “tune in” to interesting stuff.

# MVC



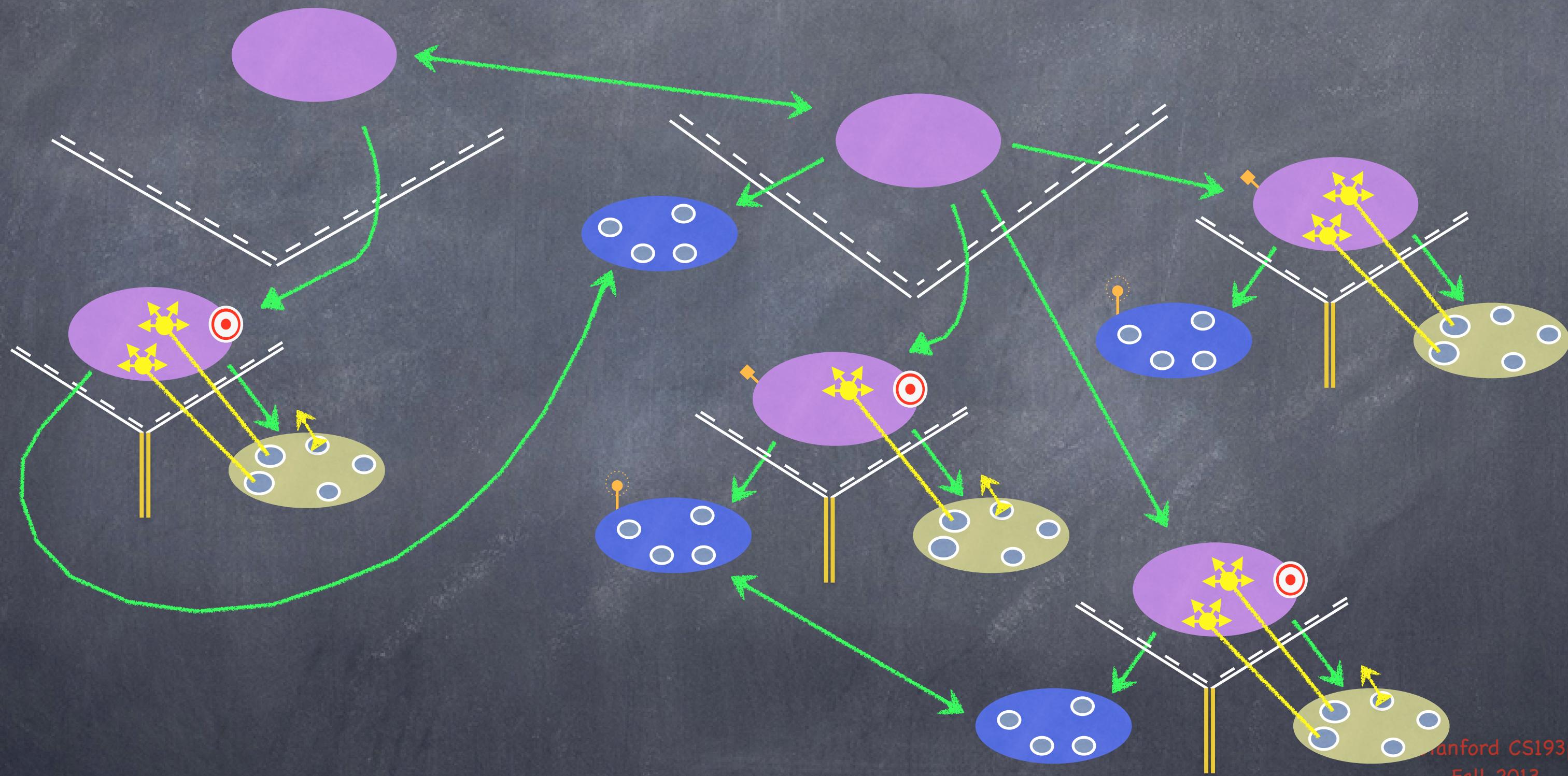
A **View** might “tune in,” but probably not to a **Model’s “station.”**

# MVC

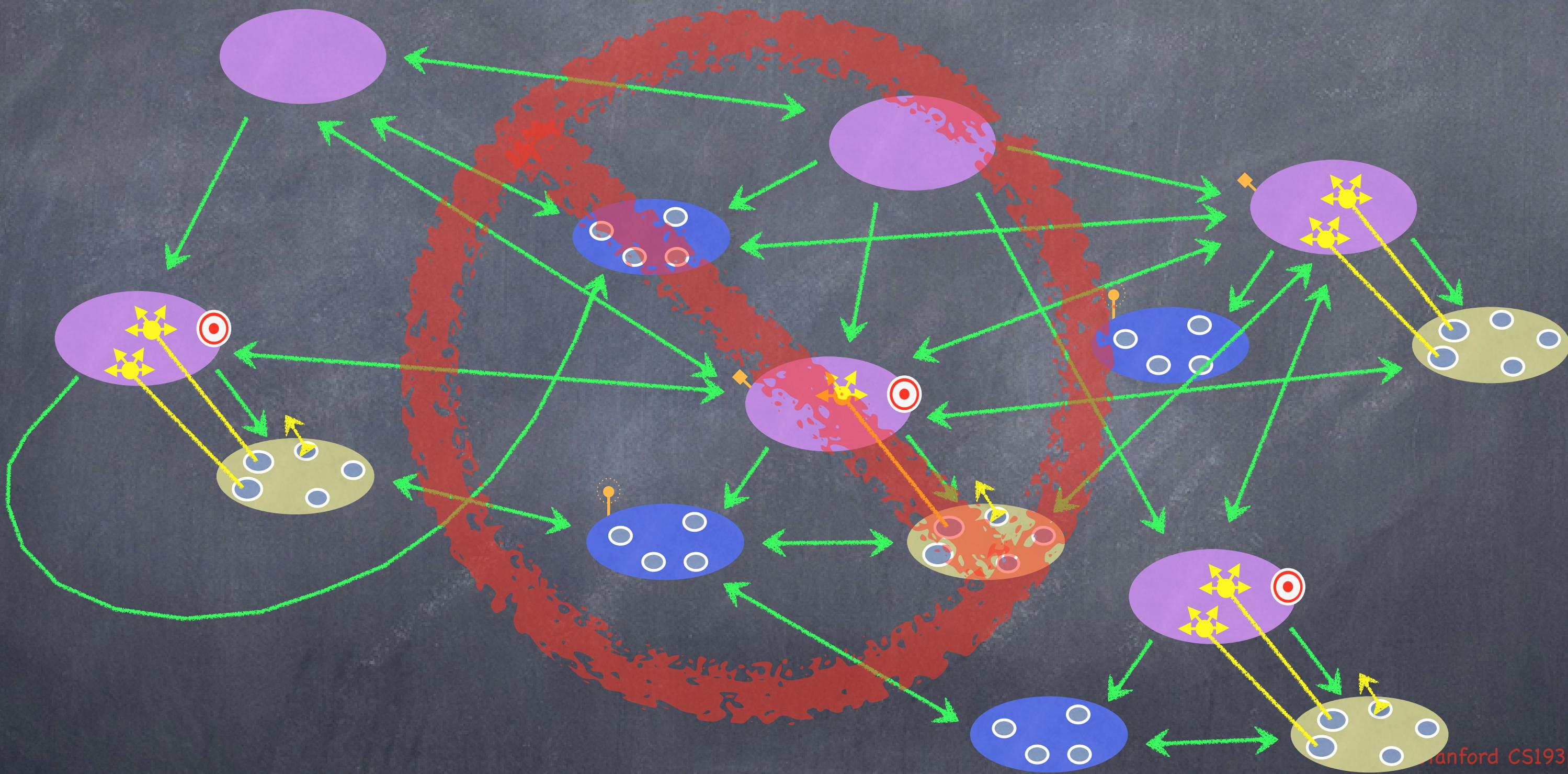


Now combine MVC groups to make complicated programs ...

# 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 (usually) 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

Card.h

Public Declarations

Card.m

Private Implementation

# Objective-C

Card.h

Card.m

Its superclass.

```
@interface Card : NSObject
```

The name  
of this class.

NSObject is the root class from which pretty  
much all iOS classes inherit  
(including the classes you author yourself).

Don't forget this!

```
@end
```

# Objective-C

Card.h

Card.m

```
@interface Card : NSObject
```

```
@implementation Card
```

Note, superclass is not specified here.

@end

@end

# Objective-C

Card.h

```
#import <Foundation/NSObject.h>
```

Superclass's header file.

```
@interface Card : NSObject
```

```
@implementation Card
```

```
@end
```

```
@end
```

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>
```

```
@interface Card : NSObject
```

If the superclass is in iOS itself, we import the entire  
“framework” that includes the superclass.

In this case, Foundation, which contains basic non-UI objects like NSObject.

```
@end
```

```
@end
```

Card.m

# Objective-C

Card.h

```
@import Foundation;
```

In fact, in iOS 7 (only), there is special syntax for  
importing an entire framework called `@import`.

```
@interface Card : NSObject
```

```
@end
```

Card.m

```
@implementation Card
```

```
@end
```

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>
```

```
@interface Card : NSObject
```

However, the old framework importing syntax is backwards-compatible in iOS 7.

```
@end
```

Card.m

```
@implementation Card
```

```
@end
```

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>
```

```
@interface Card : NSObject
```

Card.m

```
#import "Card.h"
```

```
@implementation Card
```

Our own header file must be imported  
into our implementation file.

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>
```

```
@interface Card : NSObject
```

Card.m

```
#import "Card.h"  
  
@interface Card()  
@end  
  
@implementation Card
```

Private declarations can go here.

```
@end
```

```
@end
```

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong) NSString *contents;
```

```
#import "Card.h"

@interface Card()

@end

@implementation Card
```

In iOS, we don't access instance variables directly. Instead, we use an `@property` which declares two methods: a “setter” and a “getter”. It is with those two methods that the `@property`'s instance variable is accessed (both publicly and privately).

This particular `@property` is a pointer. Specifically, a pointer to an object whose class is (or inherits from) `NSString`.

ALL objects live in the heap (i.e. are pointed to) in Objective-C! Thus you would never have a property of type “`NSString`” (rather, “`NSString *`”).

Because this `@property` is in this class's header file, it is public. Its setter and getter can be called from outside this class's `@implementation` block.

`@end`

`@end`

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong) NSString *contents;
```

**strong** means:

“keep the object that this property points to in memory until I set this property to **nil** (zero)  
(and it will stay in memory until everyone who has a **strong** pointer to it sets their property to **nil** too)”

**weak** would mean:

“if no one else has a **strong** pointer to this object,  
then you can throw it out of memory  
and set this property to **nil**  
(this can happen at any time)”

@end

```
#import "Card.h"

@interface Card()

@end

@implementation Card
```

@end

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
```

nonatomic means:

“access to this property is not thread-safe”.

We will always specify this for object pointers in this course.  
If you do not, then the compiler will generate locking code that  
will complicate your code elsewhere.

@end

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card
```

@end

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
```

This is the `@property` implementation that the compiler generates automatically for you (behind the scenes).

You are welcome to write the setter or getter yourself, but this would only be necessary if you needed to do something in addition to simply setting or getting the value of the property.

Card.m

```
#import "Card.h"

@interface Card()
    This @synthesize is the line of code that actually creates the
    backing instance variable that is set and gotten.
@end
    Notice that by default the backing variable's name is the same as
    the property's name but with an underbar in front.
@implementation Card

@synthesize contents = _contents;

- (NSString *)contents
{
    return _contents;
}

- (void)setContents:(NSString *)contents
{
    _contents = contents;
}
```

@end

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card
```

Because the compiler takes care of everything you need to implement a property, it's usually only one line of code (the `@property` declaration) to add one to your class.

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>
```

Notice no **strong** or **weak** here.

Primitive types are not stored in the heap, so there's no need to

```
@interface Card : NSObject
```

```
@property (strong, nonatomic) NSString *contents;
```

```
@property (nonatomic) BOOL chosen;
```

```
@property (nonatomic) BOOL matched;
```

Properties can be  
any C type.  
That includes **int**,  
**float**, etc., even C  
structs.

```
#import "Card.h"
```

```
@interface Card()
```

```
@end
```

```
@implementation Card
```

Let's look at some more properties.  
These are not pointers.  
They are simple **BOOLs**.

C does not define a “boolean” type.  
This **BOOL** is an Objective-C typedef.  
It’s values are **YES** or **NO**.

```
@end
```

```
@end
```

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
@property (nonatomic) BOOL chosen;
@property (nonatomic) BOOL matched;

@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@synthesize chosen = _chosen;
@synthesize matched = _matched;

- (BOOL)chosen
{
    return _chosen;
}

- (void)setChosen:(BOOL)chosen
{
    _chosen = chosen;
}

- (BOOL)matched
{
    return _matched;
}

- (void)setMatched:(BOOL)matched
{
    _matched = matched;
}

@end
```

Here's what the compiler is doing behind the scenes for these two properties.

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>
```

It is actually possible to change the name of the getter that is generated. The only time you'll ever see that done in this class (or anywhere probably) is boolean getters.

```
@interface Card : NSObject
```

```
@property (strong, nonatomic) NSString *contents;  
@property (nonatomic, getter=isChosen) BOOL chosen;  
@property (nonatomic, getter=isMatched) BOOL matched;
```

This is done simply to make the code “read” a little bit nicer. You’ll see this in action later.

```
@end
```

```
#import "Card.h"
```

```
@interface Card()  
@end
```

done in this class  
boolean getters.

```
@implementation Card
```

```
@synthesize chosen = _chosen;  
@synthesize matched = _matched;
```

```
- (BOOL)isChosen  
{  
    return _chosen;  
}  
- (void)setChosen:(BOOL)chosen  
{  
    _chosen = chosen;  
}
```

```
- (BOOL)isMatched  
{  
    return _matched;  
}  
- (void)setMatched:(BOOL)matched  
{  
    _matched = matched;  
}
```

```
@end
```

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

@end
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@end
```

Remember, unless you need to do something besides setting or getting when a property is being set or gotten, the implementation side of this will all happen automatically for you.

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
Enough properties for now.

@property (Let's take a look at defining methods.) chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;
```

Here's the declaration of a public method called `match:` which takes one argument (a pointer to a `Card`) and returns an integer.

What makes this method public?  
Because we've declared it in the header file.

@end

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

@end
```

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;
```

Here's the declaration of a public method called `match:` which takes one argument (a pointer to a `Card`) and returns an `integer`.

```
#import "Card.h"
```

```
@interface Card()
```

```
@end
```

```
@implementation Card
```

```
- (int)match:(Card *)card
```

```
{
```

```
    int score = 0;
```

`match:` is going to return a “score” which says how good a match the passed card is to the `Card` that is receiving this message. 0 means “no match”, higher numbers mean a better match.

```
    return score;
```

```
}
```

```
@end
```

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;
```

@end

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

- (int)match:(Card *)card
{
    int score = 0;

    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }

    return score;
}
```

There's a lot going on here!  
For the first time, we are seeing the  
“calling” side of properties (and methods).

For this example, we'll return 1 if the passed card has  
the same contents as we do or 0 otherwise  
(you could imagine more complex scoring).

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;
@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;
```

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

- (int)match:(Card *)card
{
    int score = 0;

    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }

    return score;
}

@end
```

Notice that we are calling the “getter” for the contents `@property` (both on our `self` and on the passed card). This calling syntax is called “dot notation.” It’s only for setters and getters.

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(Card *)card;
```

Recall that the contents  
property is an `NSString`.

```
#import "Card.h"

@interface Card()
```

@end

```
@implementation Card
```

```
- (int)match:(Card *)card
{
    int score = 0;

    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }
}
```

`isEqualToString:` is an `NSString` method  
which takes another `NSString` as an argument and  
returns a `BOOL` (`YES` if the 2 strings are the same).

Also, we see the “square bracket” notation we use to  
return `score`; send a message to an object.  
In this case, the message `isEqualToString:` is being sent  
to the `NSString` returned by the `contents` getter.

@end

@end

Card.m

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject

@property (strong, nonatomic) NSString *contents;

@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;

- (int)match:(NSArray *)otherCards;
```

We could make `match:` even more powerful by allowing it to match against multiple cards by passing an array of cards using the `NSArray` class in Foundation.

@end

Card.m

```
#import "Card.h"

@interface Card()

@end

@implementation Card

- (int)match:(NSArray *)otherCards
{
    int score = 0;

    if ([card.contents isEqualToString:self.contents]) {
        score = 1;
    }

    return score;
}
```

@end

# Objective-C

Card.h

```
#import <Foundation/Foundation.h>

@interface Card : NSObject
@property (strong, nonatomic) NSString *contents;
@property (nonatomic, getter=isChosen) BOOL chosen;
@property (nonatomic, getter=isMatched) BOOL matched;
- (int)match:(NSArray *)otherCards;
@end
```

```
#import "Card.h"

@interface Card()

@end

@implementation Card
- (int)match:(NSArray *)otherCards
{
    int score = 0;

    for (Card *card in otherCards) {
        if ([card.contents isEqualToString:self.contents]) {
            score = 1;
        }
    }
    return score;
}
@end
```

We'll implement a very simple match scoring system here which is to score 1 point if ANY of the passed otherCards' contents match the receiving Card's contents.  
(You could imagine giving more points if multiple cards match.)

Note the `for-in` looping syntax here.  
This is called “fast enumeration.”  
It works on arrays, dictionaries, etc.

Card.m

# Coming Up

## ⌚ Next Lecture

More of our Card Game Model

Xcode 5 Demonstration (start building our Card Game application)

## ⌚ Next Week

Finish off our Card Game application

Objective-C in more detail

Foundation (array, dictionary, etc.)

And much much more!