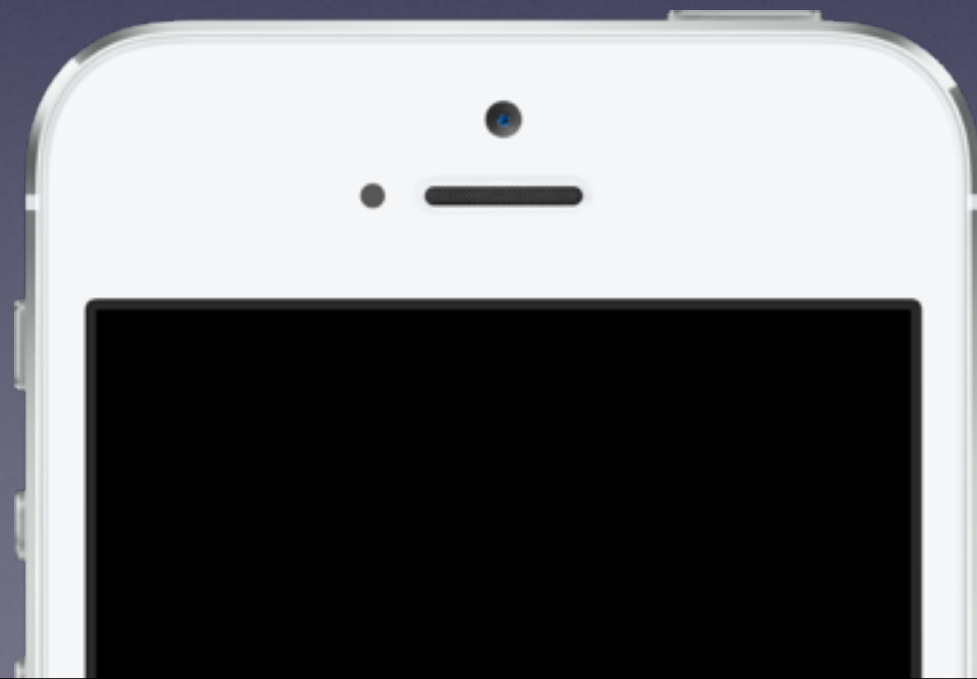


COMS W3101: Programming for iOS

Michael Vitrano



Course Goals

- Walk away with the toolset to build real world apps
- Focus on a foundation in Objective-C and core iOS concepts and frameworks
- Exposure to real-world problem domains and app categories
- A resource for transitioning from academic coding to professional coding

Course Prerequisites

- Fluency in at least one programming language
- Strong understanding of Object-Oriented programming
- Familiarity with Model-View-Controller architecture

Course Structure

- Each class is going to be a mixture of lecture and demo
 - Will be posted to the course Github page
 - <http://columbia-w3101-ios.github.io/>
 - Will focus on the building blocks of an iOS app
- One course long project
 - We're going to build a note taking app
 - There will be a set of features that your app must have
 - Over the course of the semester you'll gain the skills you need

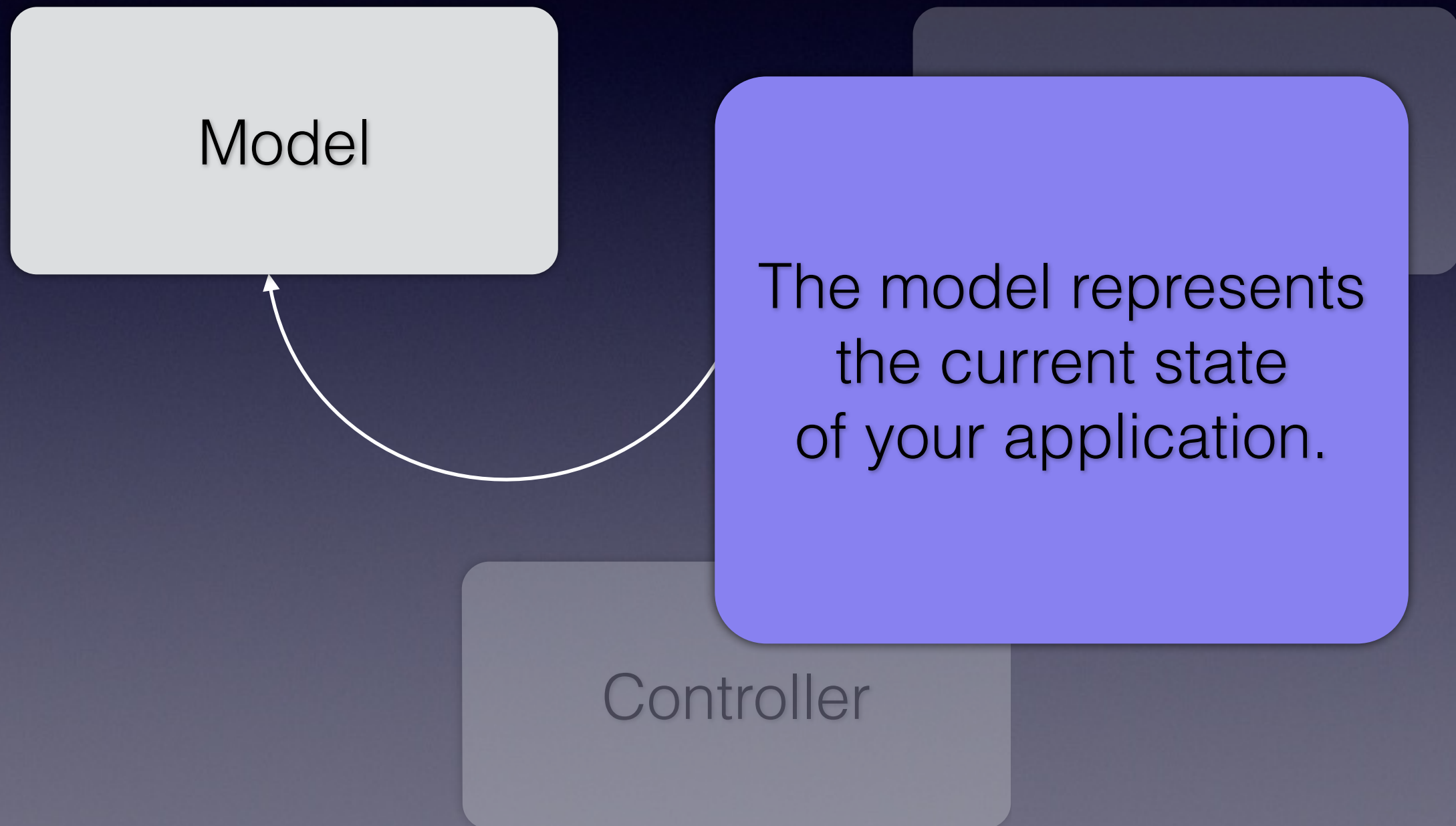
Model-View-Controller

Model

View

Controller

Model-View-Controller



Model-View-Controller

The view is how your application displays its state to the user.

View

Controller



Model-View-Controller

Model

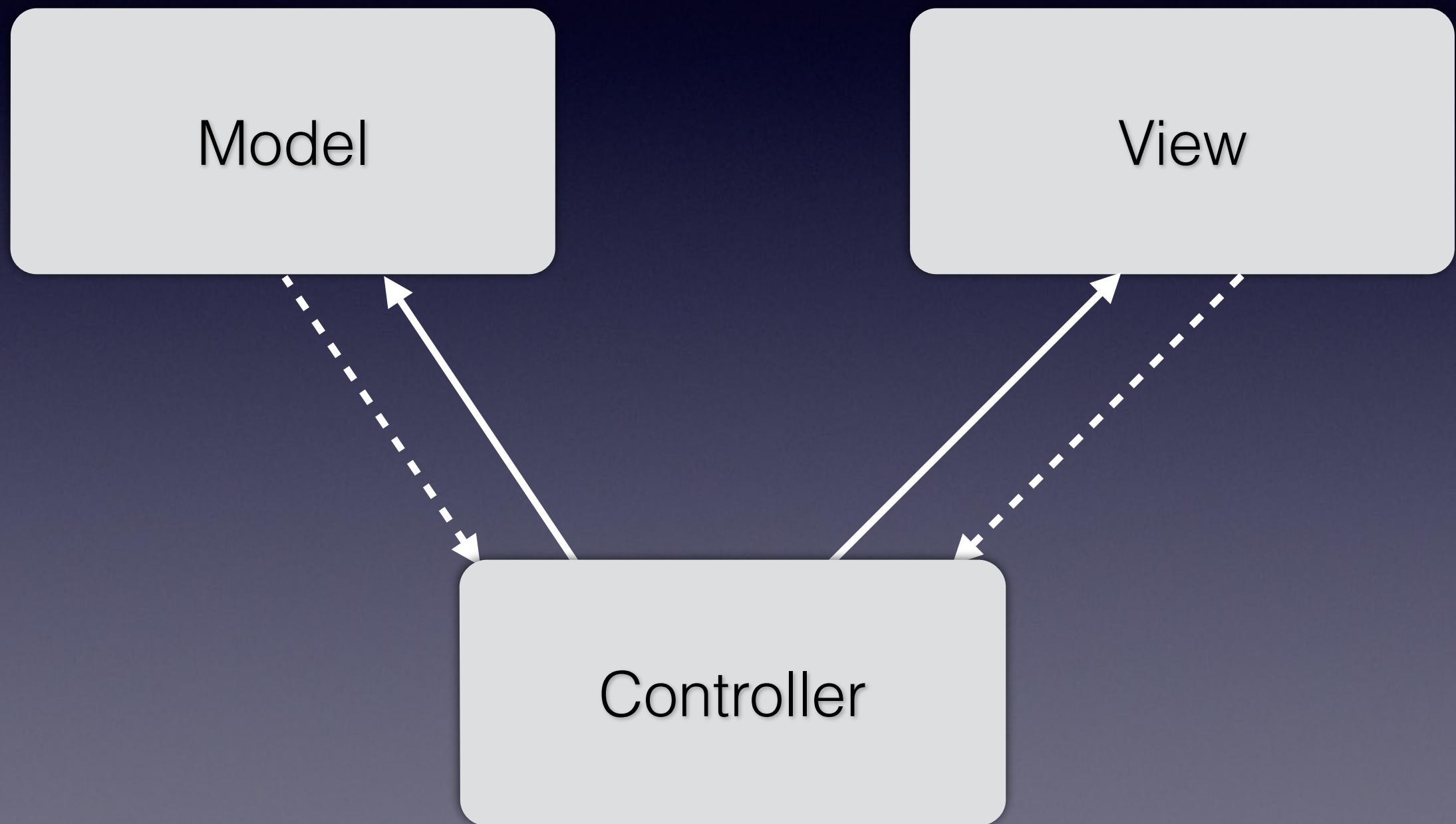
The controller mediates the communication between the model and the view.

Controller

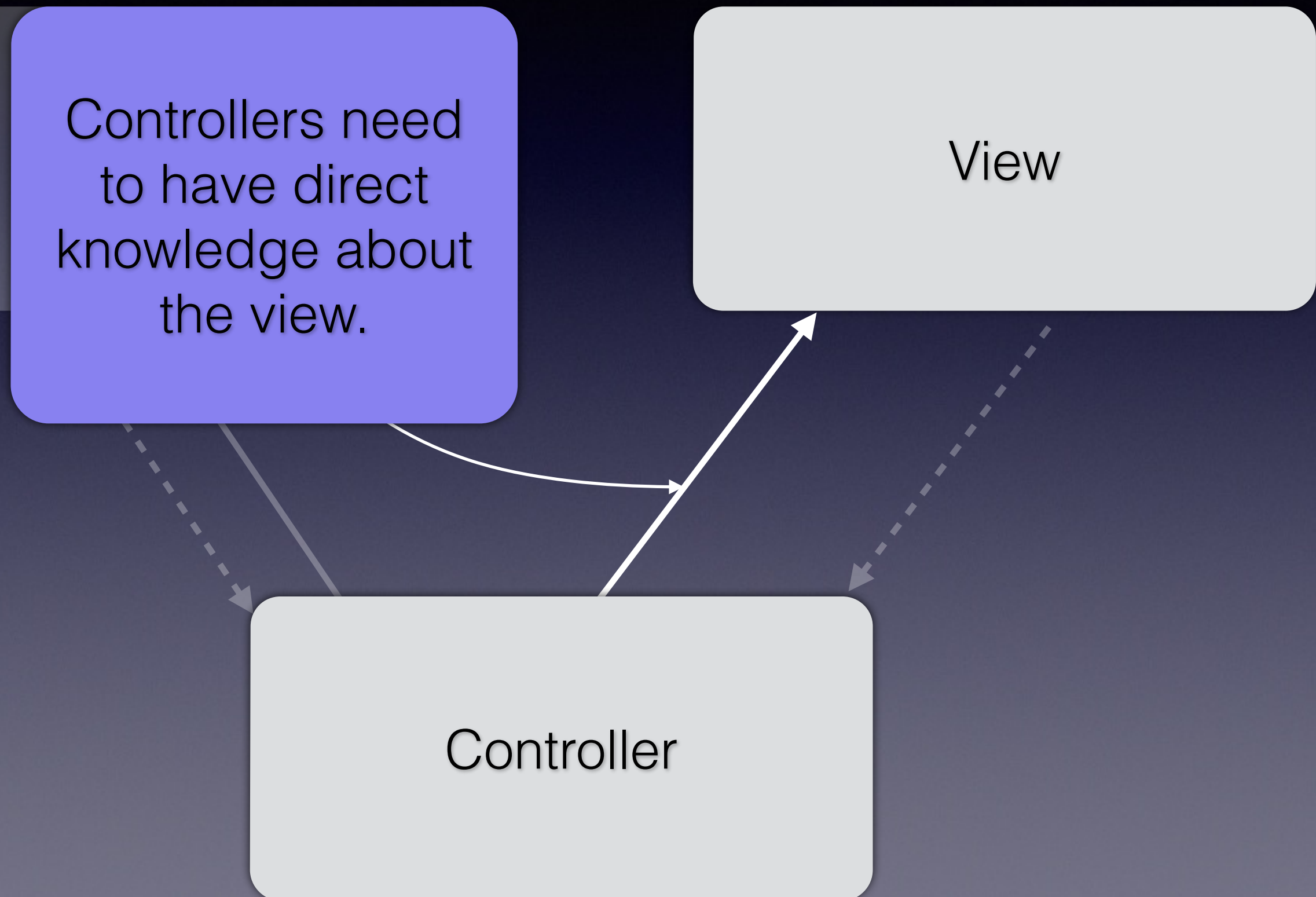
```
graph TD; Model[Model]; Controller[Controller]; Controller --> Model;
```

The diagram illustrates the MVC pattern. It features three main components: a dark gray rounded rectangle labeled 'Model' on the left, a light gray rounded rectangle labeled 'Controller' at the bottom center, and a large blue rounded rectangle on the right containing descriptive text. A curved white arrow points from the bottom of the blue box to the right side of the 'Controller' box, indicating that the controller mediates communication between the model and the view.

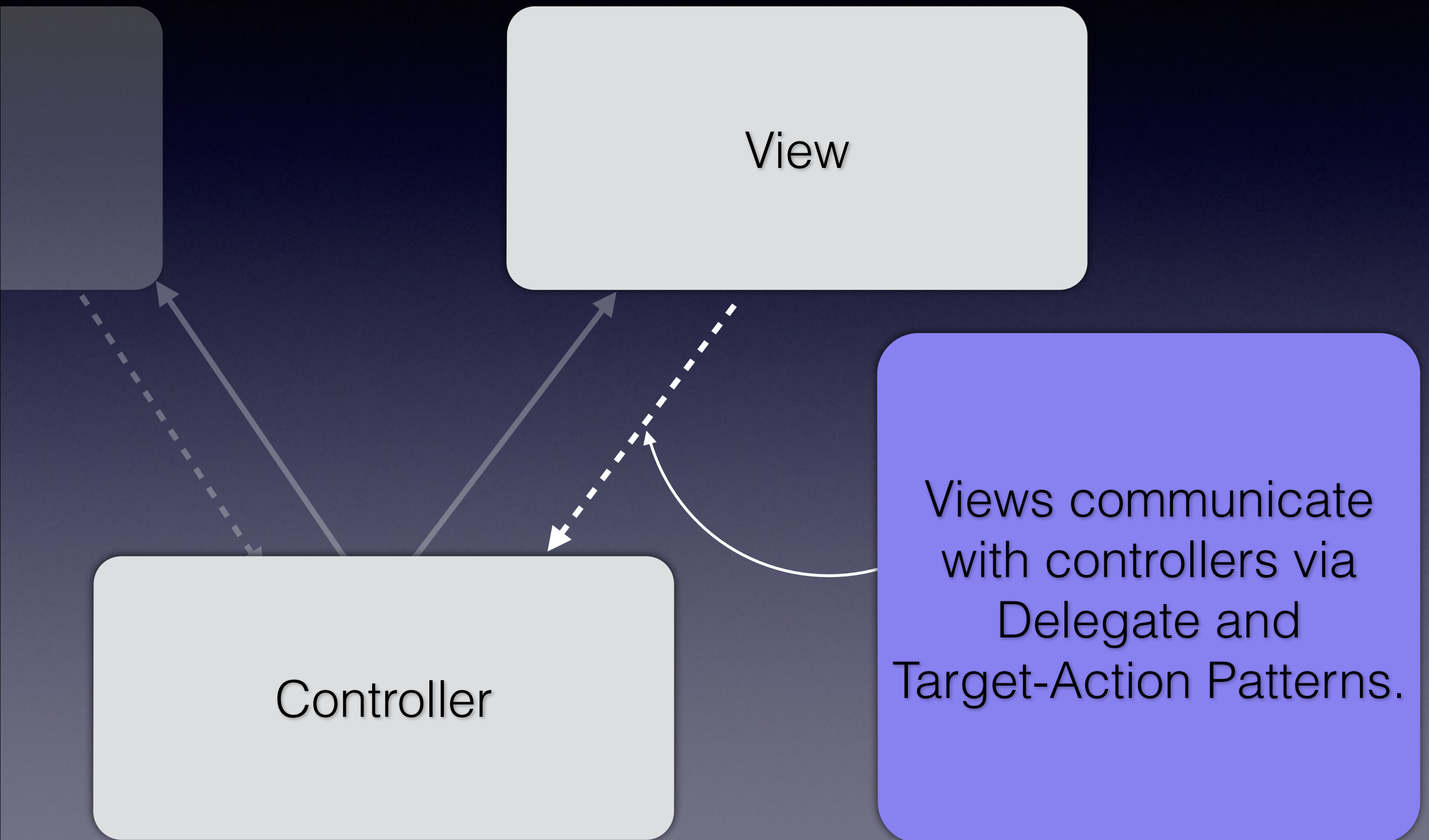
Model-View-Controller



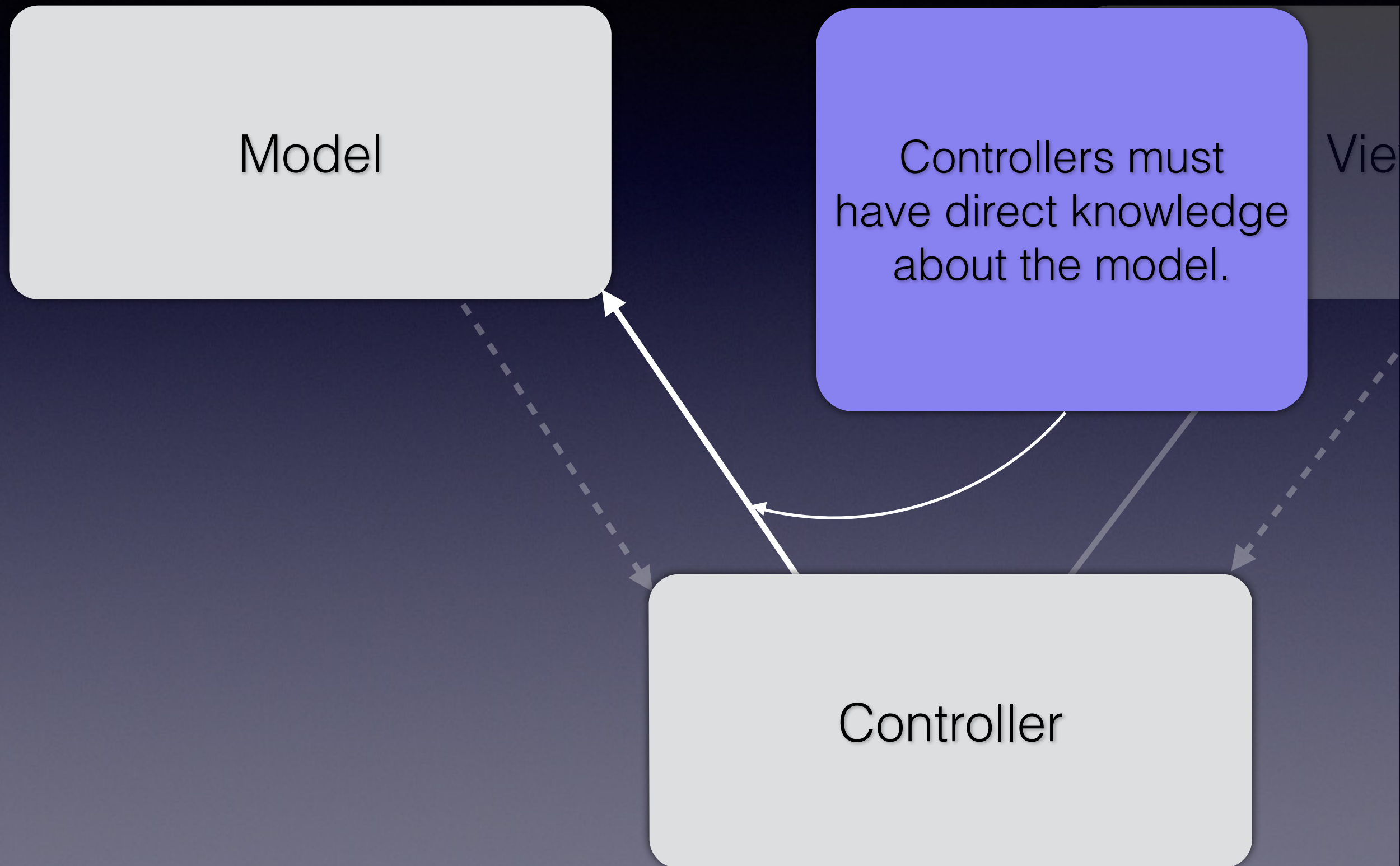
Model-View-Controller



Model-View-Controller



Model-View-Controller



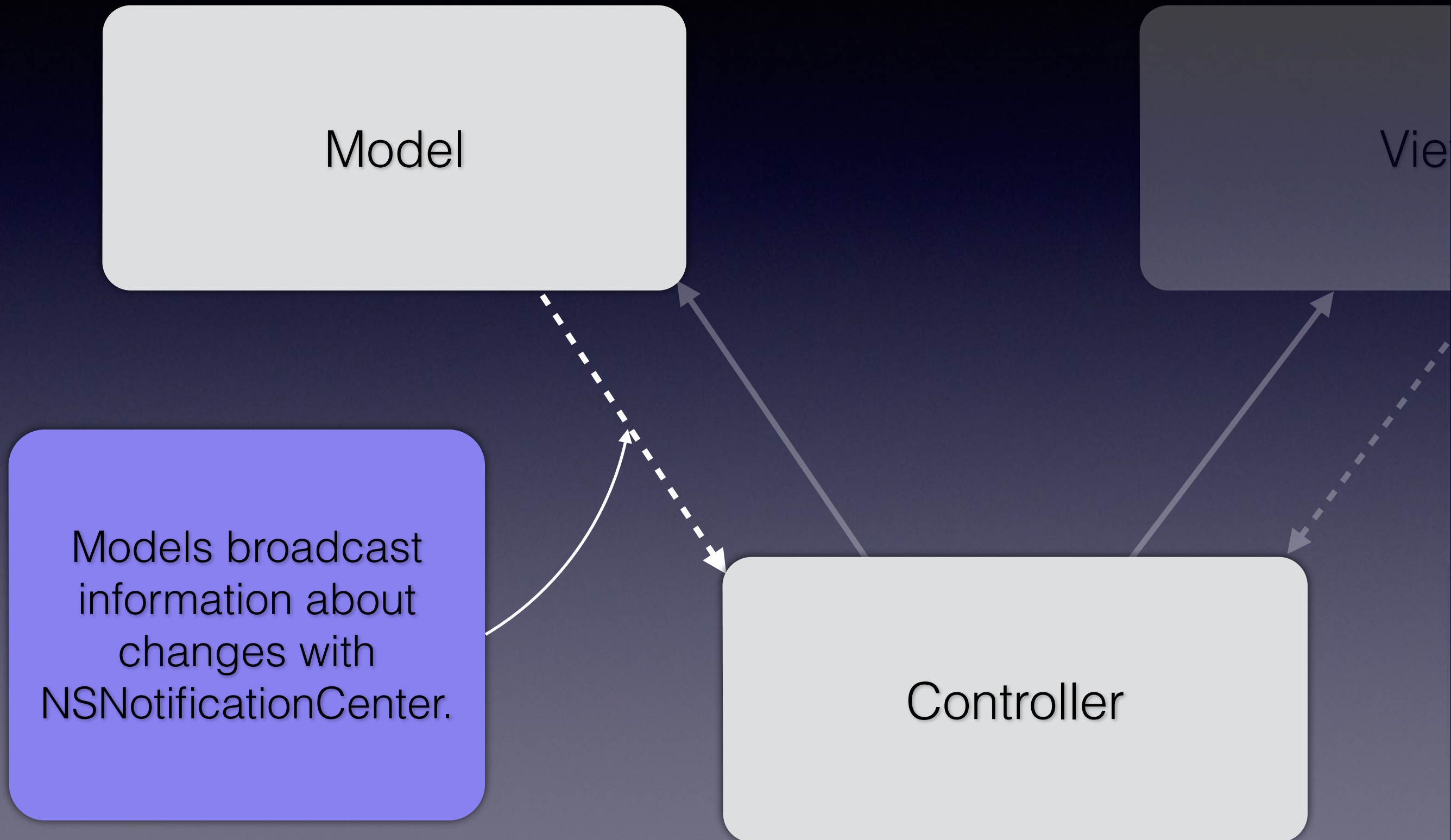
Model-View-Controller

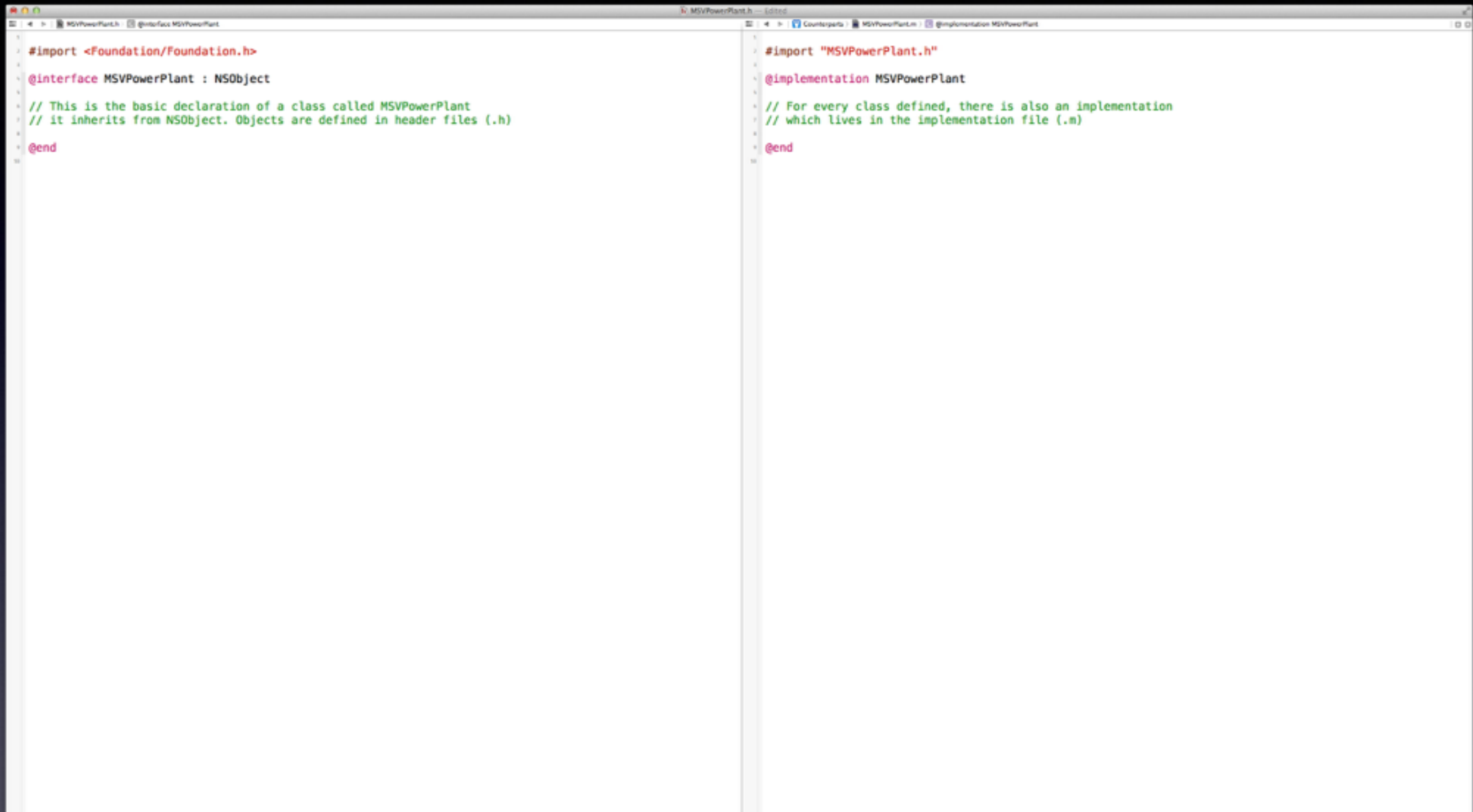
Model

View

Models broadcast
information about
changes with
NSNotificationCenter.

Controller





The image shows a code editor with two files open side-by-side. The left file is `MSVPowerPlant.h` and the right file is `MSVPowerPlant.m`. Both files contain Objective-C code for a class named `MSVPowerPlant`.

```
1 #import <Foundation/Foundation.h>
2
3 @interface MSVPowerPlant : NSObject
4
5 // This is the basic declaration of a class called MSVPowerPlant
6 // it inherits from NSObject. Objects are defined in header files (.h)
7
8 @end
```

```
1 #import "MSVPowerPlant.h"
2
3 @implementation MSVPowerPlant
4
5 // For every class defined, there is also an implementation
6 // which lives in the implementation file (.m)
7
8 @end
```

```
1 #import <Foundation/Foundation.h>
2
3 @interface MSVPowerPlant : NSObject
4
5 // Your class's public interface is defined here, in the header
6
7 @end
```

```
1 #import "MSVPowerPlant.h"
2
3 @interface MSVPowerPlant ()
4
5 // Your class's internal interface is declared inside of the
6 // implementation file in a class extension
7
8 @end
9
10 @implementation MSVPowerPlant
11
12 // The implementation of your public and private methods go
13 // here in the implementation file
14
15 @end
```

```
MSVPowerPlant.h
1 #import <Foundation/Foundation.h>
2
3 // We need to let the compiler know that there exists
4 // a class called MSVCity that we are going to use in our
5 // sendPowerToCity:amount: method. We do that with the @class
6 // directive
7 @class MSVCity;
8
9 @interface MSVPowerPlant : NSObject
10
11 // We define a method like this:
12 - (void)sendPowerToCity:(MSVCity *)city
13     amount:(double)powerInWatts;
14
15 // This method takes a city object and an amount of power
16 // to send it. It returns void.
17
18 @end
19
```

```
MSVPowerPlant.m
1 #import "MSVPowerPlant.h"
2
3 // We need to import the MSVCity header so that
4 // we can know the a MSVCity object's interface
5 #import "MSVCity.h"
6
7 @implementation MSVPowerPlant
8
9 @end
10
11 @implementation MSVPowerPlant
12
13 // We implement methods like this:
14 - (void)sendPowerToCity:(MSVCity *)city amount:(double)powerInWatts
15 {
16     // We call methods like this:
17     [city addPower:powerInWatts];
18 }
19
20 @end
21
```



```
MSVPowerPlant.h
@interface MSVPowerPlant

#import <Foundation/Foundation.h>

@class MSVCity;

@interface MSVPowerPlant : NSObject

// Properties are how we define storage for an object to
// keep state in. When declaring a property, the compiler automatically
// generates a accessor and setter for us as well as an instance variable
// to store that information in.

@property (nonatomic) double currentPower;

- (void)sendPowerToCity:(MSVCity *)city
    amount:(double)powerInWatts;

@end

MSVPowerPlant.m
#import "MSVPowerPlant.h"
#import "MSVCity.h"

@interface MSVPowerPlant ()

@end

@implementation MSVPowerPlant

- (void)sendPowerToCity:(MSVCity *)city amount:(double)powerInWatts
{
    // The accessor method is just the name of the property. In ObjC
    // we do not prepend 'get' to the beginning of our accessors
    double currentPower = [self currentPower];

    if (currentPower >= powerInWatts) {

        // The setter method is the name of the property prepended by
        // 'set'

        [self setCurrentPower:(currentPower - powerInWatts)];

        [city addPower:powerInWatts];
    }
}

@end
```

```
MSVPowerPlant.h
@interface MSVPowerPlant
1
2 #import <Foundation/Foundation.h>
3
4 @class MSVCity;
5
6 @interface MSVPowerPlant : NSObject
7
8 // Some properties should not be writable to outsiders.
9 // We restrict the writability of a property using 'readonly'.
10 // This means that only the accessor method is available outside of the class.
11
12 @property (nonatomic, readonly) double currentPower;
13
14 - (void)sendPowerToCity:(MSVCity *)city
15     amount:(double)powerInWatts;
16
17 @end
18
```

```
MSVPowerPlant.m
1
2 #import "MSVPowerPlant.h"
3 #import "MSVCity.h"
4
5 @interface MSVPowerPlant ()
6
7 // Since we still need to set the current power when sending power
8 // to a city, we need to have a writable property internal to the class.
9 // We do this by re-defining the property as writable inside the class extension
10
11 @property (nonatomic) double currentPower;
12
13 @end
14
15 @implementation MSVPowerPlant
16
17 - (void)sendPowerToCity:(MSVCity *)city amount:(double)powerInWatts
18 {
19     // The accessor method is just the name of the property. In ObjC
20     // we do not prepend 'get' to the beginning of our accessors
21     double currentPower = [self currentPower];
22
23     if (currentPower >= powerInWatts) {
24
25         // The setter method is the name of the property prepended by
26         // 'set'
27
28         [self setCurrentPower:(currentPower - powerInWatts)];
29
30         [city addPower:powerInWatts];
31     }
32 }
33
34 @end
35
```

```
MSVPowerPlant.h
1 #import <Foundation/Foundation.h>
2
3 @class MSVCity;
4
5 @interface MSVPowerPlant : NSObject
6
7 @property (nonatomic, readonly) double currentPower;
8
9 - (void)sendPowerToCity:(MSVCity *)city
10     amount:(double)powerInWatts;
11
12 @end
13
```

```
MSVPowerPlant.m
1 #import "MSVPowerPlant.h"
2 #import "MSVCity.h"
3
4 @interface MSVPowerPlant ()
5
6 @property (nonatomic) double currentPower;
7
8 @end
9
10 @implementation MSVPowerPlant
11
12 - (void)sendPowerToCity:(MSVCity *)city amount:(double)powerInWatts
13 {
14     // Properties are so important to the language that there is special syntax for accessing
15     // and setting them.
16
17     // Use dot-notation to access a property. This is equivalent to calling the accessor
18     // with the bracket notation.
19     double currentPower = self.currentPower;
20
21     if (currentPower >= powerInWatts) {
22
23         // You can also use dot-notation to write to a property. This is equivalent to calling
24         // the setter with bracket notation
25         self.currentPower = currentPower - powerInWatts;
26
27         [city addPower:powerInWatts];
28     }
29 }
30
31 @end
32
```