iPhone Programming

CMSC 498i – Spring 2010

Model View Controller

Lecture #8 – Chuck Pisula

Today's Topics

- Architecting an Application
- Model View Controller Pattern
- Communication Strategies
  - Notifications
  - Data Source, Delegation
  - Key-Value Observing, Key-Value Coding

Architecture

- Design before coding
- Spend time understanding the problem
- A good design can save you time and improve quality
- Today
    - Application design patterns
    - Strategies for handling and communicating data
    - Examples of what to do and use

Architecture

- Break code into objects and units based on functionality
- Each module has a focused purpose

Architecture

- Break code into objects and units based on functionality
- Each module has a focused purpose
- Example: “iNeedToDo” ToDo Application
  - Data only objects
  - User Interface objects
  - Coordination
  - Application Logic

ToDo Application

- In Detail…

ToDo Application

Application Delegate

Manage application
lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application

Application Delegate

Manage application lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application

Application Delegate

Manage application lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application

```
@implementation iNeedToDoAppDelegate

- (void)applicationDidFinishLaunching:(UIApplication *)app {
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
....
@end
```

Application Delegate

Manage application lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application

```
    [window addSubview:[navigationController view]];
    [window makeKeyAndVisible];
}
....
@end
```

```
@implementation UINavigationController

- (UIView *)view {
    if (!_view) {
       [self loadView];
    }
    return _view;
}

- (void)loadView {
    // Create top level container view with nav bar
    rootController = // set to ToDoListViewController in NIB
    [topView addSubview:[rootController view]];
}

@end
```

Application Delegate

Manage application lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application

```
    rootController = // set to ToDoListViewController in NIB
    [topView addSubview:[rootController view]];
}

@end
```

```
@implementation UIViewController

// ToDoListViewController is a UIViewController!
- (void)loadView {
    [NSBundle loadNibNamed: [self nibName]
                     owner: self
                   options:nil];
}

@end
```

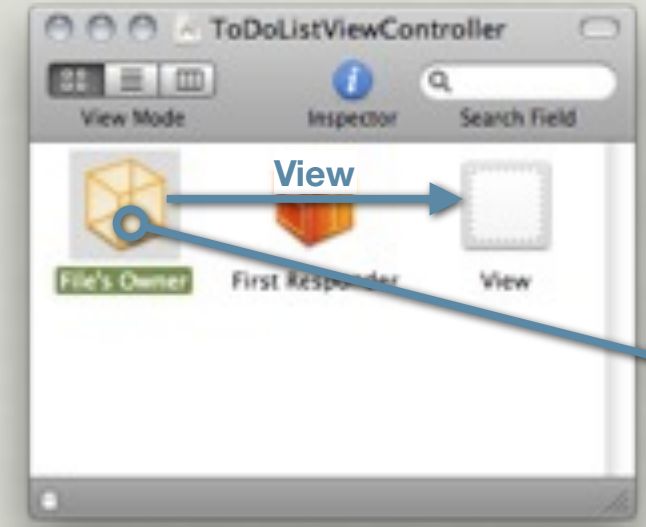
Application Delegate

Manage application lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application



Application Delegate

Manage application lifecycle and events

ToDoListViewController

Provide cells for table
Manage array of events

ToDo Application

NSDate (Additions)

date formatting strings in NSDate category

Application Delegate

Manage application lifecycle and events

UITableView

Display a list of cells provided by data source

ToDoListViewController

Provide cells for table
Manage array of events

ToDoEventCell

Display a single event in a table

ToDo Application

NSDate (Additions)

date formatting strings in NSDate category

Application Delegate

Manage application lifecycle and events

ToDoEvent

Store and provide access event data

UITableView

Display a list of cells provided by data source

ToDoListViewController

Provide cells for table
Manage array of events

ToDoEventCell

Display a single event in a table

ToDo Application

NSDate (Additions)

date formatting strings in NSDate category

Application Delegate

Manage application lifecycle and events

ToDoEvent

Store and provide access event data

UITableView

Display a list of cells provided by data source

ToDoListViewController

Provide cells for table
Manage array of events

ToDoEventCell

Display a single event in a table

Focused Purpose

- **Example** – `UITableView`
- Handles presentation of table cells in a linear layout
- Provides options to control appearance
- Provides standard UI and behaviors
  - Customize by providing a **delegate** – `UITableViewDelegate`
- Has no knowledge of any specific data
  - Cells provided by **data source** - `UITableViewDataSource`
  - Edits are implemented by the **data source**

Focused Purpose

- **Example** – `ToDoEvent`
- Stores data specific to a todo event
- Provides access to stored data through setter / getter
- Typically provide ability to archive / unarchive the data
- Has no knowledge of how it will be displayed
    - Examples
        - Might display different date formats based on available width
        - We may use “past the due date” to mean display the title in red

Focused Purpose

- **Example** – `ToDoListViewController`
- Knows about the data, and display – Coordinates it all.
- Manages and filters array of events
- Provides cells to table view for display
- Updates events in response to edits
- Tells display about changes to the data

ToDo Application

Model	View	Controller
ToDoEvent Store and provide access event data	**UITableView** Display a list of cells provided by data source **ToDoEventCell** Display a single event in a table	**ToDoListViewController** Provide cells for table Manage array of events

Model View Controller

Model - View - Controller

- MVC

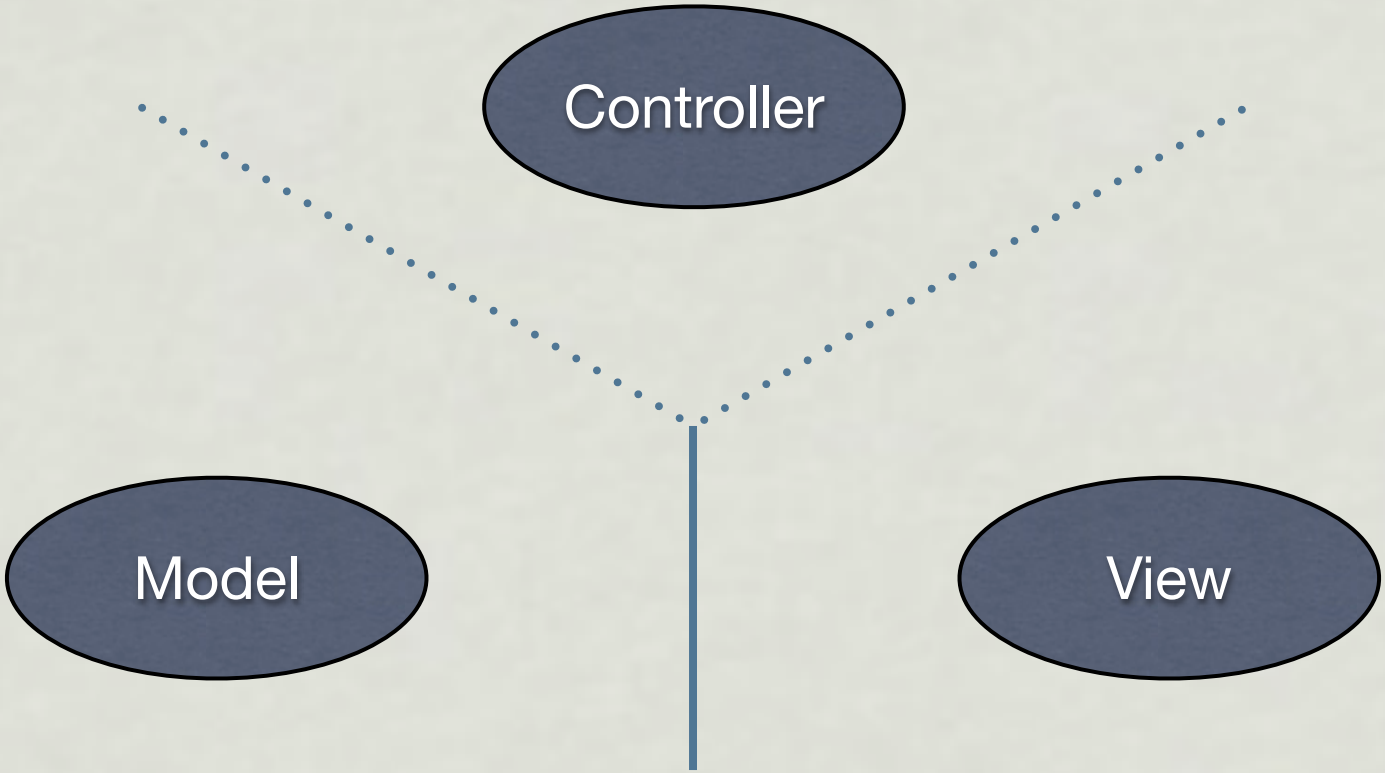
Model - View - Controller

- **MVC**
- A pattern concerned with application architecture
  - Isolate data from input and presentation
  - Each role has a specific purpose and line of communication

Model - View - Controller

- **MVC**
- A pattern concerned with application architecture
  - Isolate data from input and presentation
  - Each role has a specific purpose and line of communication
- Object roles
  - **Model** – data storage and domain specific logic
  - **View** – user interface, data presentation
  - **Controller** – respond to events, update data, implement application logic

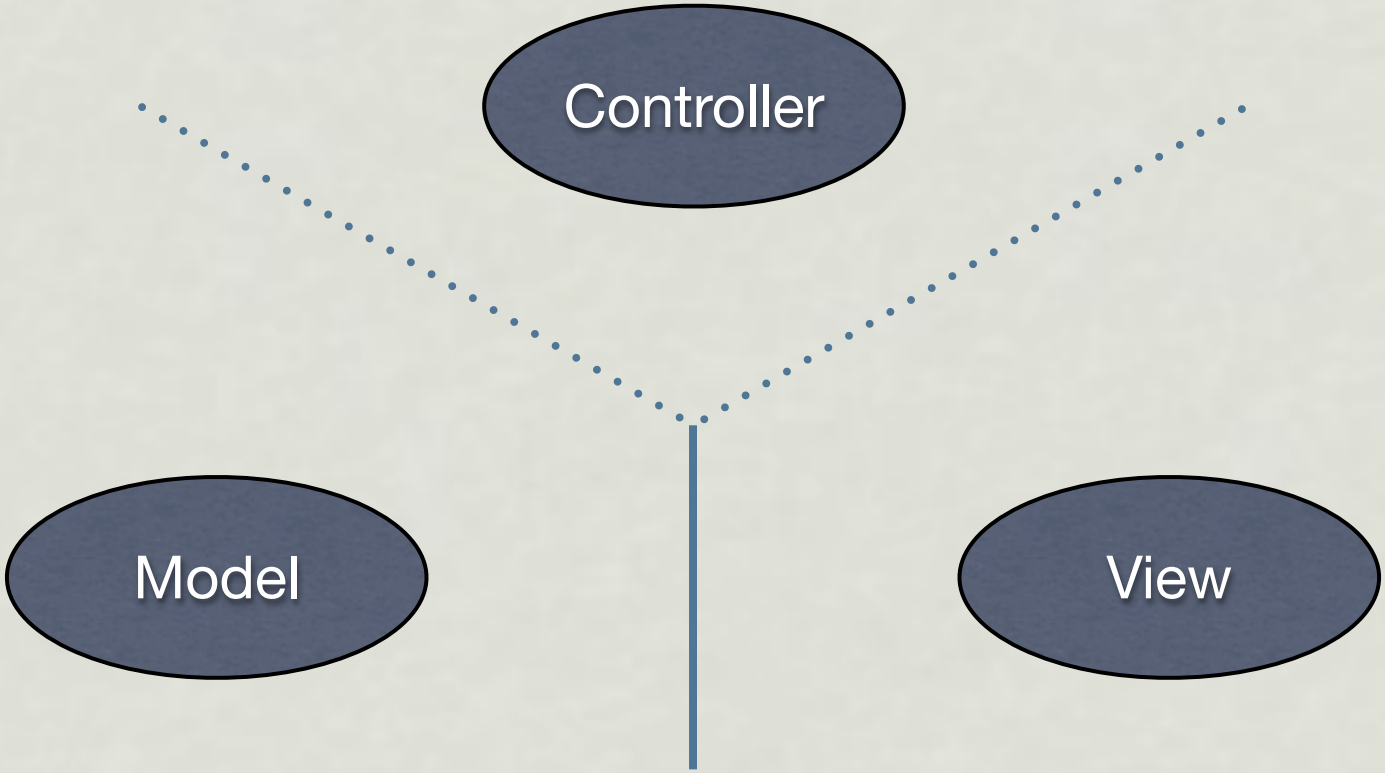
MVC



(or if you like… MCV)

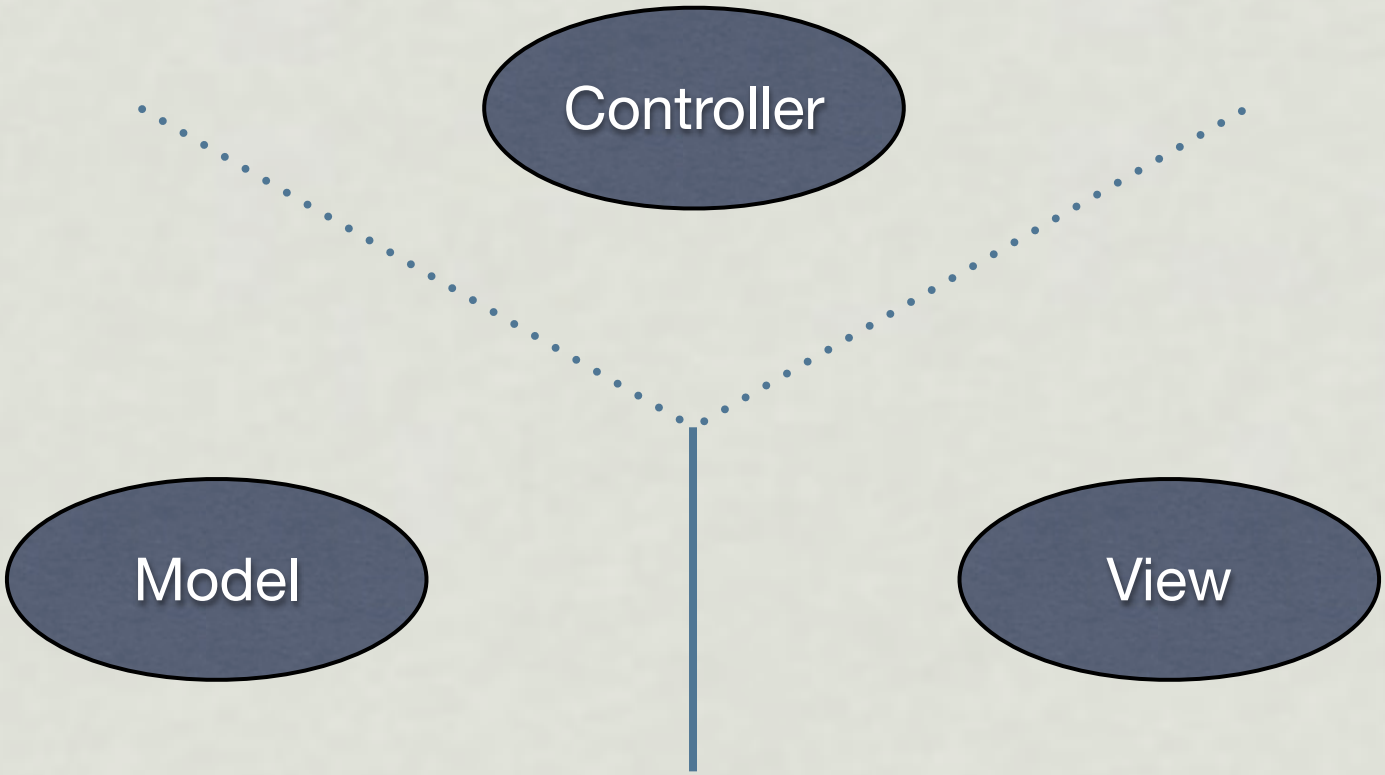
MVC

- Model and View do not communicate directly



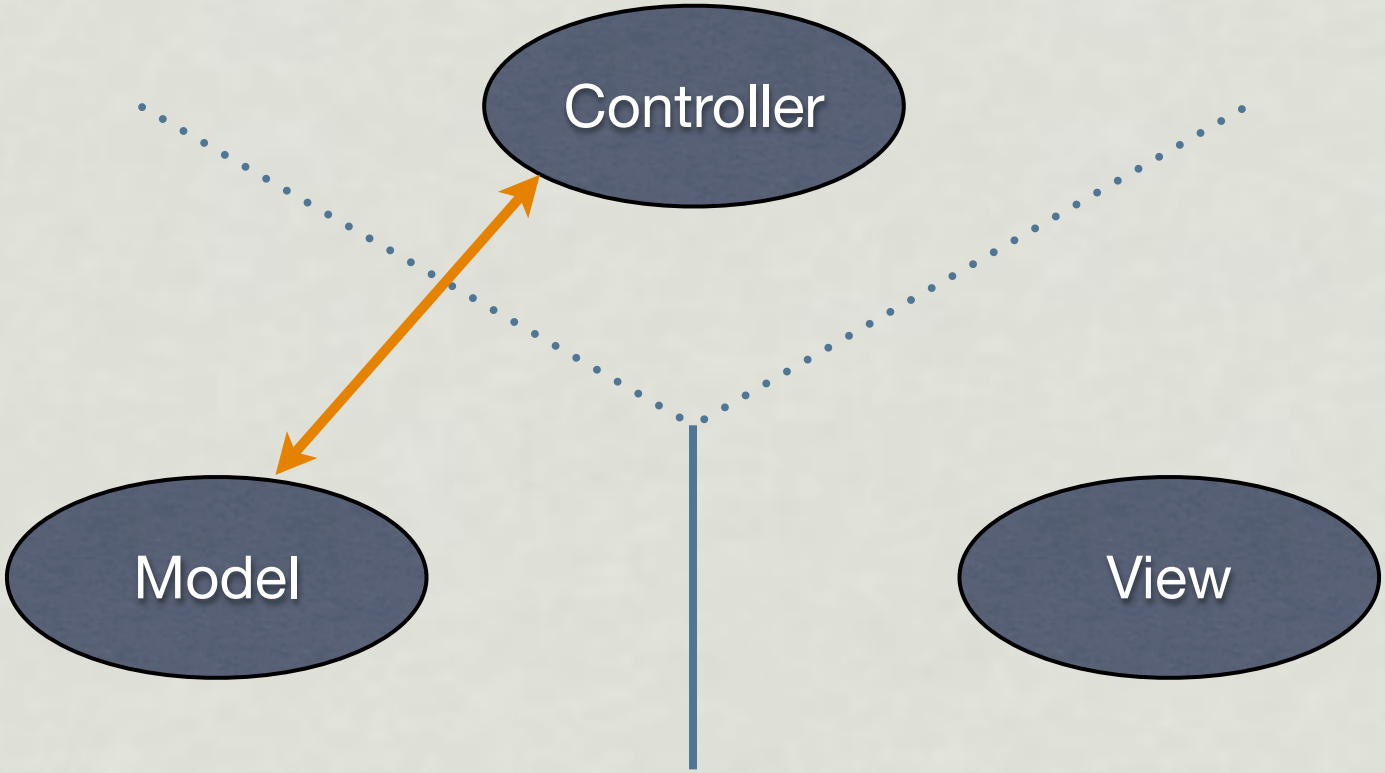
Controller & Model

- Controller updates model and watches for changes



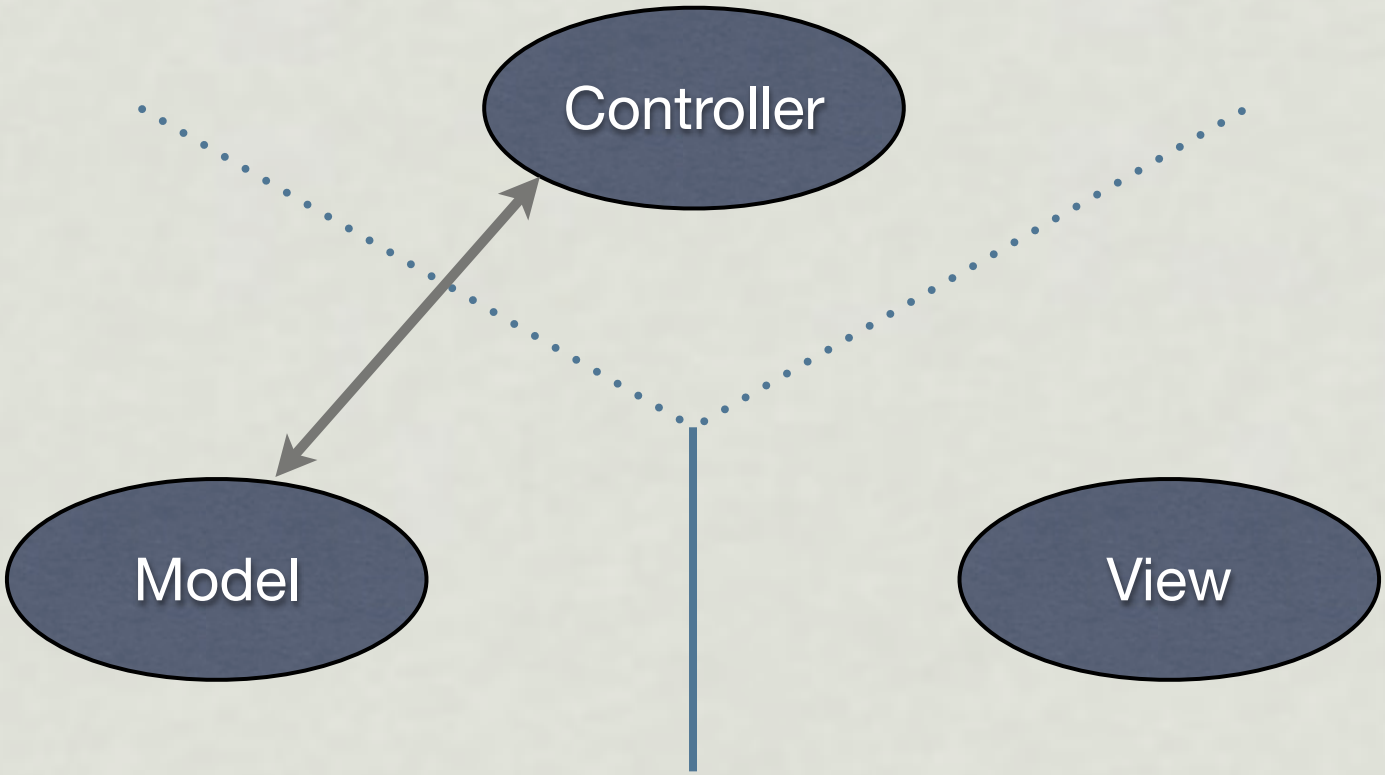
Controller & Model

- Controller updates model and watches for changes



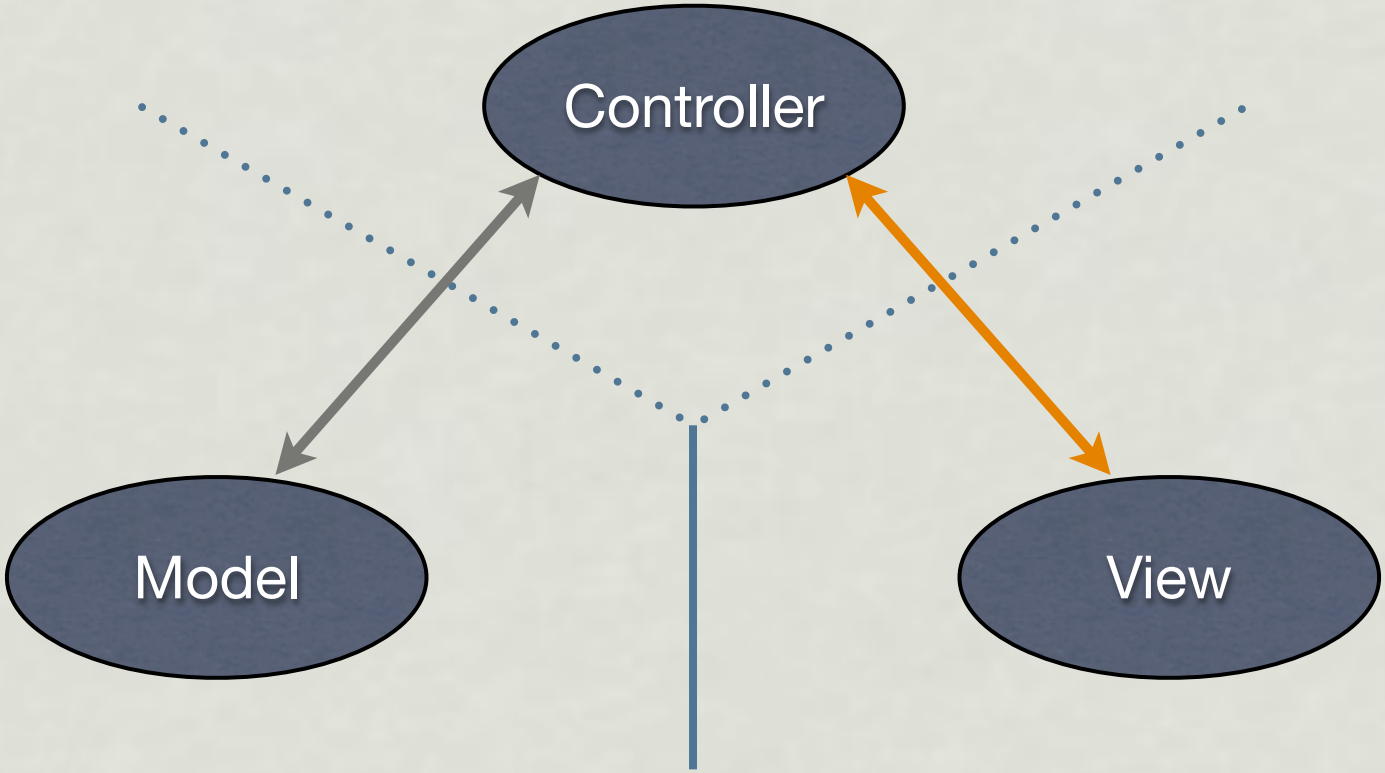
Controller & View

- Controller updates view, responds to view actions



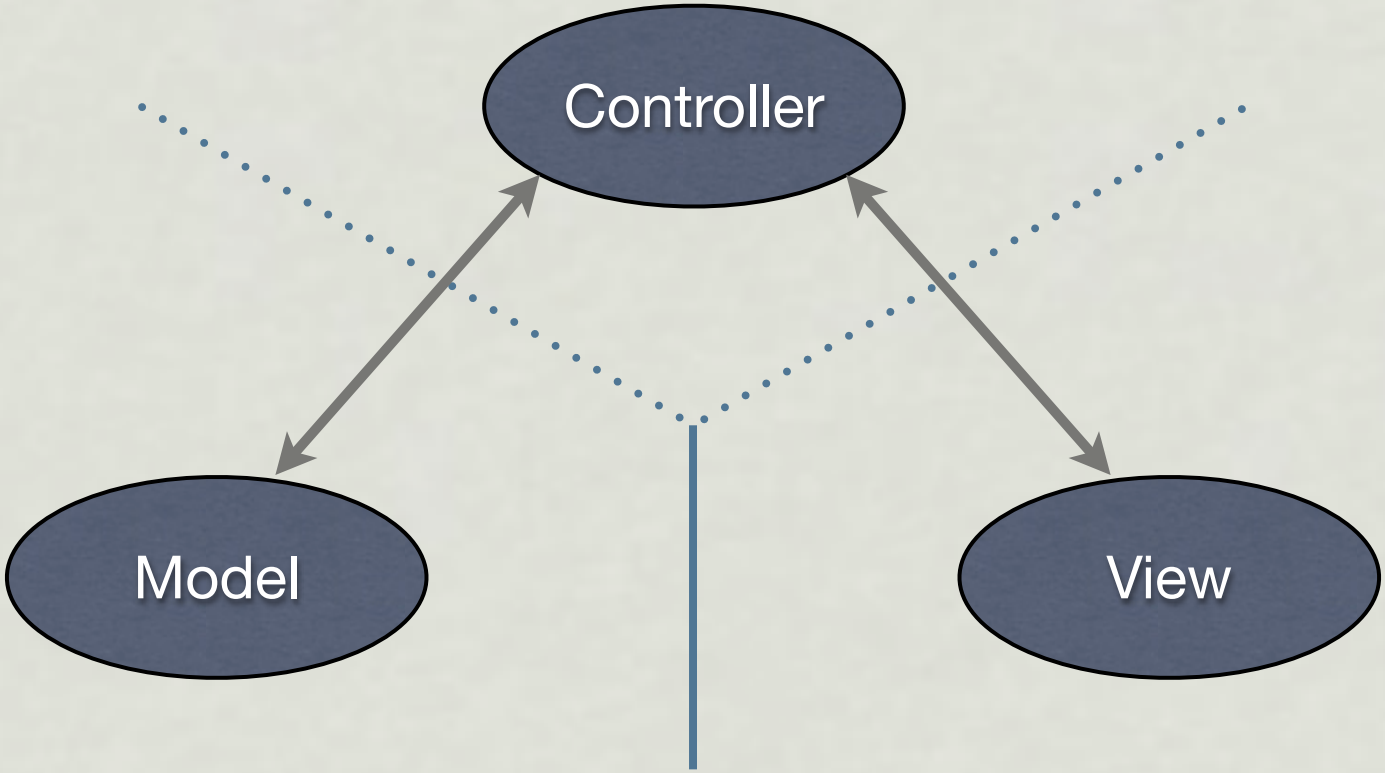
Controller & View

- Controller updates view, responds to view actions



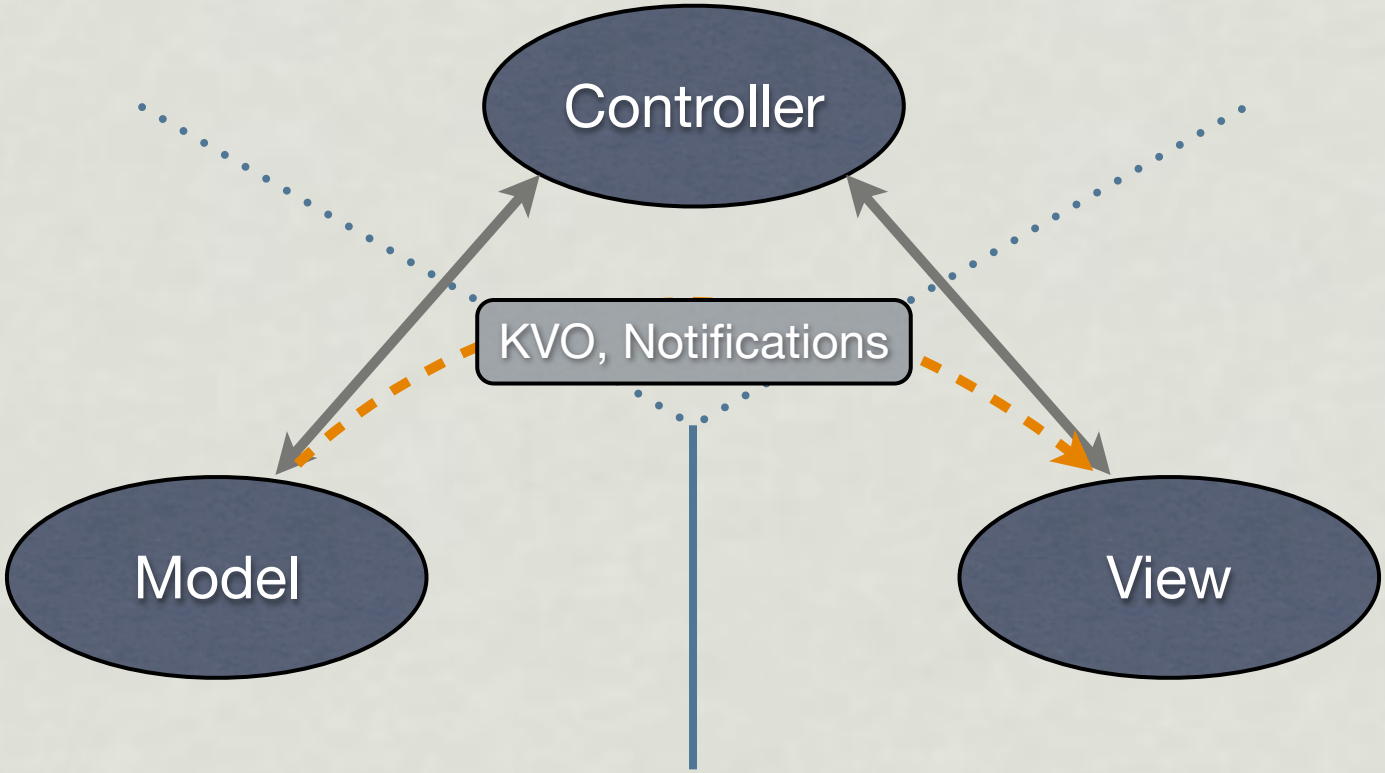
Controller & View

- Hybrid – Watch for changes without needing a controller

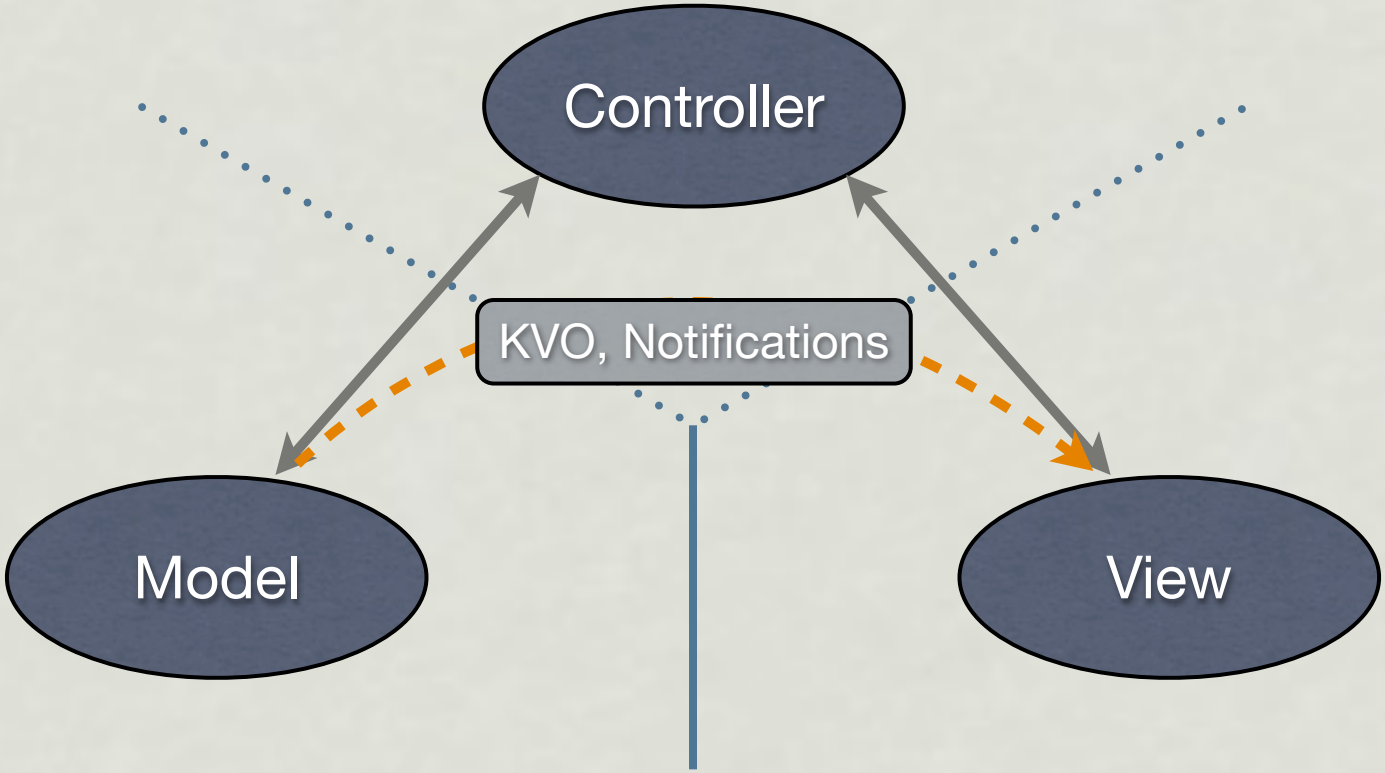


Controller & View

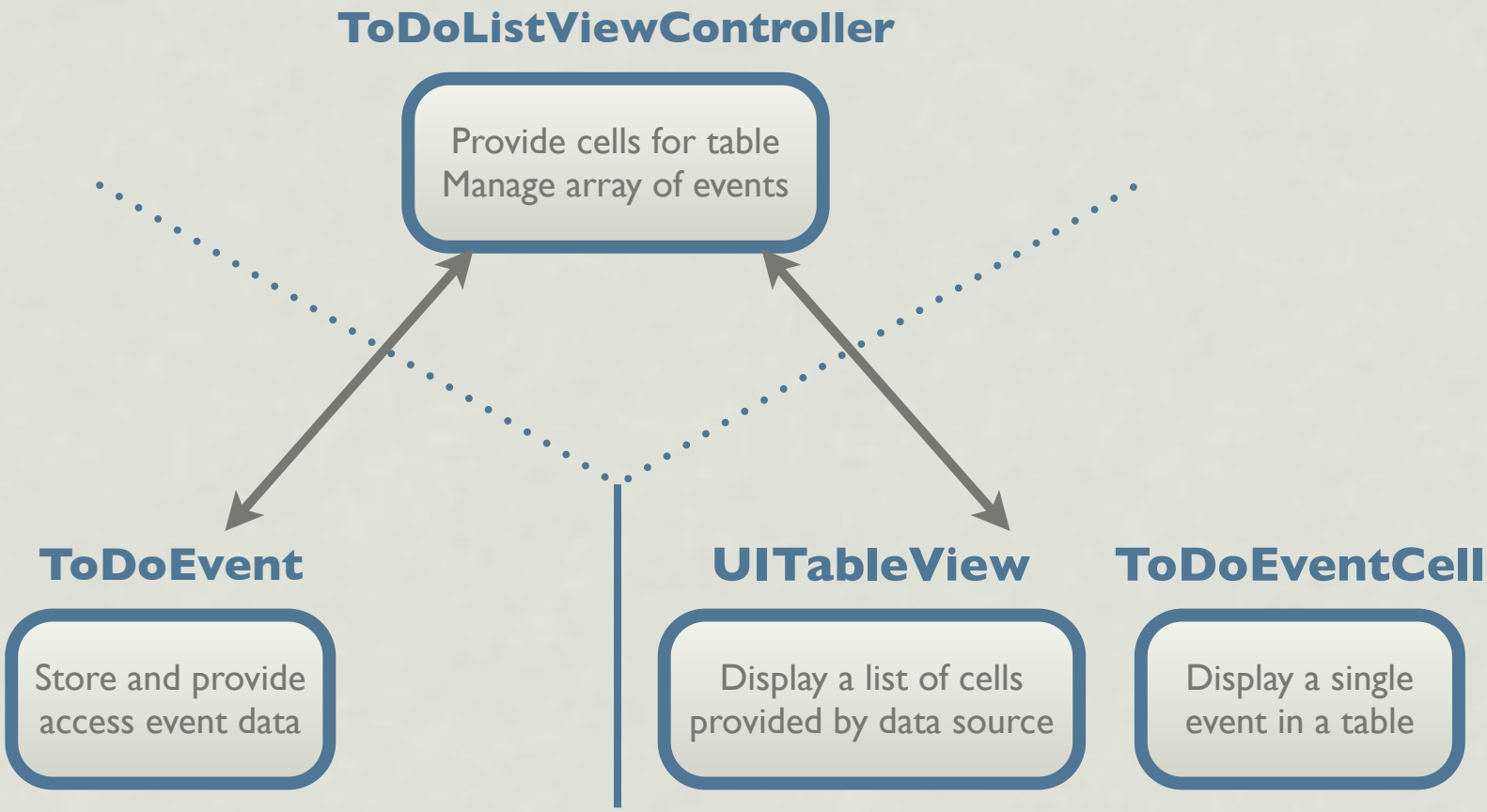
- Hybrid – Watch for changes without needing a controller



Controller & View



In Practice


ToDoListViewController
Provide cells for table
Manage array of events
ToDoEvent
Store and provide
access event data
UITableView
Display a list of cells
provided by data source
ToDoEventCell
Display a single
event in a table

Model - View - Controller

- Why are we talking about it?
- Benefits of MVC
  - **Reusability** - model, and view code tend to be reusable
  - **Reduced Complexity** – clean separation
    - *Extensibility* - code more adaptable to change
    - *Maintainability* - produce more maintainable code
  - Clean division of labor allows better independent development
  - Model objects are easily unit testable

Model - View - Controller

- Why are we talking about it?
- Benefits of MVC
  - **Reusability** - model, and view code tend to be reusable
  - **Reduced Complexity** – clean separation
    - *Extensibility* - code more adaptable to change
    - *Maintainability* - produce more maintainable code
  - Clean division of labor allows better independent development
  - Model objects are easily unit testable
- Understand MVC to effectively use iPhone APIs

Model

- Data – maintains, provides access, logic to manipulate
- Views and Controller may directly reference the Model, but not the other way around
- Models communicate with views and controllers anonymously using notifications

Model

- Data – maintains, provides access, logic to manipulate
- Views and Controller may directly reference the Model, but not the other way around
- Models communicate with views and controllers anonymously using notifications
- Reasons to keep model logic separate
  - Cleanly separated models are easily unit testable
  - Typically platform agnostic, reusable for other platforms
  - Avoids need to maintain and synchronize state when one object is responsible for storage

Model Parts

- **Instance Variables** - declare variables to hold your data
- **Accessors / Properties** - provide ways to get / set values
- **Initialization** - set instance variable to reasonable default values
- **Deallocation** - release referenced objects
- **Encoding** - provide ability to persist your object
- **Copying** - allow clients to get a snapshot of your object as a copy
- **Observation** - clients can observe object changes using built in KVO mechanism, you may want to post your own notifications

Model Example

- `ToDoEvent`
- Stores a `NSDate` instead of a formatted date string
  - Avoids making assumptions about display
  - Leaves formatting up to the view
- Observable Changes
  - Automatic through **Key-Value Observing** (*more on this later*)
  - Or coordinated by the controller

View

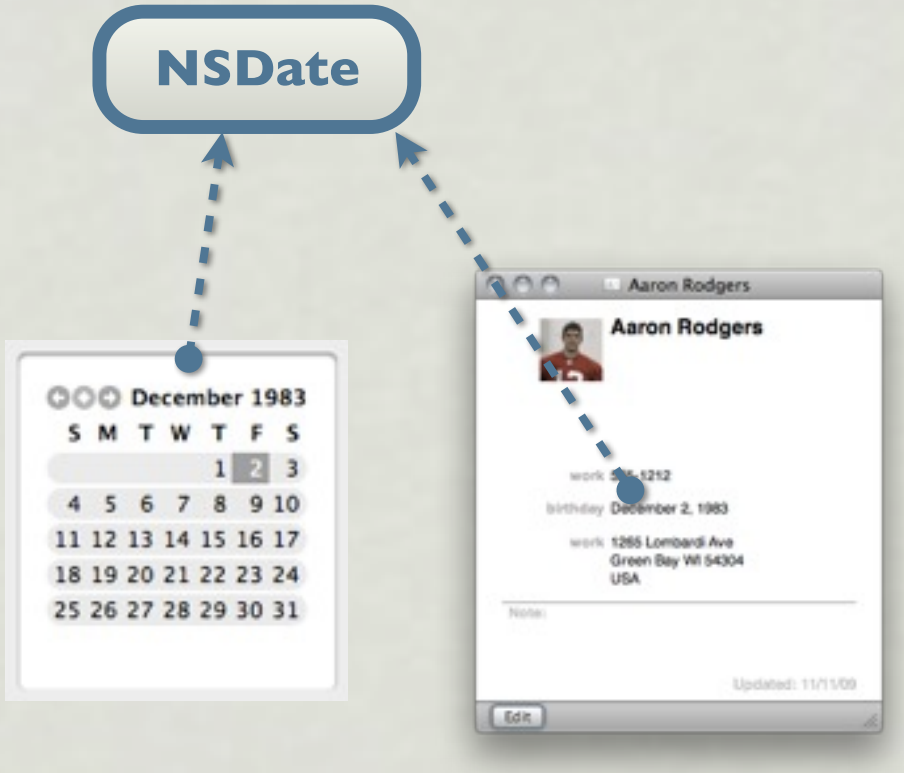
- Present information to users

View

- Present information to users
- Reasons to keep view logic separate
  - Views more reusable when generic and not specific to the data
  - Simultaneous display by different views of the same data
    - Present it in different ways
    - Present different parts of the same model

View Example

- **Model**: Date – in this case, taken from a `Person` object
- **Generic Display** – one view shows a formatted date, the other a calendar view. These could be reused by anywhere...



View Example

- **Model**: `CalcHistory` – stores calculator result history
- **Multiple Views** – one view shows the entire history, the other just the most recent item.

View Example

- **Model**: `ToDoEvent`
- **Multiple Views** – One cell displays title and time, the other shows title and full date. One displays past due in Red.

Controllers

- Coordinate data flow between Model and View
- Purpose
  - Respond to delegation messages from views
  - Respond to action messages from controls
  - Respond to notifications from models and other objects
  - Establish connections between managed objects

Controllers

- Coordinate data flow between Model and View
- Purpose
  - Respond to delegation messages from views
  - Respond to action messages from controls
  - Respond to notifications from models and other objects
  - Establish connections between managed objects
- The one object in MVC that knows a lot about the others
  - Typically (but not always) not very reusable

MVC Architecture

- **Example** – Address Book

MVC Architecture

- **Example** – Address Book
- User edits a persons phone number in a text field

MVC Architecture

- **Example** – Address Book
- User edits a persons phone number in a text field
- Change communicated to controller by an action

MVC Architecture

- **Example** – Address Book
- User edits a persons phone number in a text field
- Change communicated to controller by an action
- Controller may interpret the value
    - E.g. If not enough digits were entered, reject the change

MVC Architecture

- **Example** – Address Book
- User edits a persons phone number in a text field
- Change communicated to controller by an action
- Controller may interpret the value
    - E.g. If not enough digits were entered, reject the change
- Controller updates model object

MVC Architecture

- **Example** – Address Book
- User edits a persons phone number in a text field
- Change communicated to controller by an action
- Controller may interpret the value
    - E.g. If not enough digits were entered, reject the change
- Controller updates model object
- View is told to redisplay
    - Controller could drive this
    - View could use some other observation mechanism

Data Flow

Data Flow

- If a view knows nothing about the model
  - How does it know what to display?
  - How does it know when to display?

Data Flow

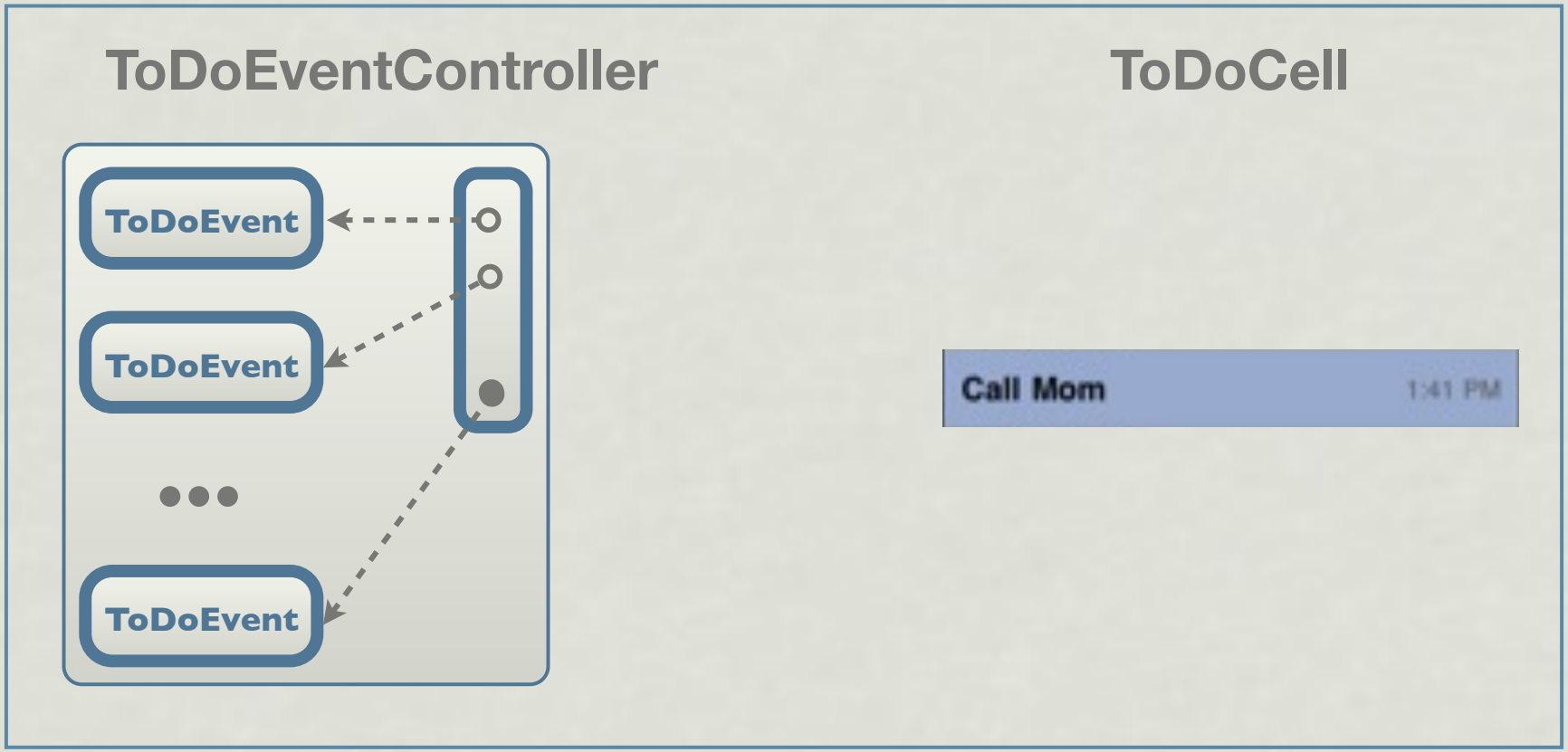
- Views really **do** know about the model
    - They don’t manage or manipulate them…

Data Flow

- Views really **do** know about the model
  - They don't manage or manipulate them…
- Getting the model
  - Push
    - Controller pushes data to the view using "setter" method
    - Changes communicated by calling setter methods again
  - Pull
    - View pulls data from controller using "data source" methods
    - Changes communicated by telling view to "reload"

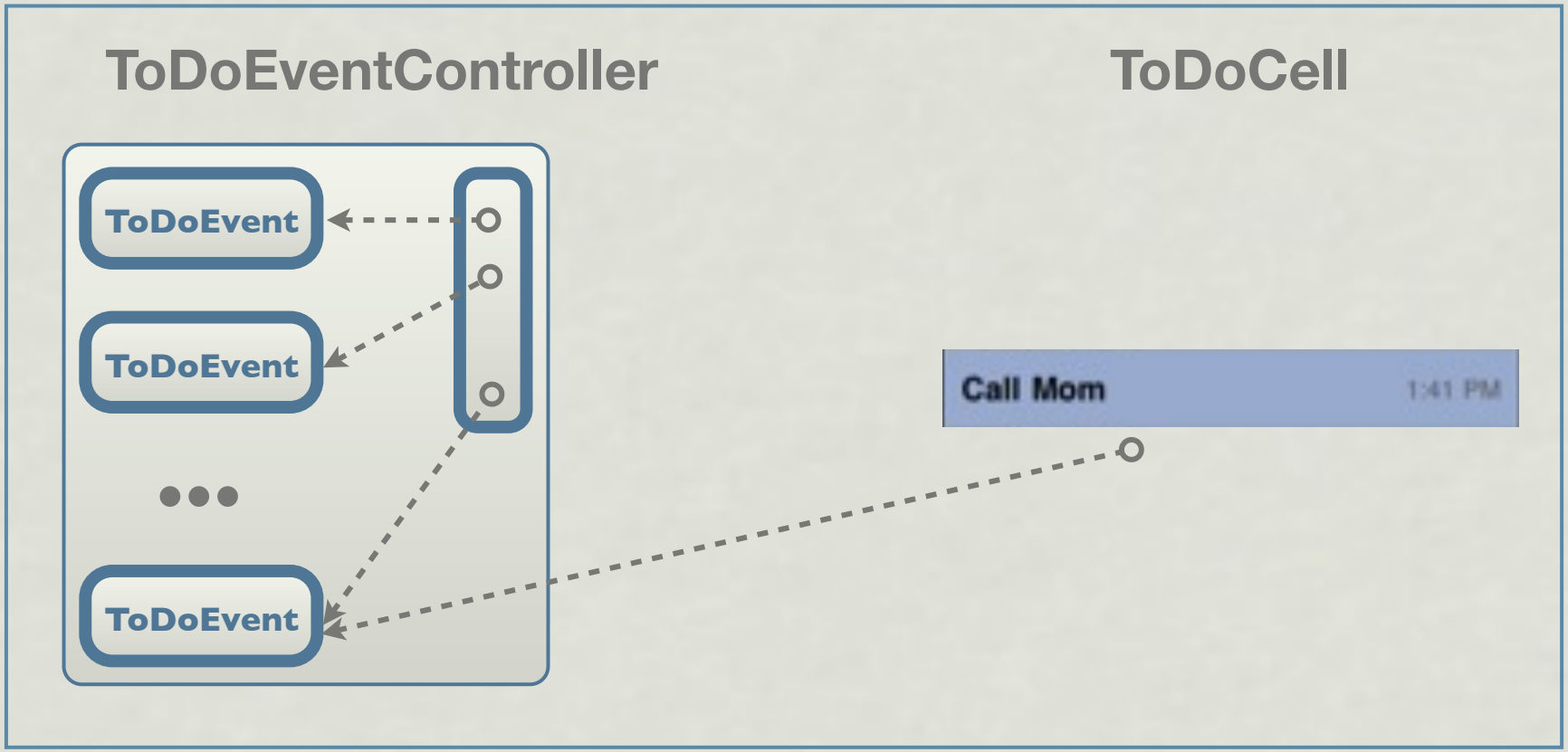
Push An Object (reference)

- Controller passes a reference by calling `[cell setToDoEvent:]`



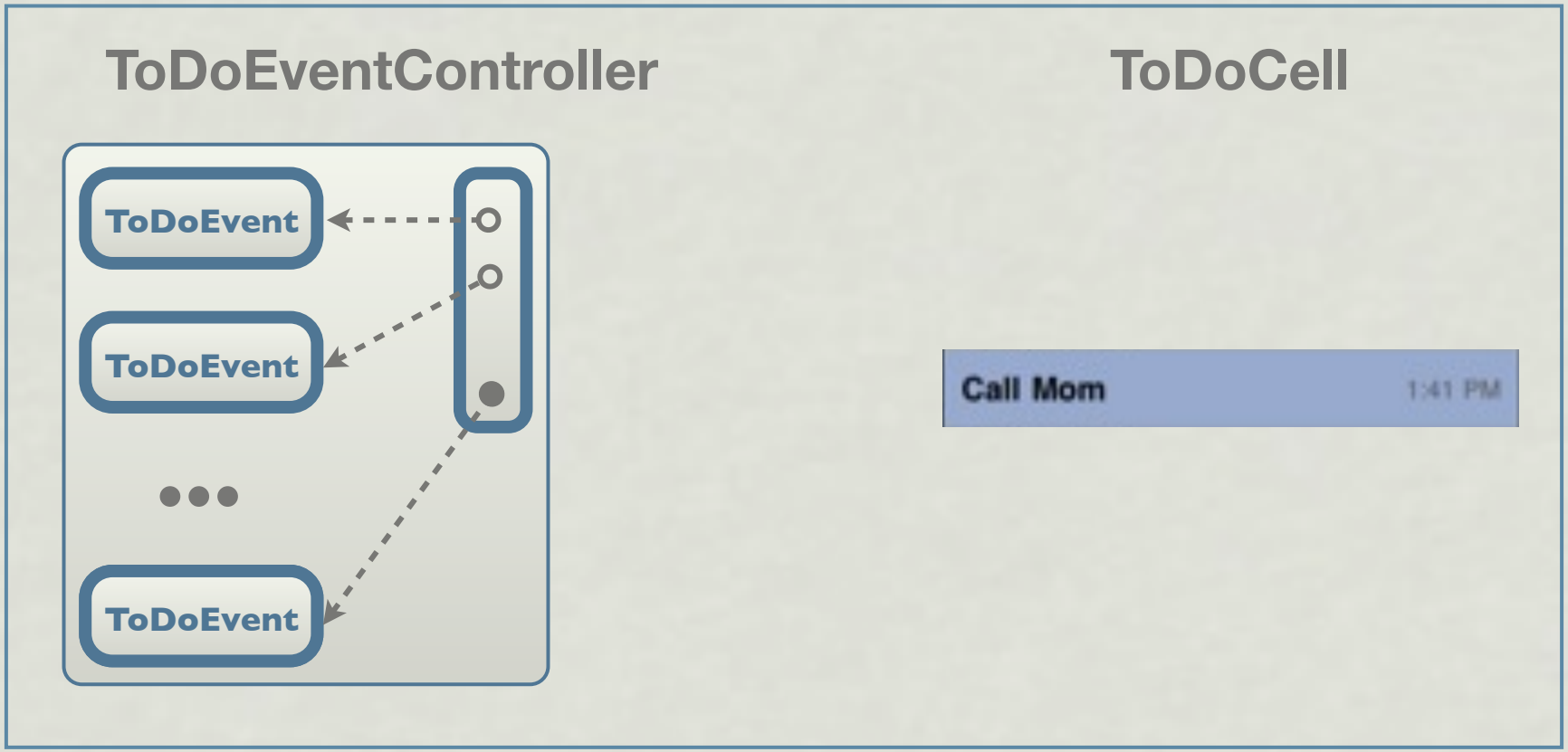
Push An Object (reference)

- Controller passes a reference by calling `[cell setToDoEvent:]`



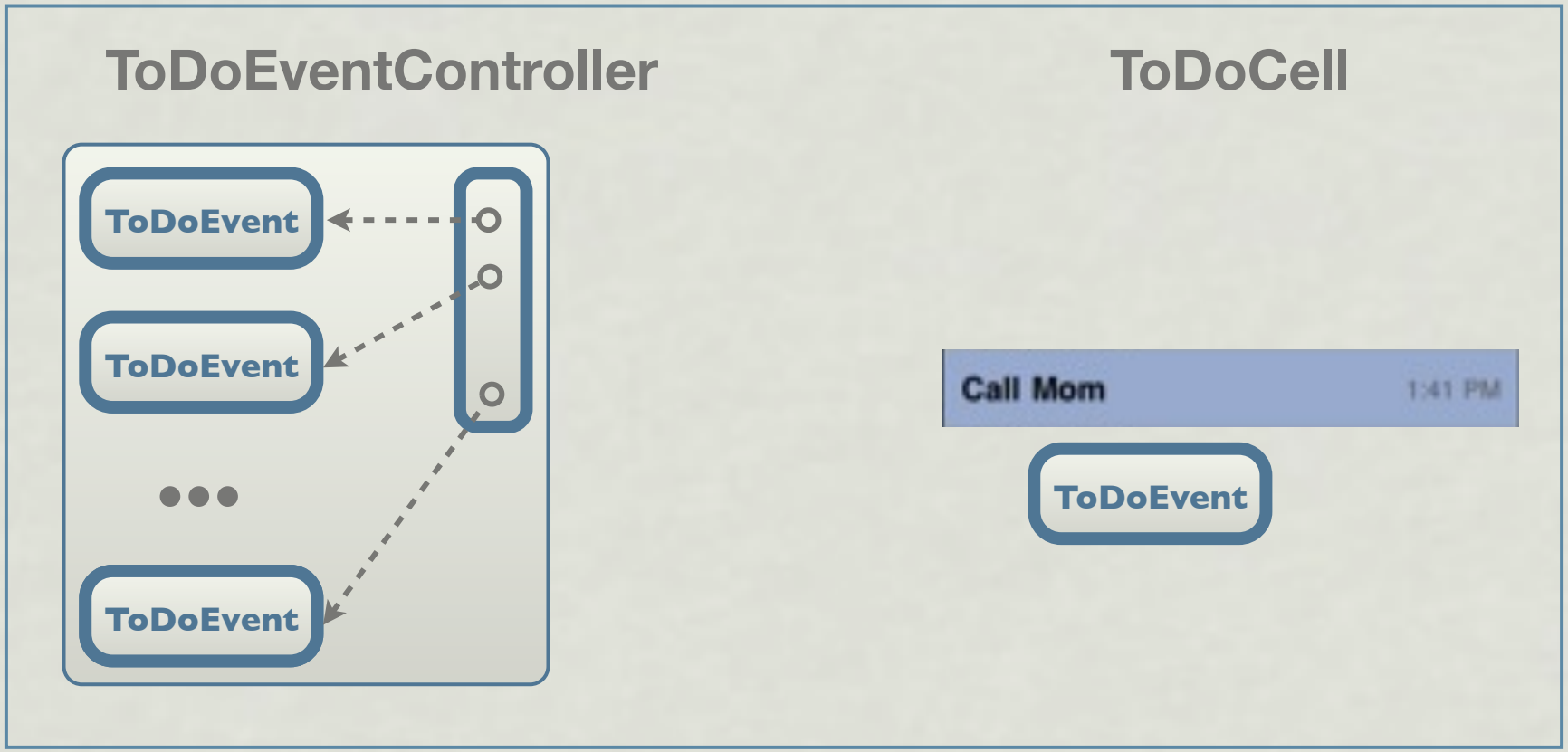
Push An Object (copy)

- Can also use a copy if the situation warrants it



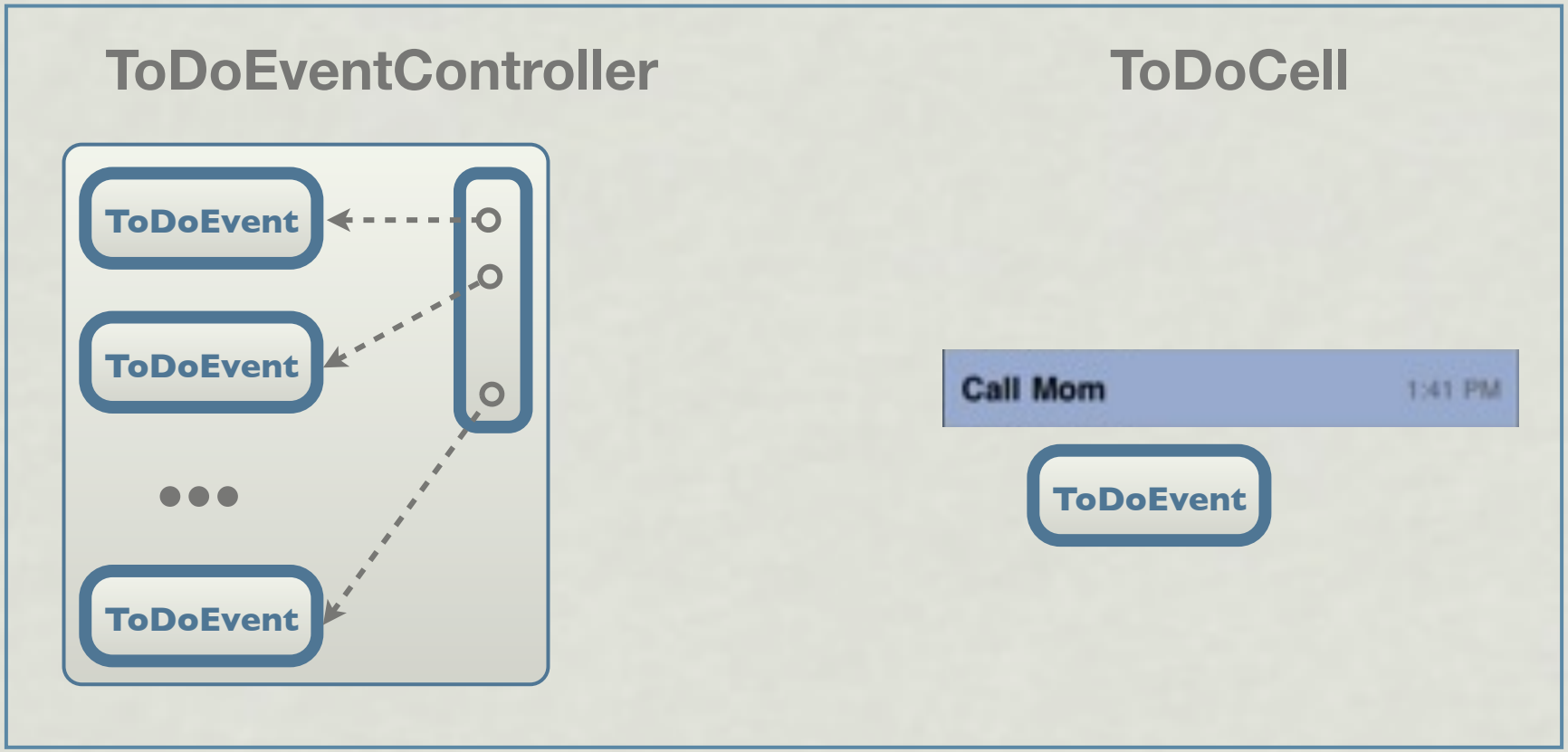
Push An Object (copy)

- Can also use a copy if the situation warrants it



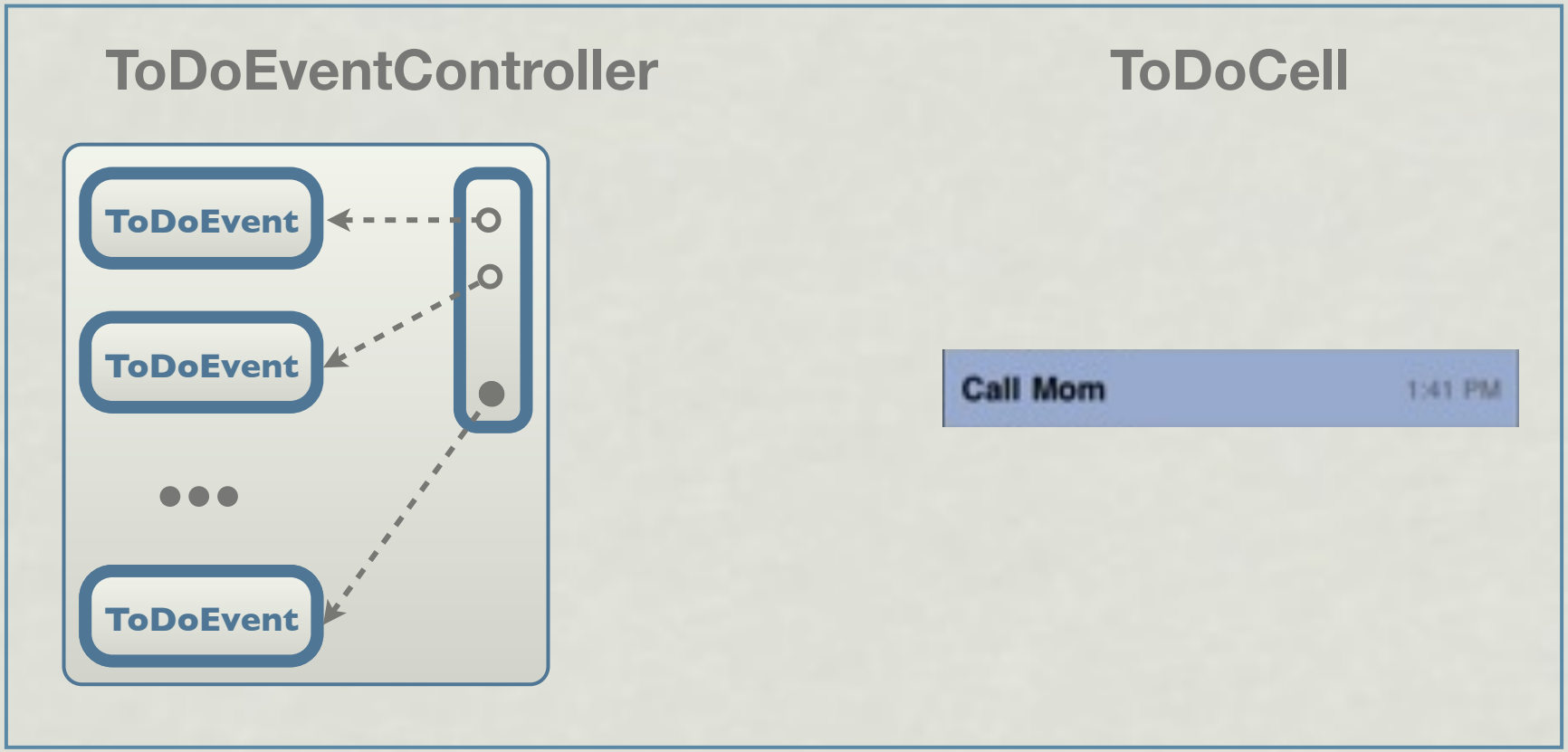
Push An Object (copy)

- Can also use a copy if the situation warrants it



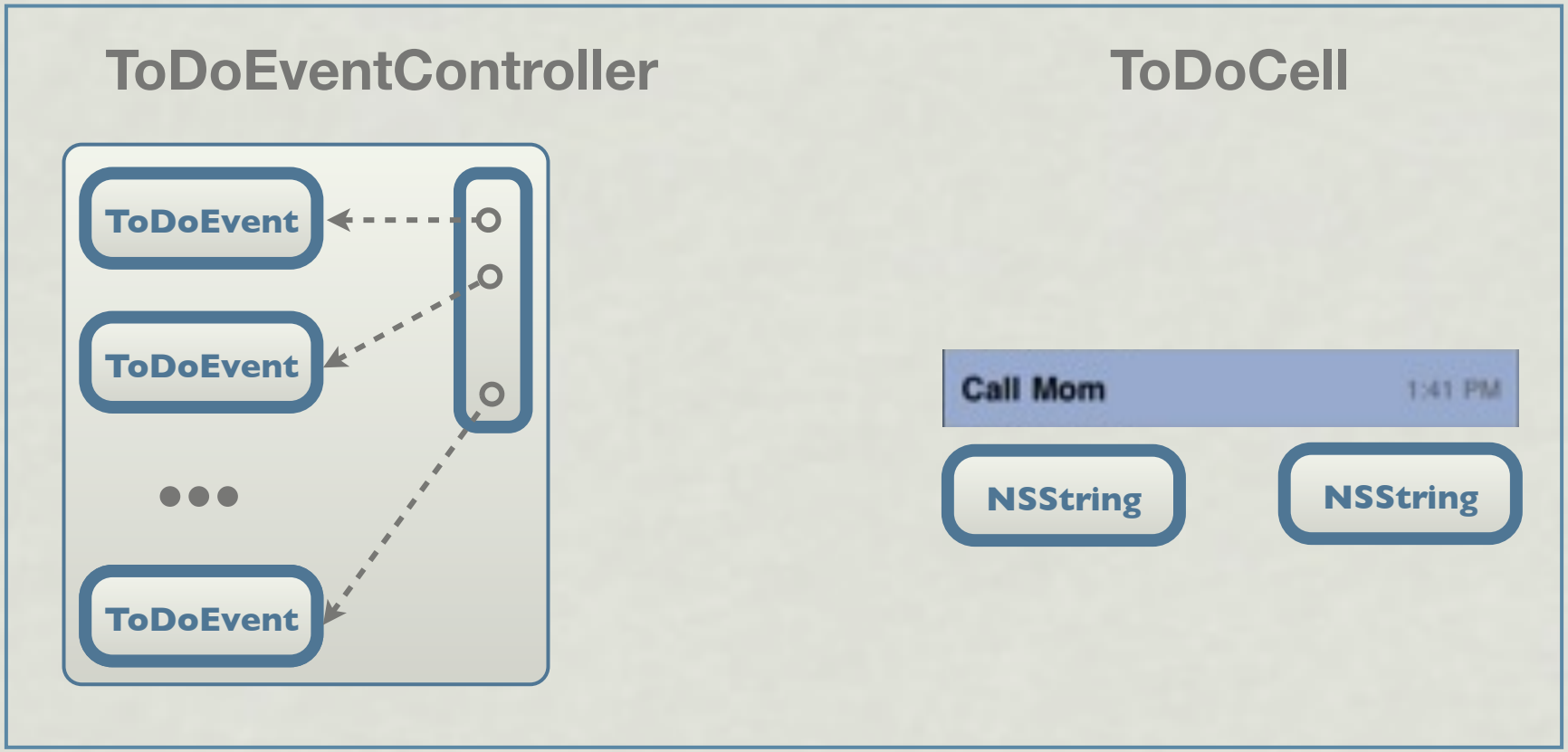
Push An Object

- Code is more reusable if you can stick to generic types



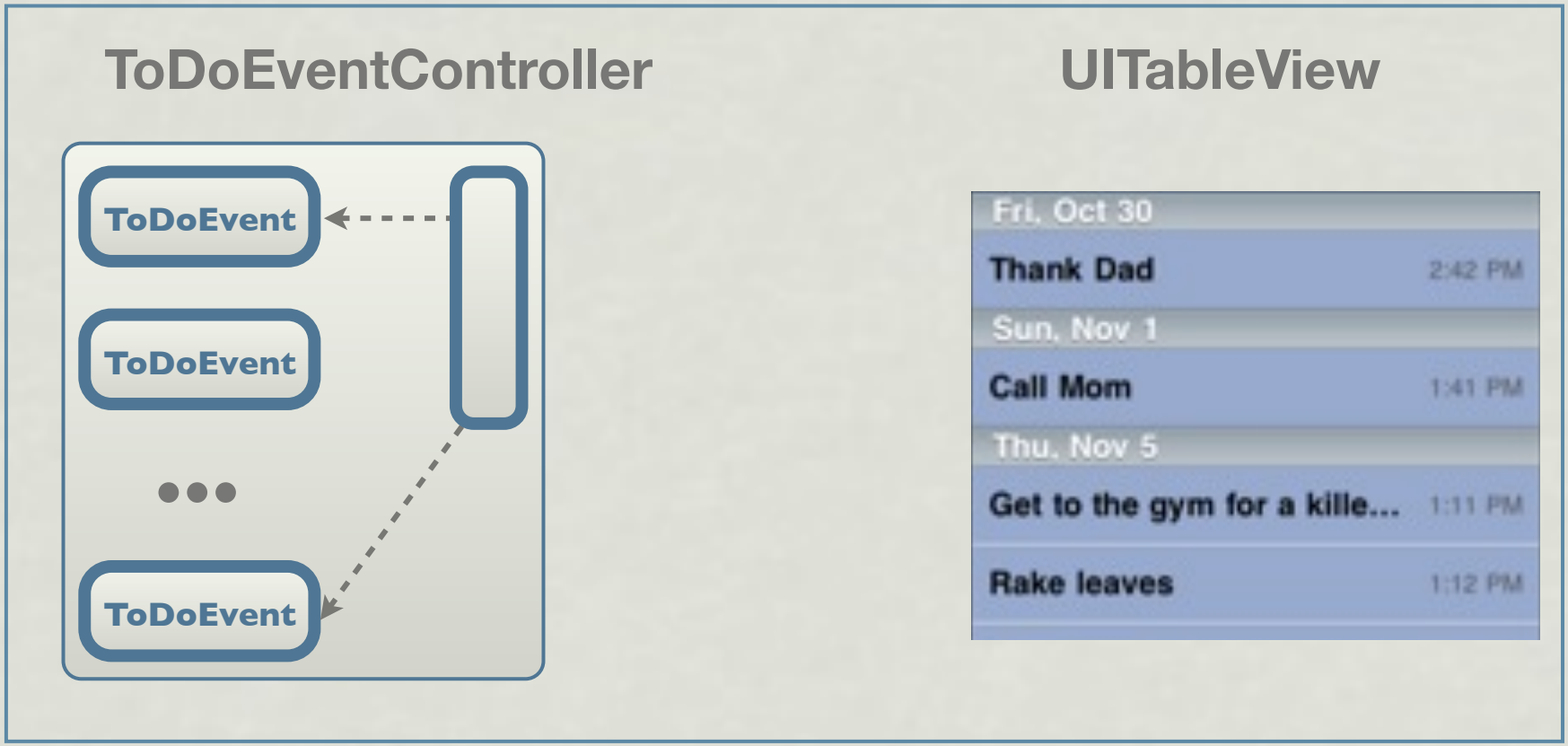
Push An Object

- Code is more reusable if you can stick to generic types



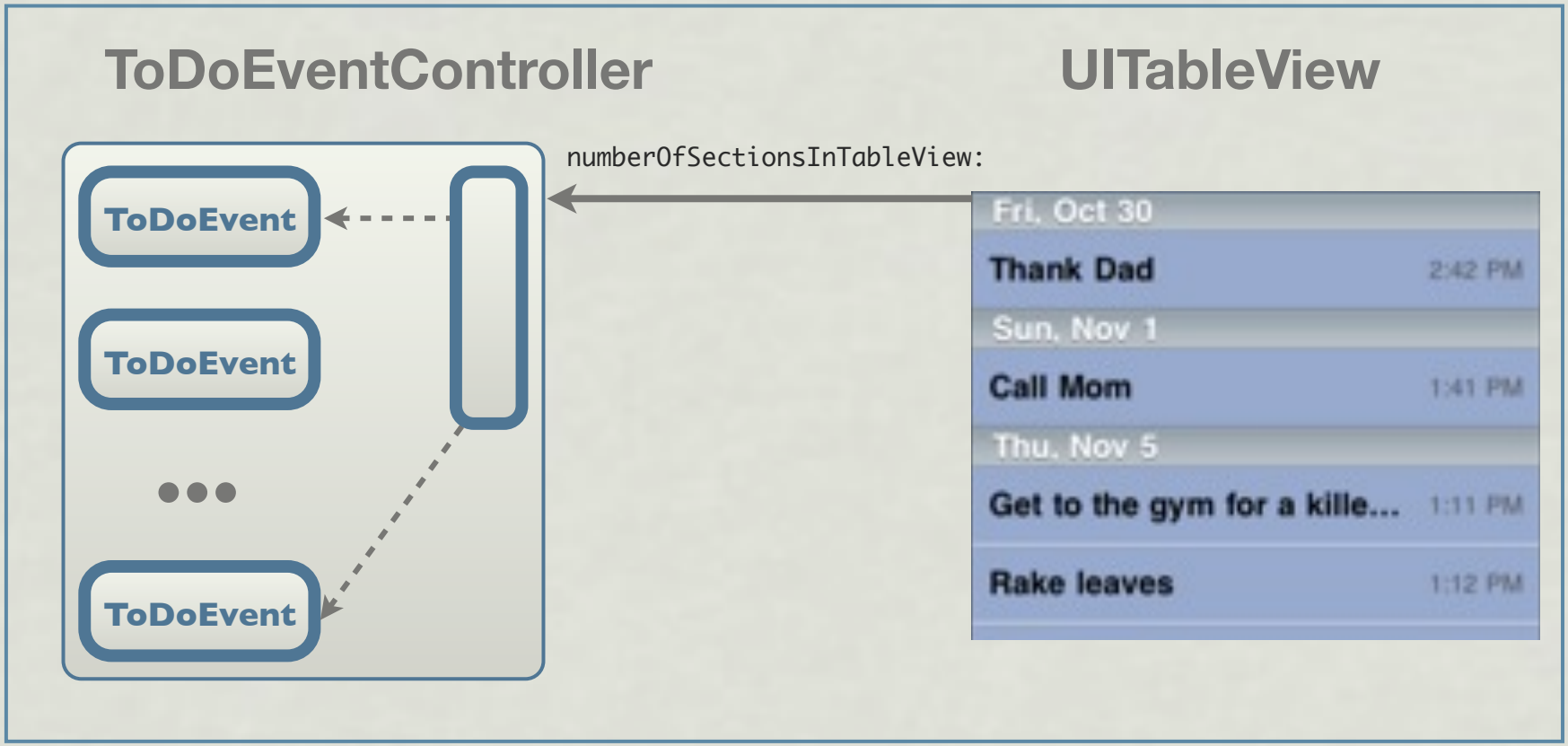
Pull – Example

- `UITableView` gets the number of sections from its data source
  - Some time after you call `[table reloadData]`



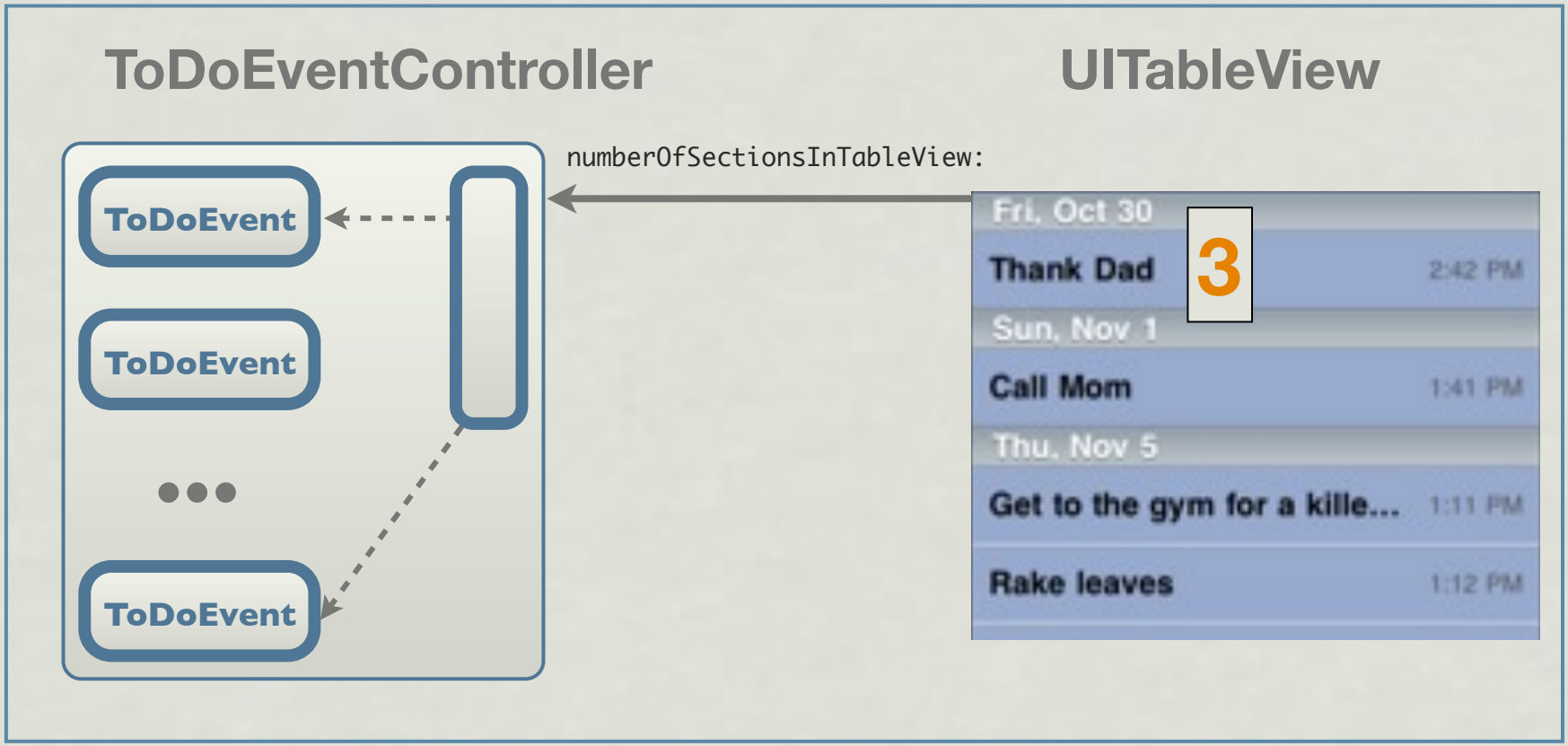
Pull – Example

- `UITableView` gets the number of sections from its data source
  - Some time after you call `[table reloadData]`



Pull – Example

- `UITableView` gets the number of sections from its data source
  - Some time after you call `[table reloadData]`



Data Changes

Updating View Data

- Controller Mediation
  - “setter” (push)
  - “reload” (pull)
- Views may also discover changes using observation
  - **NSNotification** - A model may define change notifications
  - **Key-Value Coding** – Cocoa mechanism that allows objects to be notified of changes to specific properties of other objects

Notifications

- Listen for `NSNotifications` signifying value changes

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver: self
       selector: @selector(eventDueDateChanged:)
           name: ToDoEventDueDateChangedNotification
         object: event_];
```

Notifications

- Listen for `NSNotifications` signifying value changes

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver: self
       selector: @selector(eventDueDateChanged:)
           name: ToDoEventDueDateChangedNotification
         object: event_];
```

- Register for a named notifications

Notifications

- Listen for `NSNotifications` signifying value changes

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver: self
       selector: @selector(eventDueDateChanged:)
           name: ToDoEventDueDateChangedNotification
         object: event_];
```

- Register for a named notifications
- Provide a callback – target object, and method to be invoke

Notifications

- Listen for `NSNotifications` signifying value changes

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver: self
       selector: @selector(eventDueDateChanged:)
           name: ToDoEventDueDateChangedNotification
         object: event_];
```

- Register for a named notifications
- Provide a callback – target object, and method to be invoke
- Declare which object you are interested in observing

Notifications – Example

- Monitors for changes using `NSNotifications`

```
- (void)setToDoEvent:(ToDoEvent *)event {
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    if (event != event_) {
      [nc removeObserver:self name:ToDoEventDueDateChangedNotification object:event_];
      [event_ release];
      event_ = [event retain];
      [nc addObserver:self selector:@selector(eventDueDateChanged:)
                 name:ToDoEventDueDateChangedNotification object: event_];
    }
}
```

Notifications – Example

- Monitors for changes using `NSNotifications`

```
- (void)setToDoEvent:(ToDoEvent *)event {
    NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
    if (event != event_) {
      [nc removeObserver:self name:ToDoEventDueDateChangedNotification object:event_];
      [event_ release];
      event_ = [event retain];
      [nc addObserver:self selector:@selector(eventDueDateChanged:)
                 name:ToDoEventDueDateChangedNotification object: event_];
    }
}
```

- Notification callback

```
- (void)eventDueDateChanged:(NSNotification *)note {
    ToDoEvent *event = [notification object]; // should be == event_
    NSDictionary *userInfo = [notification userInfo];

    [label_ setText:[event_ dueDate]];

}
```

Notification Posting

- Posting your own notifications

Notification Posting

- Posting your own notifications

- Declare the notification name

```
// ToDoEvent.h
extern NSString *const ToDoEventDueDateChangedNotification;
```

```
// ToDoEvent.m
NSString *const ToDoEventDueDateChangedNotification = @"ToDoEventDueDateChangedNotification";
```

Notification Posting

- Posting your own notifications
- Declare the notification name

```
// ToDoEvent.h
extern NSString *const ToDoEventDueDateChangedNotification;
```

```
// ToDoEvent.m
NSString *const ToDoEventDueDateChangedNotification = @"ToDoEventDueDateChangedNotification";
```

- Posting

```
- (void)setDueDate:(NSDate *)date {
   ....
   NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
   NSDictionary *userInfo = ........;

   [nc postNotificationName:ToDoEventDueDateChangedNotification
                     object:self
                   userInfo:userInfo];
}
```

Key-Value Observing

- Directly observe an object property for changes

```
ToDoEvent *event_;
```

```
[event_ addObserver:self
         forKeyPath:@"dueDate"
            options:0
            context:NULL];
```

Key-Value Observing

- Directly observe an object property for changes

```
ToDoEvent *event_;
```

```
[event_ addObserver:self
         forKeyPath:@"dueDate"
            options:0
            context:NULL];
```

- Observe a property by name, using a **key** (or **keypath**)

Key-Value Observing

- Directly observe an object property for changes

```
ToDoEventDetailController *detailController;
```

```
[detailController addObserver:self
                   forKeyPath:@"event.dueDate"
                      options:0
                      context:NULL];
```

- Observe a property by name, using a **key** (or **keypath**)

Key-Value Observing

- Directly observe an object property for changes

```
ToDoEvent *event_;
```

```
[event_ addObserver:self
         forKeyPath:@"dueDate"
            options:0
            context:NULL];
```

- Observe a property by name, using a **key** (or **keypath**)

Key-Value Observing

- Directly observe an object property for changes

```
ToDoEvent *event_;
```

```
[event_ addObserver:self
         forKeyPath:@"dueDate"
            options:0
            context:NULL];
```

- Observe a property by name, using a **key** (or **keypath**)
- Specify observation options - See `NSKeyValueObserving.h`

Key-Value Observing

- Directly observe an object property for changes

```
ToDoEvent *event_;
```

```
[event_ addObserver:self
         forKeyPath:@"dueDate"
            options:0
            context:NULL];
```

- Observe a property by name, using a **key** (or **keypath**)
- Specify observation options - See `NSKeyValueObserving.h`
- Supplied **context pointer** is passed to you in KVO's callback

```
- (void)observeValueForKeyPath:(NSString *)keyPath
                      ofObject:(id)object
                        change:(NSDictionary *)change
                       context:(void *)context;
```

KVO – Example

- Monitor for changes using KVO

```
- (void)setToDoEvent:(ToDoEvent *)event {
    if (event_ != event) {
        [event_ removeObserver:self forKeyPath:@"dueDate"];
        [event_ release];
        event_ = [event retain];
        [event_ addObserver:self forKeyPath:@"dueDate" options:0 context:NULL];
    }
}
```

KVO – Example

- Monitor for changes using KVO

```
- (void)setToDoEvent:(ToDoEvent *)event {
    if (event_ != event) {
        [event_ removeObserver:self forKeyPath:@"dueDate"];
        [event_ release];
        event_ = [event retain];
        [event_ addObserver:self forKeyPath:@"dueDate" options:0 context:NULL];
    }
}
```

- KVO Callback

```
- (void)observeValueForKeyPath:(NSString *)keyPath ofObject:(id)object
                        change:(NSDictionary *)change context:(void *)context
{
    if (object == event_ && [keyPath isEqual:@"dueDate"]) {
        // update display
    }
}
```

Key-Value Observing

- KVO notifications posted automatically* by Cocoa
- Automatic when you use
  - Setters : `[event setDueDate: aDueDate];`
  - Dot notation : `event.dueDate = aDueDate;`
  - Key-Value Coding : `[event setValue: aDueDate forKey:@"dueDate"];`

Key-Value Observing

- KVO notifications posted automatically* by Cocoa
- Automatic when you use
  - Setters : `[event setDueDate: aDueDate];`
  - Dot notation : `event.dueDate = aDueDate;`
  - Key-Value Coding : `[event setValue: aDueDate forKey:@"dueDate"];`
- Not automatic for directly accessed outside of a “set” method

Notification / KVO Cleanup

- Stop observing in your dealloc!
- Stop observing when you no longer care to observe

Notification / KVO Cleanup

- Stop observing in your dealloc!
- Stop observing when you no longer care to observe
- Notifications

```
// NSNotificationCenter
- (void)removeObserver:(id)observer;
- (void)removeObserver:(id)observer name:(NSString *)name object:(id)obj;
```

Notification / KVO Cleanup

- Stop observing in your dealloc!
- Stop observing when you no longer care to observe
- Notifications

```
// NSNotificationCenter
- (void)removeObserver:(id)observer;
- (void)removeObserver:(id)observer name:(NSString *)name object:(id)obj;
```

- KVO

```
// NSObject ( NSKeyValueObserverRegistration )
- (void)removeObserver:(NSObject *)observer forKeyPath:(NSString *)kpath;
```

Key-Value *

KV*

- KVO – **Key-Value Observing**
  - Allows observing property changes of another object by name
- KVC – **Key-Value Coding**
  - Allows access to properties of another object by name

KVO

- Already discussed this…

Key-Value Coding

- Can work with a value by “name” instead
- Example

```
@property(retain, readwrite) NSString *title;
```

```
[object setTitle:@”a title”];
NSLog(@”the title is %@”, [object title]);
```

Key-Value Coding

- Can work with a value by “name” instead
- Example

```
@property(retain, readwrite) NSString *title;
```

```
[object setValue:@”a title” forKey:@”title”];
NSLog(@”the title is %@”, [object valueForKey:@”title”]);
```

Key-Value Coding

- Can work with a value by “name” instead
- Example

```
- (void)setPriority:(float)priority;
- (float)priority;
```

```
[object setValue:[NSNumber numberWithFloat:3.14] forKey:@”priority”]
NSLog(@”the priority is %@”, [object valueForKey:@”priority”];
```

Key-Value Coding

- Can work with a value by “name” instead
- Example

```
- (void)setPriority:(float)priority;
- (float)priority;
```

```
[object setValue:[NSNumber numberWithFloat:3.14] forKey:@”priority”]
NSLog(@”the priority is %@”, [object valueForKey:@”priority”];
```

- KVC uses your methods if provided
- KVC will “box” and “unbox” scalar types if necessary

KV*

- KVC
- KVO

KV*

- KVC
- KVO
- KVB – **Key-Value Binding**
  - Uses KVO/KVC to keeps view's display in sync with model!
  - Establishes mediated connection between model and view
  - *Mac OS X Only*

KV*

- More on KV* online : “Key-Value Coding Programming Guide”
  - KVC method naming compliance
  - KVC defines good naming patterns to use in your own code
    - Must follow to work with KV*
  - To-Many, To-One relationships

Final Notes

Next Week

- View Controllers – Manage “screenful” of displayed data

Reading Assignments

- CocoaFundamentals.pdf
  - The Model-View-Controller Pattern: p.158 - 166
    - Ignore NSObject/Array/TreeController, …
- Key-Value Programming Guide (KeyValueCoding.pdf)
  - If Interested*
  - Intro Chapters: p.9 - 22