University of Regina

CS 455/855
Mobile Computing

Anatomy of an iOS Project

Dr. Orland Hoeber
orland.hoeber@uregina.ca
http://www.cs.uregina.ca/~hoeber/cs455/2018F

Readings

- App Programming Guide for iOS
  - About iOS App Architecture
  - Expected App Behaviours <for future reference>
  - The App Life Cycle
  - <skip four sections>
  - Performance Tips

- Online documentation, as necessary

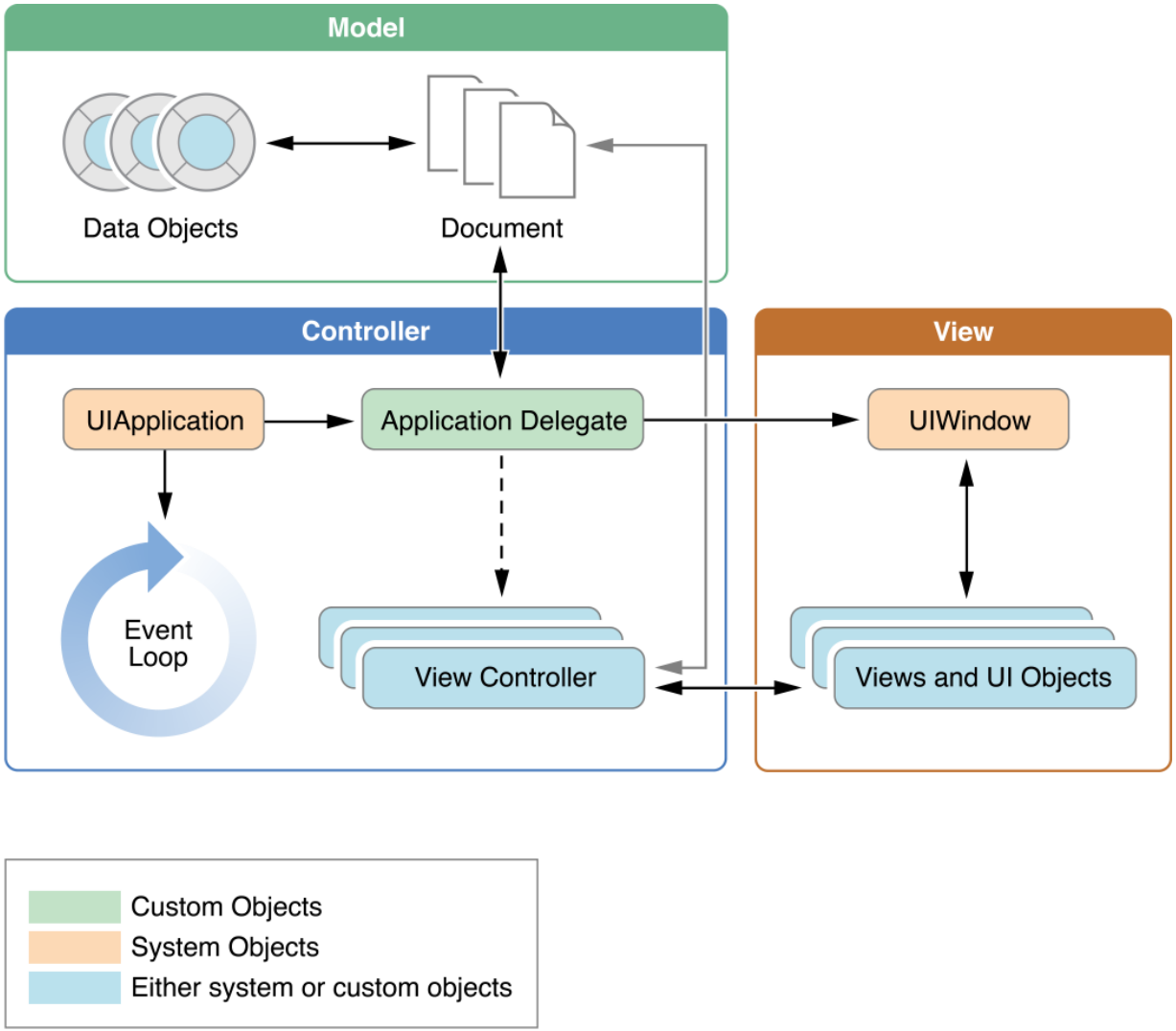
Swift & the iOS API

- App development for iOS consists a complex interplay between your code and the iOS API
  - the API and underlying system frameworks provide the basic infrastructure and overall “look & feel”
  - your code makes use of this basic functionality to do something interesting
  - what this means is that you should always try to use the built-in functionality first, and only develop new features if what you need does not exist
    - if you do build new features, try to build these by extending existing API functionality, rather than from scratch
- These principles hold for many high-level programming languages

Fundamental iOS Design Patterns

- Before we can start talking about how a typical iOS application is architected and built, we need to talk about some of the fundamental design patterns:
  - model-view-controller
    - design pattern that governs the overall architecture
  - target-action
    - design pattern that translates user interaction into the execution of specific code
  - delegation & protocols
    - a contract that specifies that a class will implement a set of pre-defined methods, so that it can be used for a specific purpose

Model-View-Controller



MVC Design Pattern

- The Model-View-Controller design pattern assigns objects in an application one of three roles:
  - model
  - view
  - controller
- The pattern defines:
  - the roles objects play
  - the way objects communicate with one another
- Each of the three types of objects is separated from the others by abstract boundaries, and communicate with objects of other types across those boundaries

Model Objects

- Model objects encapsulate the data specific to an application and defines the logic and computation that manipulates and processes the data
- Model objects can include and encapsulate other model objects
- The data that represents the persistent state of the application should reside in model objects
- Because model objects represent knowledge and expertise related to a specific problem domain, they can be reused in similar problem domains
- Ideally, a model object should have nothing to do with the presentation of the information is contains (its primary purpose is information storage and algorithmic manipulation/processing)

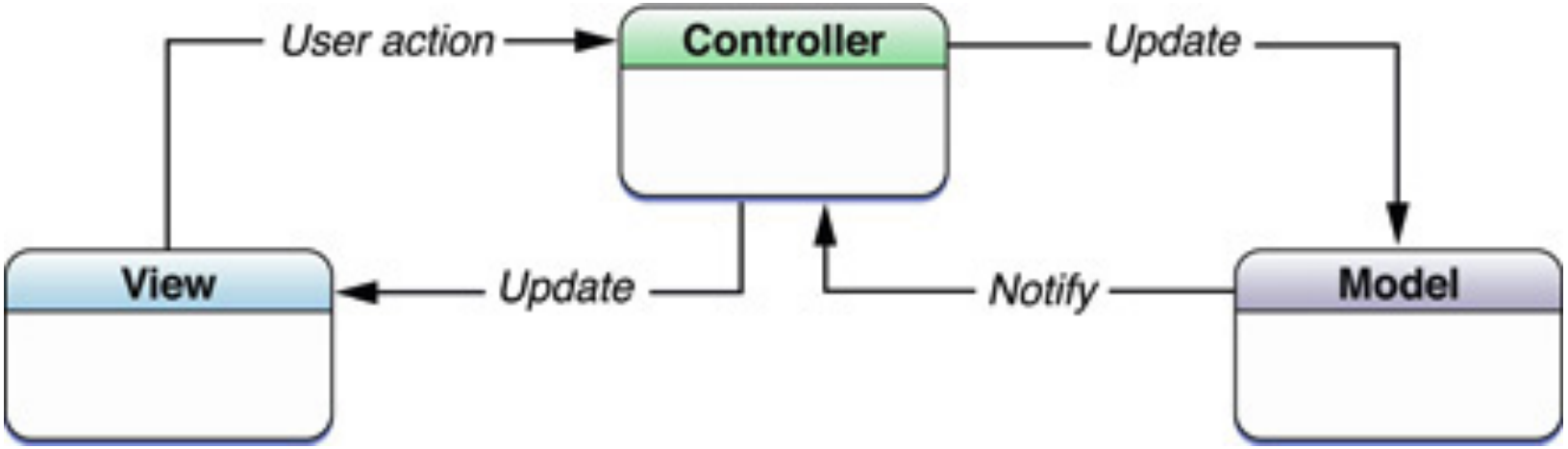
View Objects

- The primary purpose of view objects is to present information to the user, and to capture user input
  - a view object knows how to draw itself and can respond to user actions
  - we can use view objects is to display data from the application's model objects, and to enable the editing of the data
- The data (model) and the representation (view) are decoupled in order to promote object reuse

Controller Objects

- A controller object act as an intermediary between one or more of the application's view objects and one or more of its model objects
- Controller objects are a conduit through which view objects learn about changes in model objects, and vice versa
- Controller objects are the glue that binds model objects and view objects, facilitating their communication

MVC Communication



MVC Benefits

- The primary benefits for following the MVC design pattern are:
  - decoupling the model from the view promotes the reuse of these objects in other applications
  - because of the separation of model from view, changing requirements are more easily managed
  - encapsulating the different types of application logic into different types of objects makes it easier for a developer to know where to implement specific types of application behaviour
    - model: encapsulate data and data-centric behaviour
    - view: present information to the user
    - controller: tie the model and the view together

Xcode Templates & MVC

- Most of the templates provided by Xcode support the MVC design pattern
  - interface boilerplate for the view
  - boilerplate code for the controller
  - you need to add in the model
- These templates also include the code necessary to load the initial view and get things started
- What distinguishes one iOS app from another will primarily be the data it manages and how it presents this data to the user

Designing Applications Following MVC

- All of your iPhone applications developed in this course must follow the MVC design pattern
  - create separate objects to encapsulate the data (model objects)
  - use (or create) objects to present information to the user (view objects)
  - create controller objects that link the model and view objects

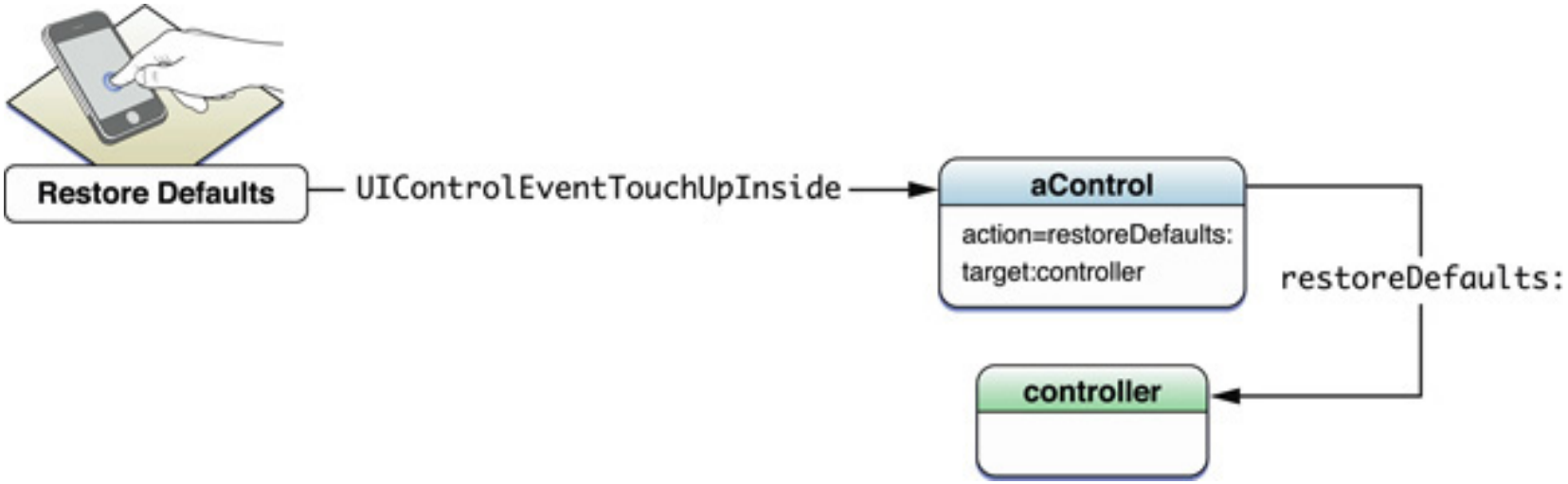
- The biggest mistake people make in following MVC is to create hybrid model-controller objects without having a good justification for doing so
  - to avoid this, think about encapsulating your model classes so that they can easily be reused in another project

Target-Action Design Pattern

- The target-action design pattern is event-based
  - an object that can can capture an event (a view, button, etc.; the target) is empowered to send a message to another object to indicate that an event has occurred (the action)
  - occurs in the View-Controller interface in MVC
  - supports the separation of view code from controller code
    - you don't write custom code in your view to tell it how to respond to an event
    - the custom code goes in the controller
    - the view is configured to know who to tell about the event

Target-Action Event Capturing

- Development steps:
  - configure the interface elements to capture the desired event (already done for built-in UI elements)
  - link the event to a method template in the View Controller
  - write the code for this method to make it do something useful when the event occurs



Target-Action & the Main Run Loop

- All events are processed by the main run loop
  - runs on the main thread to ensure that user events are processed serially and in the order they were received
  - we are primarily concerned (at this point) with touch events; but these aren't the only kinds of events
    - e.g., events may also be initiated by accessories, sensors, location-based services, or the network
  - most events target a specific responder object, but can be passed to other objects if needed

Delegation & Protocols

- Swift uses protocols to define a group of related methods
  - you can think of this as a work-around for not allowing multiple inheritance
- If a class implements these methods, then it can be used for a specific purpose by another class
  - the other class will know that it can send it messages that match the protocol because it said it conformed to the protocol
- If this seems confusing to you, think of this like networking/communication protocols

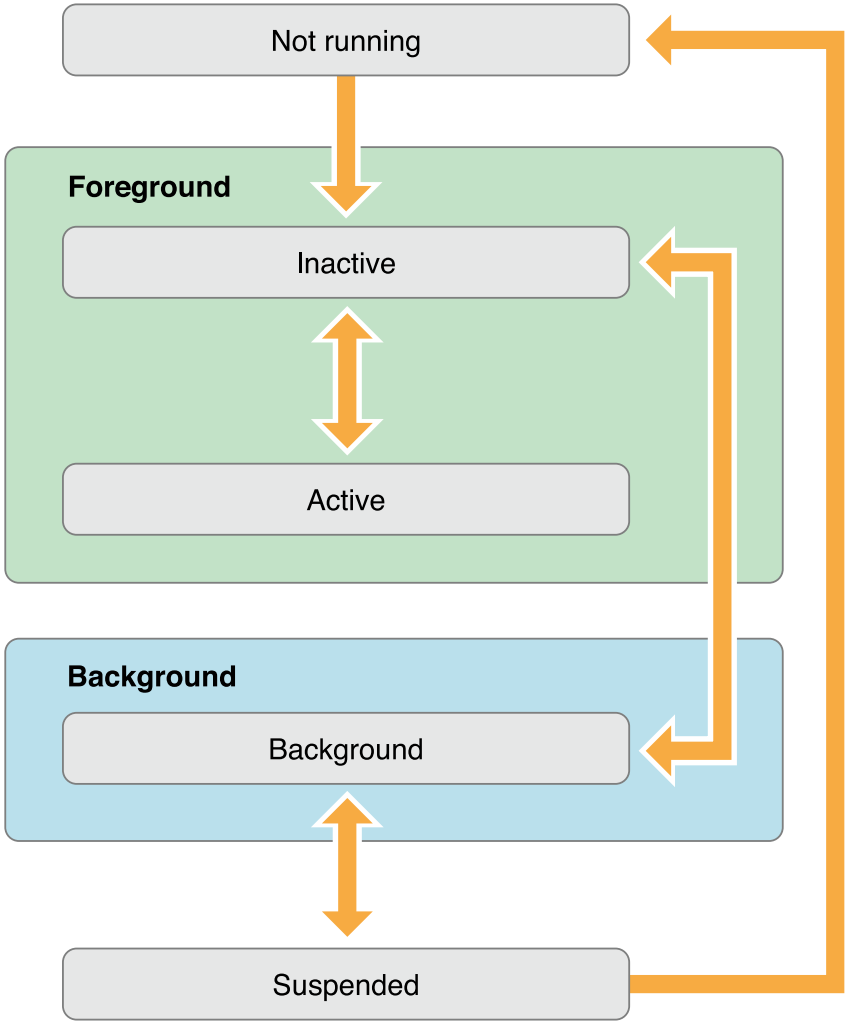
Delegation

- An example of this is when you are dealing with interface objects that allow for complex data entry.
  - in these cases, we don't want to have to put the code to manage this within the interface object
  - instead, we will tell it to talk to the view controller when certain situations arise; the view controller is the delegate for the interface object
  - these messages will be in a specific format, meaning that the view controller needs to implement specific methods
  - the protocol mechanism is how we ensure that the necessary methods are provided by the delegate object
  - example:
    - UITextField needs a view controller that conforms to the UITextFieldDelegate in order to know what to do when editing begins, when it ends, and when focus is lost

Execution States

- At any point in time, your app will be in one of four execution states
  - not running
    - has not been launched, or was terminated manually or by the system
  - inactive
    - running in the foreground, but not receiving any events
  - active
    - running in the foreground, and receiving events
  - background
    - executing code in the background
  - suspended
    - in the background, but not executing any code

Execution State Changes



App Termination

- Apps must be prepared for termination to happen at any time and without warning
  - system-initiated termination
    - normal process used to make room for a new app that the user wants to run
    - apps may also be terminated for misbehaving (excessive processor load, memory footprint, or not responding to events)
  - user-initiated termination
    - users can easily terminate an app
- You should never wait to save data or do some necessary processing

Threads and Concurrency

- iOS supports multi-threading and concurrent execution
- Use this whenever you need to do some non-trivial data processing or when you have to wait for some other resource (i.e., network programming)
  - avoids blocking the main thread
  - allows the interface to continue to be responsive
  - the threads will get a chance to do their work between the times when the user is interacting with the interface (there is lots of time between such events, even for a highly interactive interface)
- iOS provides a mechanism to support asynchronous execution, called Grand Central Dispatch (GCD), which we will talk about when we get to networking

Performance Tips

- There are a series of performance tips listed in the App Programming Guide for iOS
  - Reduce your app's power consumption
    - biggest users of power: CPU, Networking (Wi-Fi, Bluetooth, cellular network), Location Services (GPS), Accelerometers, and the Disk
    - make informed decisions regarding your use of these resources
  - Use memory efficiently
    - observe low-memory warnings
    - reduce your app's memory footprint with good programming (e.g., no memory leaks) and good software design (e.g., using appropriate data structures, loading resources lazily)
  - Move work off the main thread

Homework

- Keep up with readings (see the syllabus)
- Next topic: The User Experience & Design
- Assignment #1
  - due Oct 5
- Short Paper # 1 (CS 855)
  - due Oct 12
- Project Design
  - due Oct 17