MVC & Onwards

CS 442: Mobile App Development
Michael Saelee
Agenda

- Recap: view-controller communication
- Delegation as a general pattern
- “Observer” pattern
- Controller responsibilities & MVC
- Multiple scenes & controllers
“Simple” view-controller communication:

- target-action (view push)
  - designate method as `@IBAction`
- outlet (controller push/pull)
  - designate attribute as `@IBOutlet`
For complex views, view defines API for controller to adopt: *delegation* protocol

- enables view push & pull

- enables/simplifies reuse of complex UI widgets
- but delegation isn’t just used for view
  controller communication

- same mechanism used for relaying *app lifecycle notifications* to our custom code

- App → UIApplicationDelegate

- Recall high-level app event callbacks
from Apple’s *iOS App Programming Guide*
User switches to a different app

Foreground
- Deactivate this app

Your code
- applicationWillResignActive:

Background
- Enter background

No
- Allowed to run?
  - Yes
    - Monitor events
    - Sleep when not handling events

Yes
- Monitor events

Suspended
- App sleeps

Memory pressure
- Terminate app

from Apple’s iOS App Programming Guide
AppDelegate also receives events that are not handled by views or view controllers

- low-level touch and motion events
- functions as “catch-all” handler
iOS responder chain
- two basic ways for an object to be the designated recipient of an event
  1. *hit-testing* for touches
  2. assumes *first responder* status
- note: designated recipient doesn’t have to be a view (any subclass of UIResponder)
Classes in the view system

(from “View Controller Programming Guide for iOS”)

NSObject

UIScreen

UIResponder

UIView

UIWindow

UIControl
sometimes it’s handy to broadcast/receive notifications in non-default ways

• notifications outside our “scope”
• notifications that aren’t sent by default
• app-specific notifications
Observer Pattern

- any object can register to receive notifications of specific events
  - One mechanism: notification centers
  - Enables *action at a distance* (not necessarily a good thing!)
Delegation embodies the separation of
(1) the thing where events happen and (2) the thing that processes events

(1) is the **view**

(2) is the **view controller**
But the list of responsibilities is lopsided.

(1) just has to map *inputs* to high-level *actions*; e.g., a touch in a table area to cell selection

(2) has to (a) map an action to a *semantic intent*, then (b) actually *carry out* that intent
(a) *is* the controller’s job, but
(b) often requires *domain-specific knowledge*
to allow for reuse (and modular design) of (b) — would like to extricate this from the controller’s implementation
CHAPTER

User action -> Controller

Update -> Controller

Controller -> Model

Notify -> Model

Update -> Model

Model -> View

Update -> View

View -> Controller

User action
MVC is *central* to the architecture of virtually all native iOS apps, and is found at all levels of the development stack.
Turns out the one-controller-per-scene tale is not really accurate …

- often need a controller to manage transitions between separate, per-screen controllers

- overarching “container” controllers
Built-in controllers for this!

- **Navigation Controller**: A controller that manages navigation through a hierarchy of views.
- **Tab Bar Controller**: A controller that manages a set of view controllers that represent tab bar items.
- **Page View Controller**: Presents a sequence of view controllers as pages.
View controller classes in UIKit

(from “View Controller Programming Guide for iOS”)

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
Navigation container controller

(from “View Controller Programming Guide for iOS”)

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY
Tab bar container controller
(from “View Controller Programming Guide for iOS”)

Clock view controller
Stopwatch view controller
Timer view controller
Tab bar controller
Other tabs
Alarm view controller
Good news: intended for use without subclassing

• typically drop into Storyboard & customize in code if necessary

• segues are used as to connect controllers — e.g., to pass data

• (true even without container controllers)
A segue tracks:

1. Source controller
2. Destination controller
3. Segue “name” (unique in storyboard)
When a segue is triggered:

- the destination controller is created
- `prepareForSegue` is called in source
- an [animated] transition from source to destination controller takes place
Segues seem magical, but really just add another layer of abstraction on top of what we already have
Controller transition *without* segue:

1. Event (e.g., button tap) triggers action
2. Action method creates dest controller
3. [Pass data to dest in action method]
4. Use `presentViewController` to activate dest controller
Controller transition *with* segue:

1. Event triggers creation of segue object
2. Segue populates its src/dest controllers
3. `prepareForSegue` called in src controller
4. [Pass data to dest in `prepareForSegue`]
Controller transition with segue:

5. Segue object’s `perform` is invoked

6. `[perform method animates transition]`

7. `perform` calls `presentViewController` to complete transition
If we use built-in segues, most of this is automatic (i.e., invisible)
Can do away (*sometimes*) with cumbersome & wordy delegate mechanism
When a segue is triggered:
- the destination controller is created
- `prepareForSegue` is called in source
  - good place to send data to destination
(sometimes we want to return to an existing controller/scene)
“Unwind” segues let us return to an unwind action in a previous controller.
/* Sample unwind segue action */
- (IBAction)unwindSegue:(UIStoryboardSegue *)segue
{
    ViewController *srcController = segue.sourceViewController;

    /* retrieve data from srcController */

    /* note: manually dismissing srcController isn’t done here! */
}

Conveniently, segue stores and retains source controller — can retrieve data in destination before source goes away
§Container controllers
so far, we can create a modal VC relationship

... but that’s not always enough to build/describe a complex app
other typical VC relationships:

- hierarchical / drill-down — “navigation”
- sibling / parallel — “tabbed”
in both cases, there is some overarching context for the related VCs
$\text{UIViewController} \ & \ \text{UITabBarController} = \text{container view controllers}$
container view controllers *track and manage transitions* between other view controllers

- implement custom transitions
- simplify controller management & communication
e.g., `UINavigationController`
views managed by different VCs on screen simultaneously (previously a no-no)
NavController manages a stack of related view controllers.
NavController also exposes an API for managing the navigation & tool bars
UIViewController

Getting the Navigation controller

navigationController property

Configuring a Navigation Interface

navigationItem property

hidesBottomBarWhenPushed property

- setToolbarItems:animated property

UINavigationItem

Getting and Setting Properties

title property

backBarButtonItem property

hidesBackButton property

Customizing Views

leftBarButtonItem property

rightBarButtonItem property

UINavigationController

Creating Navigation Controllers

- initWithRootViewController:

Accessing Items on the Navigation Stack

topViewController property

visibleViewController property

viewControllers property

- setViewControllers:animated:

Pushing and Popping Stack Items

- pushViewController:animated:

- popViewControllerAnimated:

- popToRootViewControllerAnimated:

- popToViewController:animated: