Communicating with Controllers

CS 442: Mobile App Development
Michael Saelee <lee@iit.edu>
Agenda

- Recap: target-action & delegation
- Controller-Controller communication
- Container controllers
§ Recap
Controllers act as intermediaries between the UI (views) and our *application logic*
Simple views either push events to controllers via \textit{target-action}, or act as \textit{outlets} for controllers (and can be used for I/O)
A complex view, however,

1. May have many types of actions it needs to notify a controller about

2. May need to *pull* data from a controller according to some internal logic
Controller should expose a well-defined API for interacting with a complex view
Well defined API $\rightarrow$ formal protocol
§Delegation
Clean *separation of responsibilities*:

- *delegator* is the source of event/information

- *delegate (aka delegatee)* is responsible for acting on the event/information
Important: delegator and delegate are typically independent objects/actors

- weak connection!
Also: may have *multiple delegators* sending events to a *single delegate*

- use source object in API to differentiate between delegators
E.g., TableView ↔ Datasource / Delegate
@protocol UITableViewDelegate
- (CGFloat)tableView:(UITableView *)tableView
    heightForRowAtIndexPath:(NSIndexPath *)indexPath;
- (void)tableView:(UITableView *)tableView
    accessoryButtonTappedForRowWithIndexPath:(NSIndexPath *)indexPath;
- (void)tableView:(UITableView *)tableView
    didSelectRowAtIndexPath:(NSIndexPath *)indexPath;
@end

@protocol UITableViewDataSource
- (NSInteger)tableView:(UITableView *)tableView
    numberOfRowsInSection:(NSInteger)section;
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath;
- (void)tableView:(UITableView *)tableView
    moveRowAtIndexPath:(NSIndexPath *)sourceIndexPath
    toIndexPath:(NSIndexPath *)destinationIndexPath;
@end
Separation of responsibilities!

- Table View manages views
  - including caching of cells
- Controller informs model of events
  - insertion/deletion requests, etc.
- propagates changes to Table View
so far, single controller and 1+ views
= a single scene
but most interesting apps require 1+ scenes
§Controller-Controller communication
@implementation AppDelegate
- (BOOL)application:(UIApplication *)application
didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
  ...
  self.viewController1 = [[[FirstViewController alloc]
    initWithNibName:@"FirstViewController" bundle:nil]);
  self.viewController2 = [[[SecondController alloc]
    initWithNibName:@"SecondController" bundle:nil];
  self.window.rootViewController = viewController1;
  ...
}
- (void)toggleViewControllers {
  if (self.window.rootViewController == viewController1)
    self.window.rootViewController = viewController2;
  else
    self.window.rootViewController = viewController1;
}
...  
@end

@implementation FirstViewController
- (IBAction)switchButtonPressed:(id)sender {
  [((AppDelegate *)[UIApplication sharedApplication].delegate) toggleViewControllers];
}
@end
NOOO!!! iOS spaghetti code!
need a better mechanic for managing relationships between view controllers!

ideally, *decouple* VCs as much as possible
specific goals:
- only *presenter* ought know about *presentee*
- *presentee* shouldn’t know too much about its presenter or other ancestors
maximize *reusability* & *flexibility* of application components
simplest relationship: parent-child
aka “modular” relationship;
e.g., desktop dialog window
@implementation MainViewController
-(IBAction)addButtonPressed:(id)sender {
    EditingViewController *vc = [[EditingViewController alloc] ...];
    vc.delegate = self;
    [self presentViewController:vc animated:YES completion:nil];
}
@end
when/how to dismiss?

@implementation EditingViewController
- (void)textFieldDidEndEditing:(UITextField *)textField {
    [self.presentingViewController dismissViewControllerAnimated:YES completion:nil];
}
@end
but not (always) logically sound …
parent should dismiss child!

@implementation EditingViewController
- (void)textFieldDidEndEditing:(UITextField *)textField {
    MainViewController *parent = (MainViewController *)self.presentingViewController;
    [parent childControllerFinishedWithResult:textField.text];
}
@end
much better solution: develop an API for delegation!
@protocol EditingViewControllerDelegate <NSObject>
- (void)editingViewController:(EditingViewController *)viewController
textEntered:(NSString *)str;
@end

@interface EditingViewController : UIViewController
@property (weak) id<EditingViewControllerDelegate> delegate;
@end

@implementation EditingViewController
- (void)textFieldDidEndEditing:(UITextField *)textField {
    [delegate editingViewController:self textEntered:textField.text];
}
@end

@interface MainViewController : UIViewController <EditingViewControllerDelegate>
@end

@implementation MainViewController
- (IBAction)addButtonPressed:(id)sender {
    EditingViewController *vc = [[EditingViewController alloc] ...];
    vc.delegate = self;
    [self presentViewController:vc animated:YES completion:nil];
}
- (void)editingViewController:(EditingViewController *)viewController
textEntered:(NSString *)str {
    [self dismissViewControllerAnimated:YES completion:nil];
}
@end
lot of typing, but much cleaner!
Similar types of delegation are used for built-in iOS controllers; e.g.,

- mail composer
- address book
- photo picker / camera
- Twitter
Demo:

DoesItAll
A storyboard driven alternative: *segues*
Segues represent transitions between scenes (controllers)

- visual elements in storyboards

- can be accessed as objects in code
  \textit{(UIStoryboardSegue)}
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

*(slight problem)*
(sometimes we want to return to an existing controller/scene)
“Unwind” segues let us return to an unwind action in a *previous* controller
Conveniently, segue stores *and retains* source controller — can retrieve data in destination before source goes away.
Demo:

Segue
§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 VC's
UINavigationController & UITabBarController = 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
Previous view controller

- ViewController
  - navigationItem
  - UINavigationController
    - backBarButtonItem

Current view controller

- ViewController
  -.navigationItem
  - UINavigationController
    - title
      - UIBarButtonItem
        - contentView
        - UIBarButtonItem
          - UIBarButtonItem
            - UIBarButtonItem
              - UIBarButtonItem
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:

UIViewController
Getting the Navigation controller
  navigationController property
Configuring a Navigation Interface
  navigationItem property
  hidesBottomBarWhenPushed property
  - setToolbarItems:animated
  toolbarItems property

UINavigationItem
Getting and Setting Properties
  title property
  backBarButtonItem property
  hidesBackButton property
Customizing Views
  leftBarButtonItem property
  rightBarButtonItem property
Demo (ad hoc):

*NavDemo*