COMP327
Mobile Computing

Session: 2012-2013

Lecture Set 3b - View Transitions, Storyboards and Protocols

In these Slides...

- **We will cover...**
  - Navigating through Data
  - Protocols and Modal Views
  - Tab Bars
  - Storyboards and Segues

Transitions

A key component to designing any multi-view application is how to transition from one view to another. These slides will look at both traditional approaches, as well as the use of Storyboards and Segues.

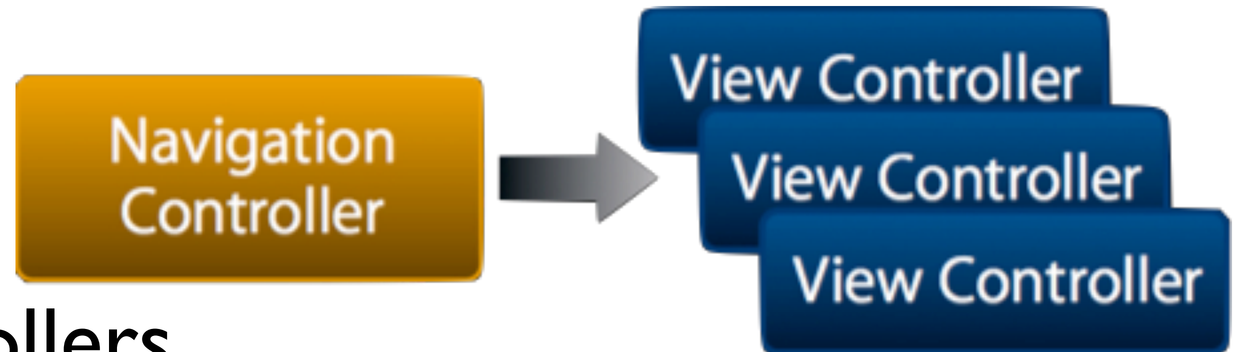
UNIVERSITY OF LIVERPOOL

Navigation Controllers

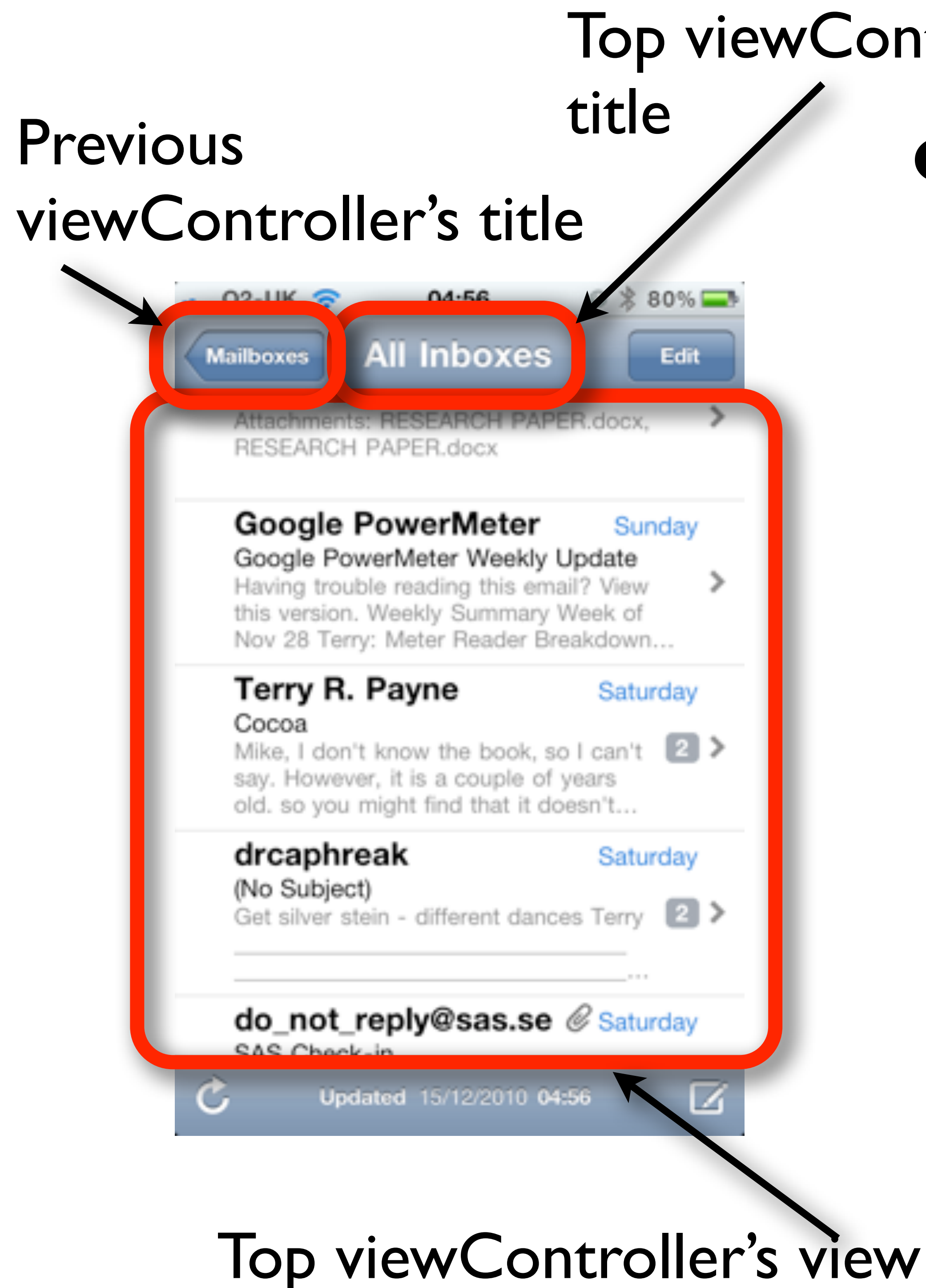
Lecture Set 3b - View Transitions, Storyboards and Protocols

Navigation Controllers

- UINavigationController
  - Provides a stack of controllers
    - As the user explores the data in new views, corresponding controllers are pushed onto the stack
    - When the user leaves these views, they are popped off the stack
  - A Navigation bar supports this navigation
    - Includes back button, title, edit buttons etc.



How it fits together



- Two main methods are used to change the content of a Navigation stack:
  - Push to add a view controller

```
- (void)pushViewController:(UIViewController *)viewController
                   animated:(BOOL)animated;
```

  - Pop to remove a view controller

```
- (void)popViewController Animated:(BOOL)animated;
```

Using a Nav Controller

- Create the view controller that should initially appear
  - As this will be pushed onto the Nav Controller stack, it is often not retained by an iVar, but simply retained on the stack.
- Then create the UINavigationController (often stored in an iVar)
  - **initWithRootController:**
    - Can either initialise with the first (i.e. root) controller
  - **pushViewController:animated:**
    - Or the root controller can be pushed onto the stack
  - By convention, no animation should be used with the root view controller

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]] autorelease];
    // Override point for customization after application launch.

    // ==========================================================================
    MyTableViewController *myRootViewController = [[MyTableViewController alloc] initWithStyle:UITableViewStylePlain];
    myNavigationController = [[UINavigationController alloc] initWithRootViewController:myRootViewController];

    [[self window] addSubview:[myNavigationController view]];
    // ==========================================================================

    self.window.backgroundColor = [UIColor whiteColor];
    [self.window makeKeyAndVisible];
    return YES;
}
```

Using a Nav Controller

- Pushing a View Controller in response to User Actions
  - Push from within a view controller on the stack

```
- (void)someAction:(id)sender {

    // Potentially create another view controller
    UIViewController *viewController = ...;

    [self.navigationController pushViewController:viewController animated:YES];
}
```

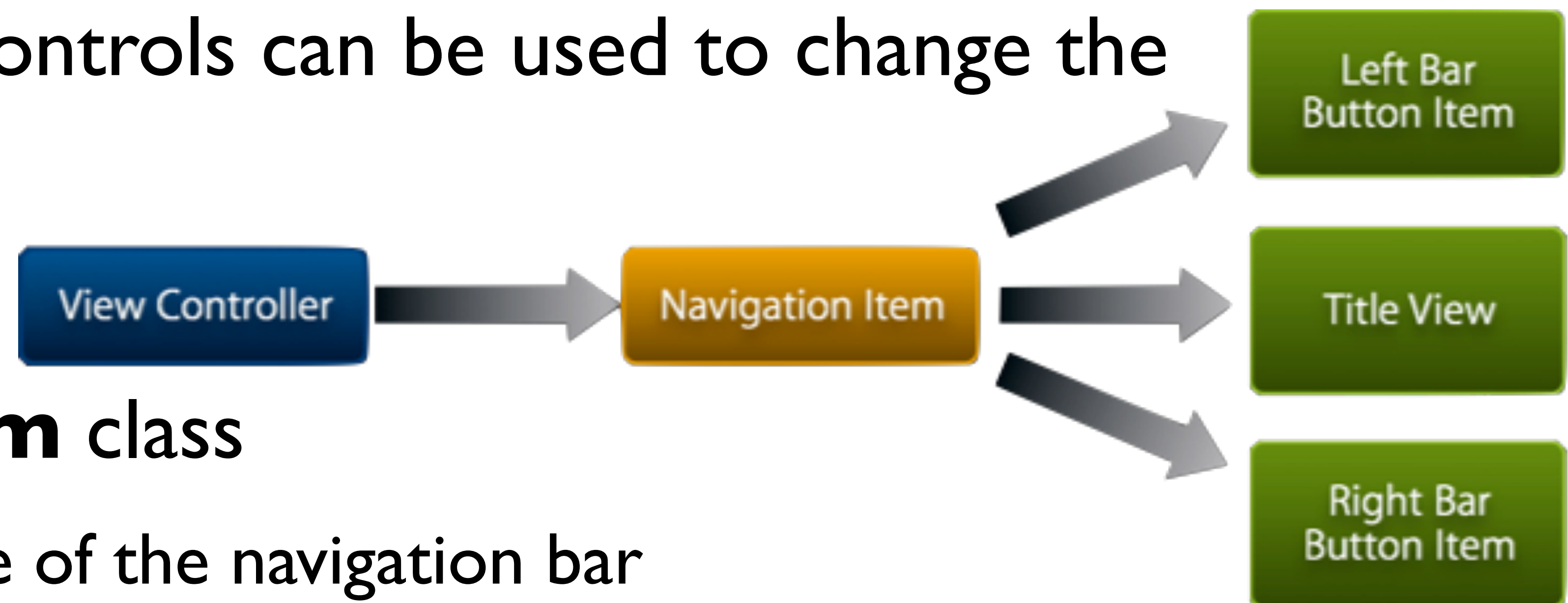
  - Stub code for UITableViewController typically includes code to do this:

```
- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath
{
    // Navigation logic may go here. Create and push another view controller.
    /*
     <#DetailViewController#> *detailViewController = [[<#DetailViewController#> alloc]
                                                         initWithNibName:@"<#Nib name#>" bundle:nil];
     // ...
     // Pass the selected object to the new view controller.
     [self.navigationController pushViewController:detailViewController animated:YES];
     */
}
```

- Almost never call pop directly!
  - Automatically invoked by the back button

Customising Navigation

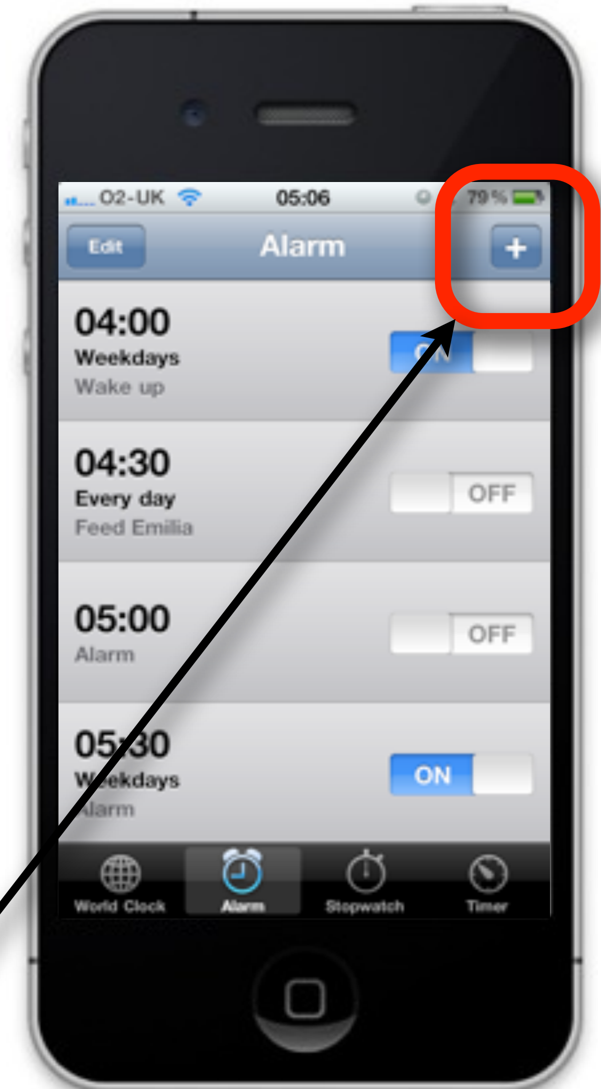
- Buttons or custom controls can be used to change the view



- **UINavigationItem** class
  - Describes appearance of the navigation bar
    - Title string or custom title view
    - Left & right bar buttons
    - More properties defined in UINavigationBar.h
  - Every view controller has a navigation item for customising
  - Displayed when view controller is on top of the stack
    - Can change in **init** or **viewDidLoad** method

Common Customisations

- **Displaying a Title**
  - UIViewController already has a title property
    - `@property(nonatomic,copy) NSString *title;`
  - Navigation item inherits automatically
  - Previous view controller's title is displayed in back button



- **Left and Right Buttons** - UIBarButtonItem
  - Special object, defines appearance & behaviour for items in navigation bars and toolbars
  - Display a string, image or predefined system item
  - Target + action (like a regular button)

```
- (void)viewDidLoad {

    UIBarButtonItem *addButton = [[UIBarButtonItem alloc] initWithBarButtonSystemItem:UIBarButtonSystemItemAdd
                        style:UIBarButtonItemStyleBordered target:self
                        action:@selector(add:)];
    self.navigationItem.rightBarButtonItem = addButton;
}
```

UNIVERSITY OF LIVERPOOL

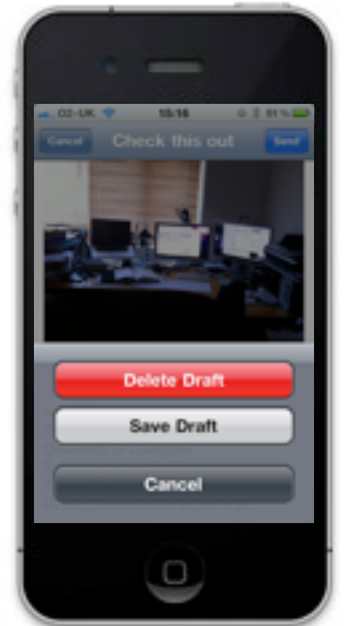
Protocols and Modal Views

Lecture Set 3b - View Transitions, Storyboards and Protocols

Using Modal Views

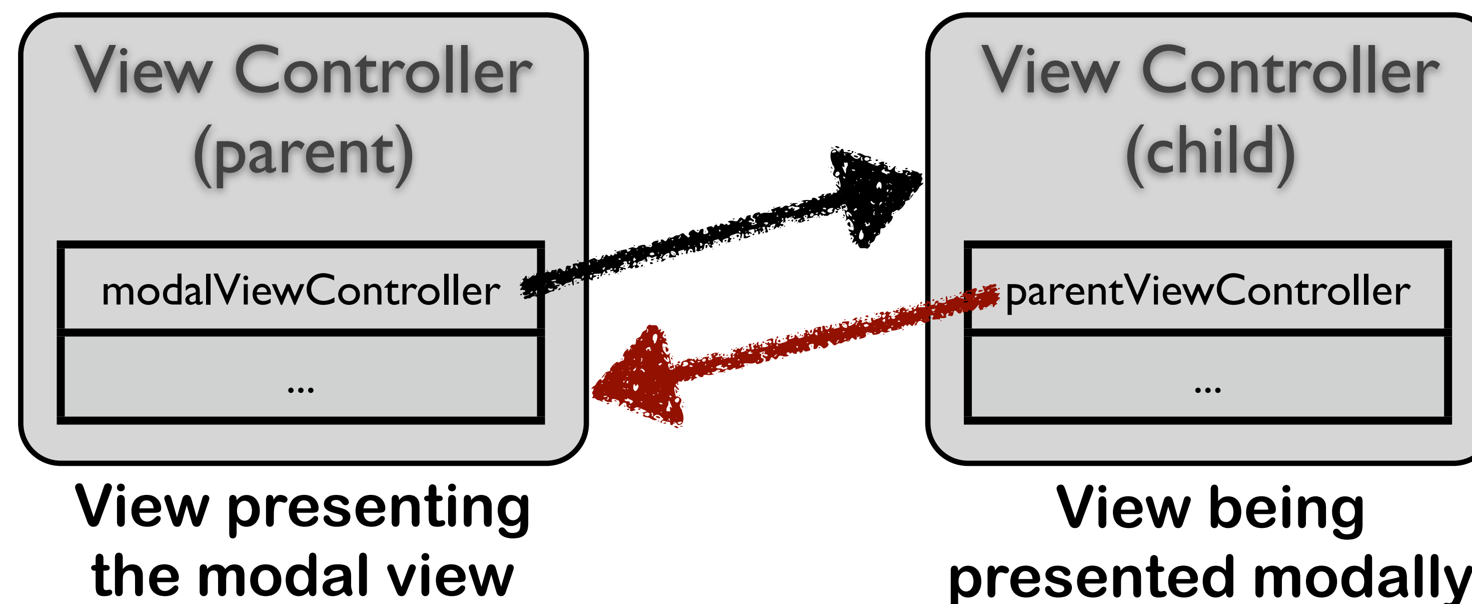
- Modal views temporarily interrupt the current workflow
  - Often used to gather new data or present information
    - Displaying preferences, such as displaying information via a flip-side information view
    - Asking the user for specific information - such as creating a new contact in AddressBook
    - Presenting alerts (e.g. with UIAlertView) or action sheets (UIActionSheet)

- Generally, any view can be presented modally
  - View remains until the view is dismissed, at which point the application returns to its previous state.



Modal Views and their Parents

- When a modal view is presented, a parent-child relationship is created between:
  - The view controller **doing** the presenting (i.e. the child)
  - The view controller **managing** the modal view (i.e. the parent)



- In iPhone apps, content always covers the visible part of the window
- In iPad apps, modal views can be presented as full-screen, as a page, or as a form.

Presenting a View Controller Modally

- There are several steps to presenting a view controller modally

1. Create the view controller you want to present
2. Set the **modalTransitionStyle** property of the view controller
3. Assign a delegate object for the modal view controller
4. Call **`presentViewController:animated:completion:`** method of the view controller parent, passing in the view controller child

Presenting a View Controller Modally

```
- (void)showFlipsideView:(id)sender {

    FlipsideModuleBrowserViewController *controller =
              [[FlipsideModuleBrowserViewController alloc]
           initWithNibName:@"FlipsideModuleBrowserViewController" bundle:nil];

    controller.delegate = self;

    controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;

    [self presentViewController:controller animated:YES completion:nil];

}
```

1: Create view Controller

3: Assign the Delegate Object

2: Set the transition style

4: Present the modal view controller

Transition Semantics

- Views can appear in a variety of ways
  - By convention, the animation style adds **meaning** to the appearance of the modal view
    - UIModalTransitionStyle**CoverVertical**
      - Interrupt the current workflow to gather info from the user
    - UIModalTransitionStyle**FlipHorizontal**
      - Change the work mode temporarily (e.g. information on an app)
    - UIModalTransitionStyle**CrossDesolve**
      - Present an alternate interface, e.g. if rotation changes

Dismissing a Modal View Controller

- Typically, the best approach is to let the parent dismiss it's own child view controller
  - Normally handled through delegation
    - Thus, the object that presents the modal view controller becomes its delegate
  - Therefore the child view controller must define a **protocol** for its delegate to implement
    - Defines actions that the delegate (i.e. the parent in most cases) should do in response to certain actions (e.g. pressing the done button)
  - Advantages of this approach
    - Allows the parent to validate data before dismissing the child
    - Promotes reuse, as the child view controller can be reused elsewhere

Protocols

- A protocol is a list of methods that is shared among classes
  - There is no existing implementation of these methods; rather it is up to the developer to implement them for that class
- A class can choose to *conform to*, or *adopt* a protocol
  - Conforming to more than one protocol is possible
    - Informally similar to multiple inheritance

Protocols

- Two types of protocol are possible
  - Informal Protocols
    - Specified only in the documentation of the class, but is not stated within source code. A class can then determine if the method exists through reflection, and invoke it if it exists.
  - Formal Protocols
    - Similar to an interface in Java
    - Defined, using the `@protocol` directive ...

Formal Protocols

```
@protocol Locking
- (void)lock;
- (void)unlock;
@end
```

The protocol is defined somewhere, using the **`@protocol`** directive, and may group together a set of related method that should be implemented if the class adopts to the protocol.

A class is then defined to adopt the protocol using the angled bracket notation.

Several protocols could be adopted, by separating the protocols with commas

```
@interface SomeClass : SomeSuperClass <Locking>
@end

@interface AnotherClass : AnotherSuperClass <Locking, Archiving>
// This class adopts two protocols!!!
@end
```

The methods can then be defined in the implementation part of the class

```
id currentObject;
...
if ([currentObject conformsTo: @protocol (Locking)] == YES) {
    // Call lock
    [currentObject lock];
```

An object can later check to see if it conforms to a protocol, using the **`conformsTo`** method

Example: Defining a protocol for a simple modal view controller :-

The child view controller (.h)

The view controller that is to be presented modally needs to declare a protocol that the parent adopts.

An ivar containing the id of the delegate (that adopts the protocol) is stored by the view controller

```
//  FlipsideModuleBrowserViewController.h

@protocol FlipsideModuleBrowserViewControllerDelegate;

@interface FlipsideModuleBrowserViewController : UIViewController {

     id <FlipsideModuleBrowserViewControllerDelegate> delegate;

}
@property (nonatomic, assign) id <FlipsideModuleBrowserViewControllerDelegate> delegate;

- (IBAction)done:(id)sender;

@end

@protocol FlipsideModuleBrowserViewControllerDelegate
(void)flipsideViewControllerDidFinish:(FlipsideModuleBrowserViewController *)controller;

@end
```

The protocol defines one (or more) methods that need to be implemented by the delegate. In this case, the delegate must implement the **`flipsideViewControllerDidFinish`** method

Example: Defining a protocol for a simple modal view controller :-
The child view controller (.m)

```
//  FlipsideModuleBrowserViewController.m

@implementation FlipsideModuleBrowserViewController

@synthesize delegate;

- (IBAction)done:(id)sender {

    // This method is called by selecting the done button
    // hence the IBAction return value

    [[self delegate] flipsideViewControllerDidFinish:self];

}
```

The delegate (normally the parent) will be stored in the iVar delegate (defined in the header file)

The action that triggers closing the modal view controller (in this case, `done:`) calls the protocol method **`flipsideViewControllerDidFinish:`** on the delegate.

Example: Defining a protocol for a simple modal view controller :-

The parent view controller (.h)

Need to assert that the parent adopts the protocol, defined in the child's header file. Remember to include the child's deader file in the parent's implementation file.

```
//  ParentViewController.h

// Repeat the forward declaration of the protocol, defined in the child's header file
@protocol FlipsideModuleBrowserViewControllerDelegate;

    ...


@interface ParentViewController : UIViewController <FlipsideModuleBrowserViewControllerDelegate> {
    ...
}

- (void)flipsideViewControllerDidFinish:(FlipsideModuleBrowserViewController *)controller;
```

Include the method prototype that implements the method defined in the protocol. Note that we pass back the child object as an argument; this simplifies managing it.

Example: Defining a protocol for a simple modal view controller :-
The parent view controller (.m)

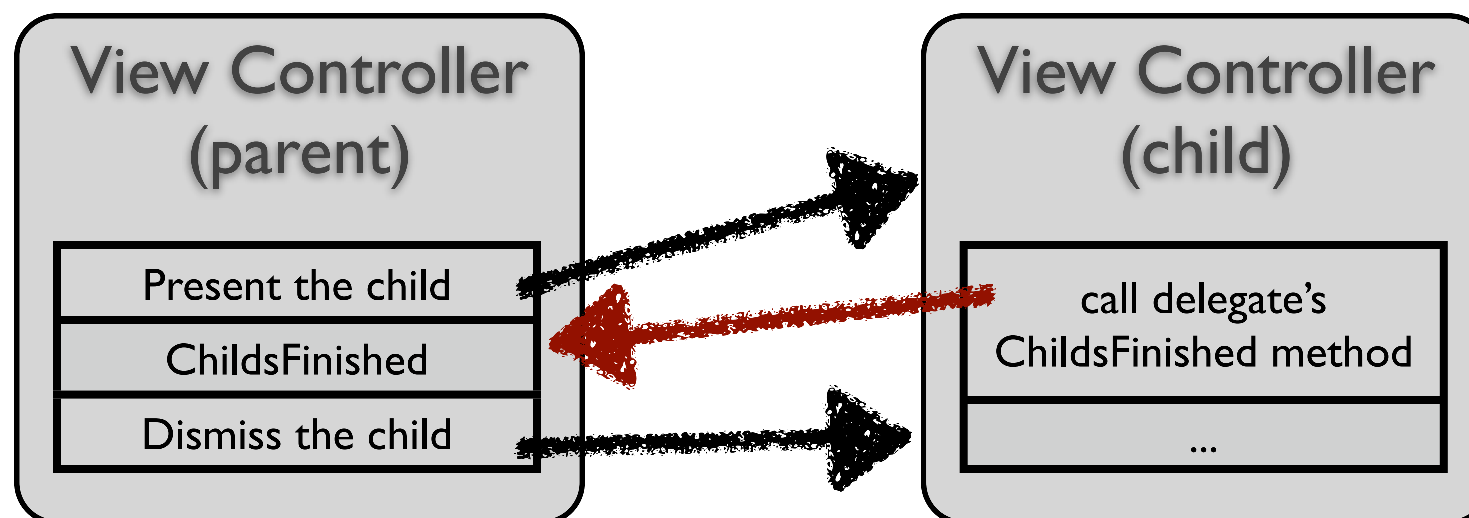
The protocol states that the delegate should implement the `flipsideViewControllerDidFinish` method. In the implementation, all it does in this case is to call (on itself)the message to dismiss the modal view controller it owns.

The parent creates the child object, configures it, and then presents it. Note that once the controller has been presented, it can be released, as presenting a controller will cause it to be retained.

```
//  ParentViewController.m

@implementation ParentViewController

- (void)flipsideViewControllerDidFinish:(FlipsideModuleBrowserViewController *)controller {

    [self dismissViewControllerAnimated:YES completion:nil];
}


- (IBAction)showFlipsideView:(id)sender {

    FlipsideModuleBrowserViewController *controller = [[FlipsideModuleBrowserViewController alloc]
                          initWithNibName:@"FlipsideModuleBrowserViewController" bundle:nil];
    controller.delegate = self;

    controller.modalTransitionStyle = UIModalTransitionStyleFlipHorizontal;
    [self presentViewController:controller animated:YES completion:nil];
}
```

Modal Views and their Parents

- To summarise:
  - The parent presents the child
  - Best practice dictates that the same object that presents should also dismiss!
    - The parent becomes the delegate, and can therefore dismisses the child later
  - The child defines a protocol which states the message it will send to the parent to tell the parent when it is done
    - The parent adopts the protocol, and implements the associated method
  - On receipt of the message, the parent dismisses its child

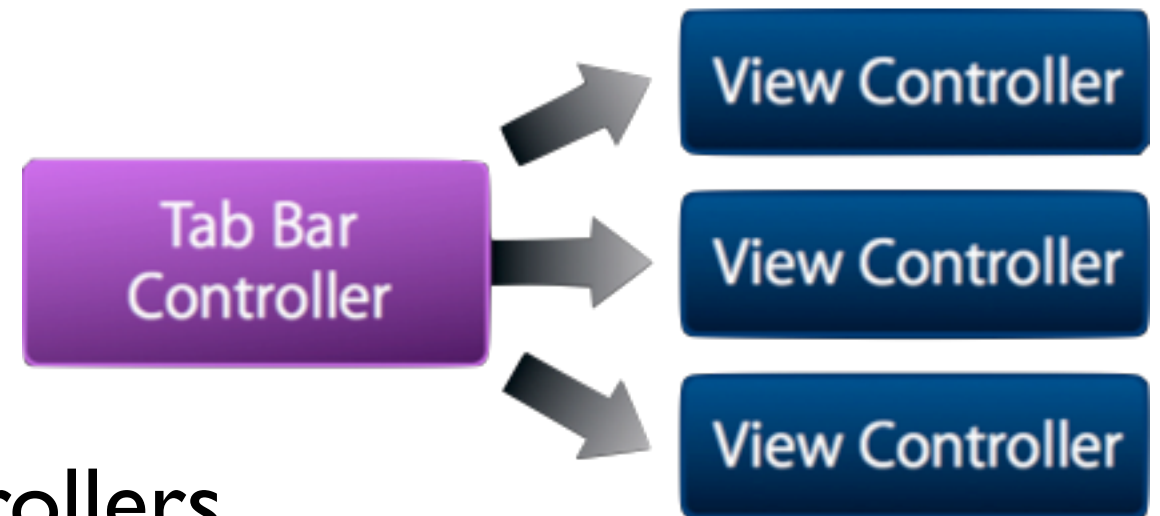


UNIVERSITY OF LIVERPOOL

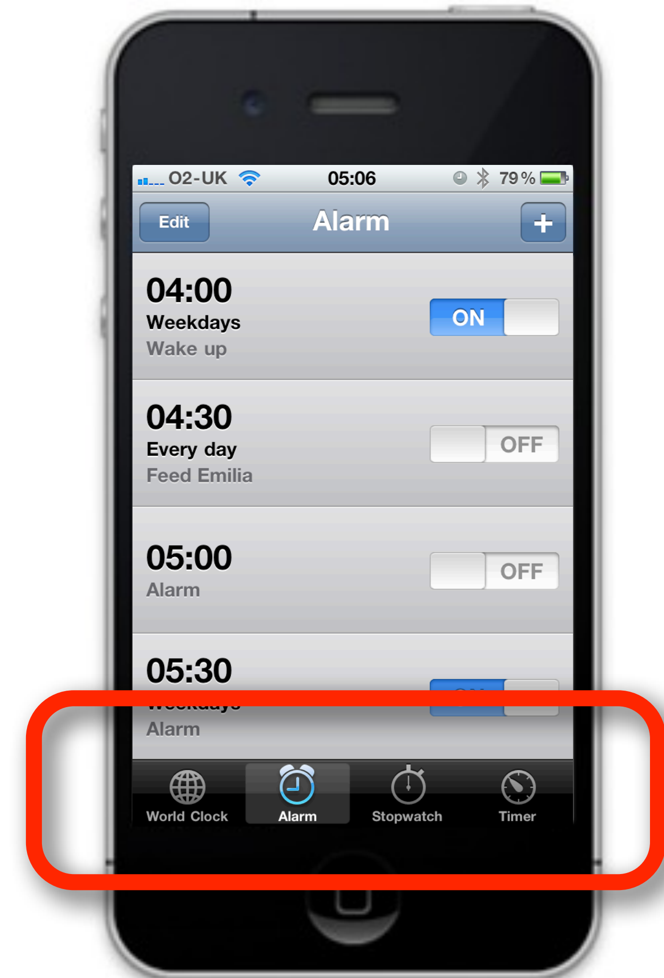
Organising Data on the display's window

Lecture Set 3b - View Transitions, Storyboards and Protocols

Tab Bar Controllers



- UITabBarController
  - Provides an array of controllers
    - User can select which view to inspect or interact with
    - The selected view appears in the main screen area
    - The corresponding image appears in blue within a tab bar that appears at the bottom of the display



Using a Tab Bar Controller

- The different controllers (to appear) are held in an array
- View controllers can define their appearance in the tab bar
  - Each view controller comes with a tab bar item for customising
  - UITabBarItem
    - Image and Title...
    - ... or system item

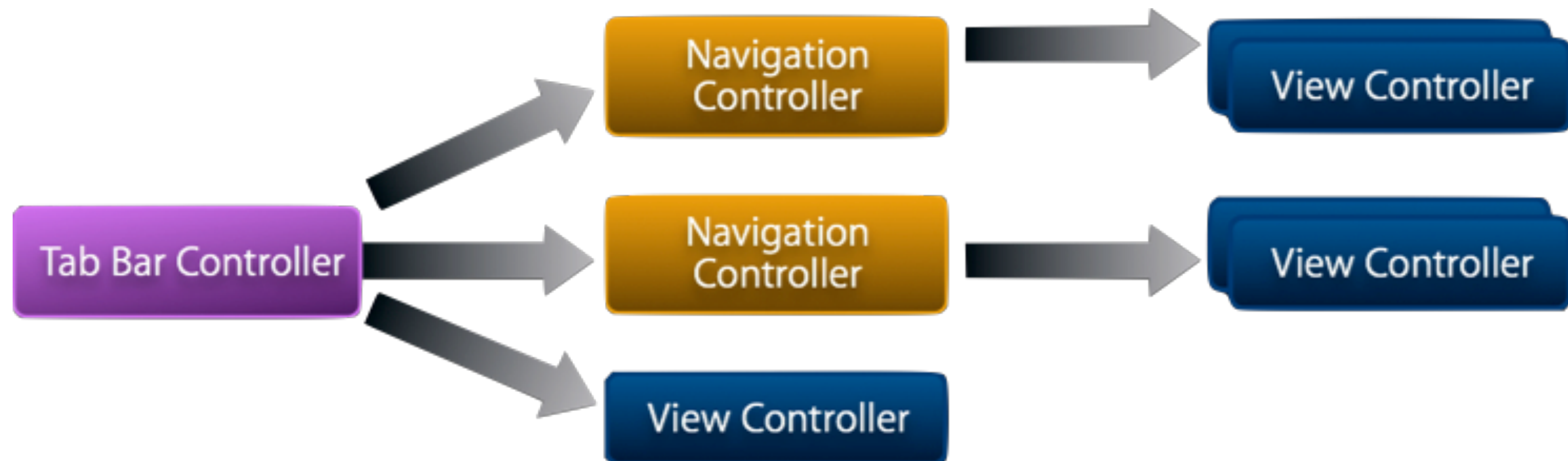
```
- (void)viewDidLoad {
    self.tabBarItem = [[UITabBarItem alloc]
                   initWithTitle:@“Playlists”
                   image:[UIImage imageNamed:@“music.png”]
                   tag:0]

    // To display a system item use
    // the following init instead
    // initWithTabBarSystemItem:UITabBarSystemItemBookmarks
}
```

More View Controllers

- What happens when a tab bar has too many view controllers to display at once?
  - "More" tab bar item is displayed automatically
    - User can navigate to remaining view controllers
    - User can also customise what appears on the tab bar

Combining Tab Bars and Navigators

- Often it is desirable to include one of the views as a navigation controller
    - First create the tab bar controller
    - Then create the navigation controllers
    - Finally, add them to the navigation controller



- A common mistake is to make the tab bar controller the first view of the navigation controller
    - This will simply disappear when the next view is pushed.

UNIVERSITY OF LIVERPOOL

Storyboards (coming soon)

Lecture Set 3b - View Transitions, Storyboards and Protocols

UNIVERSITY OF LIVERPOOL

Questions?

Lecture Set 3b - View Transitions, Storyboards and Protocols