1

Navigation

WHAT'S IN THIS CHAPTER?

- How navigation works on the iPad's split view
- Using a toolbar to rotate an image
- Implementing a simple bank account transaction tracker using a tab bar

Navigation is the process of searching through a hierarchy to arrive at the information you desire. With the iPhone and iPad, navigating through your data is achieved through the following components; each has a specific viewing philosophy:

- **Navigation bar —** Arranges the data in a hierarchy that you can navigate by drilling down, and provides a path back to the top
- **Toolbar —** Provides a number of options that act on the current view context
- **Tab bar —** Provides different views of the same set of data

This chapter presents the steps to create an application for each of these navigation components to demonstrate a simple use case for each style. The device on which your application runs determines the style of navigation you implement. Remember that there is more drilling down on an iPhone than on an iPad because of the limited viewing area. This is very important if you are planning to develop an application that will be available on both devices. The iPad should not be just a duplicate application, in terms of visual presentation.

When considering navigation design for your applications, consult Apple's User Interface Guidelines for the iPad as well as the iPhone. See Appendix D for documentation sources for these and other developer guides offered by Apple.

NAVIGATION STACK

The navigation process is stack based. These views are stacked in a last in, first out (LIFO) manner, which is managed by a View Controller. The initial view is the root View Controller, which has views placed over it. Unlike the views that are pushed upon it, the root View Controller can never be removed. The process of navigation involves responding to user interaction, pushing and popping View Controllers on and off the navigation stack. Each current view is responsible for pushing the next view onto the stack.

The object that serves as the content provider for navigation items is the *navigation bar*. Users interact with buttons on the navigation bar, and through delegate messages sent to the View Controller, the pushing or popping of views is performed.

THE NAVIGATION BAR

The navigation bar basically holds control objects that provide the process of navigating views in an application. The navigation bar provides users with all the controls necessary to push down or pop the views in the application's hierarchy. Processing of the delegate messages is handled by an associated View Controller through the use of delegate methods.

UINavigationBarDelegate Protocol

The View Controller implements the methods of this protocol when it has to either push or pop an item onto or off of the navigation stack.

The methods to implement are as follows:

- To push an item
  1. `navigationBar:shouldPushItem:`
  2. `navigationBar:didPushItem:`
- To pop an item
  1. `navigationBar:shouldPopItem:`
  2. `navigationBar:didPopItem:`

Configuring Navigation Bars

The navigation bar is located at the top of the view and will display the title of the current view. In addition to the title, the navigation bar may contain button controls that provide action within the context of the current view. To achieve this functionality, the following are available:

- `backBarButtonItem` and `leftBarButtonItem` are positioned on the left.
- `titleView` is positioned in the center.
- `rightBarButonItem` is positioned on the right.

The navigation bar itself has a few properties that can be modified:

- `barStyle`
- `translucent`
- `tintColor`

Pushing and Popping Items

To navigate from view to view, either to continue drilling down the hierarchy (pushing), or backing out back up the hierarchy (popping), the process is handled by the view controllers in your application. The process of navigation is simply a navigation controller managing a series of view controllers on the navigation stack.

A view controller is responsible for pushing (drilling down the hierarchy) and popping (backing out up the hierarchy) other view controllers on or off the navigation stack. The process of navigation is simply a navigation controller managing a series of view controllers on the navigation stack, and it works like this:

1. The `UINavigationController` is created.
2. The navigation controller pushes the view controller onto the navigation stack.
3. The view controller then presents the next view.
4. The view controller then dismisses the previous view.

A SIMPLE NAVIGATION BAR

In this application navigation will consist of displaying the numbers from 1 to 20. The grouping is even or odd numbers. Notice that with the split view of the iPad, portrait orientation presents the navigation bar as a popover; in landscape orientation, the navigation bar is in the left pane of the split view, as shown in Figure 1-1.

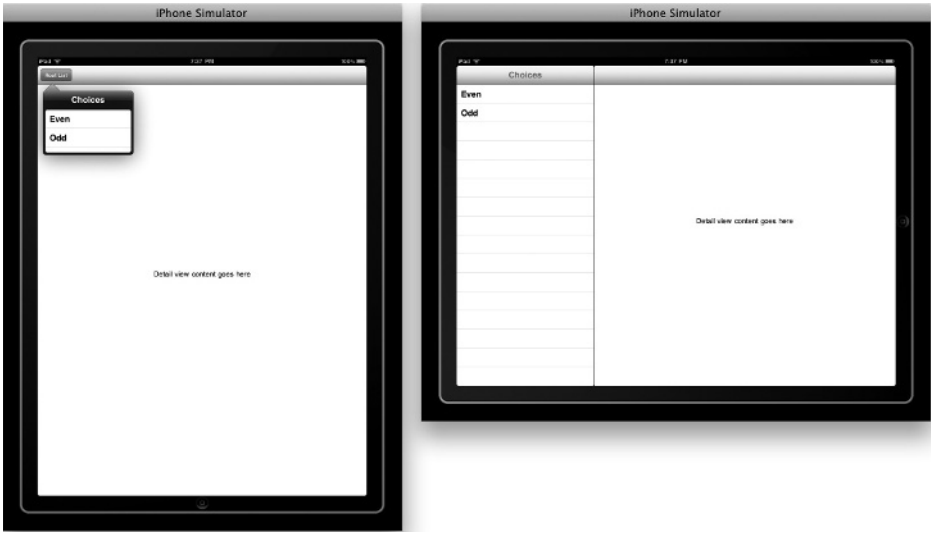

FIGURE 1-1

Tapping Odd will reveal another navigation bar with a list of the odd numbers from 1 to 20, as shown in Figure 1-2.

Tapping a number from this list will display the choice in the detail view, and the popover disappears (see Figure 1-3).

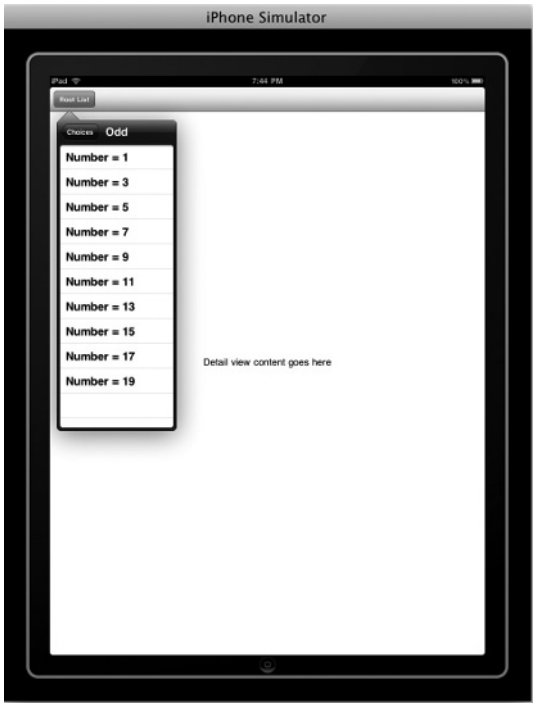

FIGURE 1-2

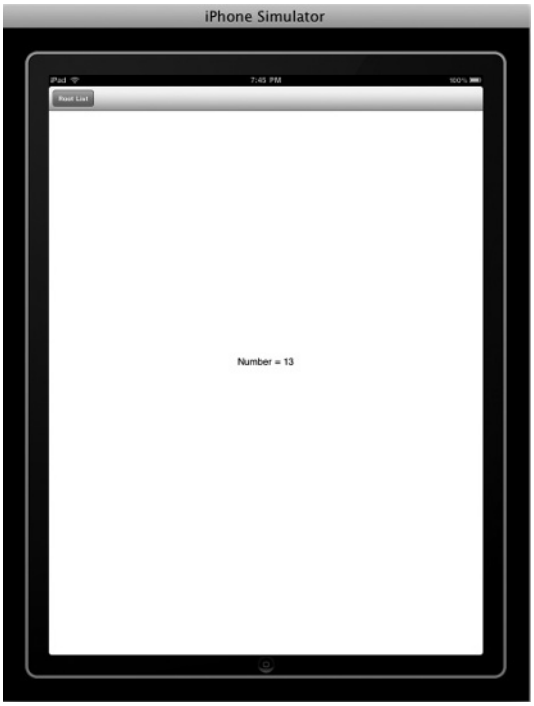

FIGURE 1-3

Development Steps: A Simple Navigation Bar

To create this application that will be the server, execute the following steps:

1. Start Xcode and create a SplitView-based application and name it **NavigationBar-iPad**. If you need to see this step, please see Appendix A for the steps to begin a SplitView-based application.
2. In the Groups & Files section of Xcode, click the Classes group. Choose File ➪ New File and select `UIViewController` subclass then check the `UITableViewController` subclass option and name it `RootDetailViewController`.

For this application, Interface Builder will not be used, as all the views will be programmatically created. Now it is time to enter your logic.

Source Code Listings for the A Simple Navigation Bar Application

For this application the `NavigationBar_iPadAppDelegate.h` and `NavigationBar_iPadAppDelegate.m` files are not modified and are used as generated.

RootViewController.h Modifications to the Template

The additions to the `RootViewController` class will be two `NSArrays` to hold the even and odd numbers. The two arrays will be stored in an `NSDictionary` with even and odd as the keys (see Listing 1-1).

LISTING 1-1: The complete RootViewController.h file (Chapter1/NavigationBar-iPad/Classes/RootViewController.h)

```
#import <UIKit/UIKit.h>

@class DetailViewController;

@interface RootViewController : UITableViewController {
    DetailViewController *detailViewController;

    NSDictionary *groupsDict;
    NSArray *evenArray;
    NSArray *oddArray;
}

@property (nonatomic, retain)
              IBOutlet DetailViewController *detailViewController;
@property (nonatomic, retain) NSDictionary *groupsDict;
@property (nonatomic, retain) NSArray *evenArray;
@property (nonatomic, retain) NSArray *oddArray;

- (void)initData;

@end
```

RootViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `RootViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-2).

LISTING 1-2: Addition of @synthesize

```
#import "RootViewController.h"
#import "DetailViewController.h"
#import "RootDetailViewController.h"


@implementation RootViewController

@synthesize detailViewController;
@synthesize groupsDict;
@synthesize evenArray;
@synthesize oddArray;
```

To initialize the view, the size of the popover is defined, and the even and odd arrays are defined and initialized (see Listing 1-3).

LISTING 1-3: Initialization of the view

```
#pragma mark -
#pragma mark View lifecycle

- (void)viewDidLoad {
```

```
    [super viewDidLoad];
    [self setClearsSelectionOnViewWillAppear:NO];
    [self setContentSizeForViewInPopover:CGSizeMake(200.0, 100.0)];
    [self setTitle:@"Choices"];
    [self initData];
}

- (void)initData {
    NSMutableArray *even = [NSMutableArray array];
    NSMutableArray *odd = [NSMutableArray array];
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    NSString *msg = nil;

    for (int i=0;i<20; i++) {
        msg = [NSString stringWithFormat:@"Number = %d", i];
        if ( i % 2 == 0 ) {
            [even addObject:msg];
        } else {
            [odd addObject:msg];
        }
    }
    [dict setObject:even forKey:@"Even"];
    [dict setObject:odd forKey:@"Odd"];
    [self setGroupsDict:dict];
}

// Ensure that the view controller supports rotation and that the split view
// can therefore show in both portrait and landscape.
- (BOOL)shouldAutorotateToInterfaceOrientation:
                                (UIInterfaceOrientation) interfaceOrientation {
    return YES;
}
```

To initialize a table view, there are two factors to consider. If you have just a single list of items, you will have one section and a number of rows that represents the items in your list. If you want to separate certain items in the list from others, you then have to consider how many sections into which the list is to be divided. For this application, there is just one list of related data, so the number of sections is one. The number of rows is simply the two arrays in the dictionary `groupDict` (see Listing 1-4).

LISTING 1-4: TableView display definition

```
#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView {
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)aTableView
                          numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    return [groupsDict count];
}
```

The `tableView:cellForRowAtIndexPath` method is where each cell of the table view is assigned a value using the method `[[cell textLabel] setText:key]` and then displayed. The value of each cell is the key value for each item in the `groupsDict`. For this application the keys will be Even for the array that contains the even numbers, and Odd for the array that contains the odd numbers (see Listing 1-5).

LISTING 1-5: TableView cell display

```
#pragma mark -
#pragma mark Table view delegate

- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSArray *keys = [[[self groupsDict] allKeys]
         sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
    NSString *key = [keys objectAtIndex:[indexPath row]];

    static NSString *CellIdentifier = @"CellIdentifier";

    // Dequeue or create a cell of the appropriate type.
    UITableViewCell *cell =
                [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                initWithStyle:UITableViewCellStyleDefault
                              reuseIdentifier:CellIdentifier] autorelease];
        cell.accessoryType = UITableViewCellAccessoryNone;
    }

    // Configure the cell.
    [[cell textLabel] setText:key];

    return cell;
}
```

In this application, if the cell labeled Even is tapped, then `RootDetailViewController` is pushed onto the navigational stack and the list of even numbers is displayed in the resulting view. If the cell labeled Odd is tapped, the odd numbers are displayed. (see Listing 1-6).

LISTING 1-6: TableView cell selected

```
#pragma mark -
#pragma mark Table view delegate

- (void)tableView:(UITableView *)aTableView
                        didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSArray *keys = [[[self groupsDict] allKeys]
       sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];

    NSString *key = [keys objectAtIndex:[indexPath row]];
    NSArray *values = [[self groupsDict] objectForKey:key];
    /*
     When a row is selected, set the detail view controller's
     detail item to the item associated with the selected row.
     */
    RootDetailViewController *rootDetailViewController =
                [[RootDetailViewController alloc]
                                initWithKey:key values:values
                              viewController:[self detailViewController]];
    [[self navigationController]
                 pushViewController:rootDetailViewController animated:YES];
    [rootDetailViewController release];
}
```

The complete `RootViewController.m` file is shown in Listing 1-7.


Available for download on Wrox.com

LISTING 1-7: The complete RootViewController.m file (Chapter1/NavigationBar-iPad/Classes/RootViewController.m)

```
#import "RootViewController.h"
#import "DetailViewController.h"
#import "RootDetailViewController.h"


@implementation RootViewController

@synthesize detailViewController;
@synthesize groupsDict;
@synthesize evenArray;
@synthesize oddArray;


#pragma mark -
#pragma mark View lifecycle

- (void)viewDidLoad {
    [super viewDidLoad];
    [self setClearsSelectionOnViewWillAppear:NO];
    [self setContentSizeForViewInPopover:CGSizeMake(200.0, 100.0)];
    [self setTitle:@"Choices"];
    [self initData];
}

- (void)initData {
    NSMutableArray *even = [NSMutableArray array];
    NSMutableArray *odd = [NSMutableArray array];
    NSMutableDictionary *dict = [NSMutableDictionary dictionary];
    NSString *msg = nil;

    for (int i=0;i<20; i++) {
        msg = [NSString stringWithFormat:@"Number = %d", i];
        if ( i % 2 == 0 ) {
            [even addObject:msg];
        } else {
            [odd addObject:msg];
        }
    }
    [dict setObject:even forKey:@"Even"];
    [dict setObject:odd forKey:@"Odd"];
    [self setGroupsDict:dict];
}

// Ensure that the view controller supports rotation and that
// the split view can therefore
// show in both portrait and landscape.
- (BOOL)shouldAutorotateToInterfaceOrientation:
                                (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}

#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)aTableView {
```

```
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)aTableView
                                numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    return [groupsDict count];
}

- (UITableViewCell *)tableView:(UITableView *)tableView
                         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSArray *keys = [[[self groupsDict] allKeys]
       sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
    NSString *key = [keys objectAtIndex:[indexPath row]];

    static NSString *CellIdentifier = @"CellIdentifier";

    // Dequeue or create a cell of the appropriate type.
    UITableViewCell *cell =
            [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier] autorelease];
        cell.accessoryType = UITableViewCellAccessoryNone;
    }

    // Configure the cell.
    [[cell textLabel] setText:key];

    return cell;
}

#pragma mark -
#pragma mark Table view delegate

- (void)tableView:(UITableView *)aTableView
                       didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    NSArray *keys = [[[self groupsDict] allKeys]
      sortedArrayUsingSelector:@selector(localizedCaseInsensitiveCompare:)];
    NSString *key = [keys objectAtIndex:[indexPath row]];
    NSArray *values = [[self groupsDict] objectForKey:key];
    /*
     When a row is selected, set the detail view controller's detail
     item to the item associated with the selected row.
     */
    RootDetailViewController *rootDetailViewController =
                  [[RootDetailViewController alloc]
                                  initWithKey:key values:values
                               viewController:[self detailViewController]];
    [[self navigationController]
                 pushViewController:rootDetailViewController animated:YES];
    [rootDetailViewController release];
}

#pragma mark -
#pragma mark Memory management

- (void)didReceiveMemoryWarning {
```

```
    [super didReceiveMemoryWarning];
}

- (void)viewDidUnload {
    // Relinquish ownership of anything that can be recreated
    // in viewDidLoad or on demand.
    // For example: self.myOutlet = nil;
    [self setGroupsDict:nil];
    [self setEvenArray:nil];
    [self setOddArray:nil];
    [self setDetailViewController:nil];
}

- (void)dealloc {
    [groupsDict release];
    [evenArray release];
    [oddArray release];
    [detailViewController release];
    [super dealloc];
}

@end
```

RootDetailViewController.h Modifications to the Template

The `RootDetailViewController` class will display the actual even or odd values; and when one of the table view cells is selected, the value of the cell will be displayed on the main detail page. The complete `RootDetailViewController` class is shown in Listing 1-8.

Available for download on Wrox.com

LISTING 1-8: The complete RootDetailViewController.h file (Chapter1/NavigationBar-iPad/Classes/RootDetailViewController.h)

```
#import <UIKit/UIKit.h>

@class DetailViewController;

@interface RootDetailViewController : UITableViewController {
    DetailViewController *detailViewController;

    NSString *key;
    NSArray *values;
}

@property (nonatomic, retain) DetailViewController *detailViewController;
@property (nonatomic, retain) NSString *key;
@property (nonatomic, retain) NSArray *values;

- initWithKey:(NSString *)aKey values:(NSArray *)aValues
                      viewController:(id)viewController;

@end
```

RootDetailViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `RootDetailViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-9).

LISTING 1-9: Addition of @synthesize

```
#import "RootDetailViewController.h"
#import "DetailViewController.h"


@implementation RootDetailViewController

@synthesize key;
@synthesize values;
@synthesize detailViewController;
```

To initialize the view, the size of the popover is defined, and the even and odd arrays are defined and initialized (see Listing 1-10).

LISTING 1-10: Initialization of the view

```
#pragma mark -
#pragma mark Initialization

- initWithKey:(NSString *)aKey values:(NSArray *)aValues
                      viewController:(id)viewController {
    [self setKey:aKey];
    [self setValues:aValues];
    [self setDetailViewController:viewController];

    return self;
}

#pragma mark -
#pragma mark View lifecycle

- (void)viewDidLoad {
    [super viewDidLoad];
    [self setClearsSelectionOnViewWillAppear:NO];
    [self setContentSizeForViewInPopover:CGSizeMake(200.0, 500.0)];
    [self setTitle:[self key]];
}

#pragma mark -
#pragma mark Rotation support

// Ensure that the view controller supports rotation and that the
// split view can therefore show in both portrait and landscape.
- (BOOL)shouldAutorotateToInterfaceOrientation:
                                (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}
```

To initialize a table view, there are two factors to consider. The first is the number of sections that your data would be divided up into — for this application there is only one group because all the data is related.

The second factor is the number of rows. This is the list from which the users will make their selections. For this application the rows of data will represent all even or all odd numbers (see Listing 1-11).

LISTING 1-11: TableView display definition

```
#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
                          numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    return [[self values] count];
}
```

The `tableView:cellForRowAtIndexPath` method is where the table view cells are populated with display details. Because the `numberOfRowsInSection` used the `values` count, the values for the table view cell display will be the actual even or odd number (see Listing 1-12).

LISTING 1-12: TableView cell display

```
#pragma mark -
#pragma mark Table view appearance

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
                       cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSString *value = [[self values] objectAtIndex:[indexPath row]];

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
            [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell...
    [[cell textLabel] setText:value];

    return cell;
}
```

When one of the table view cells is tapped, the row is selected. For this application, the tap causes the number value on the main detail view (see Listing 1-13) to display.

LISTING 1-13: TableView cell selected

```
#pragma mark -
#pragma mark Table view delegate

- (void)tableView:(UITableView *)tableView
                     didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    [[self detailViewController] setText:[[self values]
                                        objectAtIndex:[indexPath row]]];
}
```

The complete `RootDetailViewController.m` file is shown in Listing 1-14.

Available for download on Wrox.com

LISTING 1-14: The complete RootDetailViewController.m file (Chapter1/NavigationBar-iPad/ Classes/ RootDetailViewController.m)

```
#import "RootDetailViewController.h"
#import "DetailViewController.h"


@implementation RootDetailViewController

@synthesize key;
@synthesize values;
@synthesize detailViewController;

#pragma mark -
#pragma mark Initialization

- initWithKey:(NSString *)aKey values:(NSArray *)aValues
                       viewController:(id)viewController {
    [self setKey:aKey];
    [self setValues:aValues];
    [self setDetailViewController:viewController];

    return self;
}

#pragma mark -
#pragma mark View lifecycle

- (void)viewDidLoad {
    [super viewDidLoad];
    [self setClearsSelectionOnViewWillAppear:NO];
    [self setContentSizeForViewInPopover:CGSizeMake(200.0, 500.0)];
    [self setTitle:[self key]];
}

#pragma mark -
#pragma mark Rotation support

// Ensure that the view controller supports rotation and that the
// split view can therefore
// show in both portrait and landscape.
- (BOOL)shouldAutorotateToInterfaceOrientation:
                            (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}

#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 1;
}

- (NSInteger)tableView:(UITableView *)tableView
                                numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    return [[self values] count];
```

```
}

#pragma mark -
#pragma mark Table view appearance

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
                  cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    NSString *value = [[self values] objectAtIndex:[indexPath row]];

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
       [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                  initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellIdentifier] autorelease];
    }

    // Configure the cell...
    [[cell textLabel] setText:value];

    return cell;
}

#pragma mark -
#pragma mark Table view delegate

- (void)tableView:(UITableView *)tableView
                           didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    [[self detailViewController] setText:[[self values]
                          objectAtIndex:[indexPath row]]];
}

#pragma mark -
#pragma mark Memory management

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)viewDidUnload {
    // Relinquish ownership of anything that can be recreated
    // in viewDidLoad or on demand.
    // For example: self.myOutlet = nil;
    [self setDetailViewController:nil];
    [self setKey:nil];
    [self setValues:nil];
}

- (void)dealloc {
    [detailViewController release];
    [key release];
    [values release];
    [super dealloc];
}

@end
```

DetailViewController.h Modifications to the Template

When either the even or the odd values are selected from the navigation bar, the values will be displayed on a label in the center of the `DetailViewController`'s view. The complete `DetailViewController` file is shown in Listing 1-15.

LISTING 1-15: The complete DetailViewController.h file (Chapter1/NavigationBar-iPad/Classes/DetailViewController.h)

```
#import <UIKit/UIKit.h>

@interface DetailViewController : UIViewController
             <UIPopoverControllerDelegate, UISplitViewControllerDelegate> {

    UIPopoverController *popoverController;
    UIToolbar *toolbar;

    id detailItem;
    UILabel *detailDescriptionLabel;
}

@property (nonatomic, retain) IBOutlet UIToolbar *toolbar;

@property (nonatomic, retain) id detailItem;
@property (nonatomic, retain) IBOutlet UILabel *detailDescriptionLabel;

- (void)setText:(NSString *)newText;

@end
```

DetailViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `DetailViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-16).

LISTING 1-16: Addition of @synthesize

```
#import "DetailViewController.h"
#import "RootViewController.h"

@interface DetailViewController ()
@property (nonatomic, retain) UIPopoverController *popoverController;
@end

@implementation DetailViewController

@synthesize toolbar, popoverController, detailItem, detailDescriptionLabel;
```

The `setText` method, shown in Listing 1-17, sets the text on the label in the middle of the view, and dismisses the popover.

LISTING 1-17: The setText method

```
#pragma mark -
#pragma mark Managing the detail item

- (void)setText:(NSString *)newText {
```

```
    [[self detailDescriptionLabel] setText:newText];

    if (popoverController != nil) {
        [popoverController dismissPopoverAnimated:YES];
    }
}
```

The complete `DetailViewController.m` file is shown in Listing 1-18.

Available for download on Wrox.com

LISTING 1-18: The complete DetailViewController.m file (Chapter1/NavigationBar-iPad/Classes/DetailViewController.m)

```
#import "DetailViewController.h"
#import "RootViewController.h"


@interface DetailViewController ()
@property (nonatomic, retain) UIPopoverController *popoverController;
@end

@implementation DetailViewController

@synthesize toolbar, popoverController, detailItem, detailDescriptionLabel;

#pragma mark -
#pragma mark Managing the detail item

- (void)setText:(NSString *)newText {
    [[self detailDescriptionLabel] setText:newText];

    if (popoverController != nil) {
        [popoverController dismissPopoverAnimated:YES];
    }
}

#pragma mark -
#pragma mark Split view support

- (void)splitViewController: (UISplitViewController*)svc
     willHideViewController:(UIViewController *)aViewController
          withBarButtonItem:(UIBarButtonItem*)barButtonItem
       forPopoverController: (UIPopoverController*)pc {

    barButtonItem.title = @"Root List";
    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items insertObject:barButtonItem atIndex:0];
    [toolbar setItems:items animated:YES];
    [items release];
    self.popoverController = pc;
}


// Called when the view is shown again in the split view, invalidating
// the button and popover controller.
- (void)splitViewController: (UISplitViewController*)svc
                willShowViewController:(UIViewController *)aViewController
             invalidatingBarButtonItem:(UIBarButtonItem *)barButtonItem {

    NSMutableArray *items = [[toolbar items] mutableCopy];
    [items removeObjectAtIndex:0];
    [toolbar setItems:items animated:YES];
```

```
    [items release];
    self.popoverController = nil;
}


#pragma mark -
#pragma mark Rotation support

// Ensure that the view controller supports rotation and that the
// split view can therefore show in both portrait and landscape.
- (BOOL)shouldAutorotateToInterfaceOrientation:
                    (UIInterfaceOrientation)interfaceOrientation {
    return YES;
}

#pragma mark -
#pragma mark View lifecycle

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    //self.popoverController = nil;
}


#pragma mark -
#pragma mark Memory management

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)dealloc {
    [popoverController release];
    [toolbar release];

    [detailItem release];
    [detailDescriptionLabel release];
    [super dealloc];
}

@end
```

Test Your Application

Now that you have everything completed, launch the iPad Simulator. It should give you the results described at the beginning of A Simple Navigation Bar section.

THE TOOLBAR

The toolbar differs from the navigation bar initially, because it appears at the bottom of the view rather than the top, and it also differs because all the available views are are easily accessed from buttons that reside on the toolbar itself. Users do not have to drill down to the view they desire, they simply tap a button on the toolbar and the associated view is immediately displayed.

Items that are to appear on the toolbar are equally spaced and include fixed and flexible items that keep the presentation uniform. The thing to keep in mind is that the finger width will have a hard time selecting more than five items due to the size of the human finger and the limited space of the iPhone. The iPad has a larger view to handle the touch of a finger.

The items are instances of `UIBarButtonItem` and can have the following styles:

- `UIBarButtonItemStylePlain`
- `UIBarButtonItemStyleBordered`
- `UIBarButtonItemStyleDone`

They are similar to regular buttons but have additional functionality for use with navigation. The following methods are used for initialization:

- `initWithBarButtonSystemItem:target:action:`
- `initWithCustomView:`
- `initWithImage:style:target:action:`
- `initWithTitle:style:target:action:`

A SIMPLE TOOLBAR

In this application, an image is centered in the display. The user will tap several of the angle toolbar items. The image will rotate according to the angle value of the toolbar item, as shown in Figure 1-4.

FIGURE 1-4

Development Steps: A Simple Toolbar

To create a simple toolbar application, execute the following steps:

1. Start Xcode and create a View-based application for the iPhone and name it **SimpleToolbar-iPhone**. If you need to see this step, please see Appendix A for the steps to begin a View-based application.
2. You need to add one `UIImageView` and one `UIToolbar` and four `UIBarButtonItems` to the project.
   - Select Resources in Xcode's Groups & Files window on the left.
   - Choose Project ➪ Add To Project.
   - Select your image and name it, **grandpa.png.** Use an image around 300 × 300 pixels and click Add as shown in Figure 1-5.
   - Check Copy items into destination group's folder, and click Add as shown in Figure 1-6.
3. Double-click the `SimpleToolbar_iPhoneView Controller.xib` file to launch Interface Builder (see Figure 1-7).
4. From the Interface Builder Library (Tools ➪ Library), choose and drag the following to the View window. Your interface should now look like Figure 1-8:
   - One `UIImageView` with the size of 260 × 260. To accomplish this do the following:
     1. Drag your `UIImageView` to the View window, where it will take up the entire view.
     2. Choose Tools ➪ Size Inspector from the main menu and you will see W:240 W:128 just under the Frame drop-down in the upper-right corner of the inspector.
     3. Change the size of the view to W:**260** H:**260** and center it on the view.

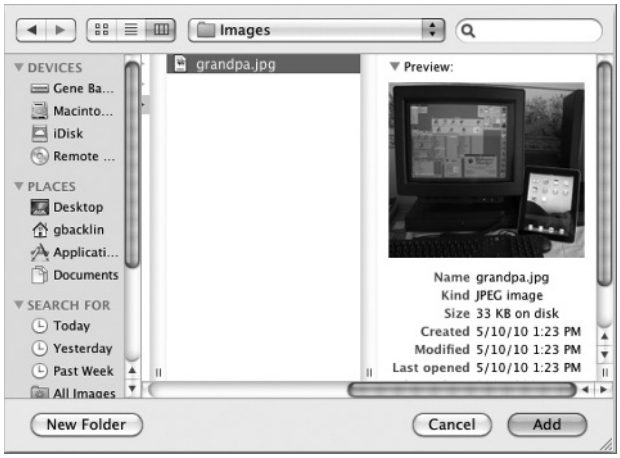

FIGURE 1-5

FIGURE 1-6

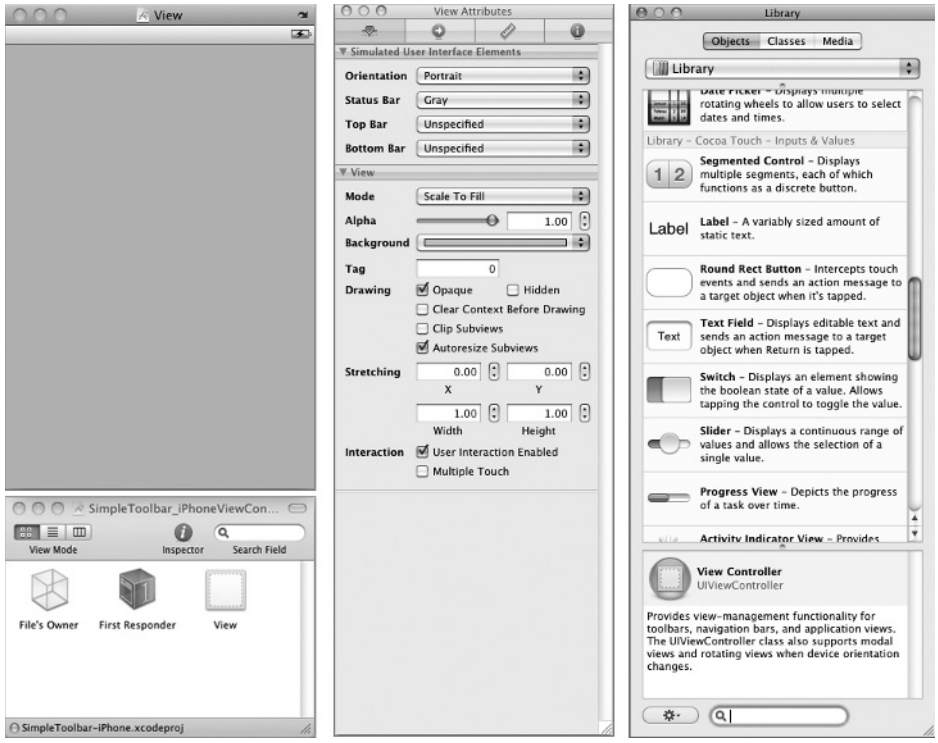

FIGURE 1-7

- One `UIToolbar` and place it at the bottom of the view.
- Three `UIBarButtonItems` and place them on the toolbar and choose Tools ➪ Attributes Inspector and enter the following:
  - +45 for the Title and 0 for the Tag for first button
  - +180 for the Title and 1 for the Tag for second button
  - −180 for the Title and 2 for the Tag for third button
  - −45 for the Title and 3 for the Tag for fourth button
- One Flexible Space Bar Button Item and place it to the left of your first button. Repeat, but place the other one to the right of the last button (see Figure 1-9).

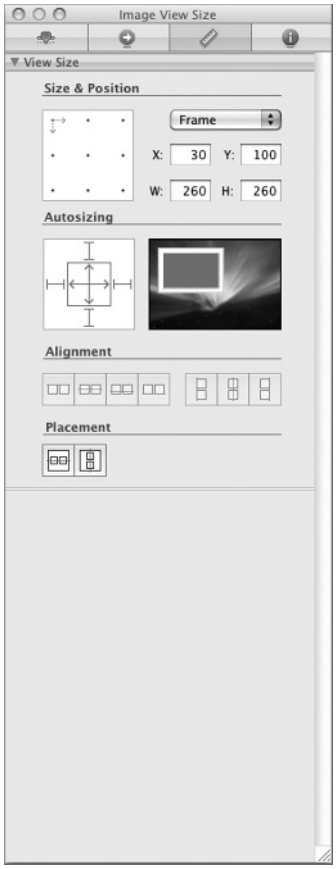

FIGURE 1-8

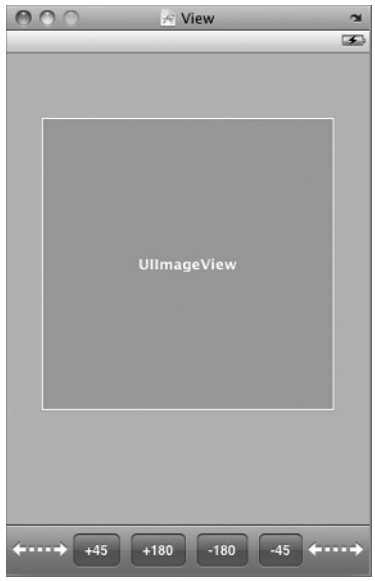

FIGURE 1-9

5. Back in the Interface Builder Library, click Classes at the top and scroll to and select your `SimpleToolbar_iPhoneViewController` class. At the bottom, now choose the Outlets button. Click the + and add the following outlet, as shown in Figure 1-10:

- `imageView` (as a `UIImageView` instead of id type)

6. Choose the Actions button. Then click the + and add the following action, as shown in Figure 1-11:
   - `rotateView`

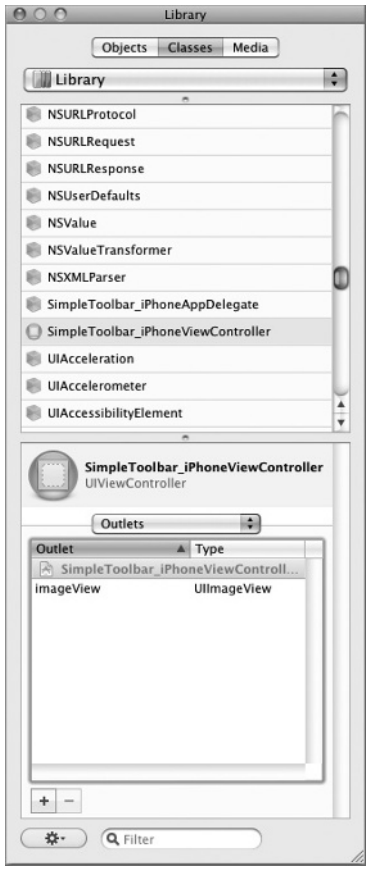

FIGURE 1-10

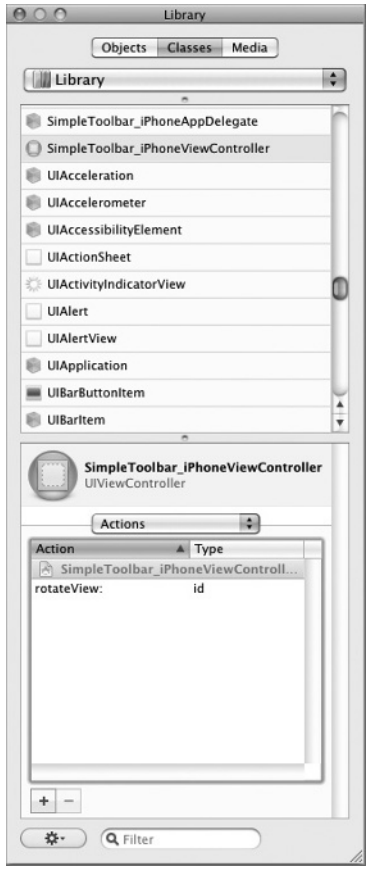

FIGURE 1-11

7. From the main menu of Interface Builder, choose File ➪ Write Class Files, select Save from the first pop-up and then Merge from the second pop-up. The `SimpleToolbar_iPhoneViewController.m` file appears with your new additions on the left and the original template on the right (see Figure 1-12).
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose File ➪ Save Merge and close the window.

8. For the next window, `SimpleToolbar_iPhoneViewController.h`, your new addition is on the left and the original template is on the right (see Figure 1-13).
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose Find ➪ Go to Next ➪ Difference.
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose File ➪ Save Merge and close the window.

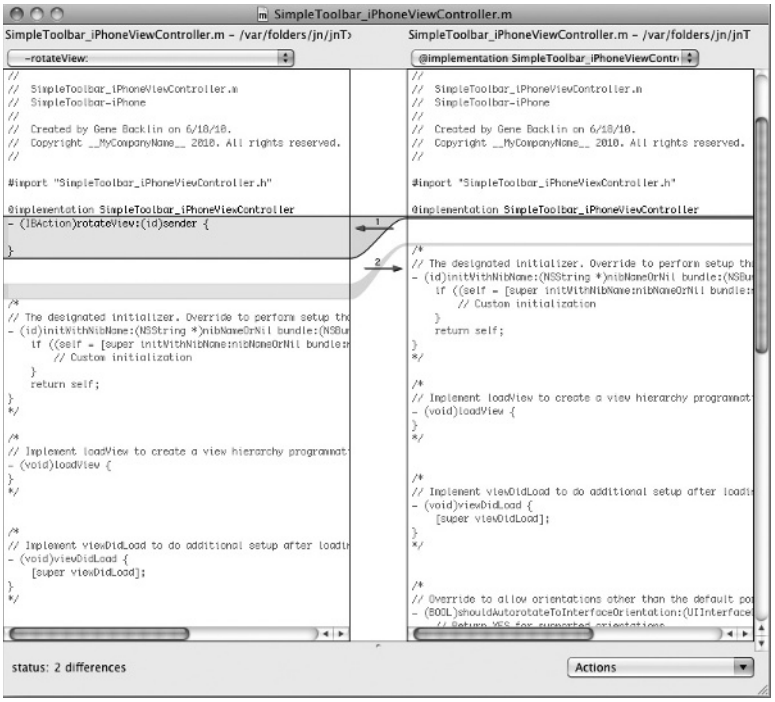

FIGURE 1-12

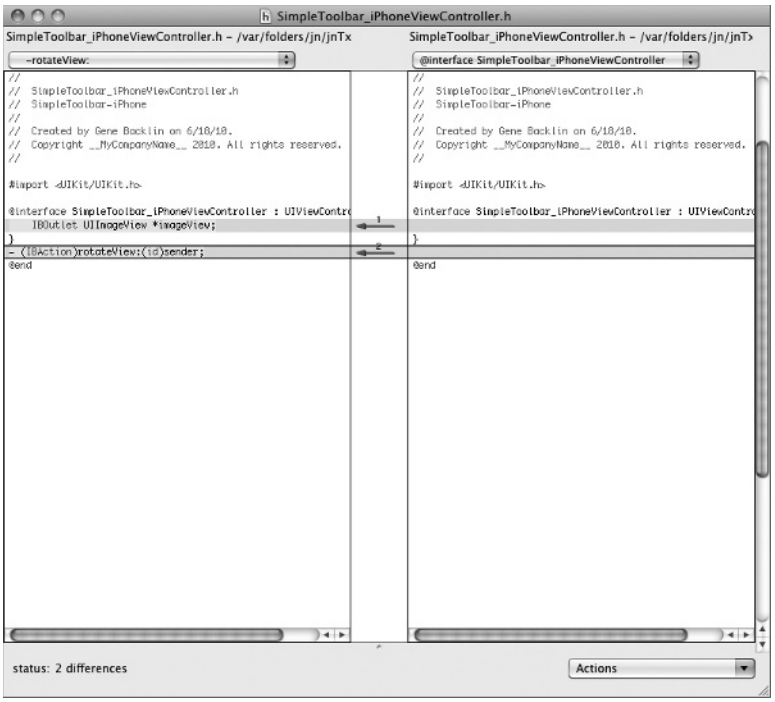

FIGURE 1-13

9. You now have an Objective-C template that holds your application's view logic. Before you begin actually programming, there is one more task to complete in Interface Builder. You have to make the connection to:

   - Identify the `UIImageView` as `imageView`

   To make the connection to identify the `UIImageView` as `imageView`, control-click on the File's Owner icon to bring up the Inspector (see Figure 1-14).

10. From the right of the File's Owner Inspector, control-drag from the circle by `imageView` to the `UIImageView` `imageView` until it is highlighted, then release the mouse. The circle will be filled, indicating that the connection has been made.

11. From the right of the File's Owner Inspector, control-drag from the circle by `rotateView` to each of the `UIBarButtonItems` (see Figure 1-15). Choose File ➪ Save and dismiss the File's Owner Inspector.

FIGURE 1-14

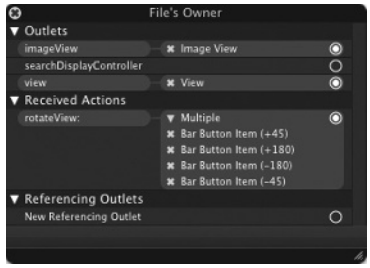

FIGURE 1-15

Now it is time to enter your logic.

Source Code Listings for A Simple Toolbar

For this application, the `SimpleToolbar_iPhoneAppDelegate.h` and `SimpleToolbar_iPhoneAppDelegate.m` files are not modified and are used as generated:

SimpleToolbar_iPhoneViewController.h Modifications to the Template

You declared the following outlet in Interface Builder:

- `imageView`

You must now define the properties for this variable in order to get and set its value (see Listing 1-19).

The `IBOutlet` was moved to the property declaration.

LISTING 1-19: The complete SimpleToolbar_iPhoneViewController.h file (/Chapter1/SimpleToolbar-iPhone/Classes/SimpleToolbar_iPhoneViewController.h)

```
#import <UIKit/UIKit.h>

@interface SimpleToolbar_iPhoneViewController : UIViewController {
    UIImageView *imageView;
}

@property (nonatomic, retain) IBOutlet UIImageView *imageView;

- (IBAction)rotateView:(id)sender;

@end
```

SimpleToolbar_iPhoneViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `SimpleToolbar_iPhoneViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-20).

LISTING 1-20: Addition of @synthesize

```
#import "SimpleToolbar_iPhoneViewController.h"

@implementation SimpleToolbar_iPhoneViewController

@synthesize imageView;
```

When the app launches, the default image, `grandpa.png`, is loaded and displayed (see Listing 1-21).

LISTING 1-21: The viewDidLoad method

```
#pragma mark -
#pragma mark View lifecycle

// Implement viewDidLoad to do additional setup after loading the view,
// typically from a nib.

- (void)viewDidLoad {
    [super viewDidLoad];

   [imageView setImage:[UIImage imageNamed:@"grandpa.jpg"]];
}
```

When the toolbar item buttons are tapped, the `rotateView` method is called and, depending on the item's tag value, determines the rotation angle, as shown in Listing 1-22.

LISTING 1-22: The rotateView method

```
#pragma mark -
#pragma mark Action methods

- (IBAction)rotateView:(id)sender {
   static CGFloat angle = 0.0;

   switch ([sender tag]) {
      case 0:
         angle += 45.0;
         break;
      case 1:
         angle += 180.0;
         break;
      case 2:
         angle -= 180.0;
         break;
      case 3:
         angle -= 45.0;
         break;
      default:
         break;
   }

   CGAffineTransform transform=CGAffineTransformMakeRotation(angle);
   [imageView setTransform:transform];
}
```

The `SimpleToolbar_iPhoneViewController.m` file is now complete. Listing 1-23 shows the complete implementation.

Available for download on Wrox.com

LISTING 1-23: The complete SimpleToolbar_iPhoneViewController.m file (/Chapter1/SimpleToolbar-iPhone/Classes/SimpleToolbar_iPhoneViewController.h

```
#import "SimpleToolbar_iPhoneViewController.h"

@implementation SimpleToolbar_iPhoneViewController

@synthesize imageView;

#pragma mark -
#pragma mark View lifecycle

// Implement viewDidLoad to do additional setup after loading the view,
// typically from a nib.
- (void)viewDidLoad {
   [super viewDidLoad];

   [imageView setImage:[UIImage imageNamed:@"grandpa.jpg"]];
}

#pragma mark -
#pragma mark Action methods

- (IBAction)rotateView:(id)sender {
   static CGFloat angle = 0.0;

   switch ([sender tag]) {
      case 0:
         angle += 45.0;
         break;
      case 1:
         angle += 180.0;
         break;
      case 2:
         angle -= 180.0;
         break;
      case 3:
         angle -= 45.0;
         break;
      default:
         break;
   }

   CGAffineTransform transform=CGAffineTransformMakeRotation(angle);
   [imageView setTransform:transform];
}

#pragma mark -
#pragma mark Memory methods

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)viewDidUnload {
   // Release any retained subviews of the main view.
   // e.g. self.myOutlet = nil;
   [self setImageView:nil];
```

```
}

- (void)dealloc {
   [imageView release];
    [super dealloc];
}

@end
```

Test Your Application

Now that you have everything completed, choose the Simulator from the Xcode panel and click Run and Build to try out your app. It should give you the results described at the beginning of the "A Simple Toolbar" section.

THE TAB BAR

The tab bar is used either when you have different operations on a single procedure or when you have different views on a single data source, such as a weather application that provides views for current temperature, hourly view, five-day forecast, and satellite images.

The tab bar provides a convenient way for users to view different perspectives on their data without having to drill down to find the results.

UITabBarDelegate Protocol

The `UITabBarDelegate` protocol defines methods of `UITabBar` delegates that provide customization of the tab bar itself.

Customizing Tab Bars

The process of customizing a tab bar involves adding, removing, or reordering items, and it uses the following methods (one method, `tabBar:didSelectItem`, is required):

- `tabBar:willBeginCustomizingItems:`
- `tabBar:didBeginCustomizingItems:`
- `tabBar:willEndCustomizingItems:changed:`
- `tabBar:didEndCustomizingItems:changed:`
- `tabBar:didSelectItem:`

Programmatically, you can customize the tab bar through the `beginCustomizingItems` method. Calling this method will create a modal view containing the items and a Done button, which when tapped dismisses the modal view.

A SIMPLE TAB BAR

This application will simulate bank account transactions, including a summary. Each of the following three options will be represented by a button on the tab bar:

- Show current balance, as shown in Figure 1-16.
- Support transactions (deposit and withdrawal), as shown in Figure 1-17.
- Display a summary of transactions, as shown in Figure 1-18.

FIGURE 1-16

FIGURE 1-17

FIGURE 1-18

Development Steps: A Simple Tab Bar

To create this application, execute the following steps:

1. Start Xcode and create a View-based application for the iPhone and name it **SimpleTabbar-iPhone**. If you need to see this step, please see Appendix A for the steps to begin a View-based application.
2. Choose File ➪ New File… and select Objective-C class as the subclass of `UIViewController`, with no options checked, as shown in Figure 1-19, and name the class **SecondViewController**.
3. Choose File ➪ New File… and select Objective-C class as the subclass of `UIViewController`, name the class **ThirdViewController,** and check the following options:
   - UITableViewController subclass
   - With XIB for user interface
4. Choose File ➪ New File… and select Objective-C class as the subclass of `NSObject`, and name the class **Transaction.**
5. Choose File ➪ New File… and select Objective-C class as the subclass of `NSObject`, and name the class **PropertyList.**
6. Double-click the `MainWindow.xib` file to launch Interface Builder. Note that the View Mode is in browser mode and select the Tab Bar Controller (see Figure 1-20).

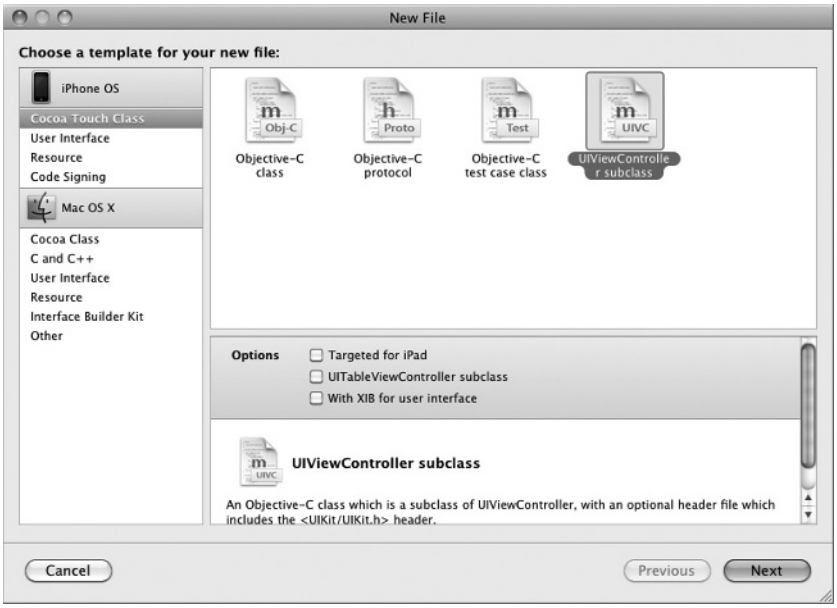

FIGURE 1-19

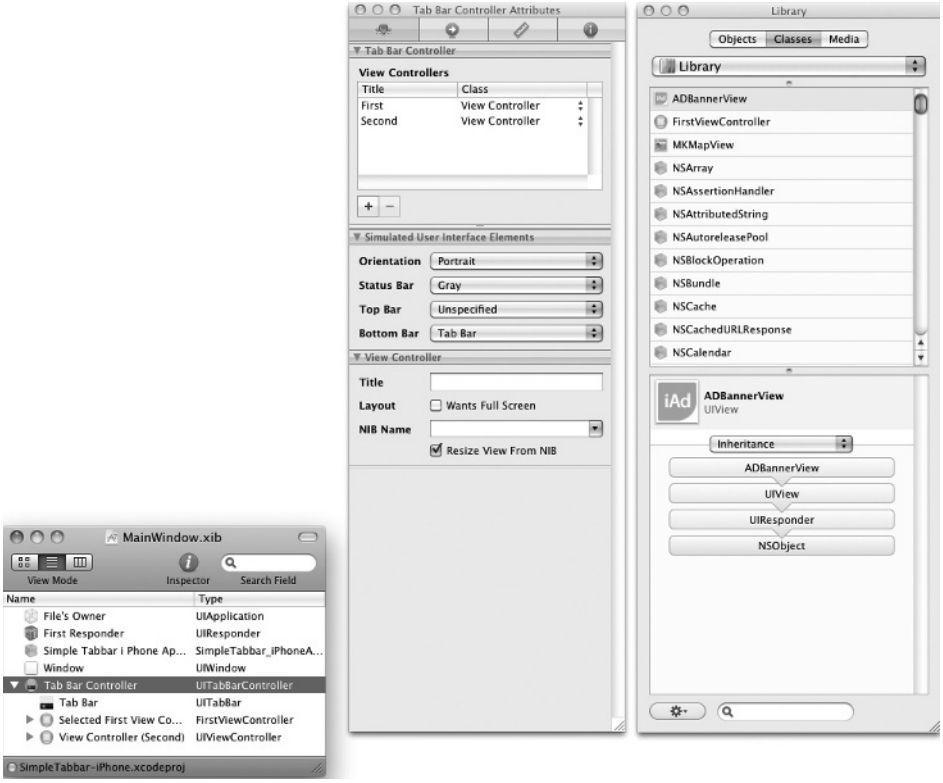

FIGURE 1-20

7. From the Interface Builder Library (Tools ➪ Attributes Inspector), click on the following in the View Controllers section (see Figure 1-21):
   - Double-click First and replace with **Balance**.
   - Double-click Second and replace with **Transaction**.
   - Click the plus button and add **Summary** for the title and `TableViewController` for the class.

8. From the `MainWindow.xib` window, click on the following under the Tab Bar Controller section (see Figure 1-22):
   - Click the third entry, `TableViewController (Summary)`; now, in the Attributes Inspector under Nib name, choose `ThirdViewController`.
   - Double-click Second and replace with **Transaction**.
   - Click the plus button and add **Summary** for the title and `TableViewController` for the class.

9. From the main menu, click Tools ➪ Identity Inspector, and choose `ThirdViewController` for the class identity.

10. Under the Tab Bar Controller entry in the `MainWindow.xib` window, choose `View Controller (Transaction)`, which is just above the Third View Controller you previously selected. From the Identity Inspector, select `SecondViewController` for the class identity.

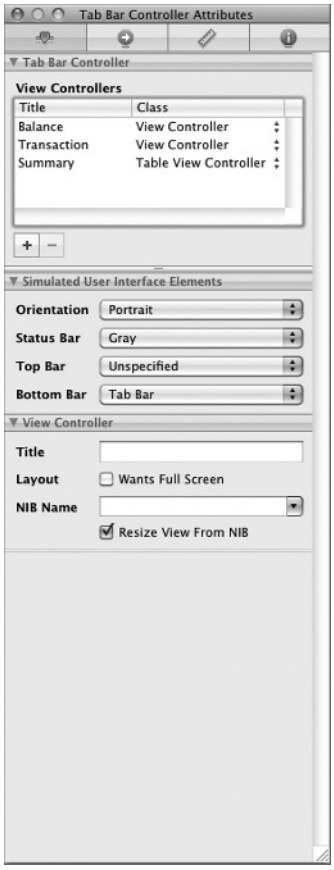

FIGURE 1-21

FIGURE 1-22

11. Double-click the Tab Bar Controller entry in the `MainWindow.xib` window, and your Tab View Controller window should appear as shown in Figure 1-23.
12. Click the third button, Summary, and then click Tools ➪ Identity Inspector and select `ThirdViewController` from the Class Identity. Finally, choose File ➪ Save.

Designing the View Controllers

In this section each button in the tab bar is associated with its own view controller. Each view controller is divided into its own series of steps.

The First View Controller

The first view controller will display the current balance in the bank account. To create this view:

1. Return back to Xcode and in the Groups & Files window, double-click on `FirstView.xib` to bring up the window, which has some labels already in it. Select and delete these labels.
2. From Interface Builder (Tools ➪ Identity Inspector), select File's Owner and `FirstViewController` as the class.
   - Add two `UILabels` next to each other in the middle of the view. Double-click the leftmost label and enter **Balance** (see Figure 1-24).
3. In the Interface Builder Library, click Classes at the top and scroll to and select your `FirstViewController` class. At the bottom, choose the Outlets button. Click the + and add the following outlet, as shown in Figure 1-25:
   - `balanceLabel` (as a `UILabel` instead of id type)
4. From the main menu of Interface Builder, choose File ➪ Write Class Files, select Save and then Merge. Close the `FirstViewController.m` window, as there are no changes.

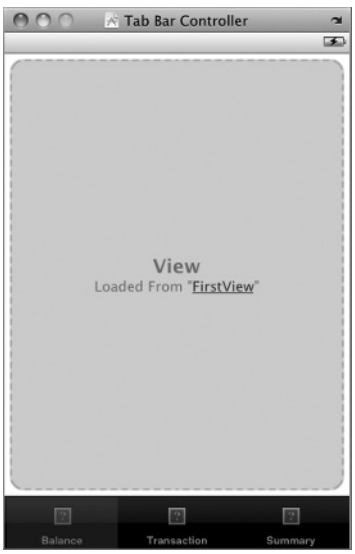

FIGURE 1-23

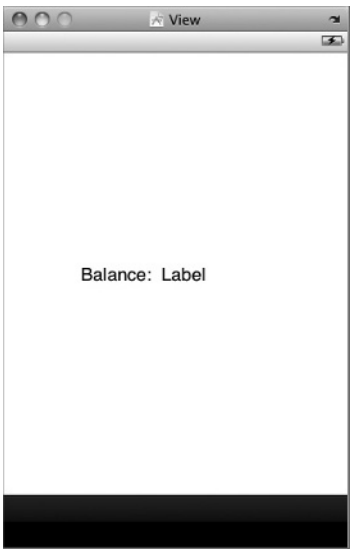

FIGURE 1-24

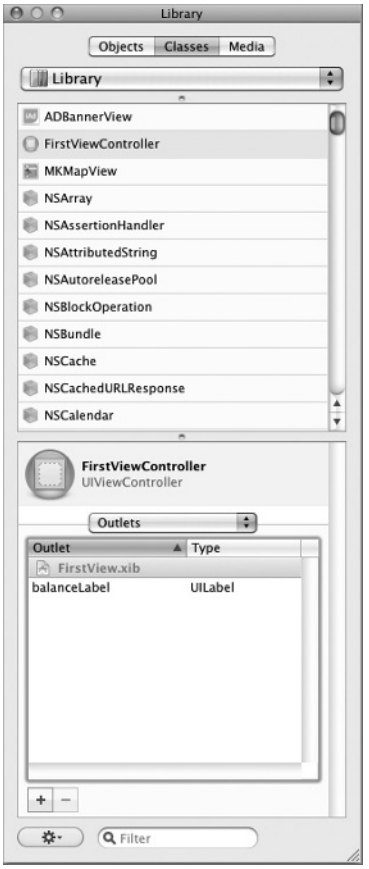

FIGURE 1-25

The `FirstViewController.h` file appears with your new additions on the left and the original template on the right (see Figure 1-26).

- In the lower-right corner, choose Actions ➪ Choose Left.
- Choose File ➪ Save Merge and close the window.

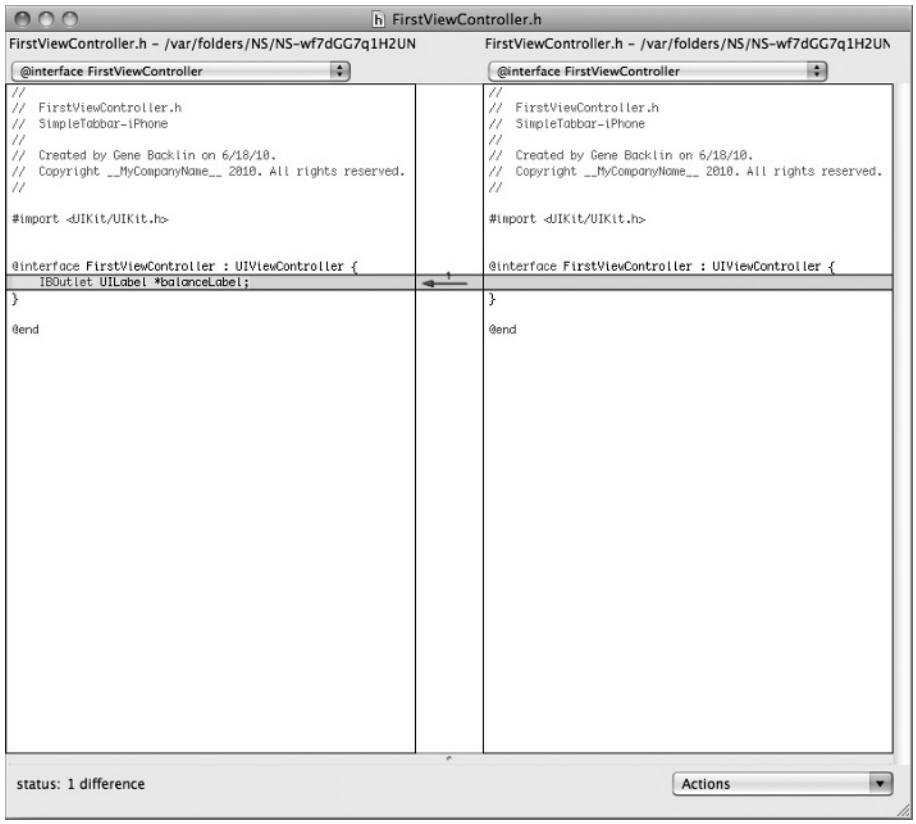

FIGURE 1-26

5. You now have an Objective-C template that holds your application's view logic. Before you begin actually programming, there is one more task to complete in Interface Builder. You have to make all the connections to:

- Identify the `UILabel` as `balanceLabel`

6. To make the connection to identify the `UILabel` as `balanceLabel`, control-click on the File's Owner icon to bring up the Inspector (see Figure 1-27).

7. From the right of the File's Owner Inspector, control-drag from the circle by `balanceLabel` to the `UILabel balanceLabel` until it is highlighted, then release the mouse. The circle will be filled, indicating that the connection has been made (see Figure 1-28). Choose File ➪ Save and dismiss the File's Owner Inspector.

FIGURE 1-27

FIGURE 1-28

The Second View Controller

The second view controller is the transaction view, where the deposits and withdrawals are entered. To create this view:

1. Back in Xcode in the Groups & Files window, double-click on `SecondView.xib` to bring up the window, which has some labels already in it. Select and delete these labels.
2. From Interface Builder (Tools ➪ Identity Inspector), select File's Owner and `SecondViewController` as the class.
   - Add one `UISegmentedControl` at the top of the view. Double-click the left segment and enter **Deposit**. Double-click the right segment and enter **Withdrawal**.
   - Add one `UILabels` and one `UITextField` placed right next to each other in the middle of the view. Double-click the label and enter **Amount:**.
   - Add one `UIButton` just below the label and text field. Double-click the button and enter **Save** (see Figure 1-29).
3. In the Interface Builder Library, click Classes at the top and scroll to and select your `FirstViewController` class. At the bottom, choose the Outlets button. Click the + and add the following outlets, as shown in Figure 1-30:
   - `amountTextField` (as a `UITextField` instead of id type)
   - `transactionType` (as a `UISegmentedControl` instead of id type)
4. Choose the Actions button. Then click the + and add the following action, as shown in Figure 1-31:
   - `saveTransaction`

FIGURE 1-29

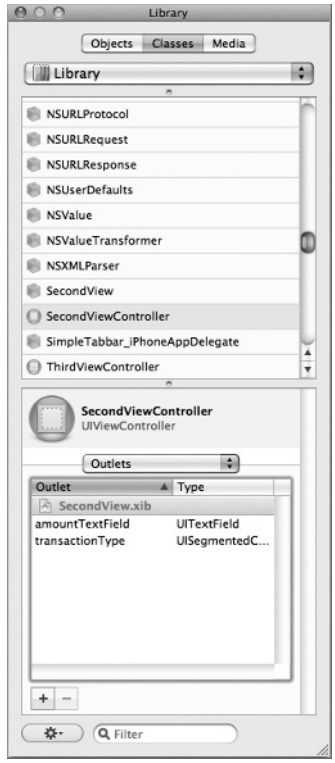

FIGURE 1-30

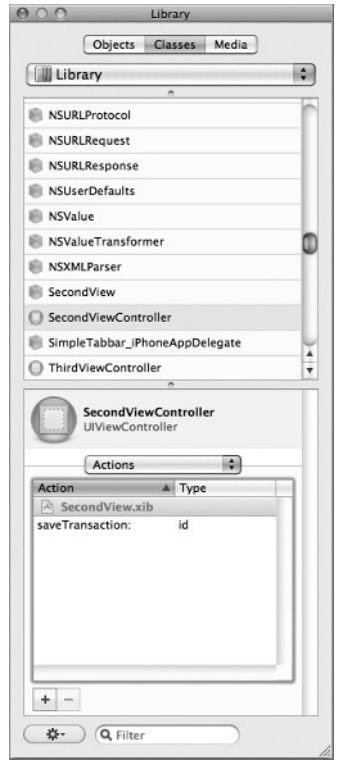

FIGURE 1-31

5. From the main menu of Interface Builder, choose File ➪ Write Class Files, select Save and then Merge. The `SecondViewController.h` file appears with your new additions on the left and the original template on the right, as shown in Figure 1-32.
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose Find ➪ Go to Next ➪ Difference.
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose File ➪ Save Merge and close the window.

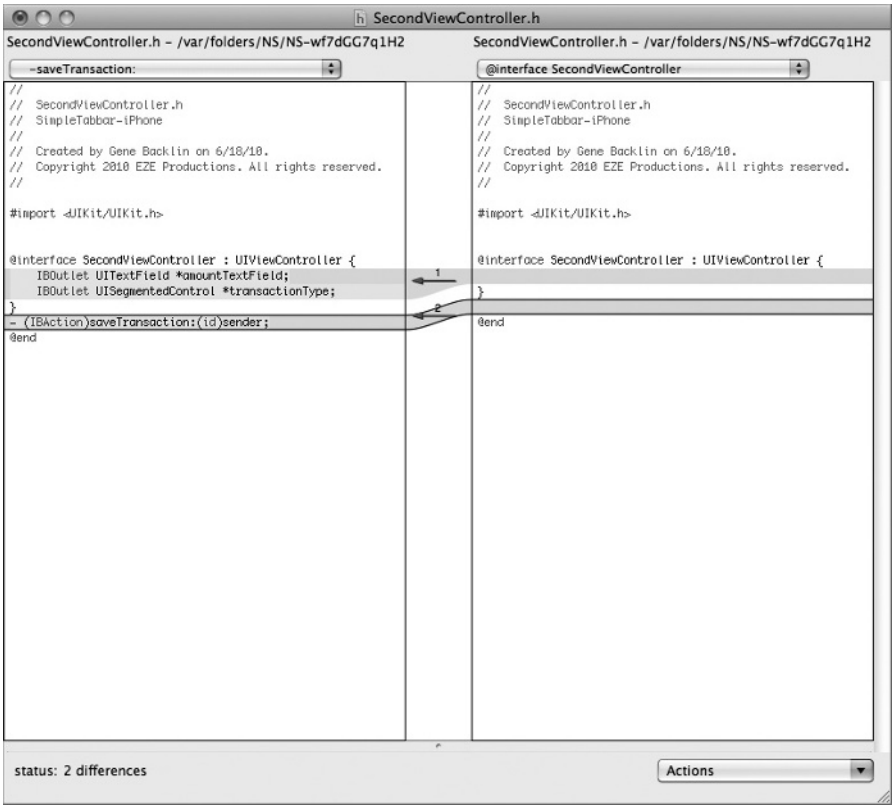

FIGURE 1-32

6. From the main menu of Interface Builder, choose File ➪ Write Class Files, select Save and then Merge. The `SecondViewController.m` file appears with your new additions on the left and the original template on the right, as shown in Figure 1-33.
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose File ➪ Save Merge and close the window.

7. You now have an Objective-C template that holds your application's view logic. Before you begin actually programming, there is one more task to complete in Interface Builder. You have to make all the connections to:
   - Identify the `UITextField` as `amountTextField`
   - Identify the `UISegmentedControl` as `transactionType`

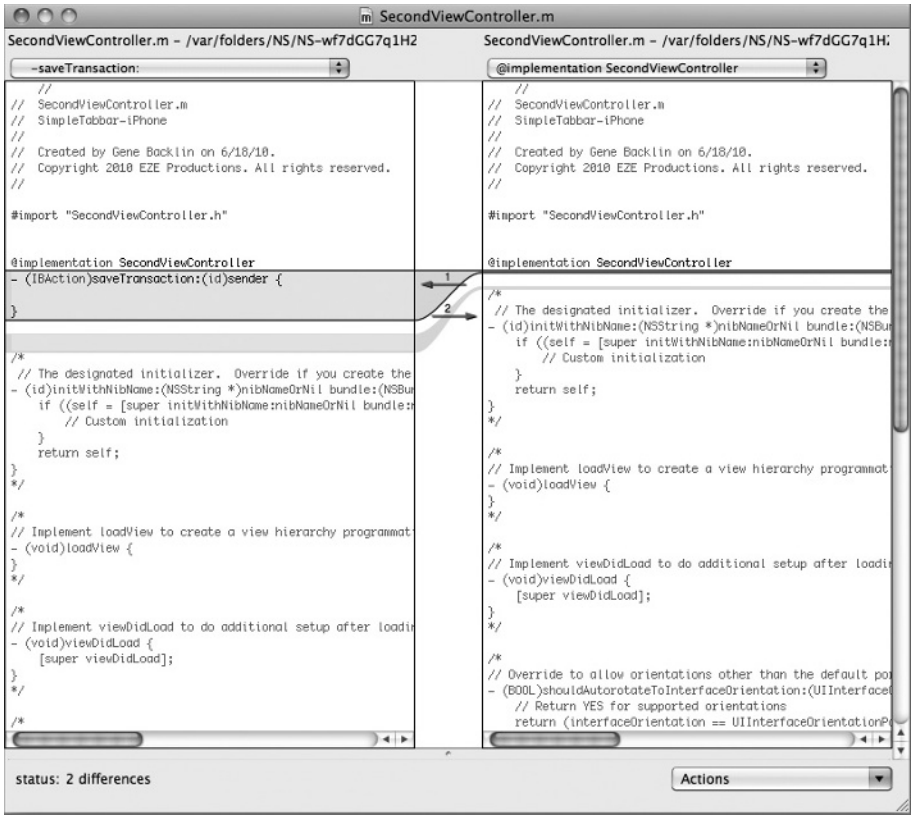

FIGURE 1-33

8. To make the connection to identify the `UITextField` as `amountTextField`, control-click on the File's Owner icon to bring up the Inspector (see Figure 1-34).
9. From the right of the File's Owner Inspector, control-drag from the circle by `amountTextField` to the `UITextField amountTextField` until it is highlighted, then release the mouse. The circle will be filled, indicating that the connection has been made.
10. From the right of the File's Owner Inspector, control-drag from the circle by `transactionType` to the `UISegmentedControl transactionType` until it is highlighted, then release the mouse.
11. From the right of the File's Owner Inspector, control-drag from the circle by `saveTransaction` to the Save button until it is highlighted, then release the mouse and choose Touch Up Inside. (see Figure 1-35). Choose File ➪ Save and dismiss the File's Owner Inspector.

FIGURE 1-34

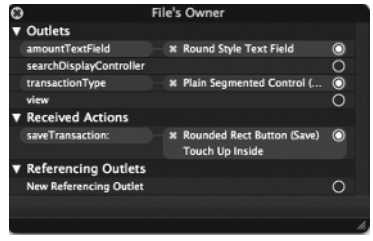

FIGURE 1-35

The Third View Controller

The third view controller displays all the transactions that have been entered into the applications. The transaction amounts are separated into deposits and withdrawal sections for clarity.

1. Back in Xcode, in the Groups & Files window, double-click on `ThirdViewController.xib` to bring up the window.
2. From Interface Builder (Tools ➪ Identity Inspector), select File's Owner and `ThirdViewController` as the class.
3. From the Interface Builder (Tools ➪ Attributes Inspector), select the table view and choose Tab Bar for the Bottom Bar in the Simulated User Interface Elements section (see Figure 1-36).
4. In the Interface Builder Library, click Classes at the top and scroll to and select your `ThirdView Controller` class. At the bottom, choose the Outlets button. Click the + and add the following outlet, as shown in Figure 1-37.
   - `detailTableView` (as a `UITableView` instead of id type)
5. From the main menu of Interface Builder, choose File ➪ Write Class Files, select Save and then Merge. Close the `ThirdViewController.m` window, as there are no changes to it. The `ThirdView Controller.h` file appears with your new additions on the left and the original template on the right (see Figure 1-38).
   - In the lower-right corner, choose Actions ➪ Choose Left.
   - Choose File ➪ Save Merge and close the window.

FIGURE 1-36

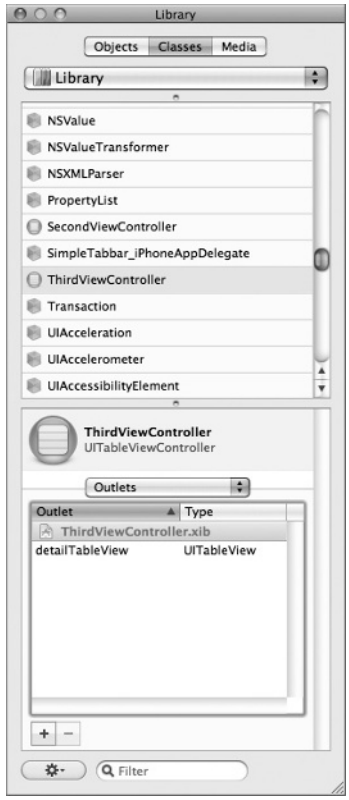

FIGURE 1-37

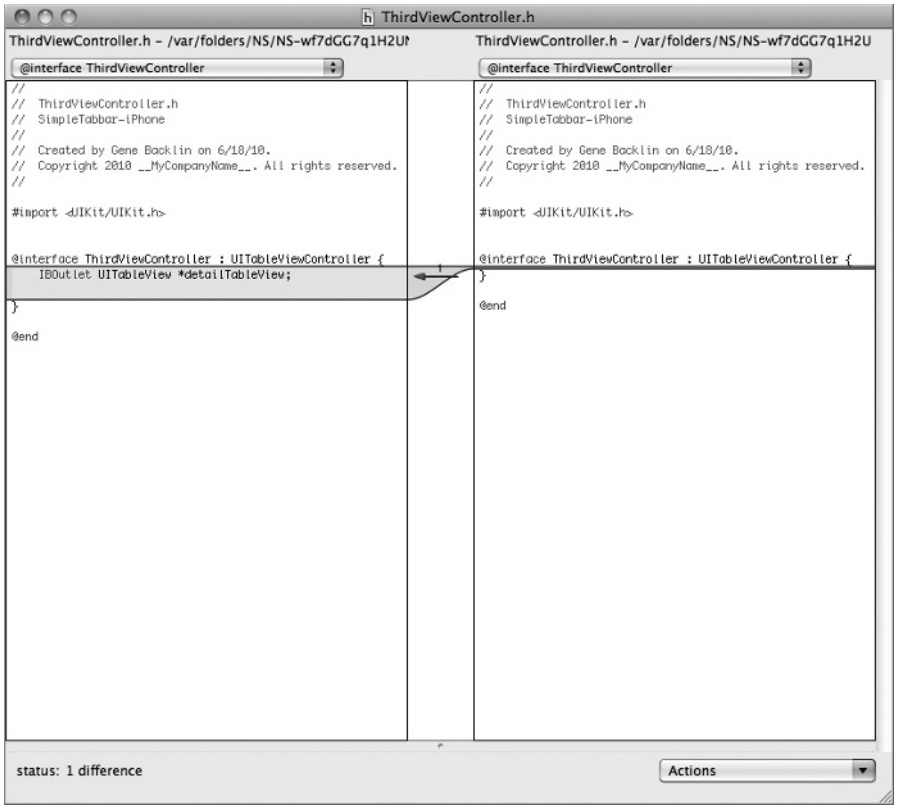

FIGURE 1-38

6. You now have an Objective-C template that holds your application's view logic. Before you begin actually programming, there is one more task to complete in Interface Builder. You have to make all the connections to:

- Identify the `UITableView` as `detailTableView`

7. To make the connection to identify the `UITableView` as `detailTableView`, click on File's Owner and select `ThirdViewController` for the Class Identity. Now control-click on the File's Owner icon to bring up the Inspector (see Figure 1-39).

8. From the right of the File's Owner Inspector, control-drag from the circle by `detailTableView` to the `UITableView` `detailTableView` until it is highlighted, then release the mouse. The circle will be filled, indicating that the connection has been made (see Figure 1-40). Choose File ➪ Save and dismiss the File's Owner Inspector.

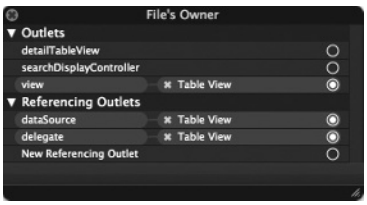

FIGURE 1-39

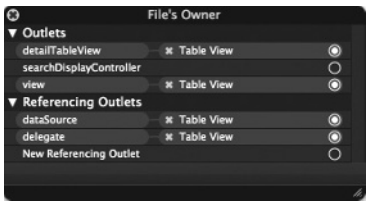

FIGURE 1-40

This concludes the user interface design section. The next section contains the source code that will provide the logic for the application.

Source Code Listings for A Simple Tab Bar

For this application, the `SimpleTabbar_iPhoneAppDelegate.h` and `SimpleTabbar_iPhoneAppDelegate.m` files are not modified and are used as generated.

FirstViewController.h Modifications to the Template

You declared the following outlet in Interface Builder:

- `balanceLabel`

You must now define the property for this variable in order to get and set its value (see Listing 1-24).

The `IBOutlet` was moved to the property declaration.

LISTING 1-24: The complete FirstViewController.h file (Chapter1/Tabbar-iPhone/Classes/FirstViewController.h)

```
#import <UIKit/UIKit.h>
#import "PropertyList.h"


@interface FirstViewController : UIViewController {
    UILabel *balanceLabel;
}

@property (nonatomic, retain) IBOutlet UILabel *balanceLabel;

@end
```

FirstViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `FirstViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-25).

LISTING 1-25: Addition of @synthesize

```
#import "FirstViewController.h"


@implementation FirstViewController

@synthesize balanceLabel;
```

When the app launches, any transactions that have occurred are stored in the file `Data.plist`, which is now loaded; and the current balance is displayed (see Listing 1-26).

LISTING 1-26: The viewWillAppear method

```
# pragma mark -
# pragma mark Initialization routines

- (void)viewWillAppear:(BOOL)animated {

    NSDictionary *d = [PropertyList readFromArchive:@"Data"];
```

```
    if(d != nil) {
        NSNumber *bal = [d objectForKey:@"balance"];
        if(bal != nil) {
            [balanceLabel setText:[bal stringValue]];
        } else {
            [balanceLabel setText:@"0"];
        }
    } else {
        [balanceLabel setText:@"0"];
    }

    [super viewWillAppear:animated];
}
```

Listing 1-27 shows the complete `FirstViewController.m` implementation.

Available for download on Wrox.com

LISTING 1-27: The complete FirstViewController.m file (Chapter1/Tabbar-iPhone/Classes/FirstViewController.m)

```
#import "FirstViewController.h"


@implementation FirstViewController

@synthesize balanceLabel;

# pragma mark -
# pragma mark Initialization routines

- (void)viewWillAppear:(BOOL)animated {

    NSDictionary *d = [PropertyList readFromArchive:@"Data"];
    if(d != nil) {
        NSNumber *bal = [d objectForKey:@"balance"];
        if(bal != nil) {
            [balanceLabel setText:[bal stringValue]];
        } else {
            [balanceLabel setText:@"0"];
        }
    } else {
        [balanceLabel setText:@"0"];
    }

    [super viewWillAppear:animated];
}

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    [self setBalanceLabel:nil];
}

- (void)dealloc {
    [balanceLabel release];
    [super dealloc];
}

@end
```

SecondViewController.h Modifications to the Template

You declared the following outlets in Interface Builder:

- `amountTextField`
- `transactionType`

You must now define the properties for these variables in order to get and set their values (see Listing 1-28).

The `IBOutlet` was moved to the property declaration.

Three variables were also added:

- `balance`
- `deposits`
- `withdrawals`

LISTING 1-28: The complete SecondViewController.h file (Chapter1/Tabbar-iPhone/Classes/SecondViewController.h)

```
#import <UIKit/UIKit.h>


@interface SecondViewController : UIViewController <UITextFieldDelegate> {
    UITextField *amountTextField;
    UISegmentedControl *transactionType;

    NSNumber *balance;
    NSArray *deposits;
    NSArray *withdrawals;
}

@property (nonatomic, retain) IBOutlet UITextField *amountTextField;
@property (nonatomic, retain) IBOutlet UISegmentedControl *transactionType;
@property (nonatomic, retain) NSNumber *balance;
@property (nonatomic, retain) NSArray *deposits;
@property (nonatomic, retain) NSArray *withdrawals;

- (IBAction)saveTransaction:(id)sender;

@end
```

SecondViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `SecondViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-29).

LISTING 1-29: Addition of @synthesize

```
#import "SecondViewController.h"
#import "PropertyList.h"


@implementation SecondViewController

@synthesize amountTextField;
@synthesize transactionType;
```

```
@synthesize balance;
@synthesize deposits;
@synthesize withdrawals;
```

When the app launches, any transactions that have occurred are stored in the file `Data.plist`, which is now loaded; and the values for the balance, deposits, and withdrawals are retained (see Listing 1-30).

LISTING 1-30: The viewWillAppear method

```
# pragma mark -
# pragma mark Initialization routines

- (void)viewWillAppear:(BOOL)animated {
    [amountTextField setDelegate:self];

    NSDictionary *d = [PropertyList readFromArchive:@"Data"];
    if(d != nil) {
        NSDictionary *localItems = [d objectForKey:@"items"];

        [self setBalance:[d objectForKey:@"balance"]];
        [self setDeposits:[localItems objectForKey:@"deposits"]];
        [self setWithdrawals:[localItems objectForKey:@"withdrawals"]];
    } else {
        [self setBalance:[NSNumber numberWithDouble:0.0]];
        [self setDeposits:[NSArray array]];
        [self setWithdrawals:[NSArray array]];
    }
    [super viewWillAppear:animated];
}
```

When the user taps the Save button, the value entered is either added or subtracted depending on which segmented button is selected (see Listing 1-31).

LISTING 1-31: The saveTransaction method

```
- (IBAction)saveTransaction:(id)sender {
    NSMutableDictionary *aDict = [NSMutableDictionary dictionary];
    NSMutableDictionary *itemDict = [NSMutableDictionary dictionary];
    NSMutableArray *transaction = nil;

    NSNumberFormatter * formatter = [[NSNumberFormatter alloc] init];
    [formatter setNumberStyle:NSNumberFormatterCurrencyStyle];
    NSString *amt = [amountTextField text];
    double localAmount = [amt doubleValue];
    double localBalance = [[self balance] doubleValue];
    [formatter release];

    if ([transactionType selectedSegmentIndex] == 0) {
        localBalance += localAmount;
        transaction = [[self deposits] mutableCopy];
        [transaction addObject:[NSNumber numberWithDouble:localAmount]];
        [self setDeposits:transaction];
        [self setBalance:[NSNumber numberWithDouble:localBalance]];
    } else {
        localBalance -= localAmount;
        transaction = [[self withdrawals] mutableCopy];
        [transaction addObject:[NSNumber numberWithDouble:localAmount]];
        [self setWithdrawals:transaction];
        [self setBalance:[NSNumber numberWithDouble:localBalance]];
```

```
    }

    [aDict setObject:[self balance] forKey:@"balance"];
    [itemDict setObject:[self deposits] forKey:@"deposits"];
    [itemDict setObject:[self withdrawals] forKey:@"withdrawals"];
    [aDict setObject:itemDict forKey:@"items"];

    if(![PropertyList writeToArchive:@"Data" fromDictionary:aDict]) {
        NSLog(@"Error writing data to pList.");
    }
}
```

When the user taps the Return button on the keyboard, the `UITextFieldDelegate` must implement the `textFieldShouldReturn` method so the keyboard will disappear (see Listing 1-32).

LISTING 1-32: The textFieldShouldReturn method

```
#pragma mark -
#pragma mark UITextFieldDelegate

- (BOOL)textFieldShouldReturn:(UITextField *)textField {
    // the user pressed the "Done" button, so dismiss the keyboard
    [textField resignFirstResponder];
    return YES;
}
```

Listing 1-33 shows the complete `SecondViewController.m` implementation.

Available for download on Wrox.com

LISTING 1-33: The complete SecondViewController.m file (Chapter1/Tabbar-iPhone/Classes/SecondViewController.m)

```
#import "FirstViewController.h"


@implementation FirstViewController

@synthesize balanceLabel;

# pragma mark -
# pragma mark Initialization routines

- (void)viewWillAppear:(BOOL)animated {

    NSDictionary *d = [PropertyList readFromArchive:@"Data"];
    if(d != nil) {
        NSNumber *bal = [d objectForKey:@"balance"];
        if(bal != nil) {
            [balanceLabel setText:[bal stringValue]];
        } else {
            [balanceLabel setText:@"0"];
        }
    } else {
        [balanceLabel setText:@"0"];
    }

    [super viewWillAppear:animated];
}

- (void)didReceiveMemoryWarning {
```

```
    [super didReceiveMemoryWarning];
}

- (void)viewDidUnload {
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
    [self setBalanceLabel:nil];
}

- (void)dealloc {
    [balanceLabel release];
    [super dealloc];
}

@end
```

ThirdViewController.h Modifications to the Template

You declared the following outlet in Interface Builder:

- `detailTableView`

You must now define the properties for this variable in order to get and set its value (see Listing 1-34).

The `IBOutlet` was moved to the property declaration.

Three variables were also added:

- `balance`
- `deposits`
- `withdrawals`

Available for download on Wrox.com

LISTING 1-34: The complete ThirdViewController.h file (Chapter1/Tabbar-iPhone/Classes/ThirdViewController.h)

```
#import <UIKit/UIKit.h>


@interface ThirdViewController : UITableViewController {
    UITableView *detailTableView;

    NSNumber *balance;
    NSArray *deposits;
    NSArray *withdrawals;
}

@property (nonatomic, retain) IBOutlet UITableView *detailTableView;
@property (nonatomic, retain) NSNumber *balance;
@property (nonatomic, retain) NSArray *deposits;
@property (nonatomic, retain) NSArray *withdrawals;

@end
```

ThirdViewController.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `ThirdViewController.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-35).

LISTING 1-35: Addition of @synthesize

```
#import "ThirdViewController.h"
#import "PropertyList.h"


@implementation ThirdViewController

@synthesize detailTableView;
@synthesize balance;
@synthesize deposits;
@synthesize withdrawals;
```

When the app launches, any transactions that have occurred are stored in the file `Data.plist`, which is now loaded; and the values for the balance, deposits, and withdrawals are retained. With the stored values now retained, the table view reload data is called; this refreshes the table view and displays the stored data (see Listing 1-36).

LISTING 1-36: The viewWillAppear method

```
- (void)viewWillAppear:(BOOL)animated {
    NSDictionary *d = [PropertyList readFromArchive:@"Data"];
    if(d != nil) {
        NSDictionary *localItems = [d objectForKey:@"items"];

        [self setBalance:[d objectForKey:@"balance"]];
        [self setDeposits:[localItems objectForKey:@"deposits"]];
        [self setWithdrawals:[localItems objectForKey:@"withdrawals"]];
    } else {
        [self setBalance:[NSNumber numberWithDouble:0.0]];
        [self setDeposits:[NSArray array]];
        [self setWithdrawals:[NSArray array]];
    }
    [[self detailTableView] reloadData];
    [super viewWillAppear:animated];
}
```

As shown in Listing 1-37, data that is used to populate a table view has to supply the table with two values:

- The number of sections into which the table view will be divided
- The number of rows in each section

LISTING 1-37: The numberOfSectionsInTableView and tableView:numberOfRowsInSection methods

```
#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 2;
}


- (NSInteger)tableView:(UITableView *)tableView
                        numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    switch (section) {
```

```
        case 0:
            return [[self deposits] count];
            break;
        case 1:
            return [[self withdrawals] count];
            break;
        default:
            return 0;
            break;
    }
}
```

There are two sections — the first displays the deposits and the second displays the withdrawals. Each section must have a section header that identifies the contents to the user (see Listing 1-38).

LISTING 1-38: The tableView:titleForHeaderInSection method

```
// Customize the Header Titles of the table view.
- (NSString *)tableView:(UITableView *)tableView
                    titleForHeaderInSection:(NSInteger)section {
    switch (section) {
        case 0:
            if([[self deposits] count] > 0) {
                return @"Deposits";
            } else {
                return @"No Deposits";
            }
            break;
        case 1:
            if([[self withdrawals] count] > 0) {
                return @"Withdrawals";
            } else {
                return @"No Withdrawals";
            }
            break;
        default:
            return @"";
            break;
    }
}
```

To display each detail row, the table view's section and row are checked. The section identifies which array to use, deposits or withdrawals; and the row identifies the element in the selected array to retrieve. Because this table view is for display only, you ignore any taps in the table view cells.

Listing 1-39 shows the `tableView:cellForRowAtIndexPath` method.

LISTING 1-39: The tableView:cellForRowAtIndexPath method

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
                        cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";
    NSString *cellText = @"";

    UITableViewCell *cell =
            [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
```

```
                initWithStyle:UITableViewCellStyleDefault
                        reuseIdentifier:CellIdentifier] autorelease];
        [cell setSelectionStyle:UITableViewCellSelectionStyleNone];
    }

    // Configure the cell...
    switch ([indexPath section]) {
        case 0:
            cellText = [NSString stringWithFormat:@"Deposit #%d = %@",
                                ([indexPath row]+1), [[self deposits]
                                      objectAtIndex:[indexPath row]]];
            break;
        case 1:
            cellText = [NSString stringWithFormat:@"Withdrawal #%d = %@",
                                ([indexPath row]+1), [[self withdrawals]
                                      objectAtIndex:[indexPath row]]];
            break;
        default:
            break;
    }
    // Configure the cell.
    [[cell textLabel]setText: cellText];

    return cell;
}
```

Listing 1-40 shows the complete `ThirdViewController.m` implementation.

Available for download on Wrox.com

LISTING 1-40: The complete ThirdViewController.m file (Chapter1/Tabbar-iPhone/Classes/ThirdViewController.m)

```
#import "ThirdViewController.h"
#import "PropertyList.h"


@implementation ThirdViewController

@synthesize detailTableView;
@synthesize balance;
@synthesize deposits;
@synthesize withdrawals;

#pragma mark -
#pragma mark View lifecycle

- (void)viewDidLoad {
    [super viewDidLoad];
}

- (void)viewWillAppear:(BOOL)animated {
    NSDictionary *d = [PropertyList readFromArchive:@"Data"];
    if(d != nil) {
        NSDictionary *localItems = [d objectForKey:@"items"];

        [self setBalance:[d objectForKey:@"balance"]];
        [self setDeposits:[localItems objectForKey:@"deposits"]];
        [self setWithdrawals:[localItems objectForKey:@"withdrawals"]];
    } else {
        [self setBalance:[NSNumber numberWithDouble:0.0]];
        [self setDeposits:[NSArray array]];
        [self setWithdrawals:[NSArray array]];
```

```
    }
    [[self detailTableView] reloadData];
    [super viewWillAppear:animated];
}

#pragma mark -
#pragma mark Table view data source

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView {
    // Return the number of sections.
    return 2;
}

- (NSInteger)tableView:(UITableView *)tableView
                        numberOfRowsInSection:(NSInteger)section {
    // Return the number of rows in the section.
    switch (section) {
        case 0:
            return [[self deposits] count];
            break;
        case 1:
            return [[self withdrawals] count];
            break;
        default:
            return 0;
            break;
    }
}

// Customize the Header Titles of the table view.
- (NSString *)tableView:(UITableView *)tableView
                      titleForHeaderInSection:(NSInteger)section {
    switch (section) {
        case 0:
            if([[self deposits] count] > 0) {
                return @"Deposits";
            } else {
                return @"No Deposits";
            }
            break;
        case 1:
            if([[self withdrawals] count] > 0) {
                return @"Withdrawals";
            } else {
                return @"No Withdrawals";
            }
            break;
        default:
            return @"";
            break;
    }
}

// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
                          cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";
    NSString *cellText = @"";

    UITableViewCell *cell =
            [tableView dequeueReusableCellWithIdentifier:CellIdentifier];
```

```
    if (cell == nil) {
        cell = [[[UITableViewCell alloc]
                    initWithStyle:UITableViewCellStyleDefault
                           reuseIdentifier:CellIdentifier] autorelease];
        [cell setSelectionStyle:UITableViewCellSelectionStyleNone];
    }

    // Configure the cell...
    switch ([indexPath section]) {
        case 0:
            cellText = [NSString stringWithFormat:@"Deposit #%d = %@",
                        ([indexPath row]+1), [[self deposits]
                                  objectAtIndex:[indexPath row]]];
            break;
        case 1:
            cellText = [NSString stringWithFormat:@"Withdrawal #%d = %@",
                        ([indexPath row]+1), [[self withdrawals]
                                  objectAtIndex:[indexPath row]]];
            break;
        default:
            break;
    }
    // Configure the cell.
    [[cell textLabel]setText: cellText];

    return cell;
}


#pragma mark -
#pragma mark Table view delegate

- (void)tableView:(UITableView *)tableView
                          didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
}


#pragma mark -
#pragma mark Memory management

- (void)didReceiveMemoryWarning {
    [super didReceiveMemoryWarning];
}

- (void)viewDidUnload {
    // Relinquish ownership of anything that can be recreated in
    // viewDidLoad or on demand.
    // For example: self.myOutlet = nil;
    [self setBalance:nil];
    [self setDeposits:nil];
    [self setWithdrawals:nil];
}

- (void)dealloc {
    [detailTableView release];
    [balance release];
    [deposits release];
    [withdrawals release];
    [super dealloc];
}

@end
```

Transaction.h Modifications to the Template

The `Transaction` class simply stores the balance, deposit, and withdrawal arrays in a `NSDictionary`, `items`. To be stored in a plist file, the `NSCoding` protocol must be implemented. The two values being stored are the balance and the transaction items.

- `balance`
- `items`

As shown in Listing 1-41, you must now define the properties for these variables in order to get and set their values.

LISTING 1-41: The complete Transaction.h file (Chapter1/Tabbar-iPhone/Classes/Transaction.h)

```
#import <Foundation/Foundation.h>


@interface Transaction : NSObject <NSCoding> {
    NSNumber *balance;
    NSDictionary *items;
}

@property (nonatomic, retain) NSNumber *balance;
@property (nonatomic, retain) NSDictionary *items;

@end
```

Transaction.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `Transaction.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-42).

LISTING 1-42: Addition of @synthesize

```
#import "Transaction.h"


@implementation Transaction

@synthesize balance;
@synthesize items;
```

The `encodeWithCoder` method defines the order in which the data will be stored, and the associated `initWithCoder` method decodes the values when they are retrieved from the file, and initializes the object with the stored values (see Listing 1-43).

LISTING 1-43: The encodeWithCoder and initWithCoder methods

```
#pragma mark -
#pragma mark NSCoder methods

- (void)encodeWithCoder:(NSCoder *)coder {
    [coder encodeObject:[self balance] forKey:@"balance"];
    [coder encodeObject:[self items] forKey:@"items"];
}

- (id)initWithCoder:(NSCoder *)coder {
```

```
    if (self = [super init]) {
        [self setBalance:[coder decodeObjectForKey:@"balance"]];
        [self setItems:[coder decodeObjectForKey:@"items"]];
    }
    return self;
}
```

Listing 1-44 shows the complete `Transaction.m` implementation.

LISTING 1-44: The complete Transaction.m file (Chapter1/Tabbar-iPhone/Classes/Transaction.m)

```
#import "Transaction.h"


@implementation Transaction

@synthesize balance;
@synthesize items;

#pragma mark -
#pragma mark NSCoder methods

- (void)encodeWithCoder:(NSCoder *)coder {
    [coder encodeObject:[self balance] forKey:@"balance"];
    [coder encodeObject:[self items] forKey:@"items"];
}

- (id)initWithCoder:(NSCoder *)coder {
    if (self = [super init]) {
        [self setBalance:[coder decodeObjectForKey:@"balance"]];
        [self setItems:[coder decodeObjectForKey:@"items"]];
    }
    return self;
}


@end
```

PropertyList.h Modifications to the Template

The `PropertyList` class provides factory methods that will retrieve the `Data.plist` file as well as store the current values after each transaction. It has one variable to hold the data:

- `pList`

You must now define the properties for this variable in order to get and set its value (see Listing 1-45).

It also supplies the following two factory methods for data processing:

- `readFromArchive`
- `writeToArchive`

LISTING 1-45: The complete PropertyList.h file (Chapter1/Tabbar-iPhone/Classes/PropertyList.h)

```
#import <Foundation/Foundation.h>


@interface PropertyList : NSObject {
    NSDictionary *pList;
```

```
}

@property (nonatomic, retain) NSDictionary *pList;

+ (NSDictionary *)readFromArchive:(NSString *)aFileName;
+ (BOOL)writeToArchive:(NSString *)aFileName fromDictionary:(NSDictionary *)aDict;

@end
```

PropertyList.m Modifications to the Template

Now that the header file has been updated to define the additions to the template, it is time to modify the `PropertyList.m` template.

For each of the view properties that were declared, you must match them with `@synthesize` (see Listing 1-46).

LISTING 1-46: Addition of @synthesize

```
#import "PropertyList.h"


@implementation PropertyList

@synthesize pList;
```

Listing 1-47 shows the complete `PropertyList.m` implementation.

Available for download on Wrox.com

LISTING 1-47: The complete PropertyList.m file (Chapter1/Tabbar-iPhone/Classes/PropertyList.m)

```
#import "PropertyList.h"


@implementation PropertyList

@synthesize pList;

+ (NSDictionary *)readFromArchive:(NSString *)aFileName {
    NSDictionary *result = nil;

    NSString *fname = [NSString stringWithFormat:@"%@.plist", aFileName];
    NSString *rootPath =
                [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                NSUserDomainMask, YES) objectAtIndex:0];
    NSString *bundlePath = [rootPath stringByAppendingPathComponent:fname];

    NSData *data = [NSData dataWithContentsOfFile:bundlePath];
    if(data != nil) {
        result = [NSKeyedUnarchiver unarchiveObjectWithData:data];
    }
        return result;
}

+ (BOOL)writeToArchive:(NSString *)aFileName
                                fromDictionary:(NSDictionary *)aDict {
    NSString *fname = [NSString stringWithFormat:@"%@.plist", aFileName];
    NSString *rootPath =
                [NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
                                NSUserDomainMask, YES) objectAtIndex:0];
```

```
    NSString *bundlePath = [rootPath stringByAppendingPathComponent:fname];

    NSData *data = [NSKeyedArchiver archivedDataWithRootObject:aDict];

    return [data writeToFile:bundlePath atomically:YES];
}

- (void)dealloc {
    [pList release];
    [super dealloc];
}

@end
```

Test Your Application

Now that you have everything completed, choose the Simulator from the Xcode panel and click Run and Build to try out your app. It should give you the results described at the beginning of the "A Simple Tab Bar" section.

SUMMARY

This chapter demonstrated three types of techniques used for navigating data on the iPhone. Navigation bars enable the use of simple taps either to traverse down a hierarchy of data to dig deeper or to return up. Toolbars allow several separate tasks to be performed on the same data within the current context, in this case the balance in the banking account. Tab bars work with common data and allow, through different views, manipulation of that data.

The choice of which navigation view to use in your application depends on how your data needs to be presented and/or modified.