Developing iOS 5 Applications

Laura J. White & Janusz Chudzynski

ePress+

iBooks Author

Copyright

About the Authors

Laura J. White *is an Assistant Professor in the Computer Science Department, and Program Director for Software Engineering at the University of West Florida. She has been teaching at the University of West Florida since 1993. She earned a B.S. in Computer Engineering from the University of New Mexico in 1984 and a M.S. in Computer Science from the Naval Postgraduate School in 1989, and a Ph.D. in Instructional Design for Online Learning at Capella University in 2006. She is a retired Surface Warfare Officer from the U.S. Navy. Laura has coauthored a book chapter on assessment, and has written numerous conference and journal publications regarding teaching programming, online learning, and software oriented architecture. She attended the Worldwide Developers Conference (WWDC) in 2010 and has been enthusiastically developing apps and teaching iPhone programming at the University of West Florida ever since.*

Janusz Chudzynski *is a Software Engineer and iPhone Programmer for the Academic Technology Center at the University of West Florida. He has earned a M.S. in Computer Science at the University of West Florida, and a M.S. in Management and Marketing with a minor in Information Systems from Warsaw University Technology in 2006. Janusz started learning iOS programming on his own and developed a few apps while he was a student, then received an Apple Scholarship to attend the Worldwide Developers Conference (WWDC) in 2009. Since then he has been enthusiastically developing apps and has taught several iPhone Programming courses for the University of West Florida.*

Dedication

Many people touch our lives as we journey through life.
We dedicate this book to a few of those who have been most influential in our lives.

Leszek and Iwona Chudzynski, Janusz's parents who showed him how to move forward in life despite all adversities, and without whose support Janusz would not be in the place that he is now.

Jiyeon Song, a very special person in Janusz's life.

In memory of Leslie A. White, Laura's twin, who was an extremely kind, generous, and creative person, with an adventurous spirit, who loved music, cats, and games. And, in memory of Mona L. White, Laura's mother who is greatly missed.

Barbara S. White, Laura's aunt who courage and grace is unmatched, and who has always provided love and support to Laura in times of joy and sorrow.

Laura's close friends, Ruth Edwards and Marcia Holland, and many friends and colleagues at the University of West Florida--especially those who regularly attend Saturday afternoon clubhouse meetings.

Preface

Since the App Store was launched in 2008, millions of apps have been created and distributed by iOS developers for the iPod Touch, the iPhone, and the iPad. This book is intended to get new iOS developers started creating apps of their own for the App Store.

The basic concepts needed to get started developing apps are presented within the context of three App development projects. The authors have developed many apps that are available for download from the iTunes App Store, and it is their belief that learning to develop apps within the context of projects will provide a satisfying and rewarding learning experience. Readers are encouraged to review relevant resources provided at Apple's developer website to deepen their understanding of the concepts integrated into each project. A companion website is available at http://www.epressplus.com/2012/01/21/developing-ios5-applications/ for the download of content needed for these projects and for support related to this book.

Chapter 1

Introduction to iOS Programming

```
//
//  Developing iOS 5 Applications
//
//  Created by Laura J. White and Janusz Chudzynski
//  Copyright (c) 2012. All rights reserved.
//

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISwitch *iPhone;
@property (weak, nonatomic) IBOutlet UISlider *iPad;

- (IBAction)learniOS5Programming:(id)sender;

@end
```

Introduction to iOS Programming

Concepts emphasized in this chapter:

- *History of iPhone/iPad programming*
- *Features and characteristics of the iPhone and the iPad*
- *Developing apps for the iPhone and the iPad*
- *The Objective-C programming language*
- *Overview of projects in this book*
- *Deploying apps*

History of iPhone/iPad Programming

The first iPhone was introduced January 9, 2007, and then released June 29, 2007. This iPhone became known as the original iPhone. The iPhone was the first smart phone with a touch screen display and integrated message texting, email, music, camera, photos, video, and internet access, as well as third-party applications referred to as apps. The next version of the iPhone was the 3G, released in July 11, 2008. The iPhone 3GS was released in June 19, 2009. The fourth generation of the iPhone was the iPhone 4 that was released on June 24, 2010. The iPhone 4S is being released on October 14, 2011. The iPhone has become so popular that over 1.7 million iPhone 4 devices were sold over the first three days after its release.

In November 2009, Apple announced that there were 1 million apps created by developers available in the App store and at that time, over 2 billion apps had been downloaded from the App store. Only apps that have been reviewed and approved by Apple are made available in the App store. The Apple review and approval policy is somewhat controversial. The advantage is that Apple is able to maintain quality control over what is made available to users, the disadvantage is that developers must comply with standards that some consider constrain creative freedom. On January 27, 2010 the iPad was introduced. It was then released April 3, 2010. The iPad was not Apple's first tablet computer, however it was the first to use the same multitouch OS that was already hugely popular by users of iPods and iPhones. The iPad released with a 9.7 inch LCD display with fingerprint

and scratch resistant glass. There were 300,000 iPads sold on the first day of its release, and 80 days later on June 21, 2010 3,000,000 iPads had been sold.

By, April 2010 when Apple announced the release of the iPad, 50 million iPhones and 35 million iPod Touches had been sold to date. These numbers indicate that many programmers are successfully developing and deploying apps. This book is intended to help those with the desire to join the ranks of iPhone and iPad developers by learning the essential concepts and developing the skills necessary to successfully develop and deploy iPhone and iPad apps.

Features and Characteristics of the iPhone and the iPad

There are many essential features and characteristics of iPhone and iPad apps. Some characteristics are required and must be satisfied in every app that is made available in the App store. The iPhone Human Interface Guidelines provides fundamental human interface design principles that should be incorporated into all iPhone apps and describes the user interface components that can be used in the development of an app and provides guidance in using these components effectively.

The iPad Human Interface Guidelines generally supplement, but occasionally supercede the guidelines provided by the iPhone Human Interface Guidelines for apps that are specifically developed for iPads. These guidelines contain four chapters (a) Key iPad Features and Characteristics, (b) From iPhone Applications to iPad Applications, (c) iPad User Experience Guidelines, and (d) iPad UI Element Guidelines.

Device characteristics specific to the iPhone are that the iPhone has a compact screen size and has a default display orientation that is portrait orientation.

Device characteristics common to both the iPhone and the iPad that should be considered when developing apps are that (a) memory is limited, (b) people see one screen at a time, (c) people interact with one application at a time, (d) onscreen user help is minimal, (e) orientation can change, (f) applications respond to manual gestures rather than mouse clicks, (g) preferences are available in the Settings application, and (h) artwork has a standard bit depth and PNG format is recommended.

Device characteristics specific to the iPad are that (a) the iPad has a screen size of 1024 by 768 pixels, (b) there is no default orientation, (c) there are user interface elements available for the iPad that are not available for the iPhone such as split views and popup views, and (d) options are available to plug the iPad into an external keyboard.

Developing Apps for the iPhone and the iPad

The iPhone is driven by the iOS, Apple's mobile operating system that is also used on the iPod Touch and the iPad. The iOS is what provides the capability for users to interact with these devices through the multi-touch screen.

Third-party applications that can be installed and used on multi-touch screen devices running iOS are call apps. These apps are created by developers using Apple's iPhone software development kit (SDK). The iOS SDK includes Xcode, a collection of development tools. Within Xcode is the Xcode integrated development environment (IDE), Interface Builder, Objective-C, Simulators for iPhone and iPad devices, and Organizer.

The Xcode IDE provides a collection of tools that support the construction, editing, compiling, execution, and debugging of source code within a single window. The IDE also provides the capability to manage testing devices and packaging iPhone apps.

Interface Builder provides capabilities for designing and developing graphical user interfaces for applications that run on multi-touch screen devices. Xcode and Interface Builder are very tightly integrated; changes made in Interface Builder are automatically synchronized with the source code files in Xcode. Early development of apps required the use of Interface Builder and Xcode as two separate applications. Since the release of Xcode 4, Interface Builder has been integrated wholly within Xcode to the extent that it is virtually transparent to new developers.

Objective-C is an object-oriented programming language that is a superset of the standard C programming language. Objective-C is extremely fast and powerful with a flexible dynamic class system. The Cocoa and Cocoa Touch frameworks with high-level application programming interfaces (APIs).

The simulators run iPhone/iPad apps in the Mac OS environment so that developers can see how apps will appear and perform on a device from within the development environment without having to actually install apps on devices while they are being developed. The iPhone simulator can be launched by clicking on the Build and Run icon from within Xcode, the simulator will appear similar to that shown in Figure 1.

Figure 1 iPhone Simulator

The iPad simulator provides the developer with the capability to build and run iPad applications in the Mac OS environment. When building and running an iPad application within Xcode, the app will appear similar to Figure 2.

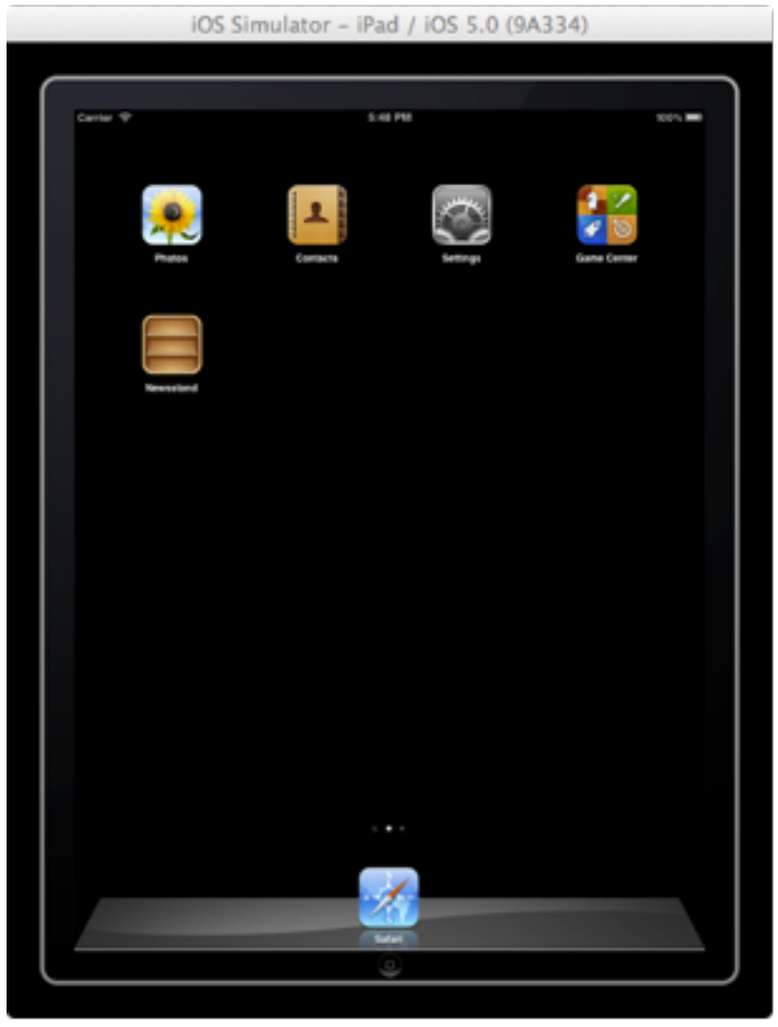

Figure 2 iPad Simulator

The Organizer provides the means to keep track of different iPhone devices and certificates. Organizer is also used when provisioning apps for the iTunes App store.

A wealth of resources and support for the development of iPhone apps is available from Apple in the iPhone Dev Center at http://developer.apple.com/devcenter/ios. The iOS Dev Center contains *Getting Started* documents and videos, Coding How-To's, SDK download links, and Featured Content based on current developer needs.

Apps for the iPhone and the iPad can be developed several ways. They can be developed entirely within Xcode—Apple's integrated development environment using the Objective-C programming language, or they can be developed using Xcode and Interface Builder in tandem where the user interface elements for an app are developed in Interface Builder and then defined inside Xcode. Generally, a new project is created within Xcode. When creating a new application the developer specifies the product as either an iPhone or an iPad application. Apps developed for the iPhone will run on the iPad in compatibility mode but will not provide the rich user experience that apps specifically designed for the iPad will generally provide. Apps developed for the iPhone will not fill the iPad screen since the size of the iPhone screen is 480 by 320 pixels, and the size of the iPad screen is 1024 by 768 pixels.

Apple has incorporated a feature on the iPad that provides the user with the option of doubling the size of the app if it detects that the resolution of the app is that of an iPhone.

It is also possible to run an iPad app in the iPhone simulator but the display of the app will be cut off since the iPhone screen is smaller than the iPad screen.

The Objective-C Programming Language

The Objective-C programming language is an objective-oriented programming language that underlies all iOS and Mac OS applications. The initial development of this language began in the early 1980s as a superset of the C programming language and has evolved over time. The Objective-C programming language has extensive libraries that support the development of high quality applications. To developed sophisticated applications, iOS developers must become familiar with the basics of the Objective-C programming language. Developers with a prior knowledge of C, C++, or Java will recognize many of the basic features of the Objective-C programming language.

Variables and Objects

Objective-C contains both primitive data types and object-oriented classes that can be used to create primitive variables and objects. Some of the most commonly used primitive types are:

int	Variables used to store/manipulate whole numbers
float	Variables used to store/manipulate real numbers
BOOL	Variables that only have to possible values such as true/false or YES/NO

A few examples of declarations for these primitive types are as follows:

```
int WholeNumber;
float realNumber;
BOOL isHappy;
```

Objects in Objective-C—as with other object oriented programming languages—are instances of classes that encapsulate attributes and behavior through methods defined in the classes. In Objective-C objects are referenced through pointers so when declared the asterisk is used to indicate that the variable is a pointer to an object of a particular class. The following declaration creates a pointer named myEmployee to an instance of the Employee class.

```
Employee *myEmployee;
```

Once the pointer is created, memory must be allocated for the object and its contents initialized, as in the following statement, which allocates and initializes memory for the instance of Employee that we just declared.

```
myEmployee=[[Employee alloc] init];
```

Operators

Objective-C contains the operators: assignment, arithmetic, modulo, combined assignment with arithmetic, increment, decrement, bitwise Boolean, and logical Boolean operators that are found in C, C++, and Java.

Decisions

Decision constructs in Objective-C include those common to C, C++, and Java. These constructs also use the use basically the same syntax as in the other languages.

Iteration

Objective-C contains four basic constructs for iteration. The while loop is intended for situations where iteration is needed zero or more times. The do while loop is intended for situations where iteration is needed one or more times. The for loop is intended for situations where iteration is intended zero or more times and the processing of an element is based on some increments of a variable. These iteration constructs are similar to those provided n C, C++, and Java.

Methods

This section will cover basic information about methods. Let's have a look at their syntax of a method declaration as shown in Figure 5. The + or – sign at the front of the definition indicates the type of the method. A + indicates that the method is a factory or class method and a – indicates that the method is an instance method. The type indicator is followed by a return type within parentheses. The return type in Figure 5 is void which means that the method does not return any value. The return type is followed by the name of the method. The name of the method in Figure 1.5 is insertObject. If the method does not require any arguments then the name of the method would be followed by a semicolon to denote the end of the method definition. If the method does require arguments as is shown in Figure 1.5 then the name of the method is followed by a colon and then an argument list of one or more arguments. Each argument in the list consists of an argument type in parenthesis and a name of the argument. A semicolon at the end of the list indicates the end of the method definition.

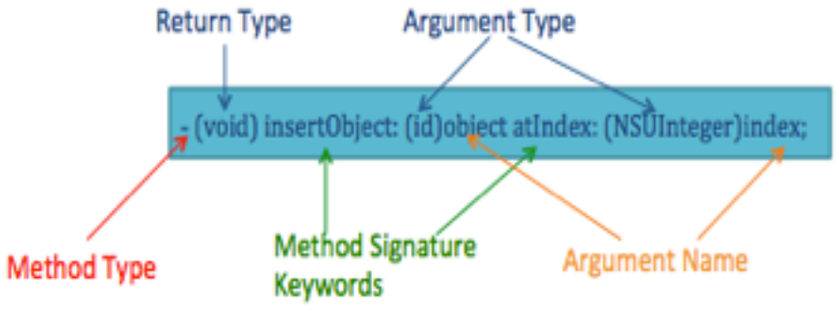

Figure 3 Method Declaration

Now let's look at the method type more closely. A class method is one that performs some operation on the class itself, such as creating a new instance of the class. The stringWithFormat method definition follows is an example of a class method.

```
+(NSString *) stringWithFormat;
```

An instance method is one that performs some operation on a particular instance of a class, such as setting its value, retrieving its value, displaying its value. The drive method definition that follows is an example of an instance method. Notice that the name of the class is not evident in the definition.

```
-(void) drive;
```

Next we look at invoking methods in Objective-C. Objective-C uses a message paradigm for invoking methods that stems from the Smalltalk programming language where a method is invoked by sending a message to it. The syntax for sending a message to a method is to enclose a class name or instance name followed by the name of the method within square brackets. For example, imagine that the class Ford has a class method named create that returns an object of class id. The method definition is +(id) create; and we send a message to it with [Ford create];

The Ford class can also have instance methods that are all named drive but with different argument lists.

```
-(void) drive;
-(void) drive: (float) speed;
-(BOOL) drive: (float) speed from:(NSString *) city1 to (NSString *) city2;
```

We can invoke these instance methods as follows.

```
[ford  drive];
[ford  drive:    50.0];
[ford  drive:    50.0 from: Destin to: Pensacola];
```

Creating Objects

Since we have two different types of methods – class and instance – there are two different ways to create objects which are instances of classes. Figure 6 shows how a new object or instance of the NSString class named string can be created. In both cases we start creating an object by providing the name of the class that the object will belong to, NSString and then the name of the object preceded with an asterisk that denotes that the name is actually a reference or pointer to the object. If we wish to create the object using a class method then we enclose the name of the class and the name of the object within square brackets and then allocation and initialization of the object is handled by the class. If we use instance

methods to create an object we enclose the class and alloc method within the square brackets, and then that message is enclosed within another message that invokes the init method for the class on the object just allocated.

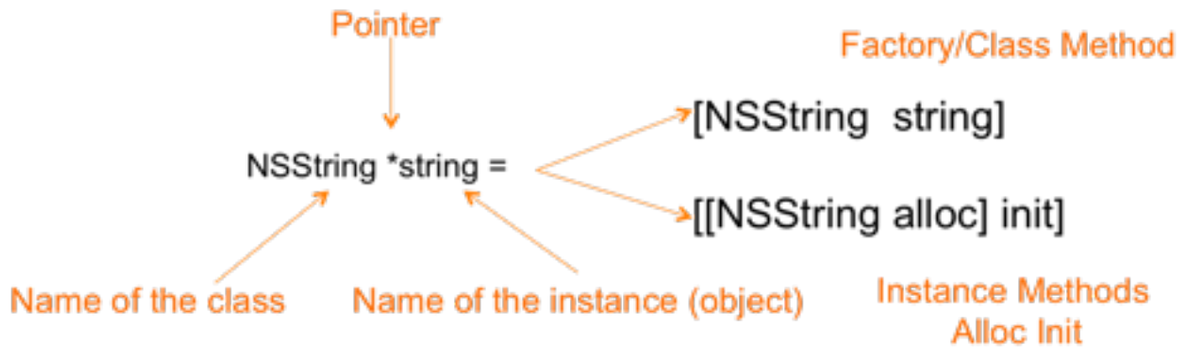

Figure 4 Using Instance and Class Methods to Create an Object

Creating an object using a class method is easier but the trade-off is that the programmer will have less control over the scope of the variable with regard to memory management than if the object is created using the alloc and init instance methods.

Objective-C Classes

In Objective-C classes are structured so that the interface for the class is defined within a .h file and the implementation for the class within a .m file. The interface contains declarations and definitions of variables and methods and links the class to its superclass. The implementation contains the method implementations--that is methods bodies with code to perform the functionality of the method. For example, the Rectangle class would be defined within the Rectangle.h file and implemented within the Rectangle.m file.

The interface file must be *included* or *imported* in any source module that depends on the class interface. For example, the first two lines in the code sample that follows are comments, and the third line imports the interface for Factory inside the Ford class. Interface files can be imported using the keywords include or import. The advantage of using import is that this ensures that the interface isn't included more than once during compilation. This is especially beneficial in large applications that consist of many classes.

The next line in the code sample presents the new class name and links it to its superclass, so in this case the class name is Ford and the superclass is NSObject. The superclass defines the position of the new class in the inheritance hierarchy. Following the first part of the interface

directive are braces that enclose declarations of instance variables for the class that can be accessed anywhere within the scope of the class. Three methods for the class are defined next, and the last element in the interface file is the end directive.

```
// Ford.h
// Interface file for the Ford class

#import "Factory.h"

@interface Ford : NSObject {
    NSString *color;
    NSString *dealer;
    int vinNumber;
}

//method definitions
-(void) repaintCar:(NSString *) newColor;
-(void) moveToDealer:(NSString *) newDealer;
-(id) initWithColor:(NSString *) initColor;
@end
```

The implementation for a class is structured very much like its interface. The next code sample contains the implementation for the Ford class. It begins with an import directive for the Ford.h file. Implementation files must import their own interface files, and can also import or include other classes as necessary that were not already imported or included in this class's interface file. The next line of code is the implementation directive. Since it imported its own interface file it does not need to include declarations and definitions already provided there. Then the method bodies for the three methods declared in the Ford.h file follow.

```
// Ford.m
// Implementation file for the Ford class

#import "Ford.h"
@implementation Ford

-(void) repaintCar: (NSString *) newColor{
    // Code to implement this behavior
}

-(void) moveToDealer: (NSString *) newDealer{
```

```
    // Code to implement this behavior
}

-(id) initWithColor: (NSString *) initColor{
    self=[super init];
    if(self) {
        // Code to implement this behavior
    }
    return self;
}

@end
```

Special Keywords

Objective-C contains some special keywords. The most common are self, super, and id. The keyword self searches for a method implementation in the usual manner, starting in the current class. The keyword super starts the search for a method implementation in the superclass of the class where super appears. The id keyword can be used as a generic type. For example if we have a method that returns a type of id, then any object can be returned. If an method argument has a type of id then any object can be passed to the method.

Format Specifiers

Objective-C contains the means to format strings and numbers through the combination of class methods and format specifiers. Consider the declaration for a string.

```
NSString *example = [NSString stringWithFormat:@"exampleText"];
```

In this example no special formatting of the string is needed, so even though we used the stringWithFormat method to create the string we simply put the text inside the double quotes. However, when we want to specify some formatting we can use various format specifiers—the three most common are:

%f for floats, and doubles

%d for integers

%@ for strings and other objects.

To use format specifiers we add them inside the double quotes, and then type a comma after the double quotes and then specify the variable to be formatted. To create a string that stores exampleText followed by the value of an int variable named value, we should use the following statement that includes the format specifier for an integer and a reference to the variable value.

```
NSString *example1 = [NSString stringWithFormat:@"exampleText %d", value];
```

The following statement includes a format specifier to display a float variable rather than an int variable named value:

```
NSString *example2 = [NSString stringWithFormat:@"exampleText %f", value]
```

To specify the minimum number of digits that should be used to display a floating point variable, a number can be added to the format specifier. The following statement displays the variable value using at least 5 places which in which the decimal point counts as one of the places.

```
NSString *example3 = [NSString stringWithFormat:@"exampleText %5f", value];
```

To specify the maximum number of digits that should be displayed after the decimal point for a float variable a decimal point followed by a number can be added to the format specifier. The following statement displays the variable value using at least 5 places with exactly 2 numbers after the decimal point:

```
NSString *example4 = [NSString stringWithFormat:@"exampleText %5.2f", value];
```

To display a string variable object we can use the @ symbol to print an object. The following example contains a string literal and a string variable in the format specifier.

```
NSString *text1 = @"More Text";
NSString *example5 = [NSString stringWithFormat:@"Some text and  %@", text1];
```

We can use multiple format specifiers in a single statement, just remember to list the variables that are associated with each format specifier separated by commas.

```
int num1=0;
float num2=10.0;
NSString *example6 = [NSString stringWithFormat:@"The numbers are  %d and %4.1f", num1, num2;
```

Memory Management

One of the new features contained in Objective-C for iOS 5 is automatic reference counting (ARC). We will use ARC in our projects. However, developers working with iOS 4 projects still need to manually manage their memory allocations. Fortunately the rules are not very complicated. First, we need to understand object ownership. Programmers are responsible for managing memory for objects they "own". So when does the programmer become the owner of an object? The programmer owns an object if it is:

- Created using an instance method (alloc-init)
- Created using the keyword new
- Called with the retain message
- Called using the copy message

Proper memory management requires that programmers relinquish ownership of objects they own when they are finished with them. Ownership of objects is relinquished by sending a release message or an autorelease message to it. You should not relinquish ownership of an object you do not own. A few simple examples of both proper and improper memory management are contained in the code sample below.

```
-(void) memoryTest {
    NSString* mem1=[[NSString alloc]initWithString:@"Some string"];
    NSMutableArray* array1=[NSMutableArray arrayWithCapacity:0];
    [array addObject:mem1];

    //Do something with the array here…
    [array1 release];   // Incorrect since it was created with a class method
    [mem1 release];  // Correct
}
```

Now let's look at another example to see how to manage memory for variables that are declared inside the interface part of a class. For this example we will use the Ford class again. As shown in the interface below, the class contains declarations for three instance variables: color, dealer, and vinNumber, along with three methods.

```
// Ford.h
// Interface file for the Ford class
```

```
#import "Factory.h"
@interface Ford : NSObject {
    NSString *color;
    NSString *dealer;
    int vinNumber;
}

//method declarations
-(void) repaintCar:(NSString *) newColor;
-(void) moveToDealer:(NSString *) newDealer;
-(id) init;
@end
```

Now let's look at the implementation file for this class. In the init method in the Ford.m file, the initial values of dealer, color and vinNumber are set. Since we need these objects outside the scope of the init method that allocates and initializes them we shouldn't release them inside the init method. Instead, we release them in the dealloc method. The role of the dealloc method is to free an object's own memory, and dispose of any resources it holds, and to relinquish ownership of instance variables. The dealloc method is invoked automatically in iOS applications – it should not be invoked or called in the code. Notice that the instance variable color is released in the dealloc method and not the other instance variables. We don't need to release vinNumber because it is a primitive variable, and we do not need to release dealer because it was created through the use of a class method and memory management is handled by the system for instances created by class methods.

```
// Ford.m
// Implementation file for the Ford class

#import "Ford.h"
@implementation Ford

-(void)repaintCar:(NSString*) newColor{
    // Code to implement this behavior
}

-(void)moveToDealer:(NSString*) newDealer{
    // Code to implement this behavior
}

-(id)init{
    self=[super init];
```

```
    if(self) {
        color=[[NSString alloc]initWithString:@"WHITE"];
        vinNumber=0;
        dealer=[NSString stringWithFormat:@"Toyota"];
    }
    return self;
}

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

@end
```

This section has provided a brief overview of essential aspects of Objective-C. For further details of the language refer to a separate text or to online tutorials and programming guides for the Objective-C programming language.

Overview of Projects in this Book

This book presents an introduction to useful resources for app developers, and then three iPhone development projects. Each project embodies new iPhone programming concepts and provides the opportunity for the developer to implement new features for iPhone apps.

The first project, in chapter 2, is the development of a Temperature Conversion App, in which readers will learn the basics of creating a simple user interface using the UIKit framework, and provided a few basic methods to implement actions for the user interface elements.

The second project is the development of a Photo Puzzle App in chapters 3 and 4. This project further reinforces concepts learned in chapter 2 and also introduces the reader to basic concepts of the Core Graphics framework, animation, modal view controllers, and more complex underlying action than was used in the Temperature Conversion App.

The third project in this book is the development of the Show Me App in chapter 5 that uses the Map Kit in chapter 4 demonstrate basics of using maps within an app.

The fourth project in this book is the development of the Treats! App in chapters 6 and 7. This project includes concepts regarding the use of touch events, timers, picker views, and displaying high scores for a game by saving and restoring data in the app.

The fifth project demonstrates the use of storyboarding, a new feature in iOS 5, to graphically build an app that consists of multiple screens or view controllers.

Deploying Apps

After an iPhone or iPad app performs as desired in the Xcode simulator, the next step is to deploy the app to an actual device. There are several steps involved in this process. In order to deploy apps to personal devices for testing, a developer must have signed up for an iOS standard, enterprise, or university developer program. In order to deploy apps to the iTunes App Store, a developer must have signed up for an iOS standard program. Once a developer has membership in one of these programs the developer can access resources in the iOS member center to obtain certificates and provisioning profiles necessary to deploy apps to the iTunes App Store. The member center How-To documents provide step-by-step instructions for submitting an app to the iTunes App Store.

Summary

Since the App Store was launched in 2008, millions of apps have been created and distributed by iOS developers for the iPod Touch, the iPhone, and the iPad. As the capability of iOS devices increases, the demand for those devices and apps for those devices also increases. Developing iOS apps is no longer a niche market for programmers. As the demand for apps dramatically increases so does the number of business that comprehend the value for establishing a mobile presence and interaction with customers via apps.

Features of the various devices—iPod Touch, iPhone, and iPad—that run iOS applications have been described and the distinctions between the devices has been emphasized. A brief introduction to the general process of developing an app using Apple products including Xcode, Interface Builder, and simulators has also been presented. The next part of this chapter presented basic principles of Objective C programming which underlies all iOS applications. This was followed by a brief description of the app development projects that are contained in the remaining chapters of this book. The last part of this chapter provided a brief overview of the process for distributing apps so that once readers have developed their skills and created their own apps they will be able to get started with the process of deploying apps to the iTunes App Store.

Review Questions

1. What app development components are provided by an iOS SDK.

2. Describe high-level characteristics of the Objective-C programming language which is used for the development of iOS apps.

3. What are the two types of methods that are indicated by a + or a - sign at the beginning of a method declaration?

4. Describe how the messaging paradigm relates to methods in Objective-C.

5. What keyword represents a generic type?

6. Explain how the keywords self and super are used.

7. What is the format specifier for an object?

8. List the conditions under which the programmer is responsible for manually managing memory.

9. What is required for a developer to deploy an app to an iOS device or to the iTunes App Store.

Chapter 2

Temperature Converter App: User Interface Controls

iBooks Author

Temperature Converter App: User Interface Controls

Concepts emphasized in this chapter:

- *Adding iOS user interface elements: buttons, labels, text Fields, and events*
- *Delegation patterns*
- *Making connections from the user interface to the code*
- *Adding custom code to implement an application*

Introduction

The first iPhone project will consist of the development of an application that converts Centigrade temperatures to Fahrenheit temperatures, and also converts Fahrenheit temperatures to Centigrade temperatures. Interface builder integrated within Xcode, basic user interface controls, and Xcode will be used in the development of this project. A plan should always be established before starting any programming project. The basic elements of the plan will be described in detail in the following sections of this chapter. The plan for this project will be to:

1. Create a new project in Xcode
2. Add an image to the project
3. Add the necessary user interface elements to the View that represents the screen of an iOS device
4. Create outlets and actions for the user interface elements
5. Add custom code to implement the functionality of the user interface elements
6. Run and test the application

Development

The development of the Temperature Converter application will be conducted in several phases. First, we will create a new project. Then we will build the user interface. Then we will write the custom code to provide the desired functionality for the user interface elements.

Create a New Project in Xcode

Launch Xcode. Xcode is often located at Macintosh HD > Developer > Applications > Xcode. When Xcode launches, it presents a window that displays, as shown in Figure 1, four options in the left side of the window and displays the names of projects previously created in Xcode in a panel in the right side of the window. Three of the options ***Connect to a repository, Learn about using Xcode, Go to Apple's developer portal*** may be explored before starting this project, or can be revisited later. After saving a new project and exiting Xcode, work can be continued on a project by selecting it in the *Recent Projects* panel. To get started with our temperature project, click on **Create a new Xcode project.**

Figure 1 Xcode Window

Choose the Single View Application template as shown in Figure 2, then click on the Next button.

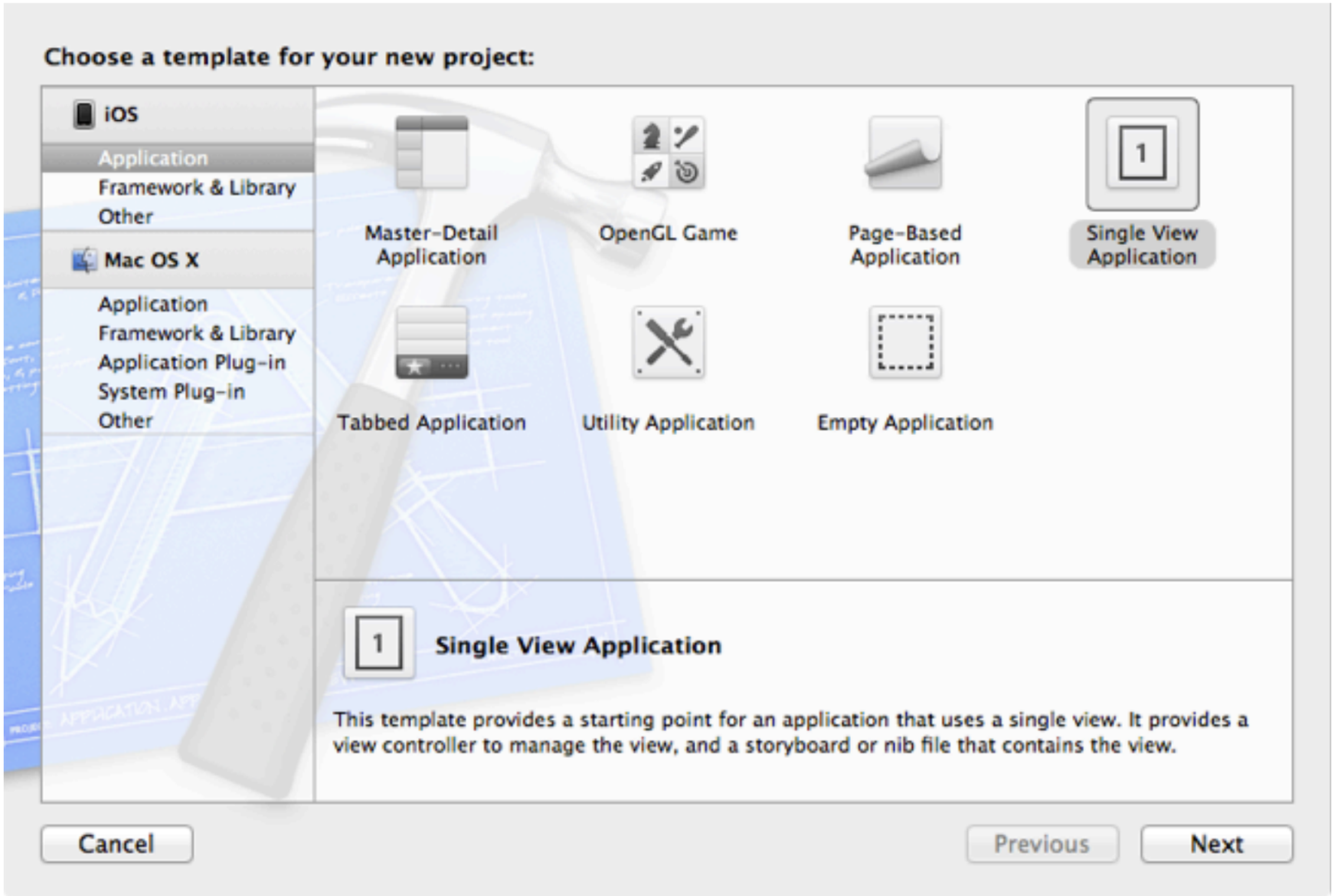

Figure 2 Creating a New Project – Selecting a Template

Enter *Converter* for the ***Product Name*** as shown in Figure 3. A different name can be used, but it will be easier to follow along with this project if you use the same name.

Enter *com* for the ***Company Identifier*** or leave it with the default value as shown in Figure 3.

Select Universal in the Device-Family menu.

Select the Use Automatic Reference Counting checkbox, but do not select the other checkboxes as shown in Figure 3. Click on the Next button.

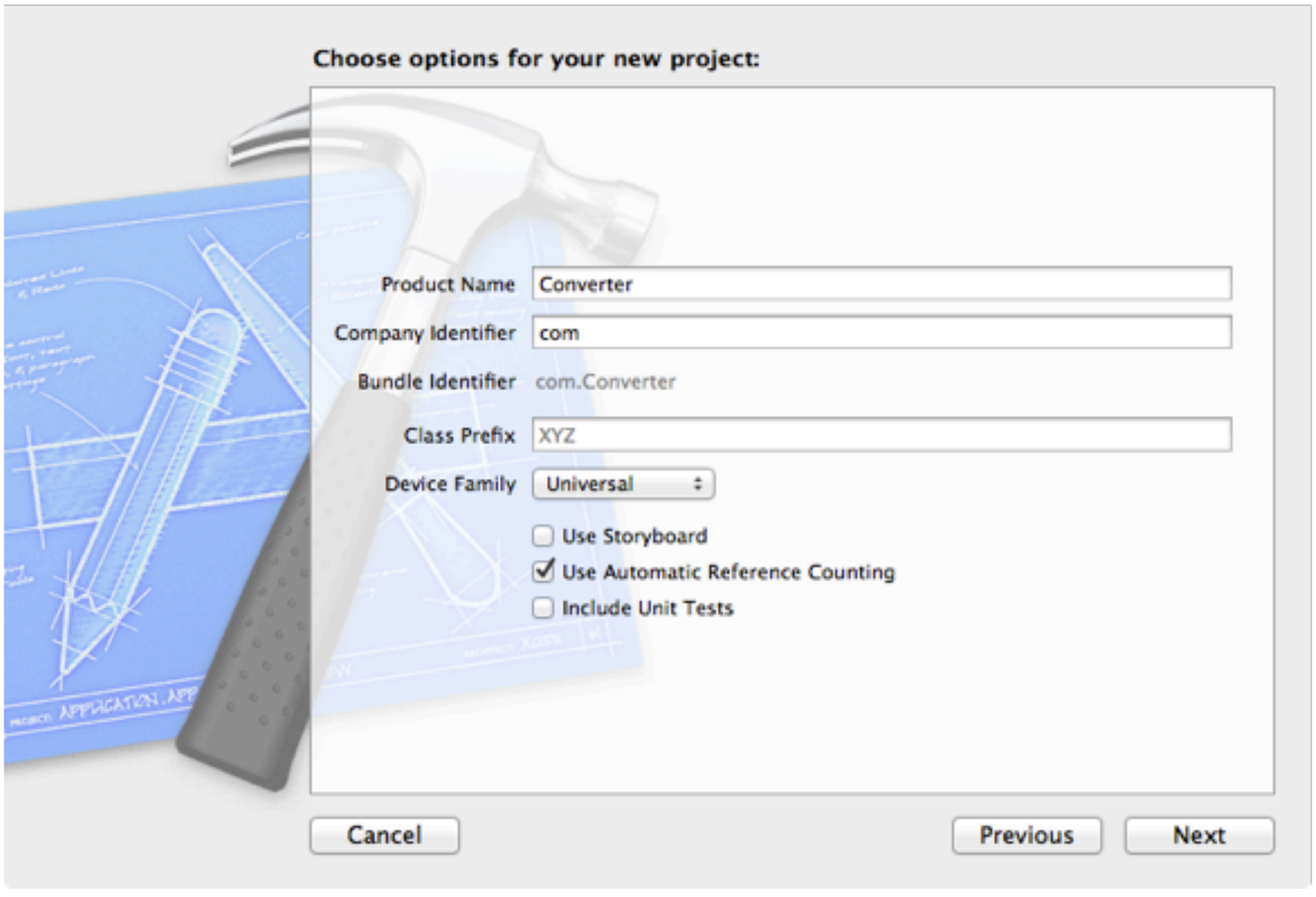

Figure 3 Creating a New Project – Naming the Project

Choose a location for your project as shown in Figure 4. Do not select the Source Control checkbox at the bottom of the window, then click on the ***Create*** button.

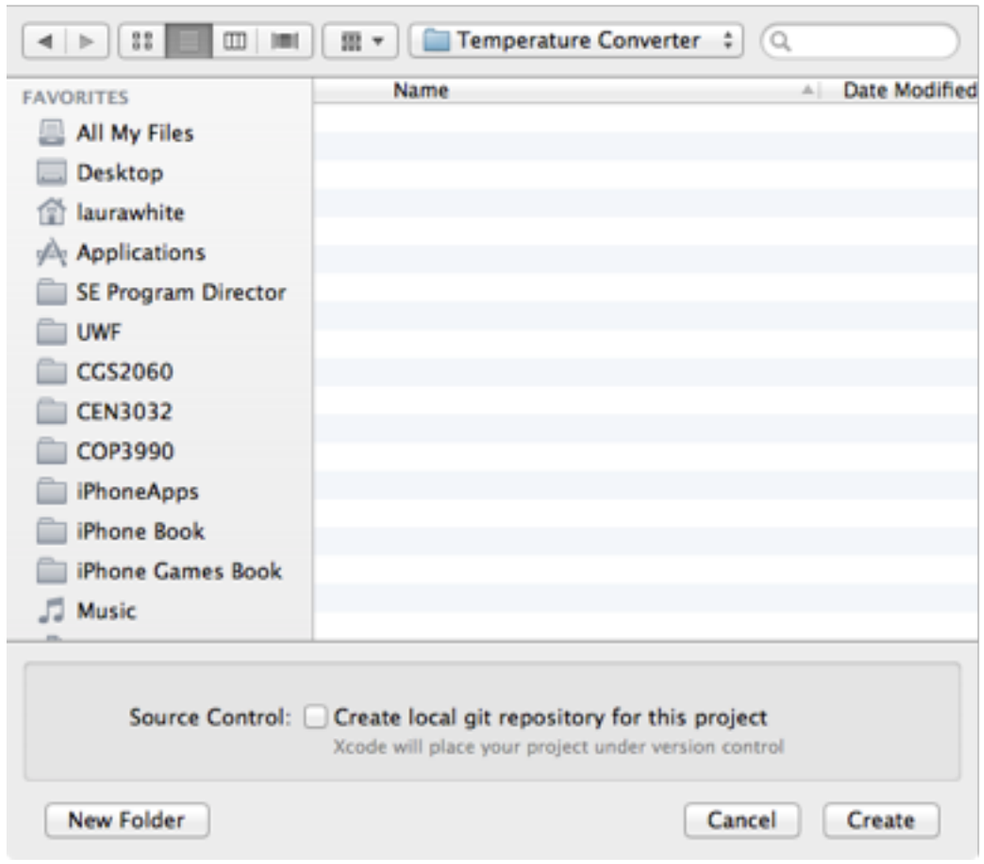

Figure 4 Creating a New Project – Selecting the Location for the Project

The Xcode Workspace Window

Once the project is created a single Xcode workspace window is launched as shown in Figure 5. Xcode fully integrates the Interface Builder design tool and the LLVM compiler into the Xcode Integrated Development Environment. It looks complex, but don't worry we will provide a short introduction of the main functions of Xcode.

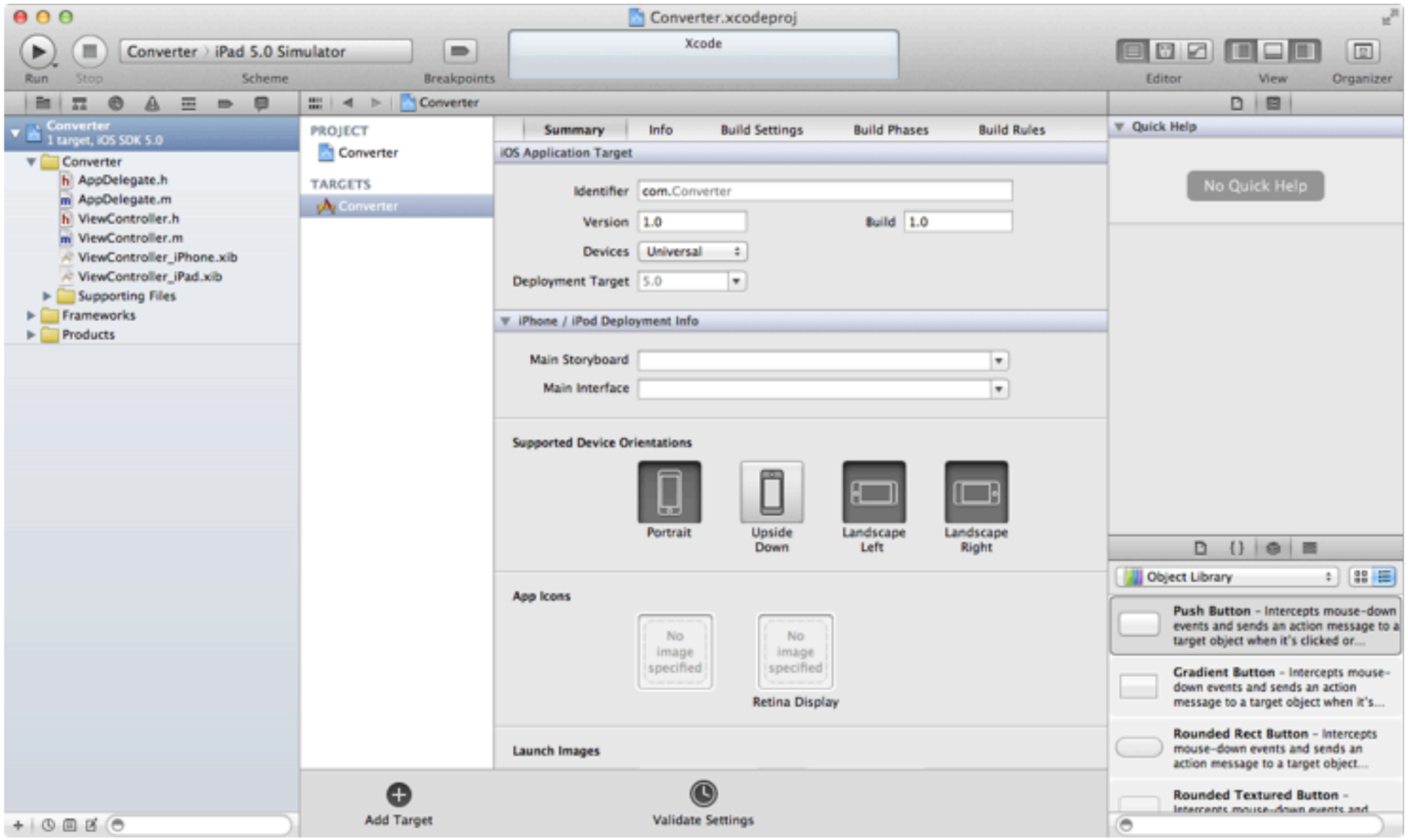

Figure 5 The Converter Project Inside the Xcode Workspace

Across the top of the Xcode workspace window is the Toolbar. Starting at the left of the Toolbar, the first things we see are the Run and Stop buttons that are used to run or stop our application inside the workspace. Notice the small down arrow associated with the Run button. There are five execute actions that can be performed in Xcode. These actions are Run, Test, Profile, Analyze, and Archive.

The next element in the Toolbar is the Scheme menu, as shown in Figure 6. In Xcode, the developer can select and create Schemes for their projects. A Scheme specifies what targets to build and what configurations to use when building a project.

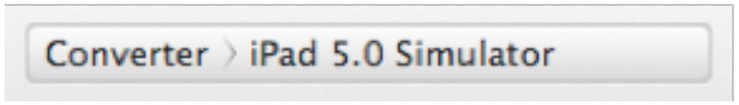

Figure 6 Xcode - Scheme Menu

The default schemes for an iOS project includes devices and simulators. When we click on the Scheme menu a pop-up window will appear with device options. Since we selected the Universal device family for our project, we will have options for an iOS Device, the iPad simulator, and the iPhone simulator as shown in Figure 7. A developer must be a member of one of Apple's iOS Developer Programs in order to run applications on an iOS device. Don't worry about the Update Simulators... option for now. In this chapter we will test our application in the iPhone 5.0 Simulator Environment so select iPhone 5.0 Simulator.

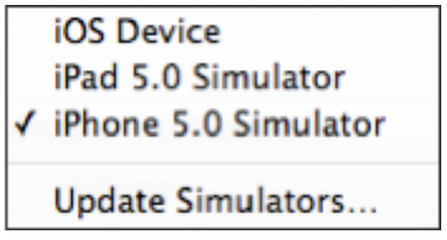

Figure 7 Choosing a Device to Build Inside Xcode

Notice the Breakpoints icon to the right of our scheme. It is used for debugging code. We are not going to use this option in this project. The center of the Toolbar contains a display error for messages related to actions performed in the workspace. Now look at the top right of the Toolbar. This is where the editor, view, and organizer options are located. It should appear similar to Figure 8.

Figure 8 View and Layout Options

The first group of Editor buttons in this part of the Toolbar are the editor selector buttons that are used to display different views of files within editor panes inside the Editor area of the Xcode workspace window. The first button selects the Standard editor which displays a file in a single editor pane. The second button selects the Assistant editor which is generally used to display a second file closely related to the file in the Standard editor within a second editor pane using the split editor pane. The third button selects Version editor which is generally used to display two versions of the same file within the split editor pane.

The next group of buttons are the View group used to select different areas of Xcode. The first button is used to select the Project's Navigator area on the left of the workspace window. The second button is used to display the Debug area at the bottom of the workspace window, and the third button is used to display the Utility area on the right side of the workspace window. Select the ViewController.h file in the Navigator area on the left. The Xcode workspace window should now appear similar to Figure 9.

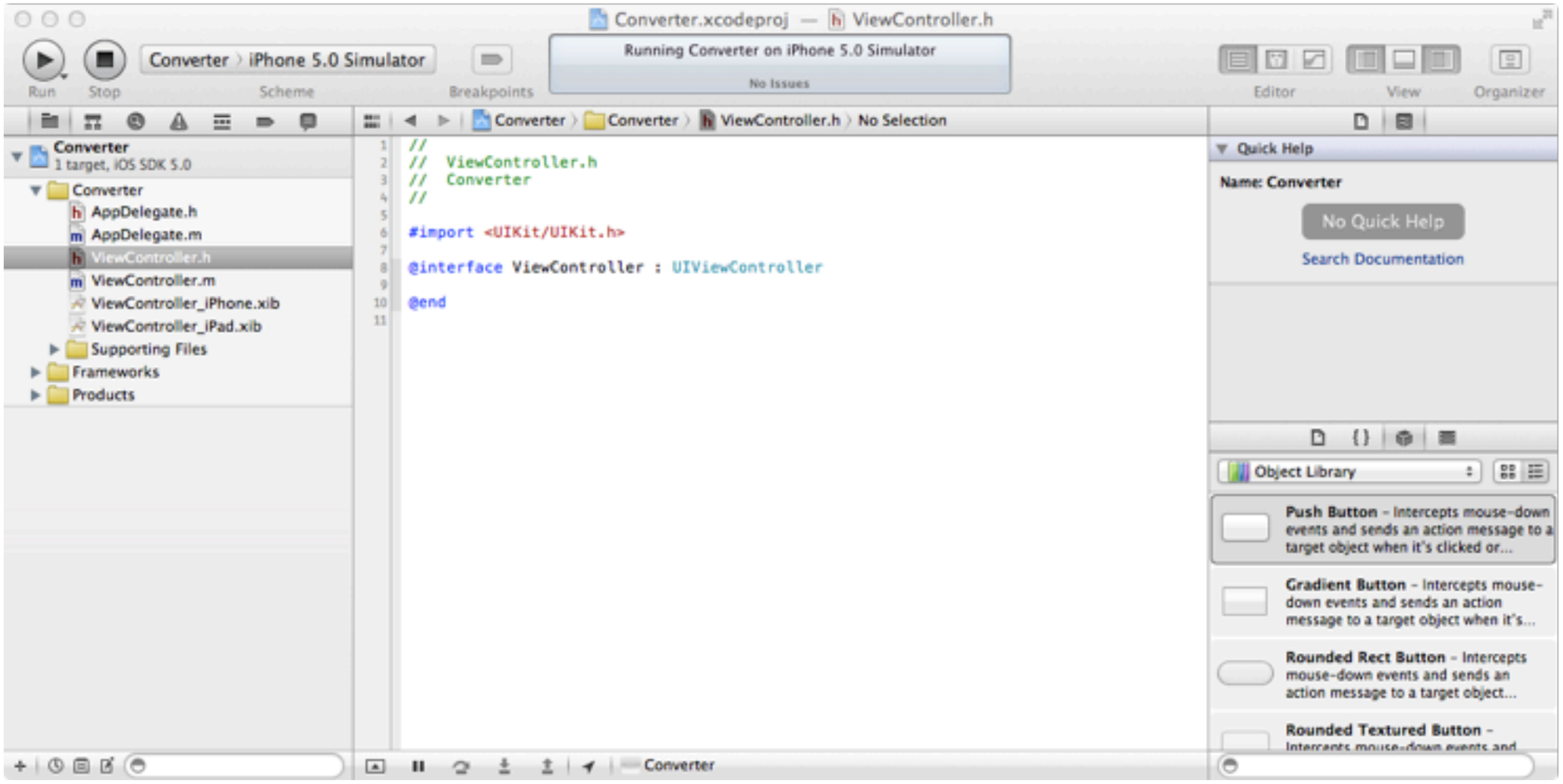

Figure 9 Xcode Workspace with a Header File in the Editor Area

It is important to understand how a project is structured inside Xcode so expand all of the folders in the project navigator area. The panel should appear similar to Figure 10.

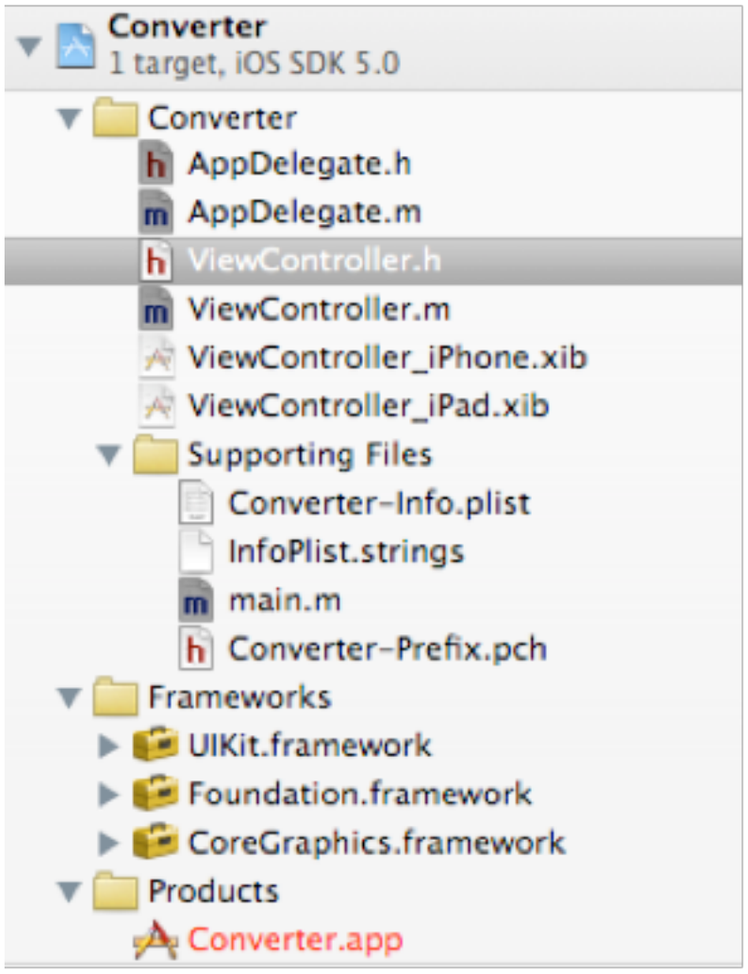

Figure 10 Project Navigator Area

All the files listed in the navigator area were created as a result of selecting the View Based Template when we created this project. The first two files listed in the Converter folder are AppDelegate.h and AppDelegate.m. All Objective-C classes are defined in two files. Just as in C, the .h files contain declarations and definitions of methods and variables, and the .m files contain the implementation of those methods.

Each iPhone project contains an AppDelegate class. Instead of subclassing and overriding methods for a complex object as is done in some object-oriented programming languages, in Objective-C we create objects from unmodified classes and then put our own custom code inside a delegate object that defines special characteristics for that object. As interesting events occur, the complex object sends messages to our delegate object. We use these messages to execute our custom code to implement the behavior we need. An AppDelegate class is generally responsible for handling critical system messages, such as moving to the background, suspending an application, and starting an application. The AppDelegate class sets the main view controller and the view that will be displayed on the screen. We will not modify the AppDelegate class for the project in this chapter.

The next four files in the navigator pane are the ViewController.h, ViewController.m, ViewController_iPhone.xib, and ViewController_iPad files. These files define a subclass of the UIViewController for our project. View controllers conform to an important concept in iOS programming, that uses the Model View Controller (MVC) pattern. This pattern provides a logical separation between that data that is used, and the view or user interface. The UIViewcontroller works as a mediator between the data and the user interface, and as such controls the operations used to process the data, and responds to changes and events that occur on the user interface. The MVC pattern is illustrated in the Figure 11.

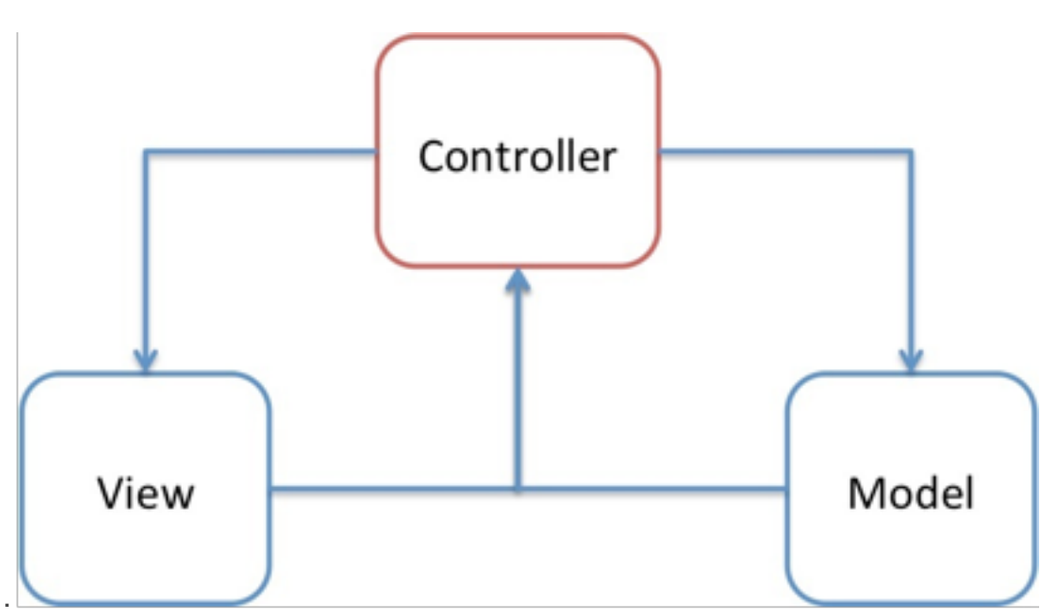

Figure 11 Model View Controller Pattern

There are numerous benefits of adopting the MVC pattern for iOS programming. These benefits include well-defined interfaces, reusability, and extensibility. How do we map the model in this figure to our application? The files ViewController.h and ViewController.m contain the code that defines the behavior for the application, and is where we will place our custom code later on in this chapter. These files are represented as the Controller in Figure 11. The ViewController.xib file stores the information about the user interface associated with the ViewController and is represented as the View in Figure 11. We will use this file to contain the user interface elements for our application. In applications that involve the use of data, that data represents the state of the application. This data is structured and stored within the application. This aspect of an application is represented by the model element in Figure 11. In simple applications this data can be embedded inside the view controller, which is what we will do in this chapter's project. In more complex applications the data might be managed within a database, plist files. xml files, or some other appropriate structure.

The next element in the Navigator pane is the Supporting Files folder which contains the following files:

- The Converter-Info.plist file contains the information about the application configuration. We will not make any changes to this file.
- The InfoPlist.strings file contains localized strings.
- The Converter–Prefix.pch file contains the header that is included in all source files for a project.
- The main.m file is responsible for starting the application. All iOS applications contain this file.

The next folder is the Frameworks folder which contains the iOS frameworks that are used within iPhone applications. By default all projects contain the (a) UIKit framework that is responsible for User Interface and Touch elements, (b) Foundation framework that contains basic data structures such as strings, and arrays, and (c) Core Graphics framework for drawing on the screen. Other frameworks may be added to a project by the developer as needed.

The last folder in our project is the Products folder that will contain the .app file that will contain the executable code for the application after it is built. We currently see the name we selected when we created this project with .app appended at the end.

Adding Resources to a Project

The developer may want to add additional resources to a project. In this section we will demonstrate how this is done, by adding an image to our project, using the following steps.

First, in the newly opened Xcode window, ctrl-click on the Project name at the top of the Navigator area in and then select ***New Group*** in the popup menu as shown in Figure 12. Then double click on the label for the New Group folder and change its name to Resources.

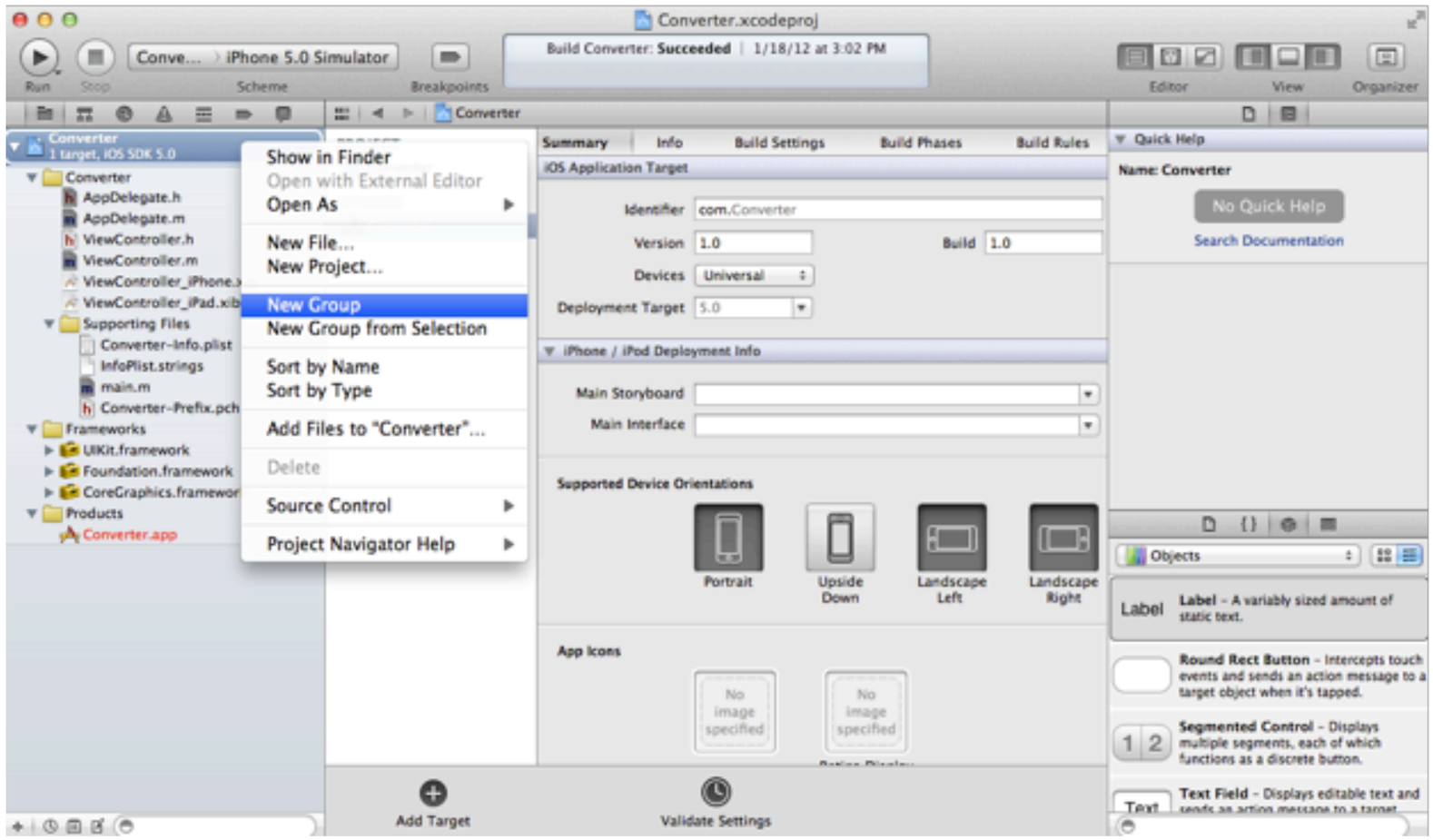

Figure 12 Creating a New Group in a Project

Now, download the image for the Converter app from the companion website for this book at http://www.epressplus.com/2012/01/21/developing-ios5-applications/. Select the logo.png image from the Chapter 2 folder and drag it to the newly created Resources group folder—be sure to check the *Copy items into destination group's folder (if needed)* as shown in Figure 13 and then click on the Finish button.

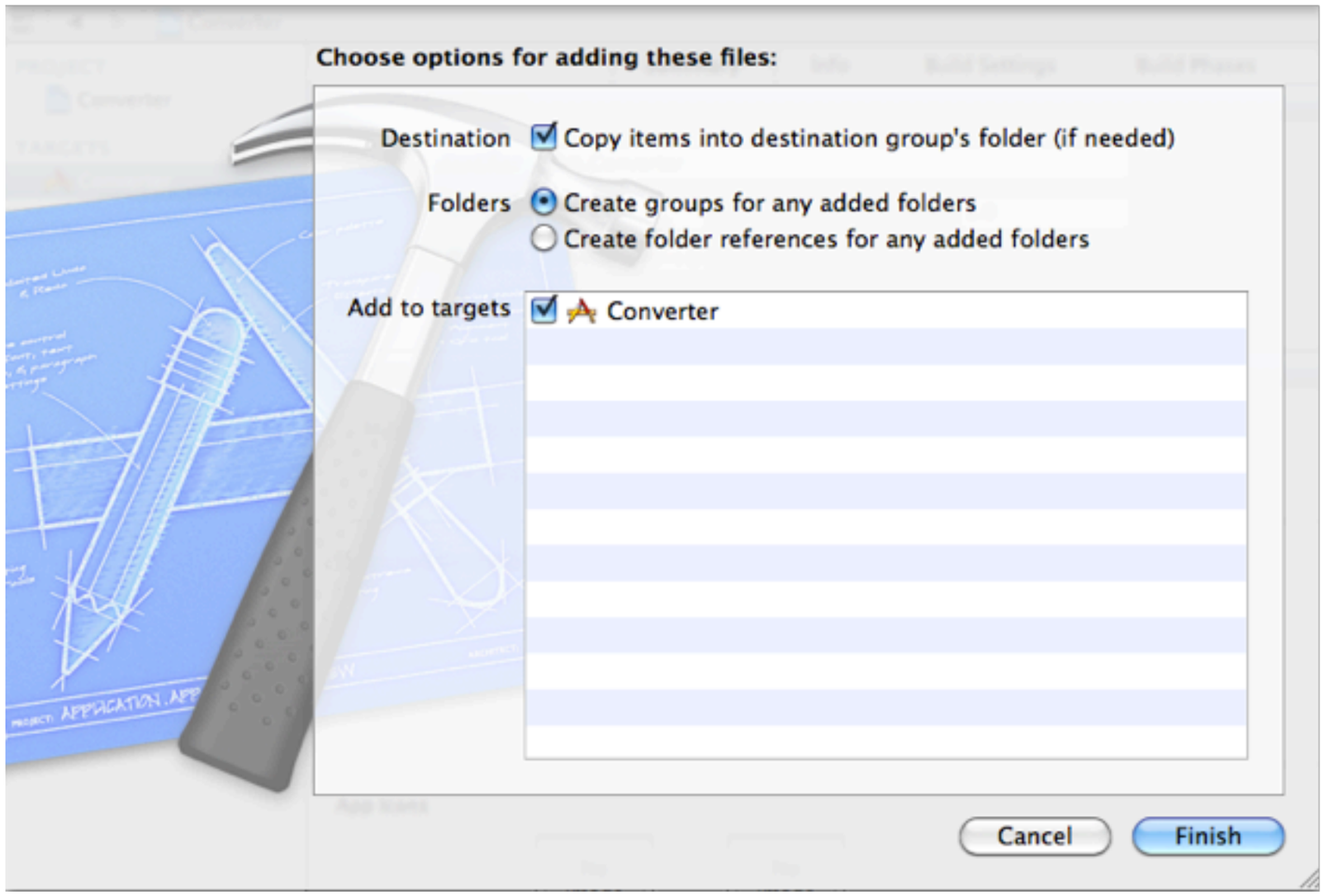

Figure 13 Adding Images to a Project

If everything went right the logo.png image will now be listed inside the Resources folder.

KEY POINT

Remember that images must be dragged into the Xcode Navigator panel in order to be added to a project. Copying files to the Applications folder will not suffice.

In this section, you created a new project in Xcode, and learned how to add resources to it. Good job! Now we can start adding elements to the user interface.

Adding the Necessary User Interface Elements

In this section we will create user interface elements for the application. Traditionally, this was done inside the Interface Builder application which was a separate application from Xcode, but these applications worked very well together to support the development of applications. Now, Interface Builder is fully integrated within Xcode. Information about the User Interface elements are contained in files with an .xib file extension. Expand the Converter folder if it isn't already. Then click on the ViewController_iPhone.xib to open the user interface editor panel inside the Xcode editing window. Before we start adding elements, double-check that the selected scheme is the iPhone 5.0 simulator as shown in the Figure 14.

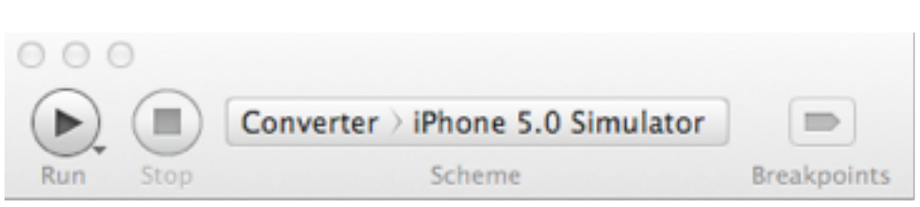

Figure 14 Simulator Running Mode

The gray View window shown in Figure 15 displays the same content that the application will display on the screen of an iPhone application. Currently it doesn't look too exciting -- nothing more than a grey rectangle – because we haven't added any elements to it yet. Now click on the ***Run*** button in Xcode to compile and run the application in the iPhone simulator. As shown in Figure 16, the application simply displays a grey rectangle in the simulator, just as it does in the View window.

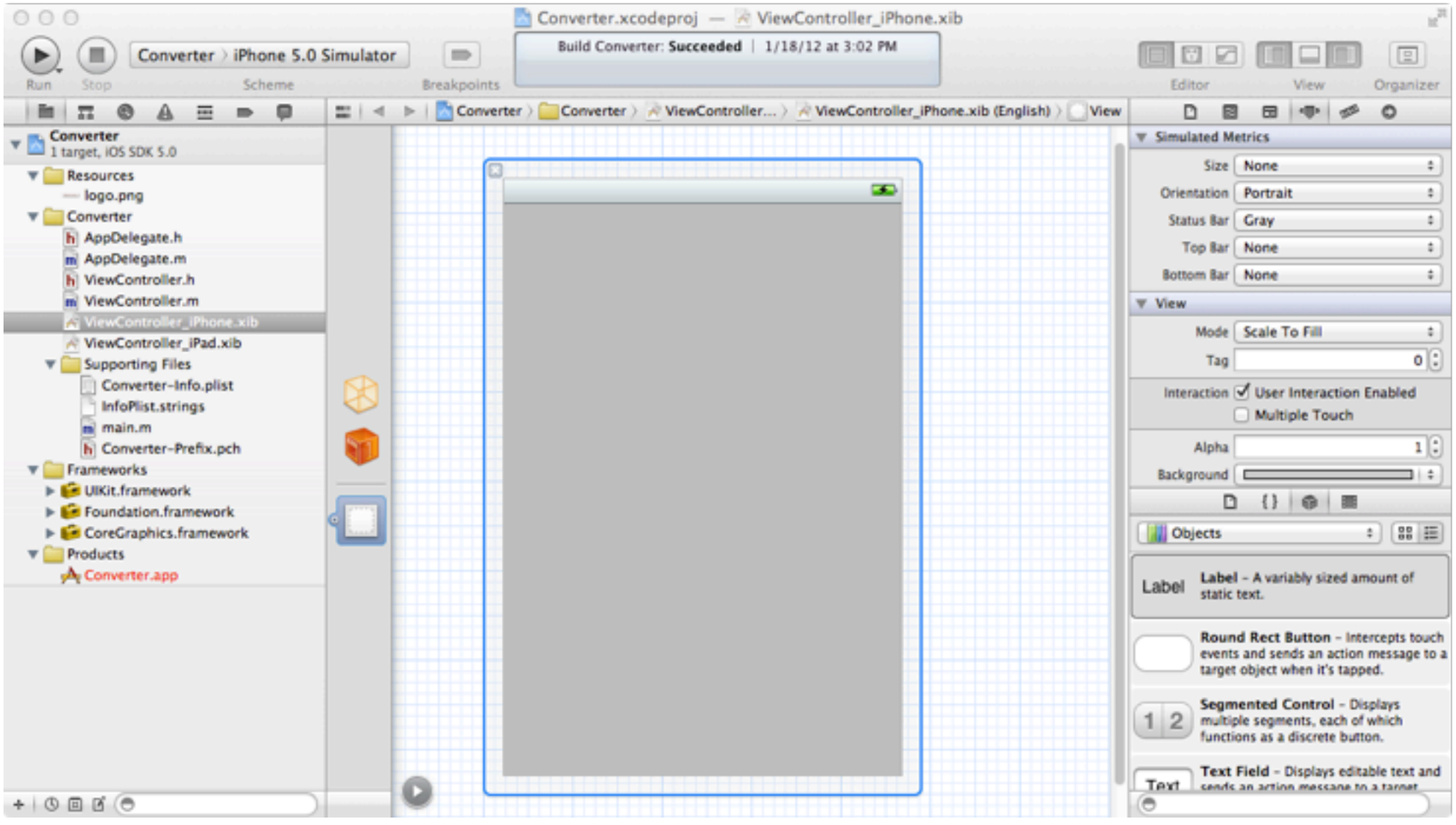

Figure 15 View Window in Editor Area

Figure 16 iPhone Simulator

We are now ready to start adding elements to our user interface. With the ViewController_iPhone.xib file selected in the Navigator area, navigate to View > Utilities > Show Object Library. The Utilities panel that contains the Object Library will open on the right side of the Xcode window in the Utilities area. Notice the buttons to the right of the drop down menu for the Object Library toward the bottom of the Utilities area. The Object Library can be displayed either as a list of elements as shown in Figure 17 or as icons as shown in Figure 18.

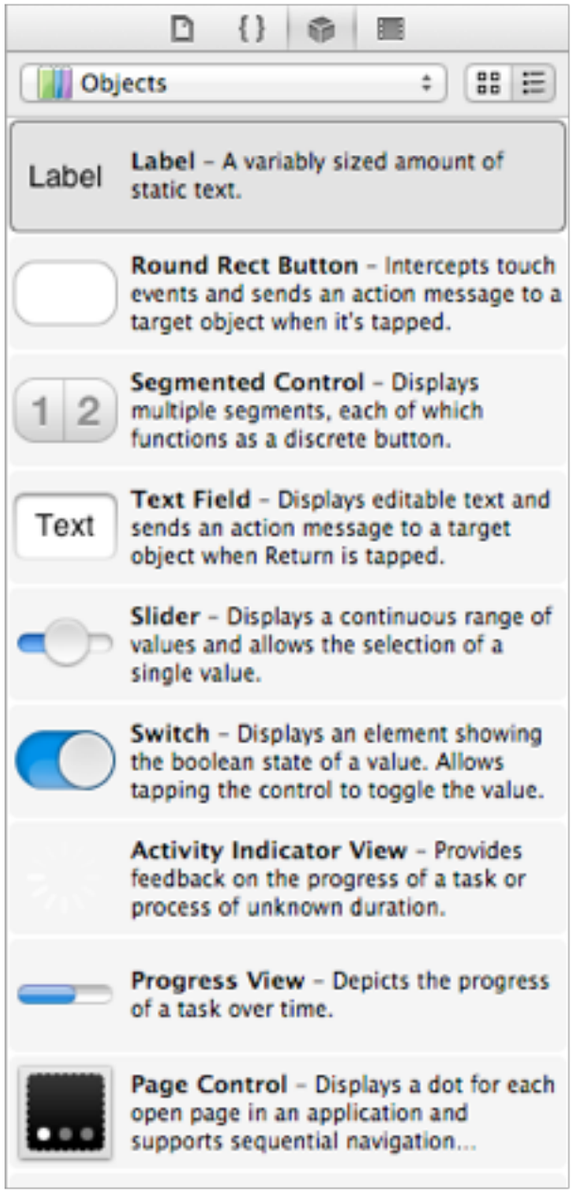

Figure 17 Objects Library - List Display

Figure 18 Objects Library icon Display

The Object Library contains the objects that can be used as elements in the user interface. We can categorize the objects in the library into several groups that can be viewed by clicking on the Object Library dropdown menu as shown in Figure 19.

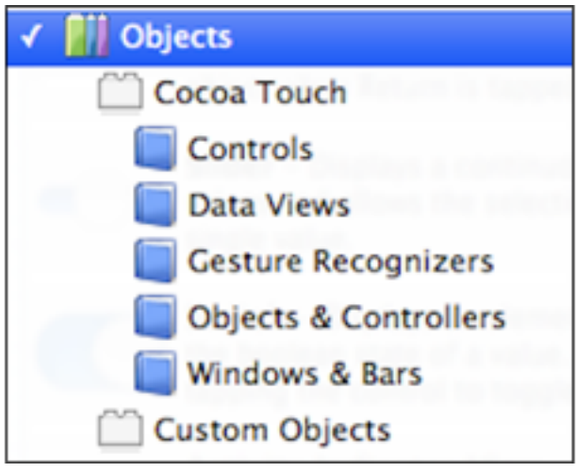

Figure 19 Objects Library Groups

Cocoa Touch – Objects & Controllers

Objects & Controllers are used to create view controllers and different types of user interface controllers such as tab bars and image pickers. View controllers are often associated with the screens of an application, tab bar controllers are used in an app to provide navigation to different screens.

Cocoa Touch – Data Views

Data Views are used to display data and information in different ways such as a Table View that displays data in the format of a table, a Web View for displaying HTML content, a Scroll View for displaying views that are bigger than the application window, an Image View for displaying images, and more. We will use an image view in our application, so click on the Image View icon in the Object Library panel and drag it into the View window. A UIImageView object is now displayed in the View window. This object's label indicates that it belongs to the UIImageView class. By default the Image View object will fill the entire View window. We change its size by clicking on any of the handles positioned around the border of the image view and then drag to the desired shape. For this project, resize the Image View object and position it so that so that it appears similar to Figure 20.

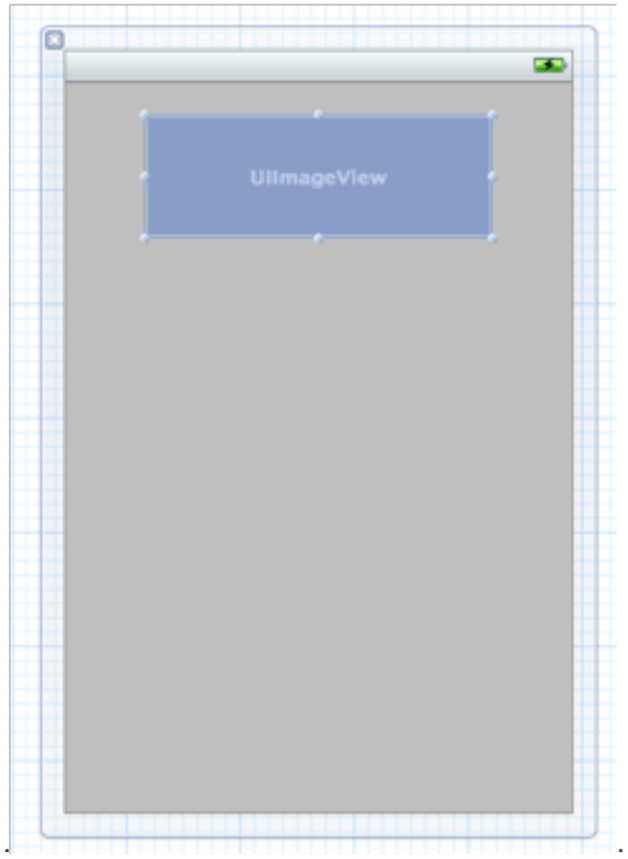

Figure 20 View Window

Cocoa Touch – Controls

Controls are used to create labels, buttons, sliders, segmented controls, text fields, and switches. We will use several of these objects in our Converter application. Click and drag the following four elements from the library to the View window: Segmented Control, Text Field, Label, and Round Rect Button. Click and drag the elements within the View window so that the objects appear in the View window similar to Figure 21. Once the objects have been placed in the View window they can then be connected to code that implements the desired behaviors. We will do this later in this chapter. This is a good time to save your work. Navigate to File > Save.

Figure 21 View with User Interface Elements.

There are two ways that information regarding File's Owner, First Responder, and View objects are accessed for a xib file. Notice the narrow gray panel between the Navigator area and the Editor area in Figure 22. It contains three icons, the gray box with the enclosed white rounded square selects the view, the orange box selects the First Responder interface in the Utilities panel, and the translucent yellow box selects the File's Owner interface. Alternatively, clicking on the gray circle with the white arrow in the lower left of the Editor area will reveal a detailed view for this information, as shown in Figure 23. These objects may be listed in a different order, which is fine, however, if any of these are missing it is necessary to retrace the previous steps in this section. If all of the objects are listed then we may now set the attributes for the objects that comprise our user interface elements.

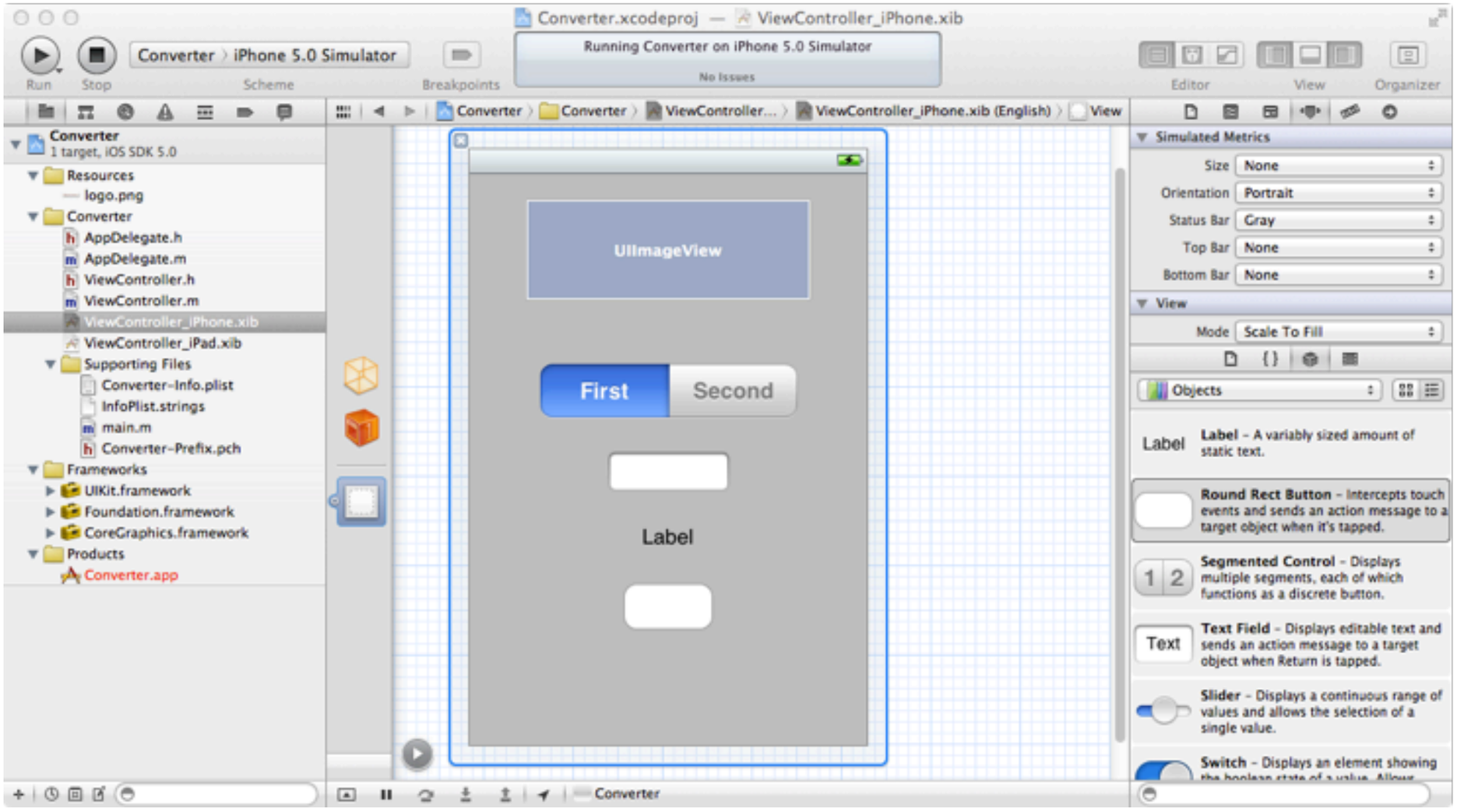

Figure 22 ViewController_iPhone.xib File

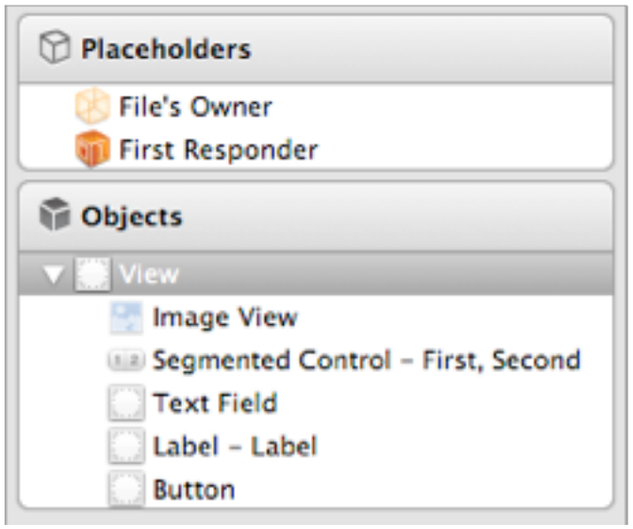

Figure 23 User Interface Elements for Temperature Converter Application

Setting Attributes for User Interface Elements

To set the attributes for the user interface elements, click on the View to select it and then navigate to View > Utilities > Show Attributes Inspector. to reveal the attributes, shown in Figure 24, for the View in the Utilities area. There are several attributes that can be defined, such as the orientation, navigation bars, background color, and more. Click on the Background menu in this pane and change the background to white.

Next click on the UIImageView object in the View window to reveal its attributes in the Utilities area.

Click on the Image menu and select logo.png in the list. The View window now has the Temperature Converter image from the companion website added to the project earlier in the location where the UIImageView object placeholder was previously.

Select the image in the View window then click on the Mode menu in the Image Attributes window and then select Aspect Fit – this will adjust the image to display proportionally within the Image View object. The attributes will now appear as shown in Figure 25 and the View window should now appear similar to Figure 26.

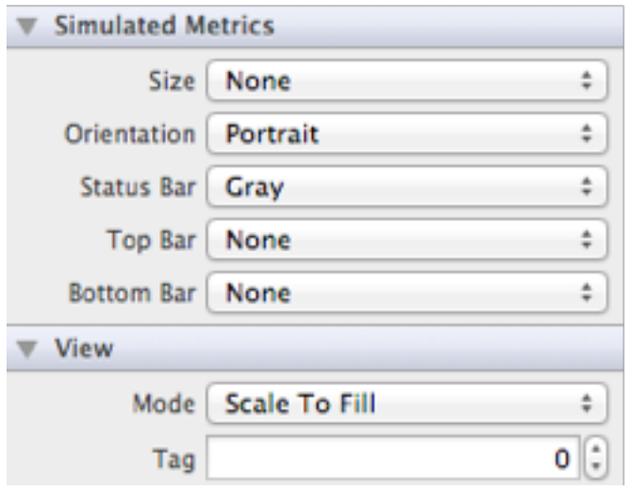

Figure 24 View Attributes Pane

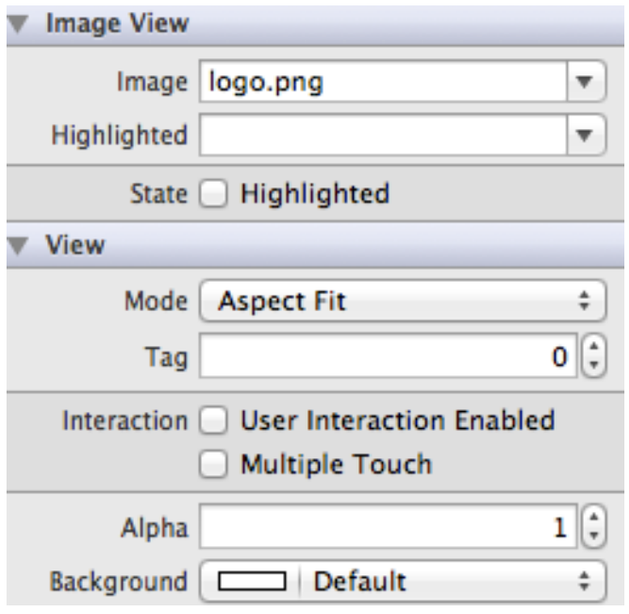

Figure 25 Image View Attributes Pane

Figure 26 User Interface Elements in Converter View Window

Save your work by navigating to File > Save. Then click on the Run button in the Xcode Toolbar to see how the application appears in the iPhone simulator. The iPhone simulator should appear similar to Figure 27. Good job! You have learned how to work with iOS image views and how to change the background of a project!

Figure 27 Partially Developed Temperature Converter App Displayed in the iPhone Simulator

The next user interface object in the View window is a segmented control. This element can be treated as a set of segments from which each segment functions as a button. Go ahead and click on the segmented control in the View window and then click on the View > Utilities > Show Attributes Inspector. A Segmented Control Attributes window similar to Figure 28 will appear. The Attributes Inspector is used to change the properties of the segmented control. Notice that the Segmented Control Attributes pane is divided vertically into three sections labeled Segmented Control, Control, and View. Let's have a look at the properties that can be changed for segmented control elements in the Segmented Control section.

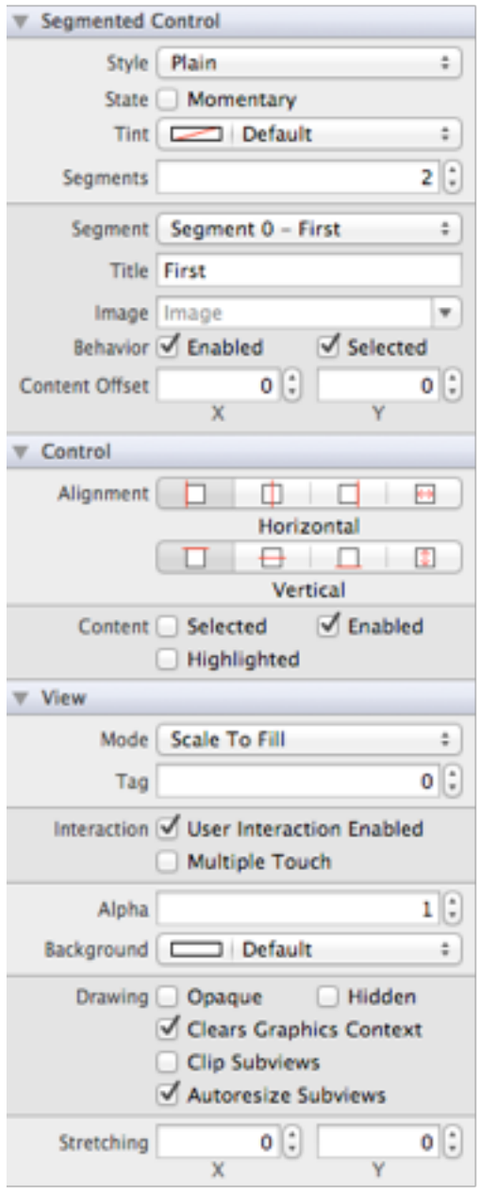

Figure 28 Segmented Control Attributes Pane

The first attribute, Style, provides the capability to apply different styles:

- Plain – the default view
- Bordered – plain segmented control with border
- Bar – a bit sleeker with grey coloring and white labels
- Bezeled – similar to bar but a bit wider with white and gray labels

Go ahead and play with the Style segments and choose the one that is most appealing. We chose bar for the value of this attribute.

The next attribute in the pane is Momentary which is selected or not with a checkbox. This attribute determines whether a momentary change of appearance or a permanent change in appearance is desired when the user taps on one of the segments in the segmented control. We do not check the box so that the change in appearance will be permanent. The next attribute in the Attributes Inspector window is Tint, which allows developers to change the color of the control. We will retain the default color scheme for our application. We also set the value of Segments to 2 since we will use two segments to switch between Celsius and Fahrenheit functionality.

The next section in the Segmented Control Attributes pane relates to attributes for each segment in the segmented control. The attributes that pertain to each segment are:

- Title
- Image
- Enabled Checkbox
- Selected Checkbox
- Content Offset

Each segmented control is enumerated from left to right as Segment 0 ... Segment n. By default Segment 0 is selected. Set the title for the first segment by typing Fahrenheit to Celsius into the Title text box. Set the title for the second segment by selecting Segment 1 and then typing Celsius

to Fahrenheit into the Title text box. An image could be applied to the segments but that is not desired for this application. The Enabled checkbox should be selected for each segment. The Selected checkbox should be checked for whichever segment should be selected by default. The offset attribute is not desired for this segmented control. At this point, the View window should appear similar to Figure 29. Now is a good time to Run the application in the simulator again. Click the Run button in the Xcode Toolbar. The screen in the iPhone simulator should appear similar to the View window in Figure 29.

Figure 29 View Window After Setting Segmented Control Attributes

The next two elements in the View window are a Text Field and a Label. We will look at the attributes for these elements now, but won't actually change any of these attributes at this point – we will change the values of the attributes for these elements programmatically later in this chapter.

Text Fields are used to process textual data provided by a user. Click on the Text Field in the View window and then navigate to View > Utilities > Show Attributes Inspector. The Text Field Attributes pane will appear similar to Figure 30. Notice that the Text Field Attributes pane contains three major sections: Text Field, Control, and View. The attributes within the Text Field section can be used to set the appearance, behavior, and input traits for the text fields.

To examine the attributes for the Label element that appears next in the View window, click on the Label element in the View window and then navigate to View > Utilities > Show Attributes Inspector. A Labels Attributes pane similar to Figure 31 will appear. Labels are one of the most commonly used user interface elements. A label is used to display text. Look at the attributes. The first and the most important one is Text where the desired text for the label may be entered. Notice that there is quite an assortment of attributes such as font, font size, and alignment that can be set to control how labels appear. For this project, we will set the attributes we need for this project programmatically later on.

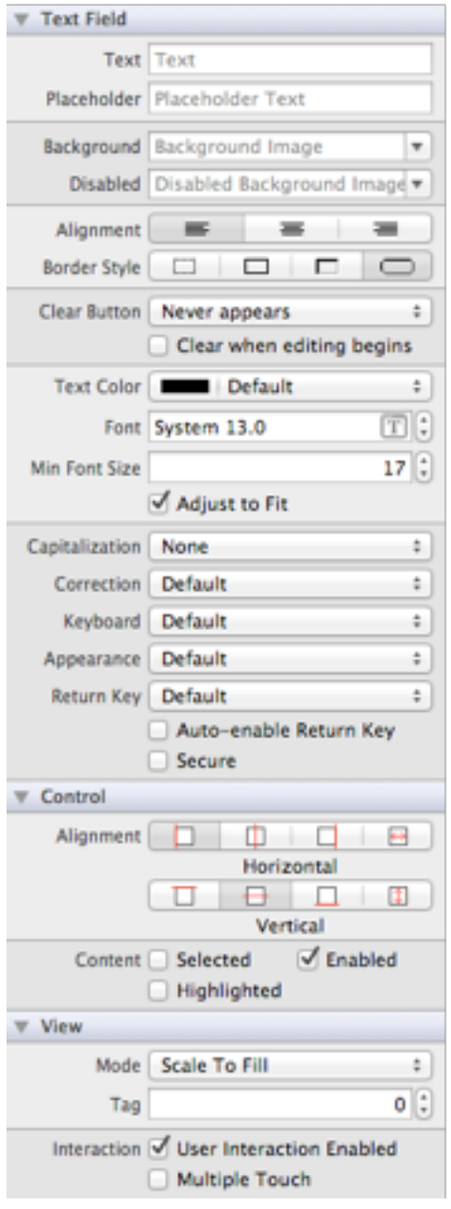

Figure 30 Text Field Attributes Pane

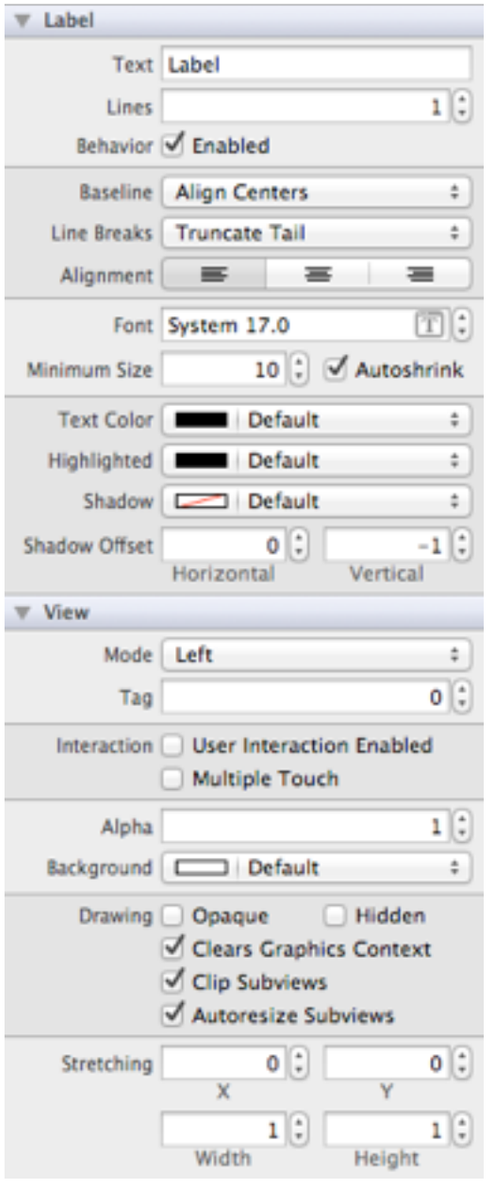

Figure 31 Label Attributes Pane

The next element in our View window is a Button. Buttons trigger methods when user interaction is detected. Click on the button in the View window and navigate to View > Utilities > Show Attributes Inspector to open the Button Attributes pane similar to Figure 32.

The Button Attributes pane is divided into three sections, Button, Control, and View. The first attribute in the Button section is Type, this allows the developer to select the type of button desired. The various types are:

- Custom: a transparent rectangle.
- Rounded Rectangle: a rectangle with rounded corners, the default form of the button.
- Detail Disclosure: a greater than symbol in a round button that is typically used to display additional details on a new screen.
- Info Light: displays the typical info icon inside a light colored button.
- Info Dark: displays the typical info icon inside a dark colored button.
- Add Contact: the button with the plus symbol typically used for adding an additional element, such as a new contact in the Apple Address Book.

Select Rounded Rect for the style of the button for the Converter project.

KEY POINT

There are several types of button options that have typical uses in native applications. It is important for developers to maintain use of buttons in a manner that is typical in native applications. This ensures that apps retain the behavior that is intuitive to users. This is an important consideration related to human interactions in high quality applications

The next attribute is the State Configuration to set the initial state of the button -- the default is fine for this project. The Title attribute contains the title of the button. For this application the word ***Convert*** should be typed into the Title text field. The button may need to be resized and repositioned to accommodate the title. There are also attributes to select an image for the button title rather than plain text, or to select a title background image for the button. Additional attributes are available to select colors for the text and shadows. At this point our View window should appear similar to Figure 33. This is a good time to confirm that everything we have created so far displays as expected inside the iPhone simulator. Navigate to File > Save to save your application, and then click on the Run button to view the Temperature Conversion application inside the iPhone simulator.

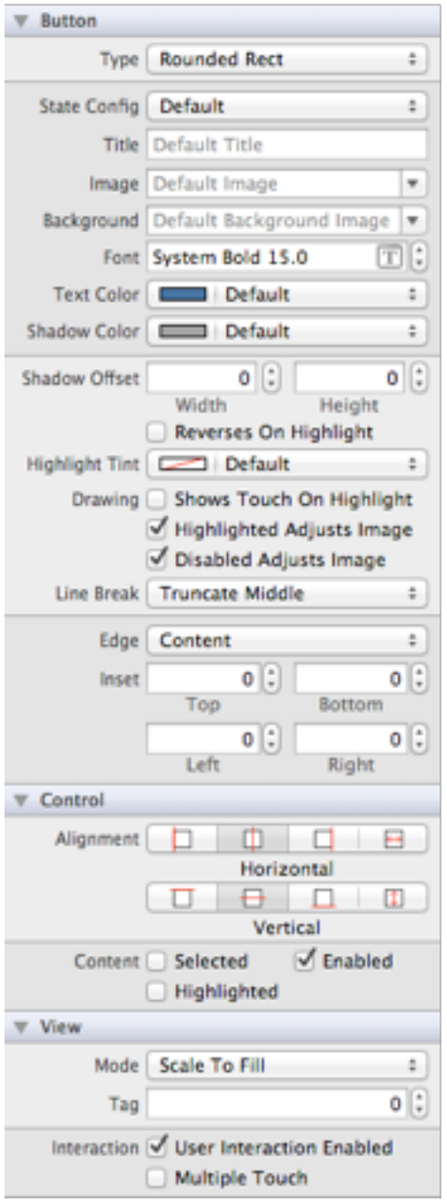

Figure 32 Button Attributes Pane

Figure 33 View Window

Creating Functionality for User Interface Objects in Xcode

So far we have created the user interface elements for the Temperature Converter application and defined attributes for some of those elements. We now need to create the functionality in the Objective-C code for the user interface elements in order to have a working Temperature Converter application.

The classes that implement the user interface elements for the Temperature Converter application are UIimageView for the image view element, UISegmentedControl for the segmented control, UITextField for the text field, UILabel for the label, and UIButton for the button. No functionality is needed for the image view element, and no functionality specific to the button itself is needed, however a method to compute a conversion to

Celsius and a conversion to Fahrenheit when the button is tapped is needed. We also need to create and customize instances of UISegmentedControl, UITextField, and UILabel in the code.

Connecting User Interface Elements to the Code

The first step in defining the behavior of the user interface elements is to connect them to the code. Connections link the user interface elements to the code so that they can be accessed and modified via the code when the application is running. Select the ViewController_iPhone.xib file inside the Navigator area then navigate to View > Assistant Editor > Show Assistant Editor, or click on the Assistant Editor button in the Xcode Toolbar. The editor area should appear similar to Figure 34 with the ViewController.h file shown in the Assistant Editor--note that we closed the Utility area using the button in the top right of the Toolbar. Recall, that the Assistant Editor will always open a file related to the one that is already displayed in the single editor area before the Assistant Editor is selected. For the ViewController_iPhone.xib file, there are two closely related files. ViewController.h and ViewController.m. An item in the jump bar above the editor, shown in Figure 35, provides the number of related files and arrows to select these files. In our case there is a 2 since there are two related files. If the ViewController.m file is displayed in the assistant editor rather than the ViewController.h file, then use one of the arrows to switch to the ViewController.h file which should appear as shown in Figure 34.

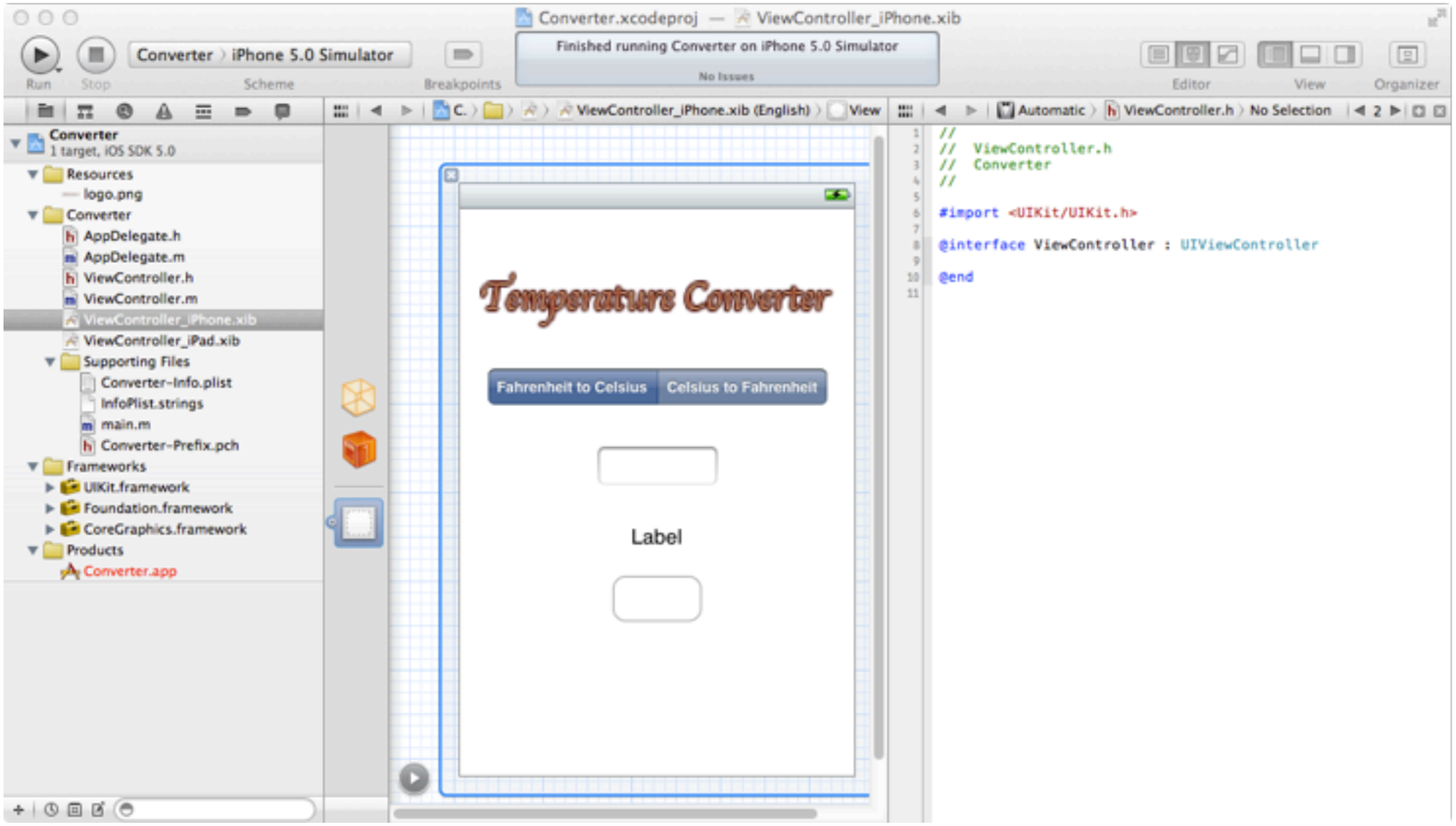

Figure 34 Editor Area Displaying the Assistant Editor

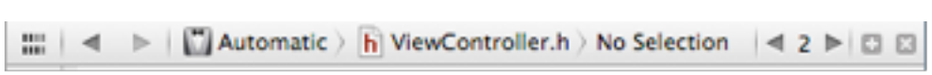

Figure 35 Editor Area Jump Bar

Now ctrl-click on the segmented control element in the view window, a new window will popup that should look similar to that shown in Figure 36.

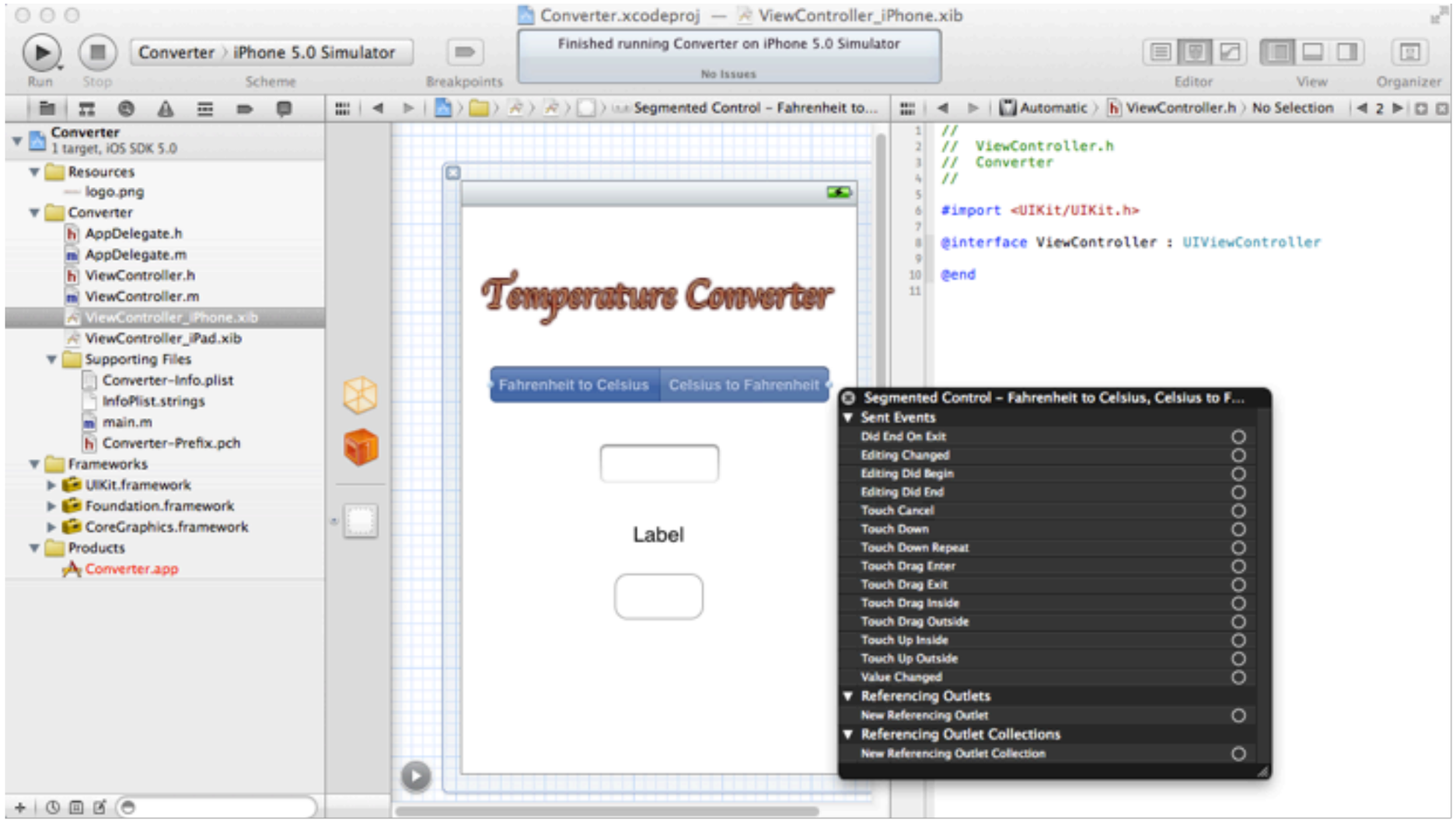

Figure 36 Segmented Control Events and Outlets

The Segmented Control window contains three sections. The Sent Events section contains events that can be used when working with a segmented control. The Referencing Outlets section provides the means to create a connection between an object in the current object to an outlet in another object. The third section allows us to create a outlet connections to a collection object. Sometimes you will need to create a complex User Interface for your applications and creating outlets for each of them might be a tedious process. You can use the outlet collections that will store User Interface elements in an array to reduce the need to define the behavior individually for each element.

Click on the circle to the far right of the New Referencing Outlet as shown in Figure 37 and then drag to the line above the end directive in the ViewController.h file. A popup window will appear as shown in Figure 37 when the mouse is released. Enter segmentedControl for the name of the Segmented Control outlet as shown in Figure 38.

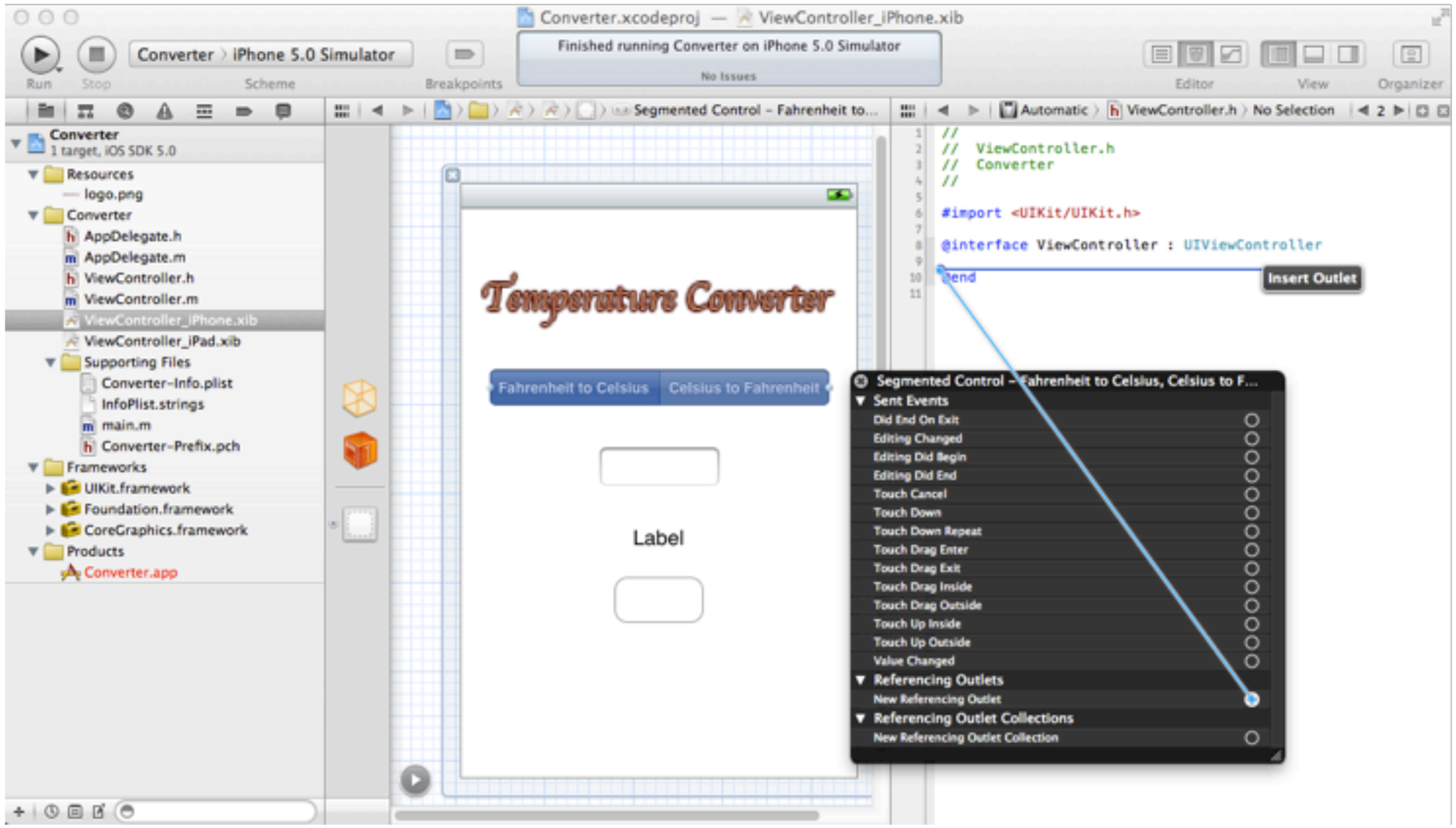

Figure 37 Connecting the Segmented Control to the Code

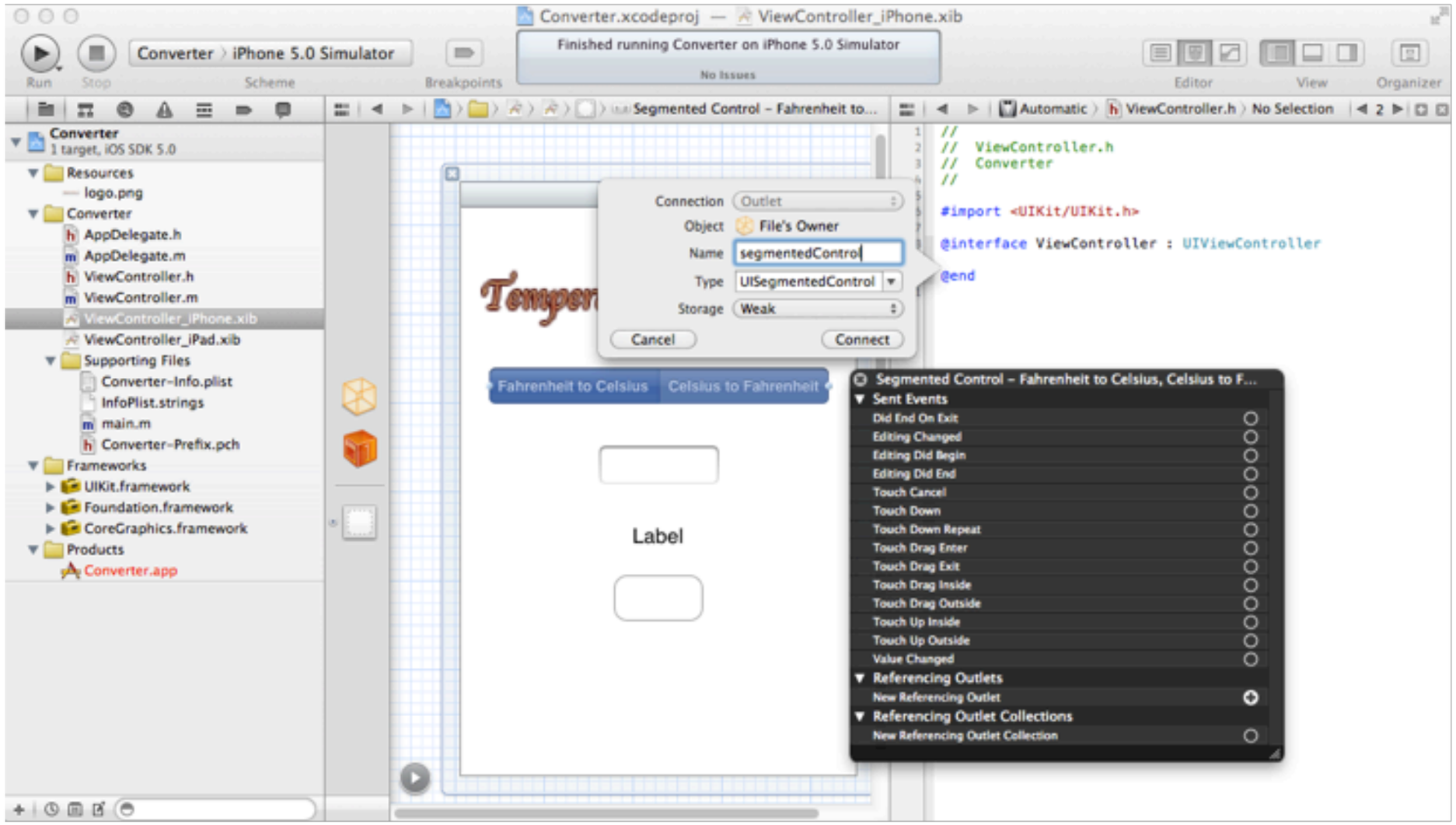

Figure 38 Creating the Outlet Connection for the Segmented Control

Notice the new declaration for the IBOutlet inserted in the following code.

```
//
//  ViewController.h
//  Converter
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segmentedControl;
@end
```

Repeat this same process of ctrl-clicking on an item in the View window then making a connection to the ViewController.h file for the text field and label. Name the outlet connection textField for the text field outlet and label the label label.
Now the ViewController.h file should appear similar to the code displayed below with IBOutlet properties for the segmented control, the text field and the label on the three lines above the end directive.

```
//
//  ViewController.h
//  Converter
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UISegmentedControl *segmentedControl;
@property (weak, nonatomic) IBOutlet UITextField *textField;
@property (weak, nonatomic) IBOutlet UILabel *label;
@end
```

KEY POINT

Notice that the declarations for the UILabel, UITextField, and UISegmentedControl instance variables are preceded with the IBOutlet keyword. This keyword is used whenever a connection will be needed between the object in the code and the element in the user interface.

Synthesizing Properties and Memory Management

Navigate to the ViewController.m and notice the three synthesize directives near the top of the file as shown in the code below. Whenever properties are declared they must also be synthesized which creates accessor and mutator methods for these objects. In iOS 4.0 and later Xcode automatically adds the @property and the @synthesize directives for all reference outlets created as we have just done for the segmented control, text field, and label. Notice the synthesize directives inserted by Xcode into the ViewController.m file as shown in the following code.

```
//
//  ViewController.m
//  Converter
//

#import "ViewController.h"

@implementation ViewController
@synthesize segmentedControl;
@synthesize textField;
@synthesize label;
```

Xcode also inserted statements into the code that assigns a nil value to the references to the outlet connections as a result of the way that we created the outlet connections. Scroll down in the ViewController.m file and find the ViewDidUnload method. The first three statements in the ViewDidUnload method use an accessor method for each object to assign nil to those references in the following code.

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

Now that we have created connections between the button, text field, and label in the user interface to the ViewController class, when a user taps on the button we will be able to get the value of the text field and change the value of the label programmatically from within the code when running the application. Now we need to create a method to do that.

Creating User Interface Actions

In the previous section we created instances of the user interface elements, so that we can manipulate with their properties programmatically. Now it's time to create an action that will be triggered whenever a user taps on the Convert button on the screen. To do this, go back to the ViewController_iPhone.xib file and open the Assistant Editor for the ViewController.h file, and then ctrl-click on the Convert button. This time drag from the circle to the far right of the Touch Up Inside event to just above the end directive to insert an action. When you release the mouse, a Connection popup window will appear. Enter ***convertTemperature*** for the name of the action as shown in Figure 39, and then click on Connect. This will declare a new method for the class named convertTemperature that will execute whenever a user taps on the Convert Button. Notice the new declaration for the convertTemperature IBAction just above the end directive in the following code.

```
//
//  ViewController.h
//  Converter
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UISegmentedControl *segmentedControl;
@property (weak, nonatomic) IBOutlet UITextField *textField;
@property (weak, nonatomic) IBOutlet UILabel *label;
- (IBAction)convertTemperature:(id)sender;
@end
```

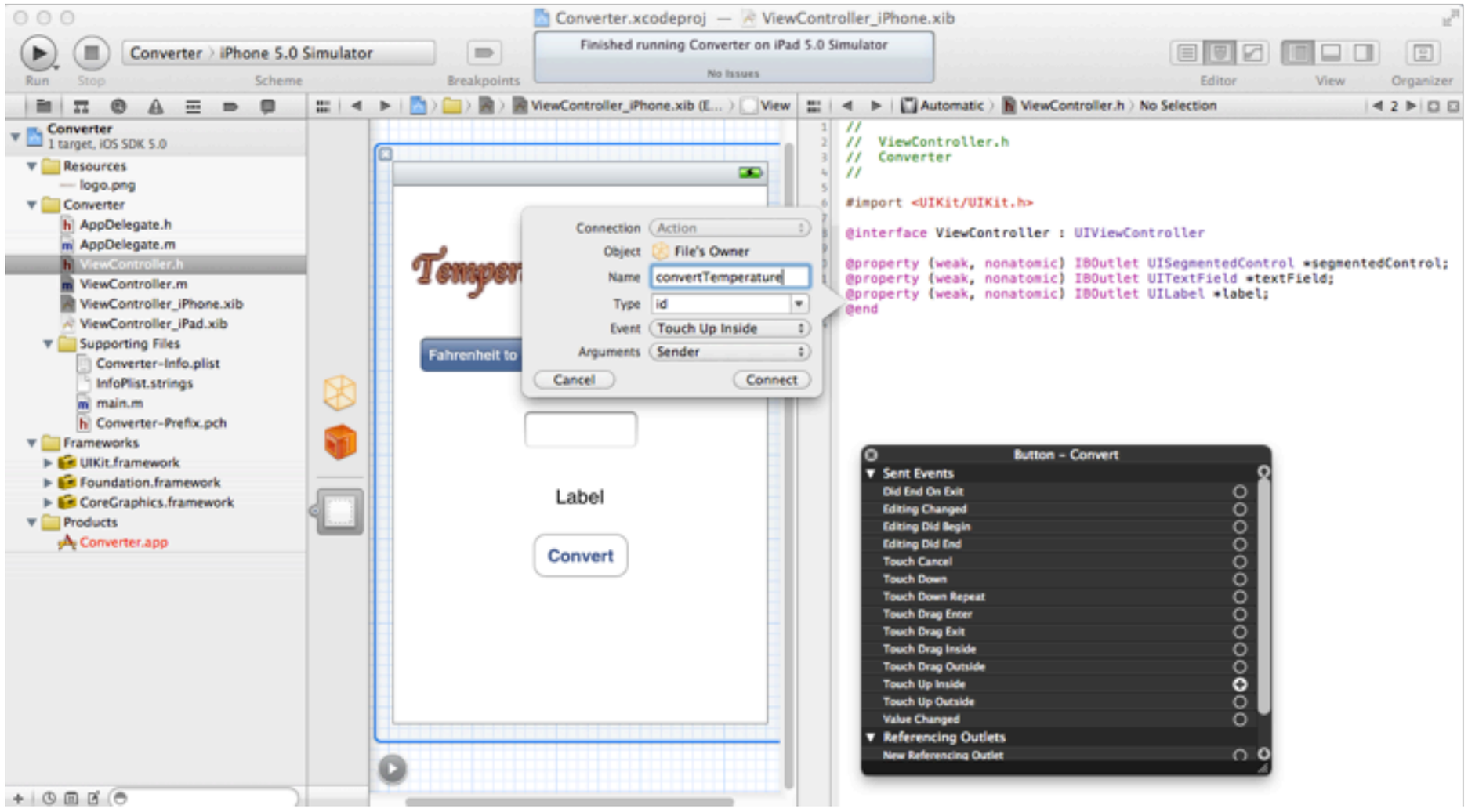

Figure 39 Creating the ConvertTemperature Action

Once the action is created, there will be a new method declaration for the action in the ViewController.h file and a new method definition in the ViewController.m file. The body of the convertTemperature method in the ViewController.m file is empty. It is up to us to add the custom code to perform the desired behavior for the button. Before we do that let's review the connections that we have made. Ctrl-click on the File's Owner icon in

the pane just to the left of the Editor area. A File's Owner window should popup similar to that shown in Figure 40. The Outlets part of the File's Owner window lists all the IBOutlets that have been created so far and the convertTemperature method listed under the Received Actions section of the File's Owner window.

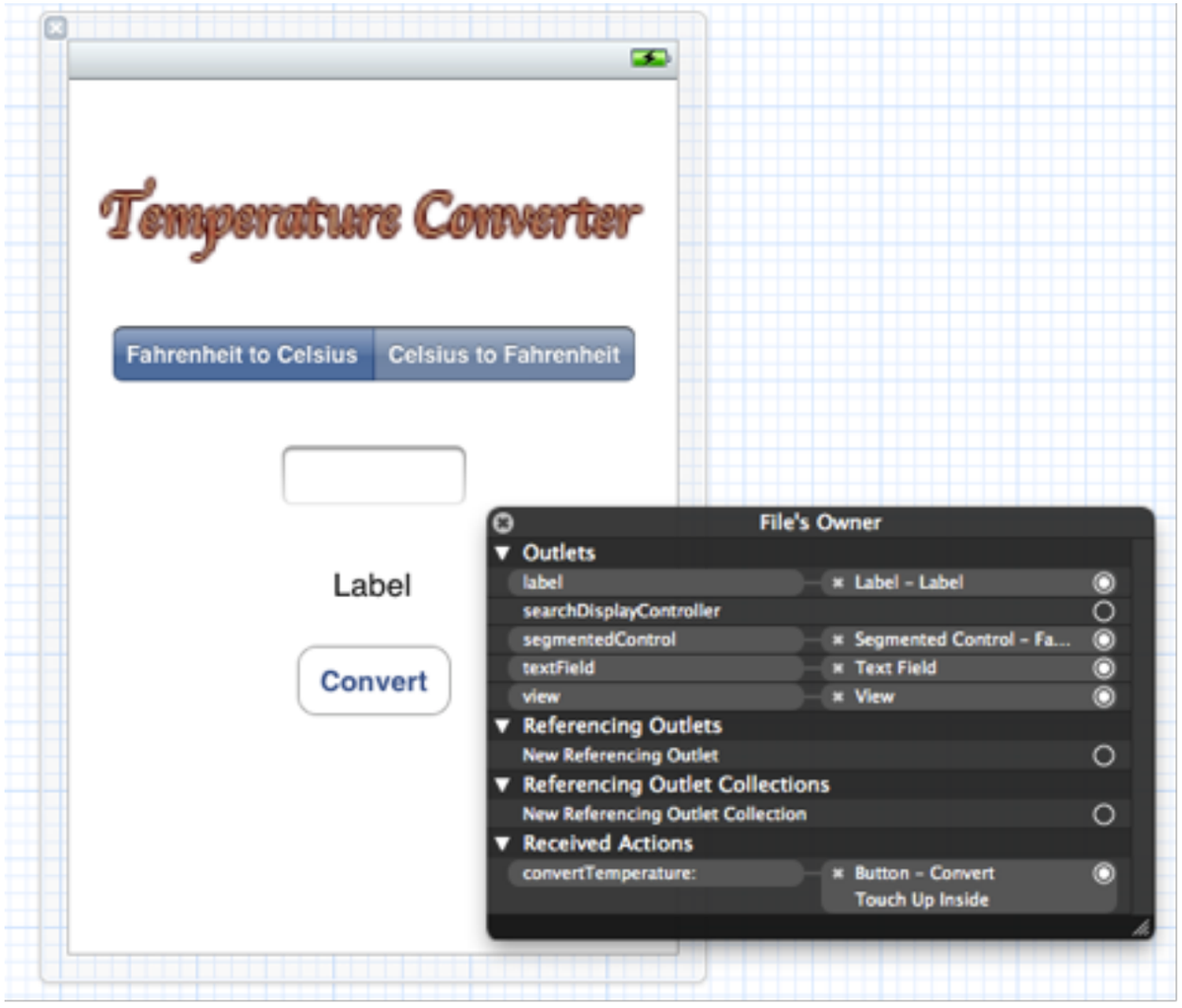

Figure 40 Outlets and Actions for the ConvertViewController

Adding Custom Code

So far Xcode has generated the code for us that is required for the outlet and action connections that we created for the elements in our user interface. Now we need to define the functionality we want the app to perform in order to perform temperature conversions inside the convertTemperature method in the ViewController.m file.

The first thing we will need is a variable to store the state of the segmented control so we know whether to convert the input the user enters into the text field from Celsius to Fahrenheit or from Fahrenheit to Celsius. Since there are only two segments in our segmented control we can define a boolean variable that be true if the first segment is selected so the value entered is a fahrenheit value that should be converted to celsius when the user taps the Convert button, or indicates will be false if the second segment is selected so the value entered into the text field is a celsius value that should be converted to fahrenheit. We can do this by adding the declaration for the fahrenheit variable shown in the code below. We also need a variable to store the value of the temperature the user enters into the text box, and the value of the converted temperature. Add the three declarations shown in the following code to the convertTemperature method in the ViewController.m file. Notice that once these declarations are added, yellow warning indicators popup in the left gutter of the Editor Area. Click on the triangles to see what the issues are. You will see that Xcode is informing you that these variables are not used in your program. This is a good type of warning because this situation often reveals a mistake by the programmer. However, in this case it is not a problem because we aren't done yet, and we will use these variables before we are done.

```
- (IBAction)convertTemperature:(id)sender
{
    BOOL fahrenheit;
    float userTemperature;
    float convertedTemperature;
}
@end
```

The next step is to add code that determines which segment is selected and then assign a boolean value to the fahrenheit variable as appropriate. Add the if-else construct that determines which as shown in the code segment below, just after the declarations. Notice that now that we have assigned values to the fahrenheit variable inside the if-else construct that there isn't a yellow warning symbol next to its declaration in Xcode anymore.

```
    if(segmentedControl.selectedSegmentIndex==0) {
        fahrenheit=FALSE;
```

```
}
else {
    fahrenheit=TRUE;
}
```

Now that we have code that determines what conversion the user desires, we can get the value from the text field and then perform the temperature conversion. The formulas for converting temperatures are as follows:

Centigrade = (5/9)(Fahrenheit-32) and Fahrenheit = (9/5)Centigrade + 32

First we get the value from the text field and assign it value to our conversion temperature variable. Notice that we get that value using the text property of our textField object and then convert it to a float value using the floatValue method so that we can assign it to our userTemperature variable. After this, we use an if-else construct to execute the proper formula for a temperature conversion depending on the value of the BOOL fahrenheit variable. Add the assignment and if-else construct in the following code segment after the code just previously added.

```
userTemperature=[textField.text floatValue];
if(fahrenheit) {
    convertedTemperature=((9.0/5.0)*userTemperature)+32;
}
else {
    convertedTemperature=(5.0/9.0)*(userTemperature-32);
}
```

The last two things we need to do in the convertTemperature method is to assign the value of the converted temperature to the text field on the screen, and then lower the keyboard after the user enters a temperature value and taps on the Convert button. In order to display the converted temperature in the text field we simply need to assign it to the text property of the label. But before we can do that we need to create a string that contains that value since the text property must be a string. We do this with the stringWithFormat method so that we can use a format specifier (%. 1f" to create a string that displays numbers with one place to the right of the decimal point, as in 32.5.

We also realize at this point that we have a minor problem. Once the user taps on the text field it becomes the first responder for the user interface and the keyboard animates from the bottom of the screen and covers the lower part of the screen and the keyboard covers the Convert button. Since we want to attach the method invocation that lowers the keyboard to the Convert button, we will need to rearrange the elements on our screen so that the Convert button is accessible while the keyboard is raised. We will do this once we are finished with the code for the convertTemperature method. Add the two statements in the following code segment to the end of the convertTemperature method.

```
label.text=[NSString stringWithFormat:@"%.1f", convertedTemperature];
```

```
    [textField resignFirstResponder];
}
@end
```

We have now added the custom code necessary to add the functionality for the temperature converter app. The complete method is as follows.

```
- (IBAction)convertTemperature:(id)sender
{
    BOOL fahrenheit;
    float userTemperature;
    float convertedTemperature;
    if(segmentedControl.selectedSegmentIndex==0) {
        fahrenheit=FALSE;
    }
    else {
        fahrenheit=TRUE;
    }
    userTemperature=[textField.text floatValue];
    if(fahrenheit) {
        convertedTemperature=((9.0/5.0)*userTemperature)+32;
    }
    else {
        convertedTemperature=(5.0/9.0)*(userTemperature-32);
    }
    label.text=[NSString stringWithFormat:@"%.1f", convertedTemperature];
    [textField resignFirstResponder];
}
@end
```

Now, go back to the View window for the user interface and rearrange the user interface elements so that the Convert button is positioned in the top part of the View window. You may also wish to adjust attributes in the Utilities area for the user interface elements to change colors, and fonts to suit your own preferences. We adjusted the attributes so that the final implementation of our app appears as shown in Figure 2.41.

Figure 41 Final Temperature Converter App in the iPhone Simulator

Save all the files and click on the Run button in Xcode to test your application. Now the application should behave as expected inside the iPhone simulator. Congratulations! There is one more thing that needs to be taken care of before we are finished which we will take care of now.

Universal Apps

Recall that we selected Universal for the device family when we first created this app. We have three choices for device family in Xcode 4. These choices are iPhone, iPad, and Universal. Apps that are developed specifically for the iPhone device family are generally optimized for the iPhone and iPod Touch platforms. These apps can be downloaded and run on iPads but don't generally provide the rich user experience that Apple intended for iPads. Developing Universal apps provides the greatest efficiency and maintainability for apps that incorporate best practices described in Apple's iOS Human Interface Guidelines. Apps developed specifically for the iPad don't run in the iPhone device family. Developers can create two completely different versions of the same app: one for the iPhone and one for the iPad. However, this can lead to a lot of redundancy and commonly result in maintainability difficulties.

The Universal device family allows developers to create one project for an app that maximizes the user experience uniquely for both the iPhone and the iPad. Since we chose the Universal device family when we created this Temperature Converter project. Xcode created two different View Controller xib files for our project. One view controller was named ViewController_iPhone.xib and one view controller was named ViewController_iPad.xib. So far we have only customized the ViewController_iPhone.xib file, next we need to customize the ViewController_iPad.xib file. But before we do, let's look at the mechanism within the app that facilitates the Universal app. Select the AppDelegate.m file in the Navigator area of Xcode and display this file in the Editor area and then find the didFinishLaunchingWithOptions method, as follows.

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    self.window = [[UIWindow alloc] initWithFrame:[[UIScreen mainScreen] bounds]];
    // Override point for customization after application launch.
    if ([[UIDevice currentDevice] userInterfaceIdiom] == UIUserInterfaceIdiomPhone) {
        self.viewController = [[ViewController alloc] initWithNibName:@"ViewController_iPhone" bundle:nil];
    } else {
        self.viewController = [[ViewController alloc] initWithNibName:@"ViewController_iPad" bundle:nil];
    }
    self.window.rootViewController = self.viewController;
    [self.window makeKeyAndVisible];
    return YES;
}
```

The first thing this app does when it is launched is that it checks what kind of a device the app is running on. If the app is an iPhone then it sets the view controller to the ViewController_iPhone.xib file, otherwise it sets the view controller to the ViewController_iPad.xib file. We added simple elements to the ViewController.xib file and created IBOutlet and IBAction connections to the ViewController.h and ViewController.m files just as we had for the iPhone interface. We used the convention of using the same names for these elements as we had previously but with iPad appended to the first part of the name. Then we copied the convertTemperature method that we used for the iPhone interface and customized it for the iPad interface. After making similar changes, select the iPad 5.0 Simulator as the scheme and then click on the Run button to test your app in the iPad simulator. The screenshot for our app in the iPad simulator is shown in Figure 2.42. The complete code for this universal app is available on the companion website for this book.

Figure 42 Temperature Converter in the iPad Simulator

Summary

In this chapter we have learned about basic concepts of iOS programming and created a simple iPhone application. First we learned how to create and use Xcode templates, specifically we used the View Based Application template. In the latter part of the chapter we described the following user interface elements, a UIImageView element for displaying images, a UIButton element for facilitating execution of a given method when a button is touched by a user, a UILabel element for displaying text on the screen, a UISegmentedControl element for providing alternative options, and a UITextField element for receiving user input. The general process of incorporating user interface controls in an application is as follows:

1. Drag the UI controls to the iPhone View window for the .xib file.

2. Set the attributes for the elements in the Utilities panel in Xcode.

3. If access to the attributes of the user interface controls is necessary within the code then create Outlet connections and between the .xib and .h files

4. Provide the code inside the appropriate method in the .m file that changes the appearance or behavior of the UI control or user interface element.

Another important concept covered in this chapter was the delegation pattern. Delegation is a simple mechanism that allows the developer to assign responsibilities of one object to another object. For the Temperature Converter application the text field sends notifications to its delegate, the ViewController to resign control and lower the keyboard when the return button on the keyboard is tapped.

Review Questions

1. What is an IBOutlet? Why is it used? How is it used?

2. Which class is used to display text on an iPhone screen? Which property of the class is used to update the text?

3. What is an IBAction and why is it used?

4. Which event is used to implement the functionality of a button?

5. How are connections between user interface elements and the code created? Describe the steps.

6. Which property of the UISegmentedControl is used to check which segment is currently selected?

7. Which delegate method is used to lower a keyboard off the screen?

8. How do you relinquish ownership of an object? Which method should you use to perform that action?

9. What does the resignFirstResponder method do?

Exercises

1. Write a short program that displays an image on the screen.

2. Create a program that displays the following UI elements on the screen -- customize the appearance of all of the UI elements:

    a. UISegmentedControl

    b. UIButton

    c. UILabel

    d. UISlider

    e. UITextField

3. Modify the application developed in this chapter to convert pounds to kilograms, kilograms to pounds, meters to feet, and feet to meters.

4. Develop a simple calculator with the capability to add and subtract numbers.

Chapter 3

Photo Puzzle App - Part 1: User Interface

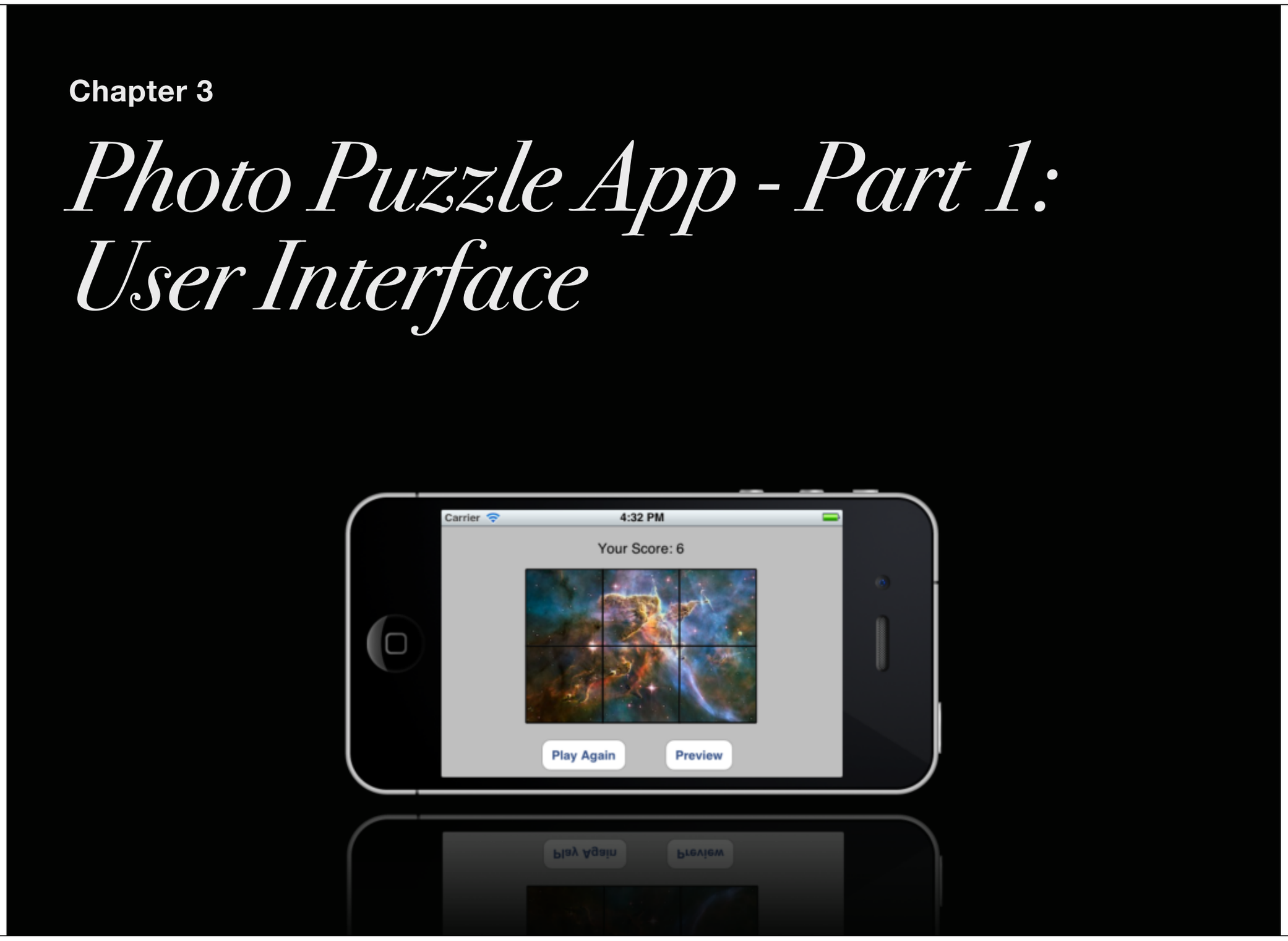

Photo Puzzle App Part 1: User Interface

Concepts emphasized in this chapter:

- Designing an app
- Building a user interface and making connections
- Using different screen orientations
- Handling touch events to move elements on the screen

Introduction

This chapter presents the first part of a Photo Puzzle Project. This application will allow a user to move puzzle pieces placed around a puzzle board on the iPhone screen to their proper position to reform the original photo. If puzzle pieces are moved to the wrong position, the application sends them back to the outside of the puzzle board, once a puzzle piece is placed in its proper place it is then locked in place. The user may access a preview or the image and may play again.

The emphasis in the first part of the development will be on the design phase, techniques needed to create the user interface, and learning about touch events. The design phase will consist of identifying the objects to be used in the puzzle. During the implementation phase the behavior of the objects will be defined by describing their relationships and attributes using Objective-C/Cocoa Touch. Testing will be conducted very frequently throughout the development.

Design

The first phase in the development of any application is design. This is a very important phase because it provides a blueprint to be used throughout the rest of the development. During the design phase the developer will consider the intended users of the app, plan the storyline, sketch a storyboard, and then define the elements for the app by defining the objects and the relationships between the objects. So let's start designing the photo puzzle app. by defining a storyline and constructing a traditional storyboard

Storyline

For our Photo Puzzle app a user will move pieces of a photo into a display area so that the arrangement of the pieces displays the original image correctly—similar to putting a jigsaw puzzle together. The photo puzzle app will start with all of the puzzle pieces around the outside edges of the display area. The puzzle pieces will appear to the user as if they are arranged randomly. The user may then move pieces of the photo into a position inside the display area with their finger. The user's score will be increased each time a puzzle piece is placed correctly. If a puzzle piece is moved to an incorrect location, the score will not increase and the piece will animate back to its original placement to the outside of the display area. This continues until the puzzle pieces are all arranged so that the original photo is displayed correctly.

After defining a storyline, the next part of the design consists of creating a graphical representation of the user interface. This can be done several ways—possibly inside Microsoft PowerPoint, inside Photoshop, or simply by drawing it on a sheet of paper. The user interface of iOS apps must accommodate the physical display characteristics of the device that the app will be used on. This app will be designed for an iPhone display screen. The resolution for the screen on the iPhone 4 and iPhone 4S is 640 pixels by 960 pixels, and the resolution for other iPhone devices is 320 pixels by 480 pixels. Whatever the resolution of a device, the size is referred to as points, so the screen size for all iPhone devices is 320 by 480 points with the origin at the top left corner. The graphical representation of the user interface for our Photo Puzzle app is shown in Figure 1. Notice that this representation indicates that the puzzle will be displayed in landscape mode on the iPhone.

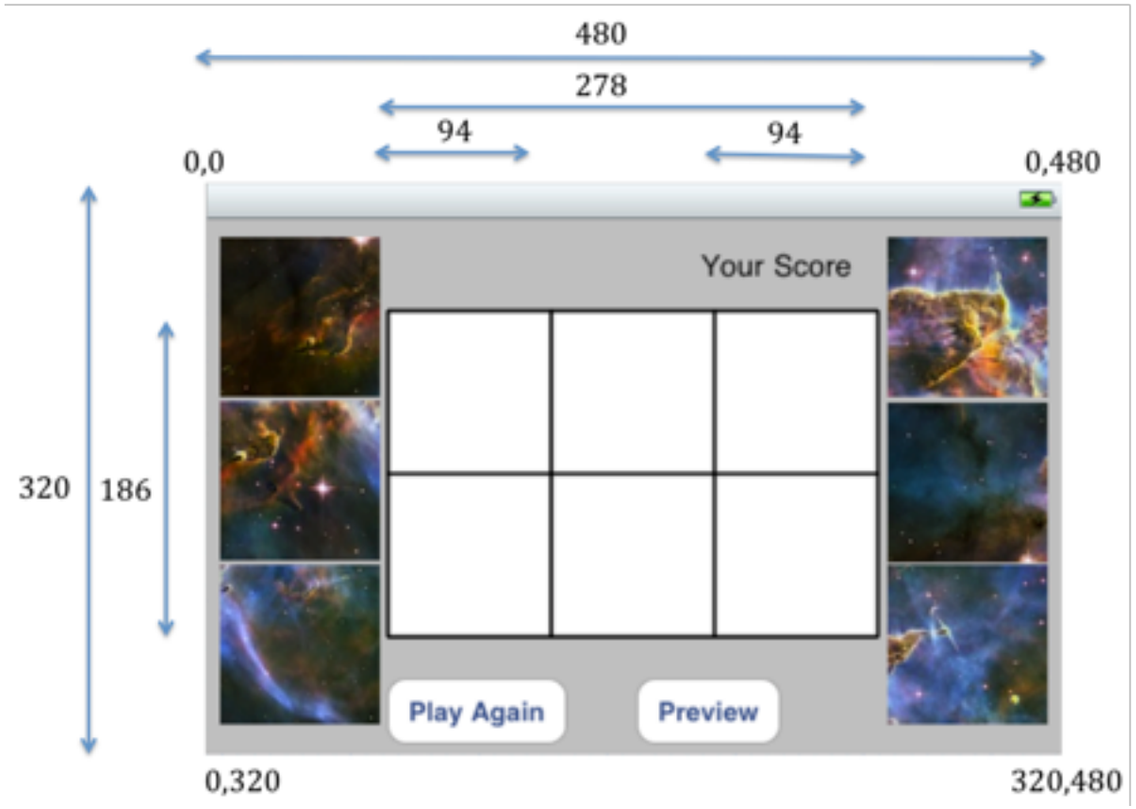

Figure 1 User interface for Photo Puzzle app

Figure 2 displays images that represent different screens during user interaction with the app. The app starts by launching a screen that has 6 puzzle pieces and a background. For this app we use an image resized to 270 points by 180 points and then divided into 6 puzzle pieces that are each 90 points by 90 points. Developers can use the image that we used and provided on the companion website for this book, or use an image of their own.

Our image was resized and tiled using Photoshop, however this can be done within many applications that include an image editor. The first screen in our storyboard is the puzzle board that will contain a label that displays the score once a game is started, and two buttons: a Play Again button and a Preview button. The Play Again button will allow users to solve the puzzle again. The Preview button will display the second screen that

displays the photo puzzle arranged correctly and also contains a Back button to return to the puzzle board. When the puzzle is solved, the user will be presented with the second screen.

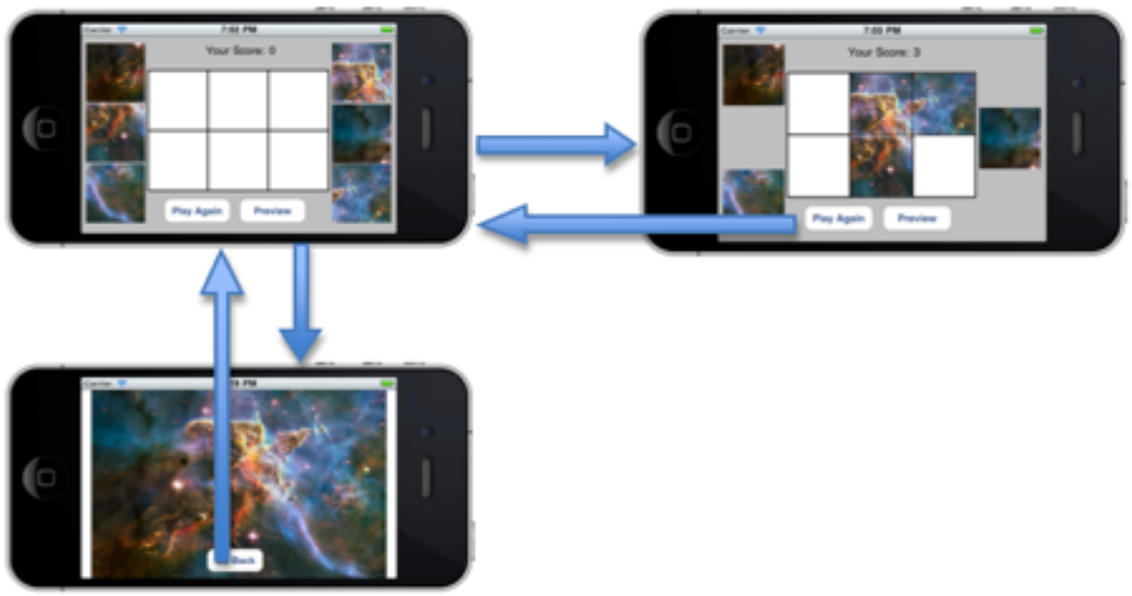

Figure 2 Photo Puzzle Game Flow

User Interface Elements

For this particular app, we can see within the storyboard that we need the following user interface elements for our app:

- A background image (grid)
- A set of 6 image pieces that serve as the puzzles pieces
- A label that will store/display the score

- The full image for the preview
- A Play Again button
- A Preview button
- A Back button

Once the elements are identified, the next step is to define the roles for the elements. In this app the roles of the elements were well defined within the storyboard.

- A background for the photo puzzle will enhance the visual appeal of the app.
- The 6 puzzle pieces will be movable from outside the completed puzzle area to their proper position within the original image.
- A label will display the score each time the puzzle app is played.
- A window with the original photo will provide a preview functionality for the app.
- A Play Again button will permit a user to build the puzzle again.
- A Preview button will display a screen with the puzzle pieces arranged properly.
- A Back button will take the user back to the main window with a puzzle in progress, and to the main window at the launch point of the app.

This completes the design phase of the photo puzzle app. We are now ready to proceed to the implementation phase.

Implementation

The implementation phase for the Photo Puzzle app will consist of (a) setting up the programming environment as needed to develop the app, (b) creating the user interface, (c) creating and connecting outlets for elements in the user interface, and (d) programming the touch events to provide the capability for moving the puzzle piece images.

Setting up the Programming Environment

For this project we will set up the programming environment as follows. Start Xcode and create a new Project. Xcode is often located at Macintosh HD > Developer > Applications > Xcode. Select Application in the iOS section in the left panel of the popup window, and also select the Single View Application template to the right, as shown in Figure 3, then click on the Next button. The project options window will appear.

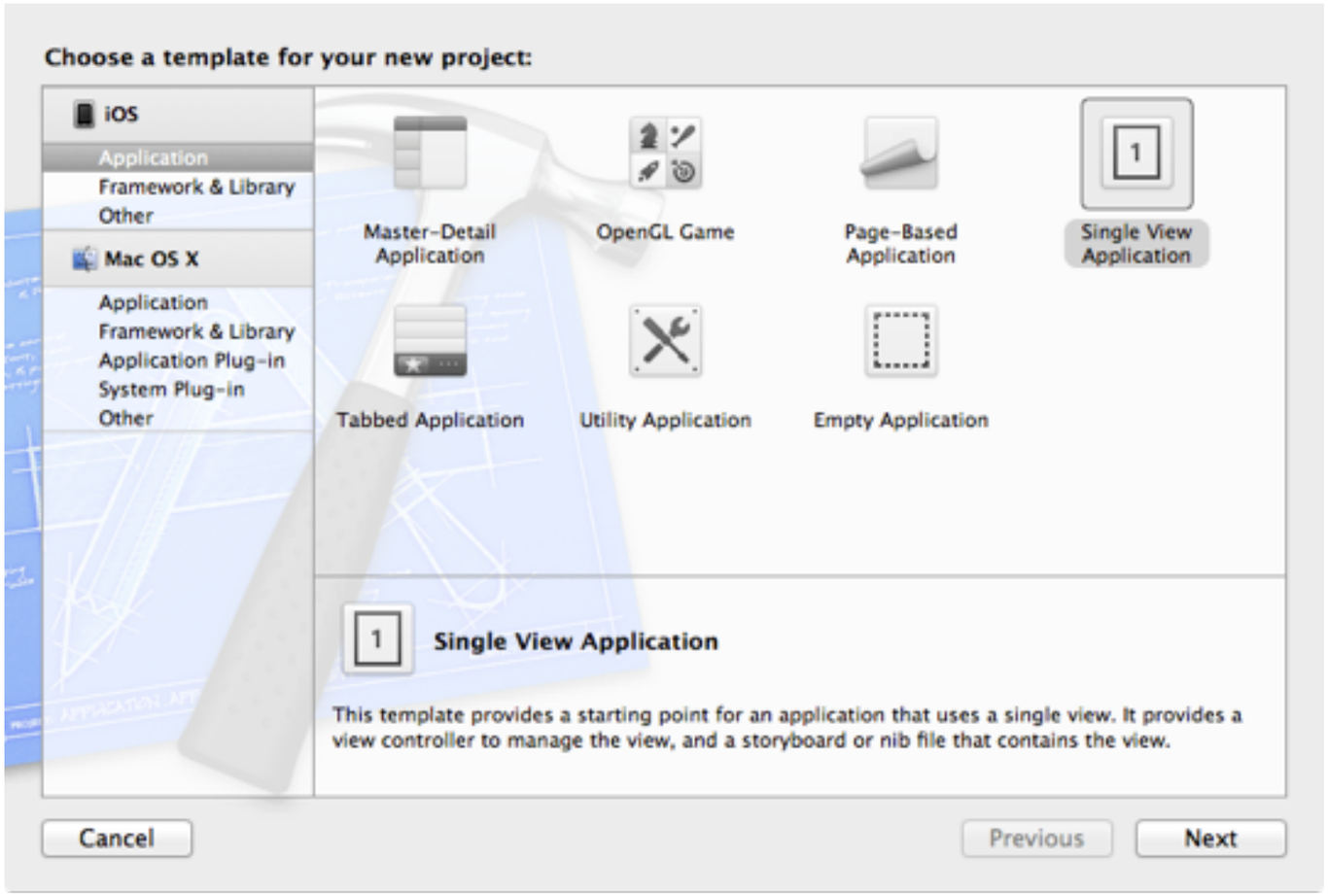

Figure 3 Creating a New Single View Application

Enter *Puzzles* into the Product Name text field as shown in Figure 4. You can use a different name but it will be easier to follow along with this project if you use the same name.

Also accept the default Company Identifier or enter *com*, select iPhone in the Device-Family menu, and select the Use Automatic Reference Counting checkbox as shown in Figure 4.

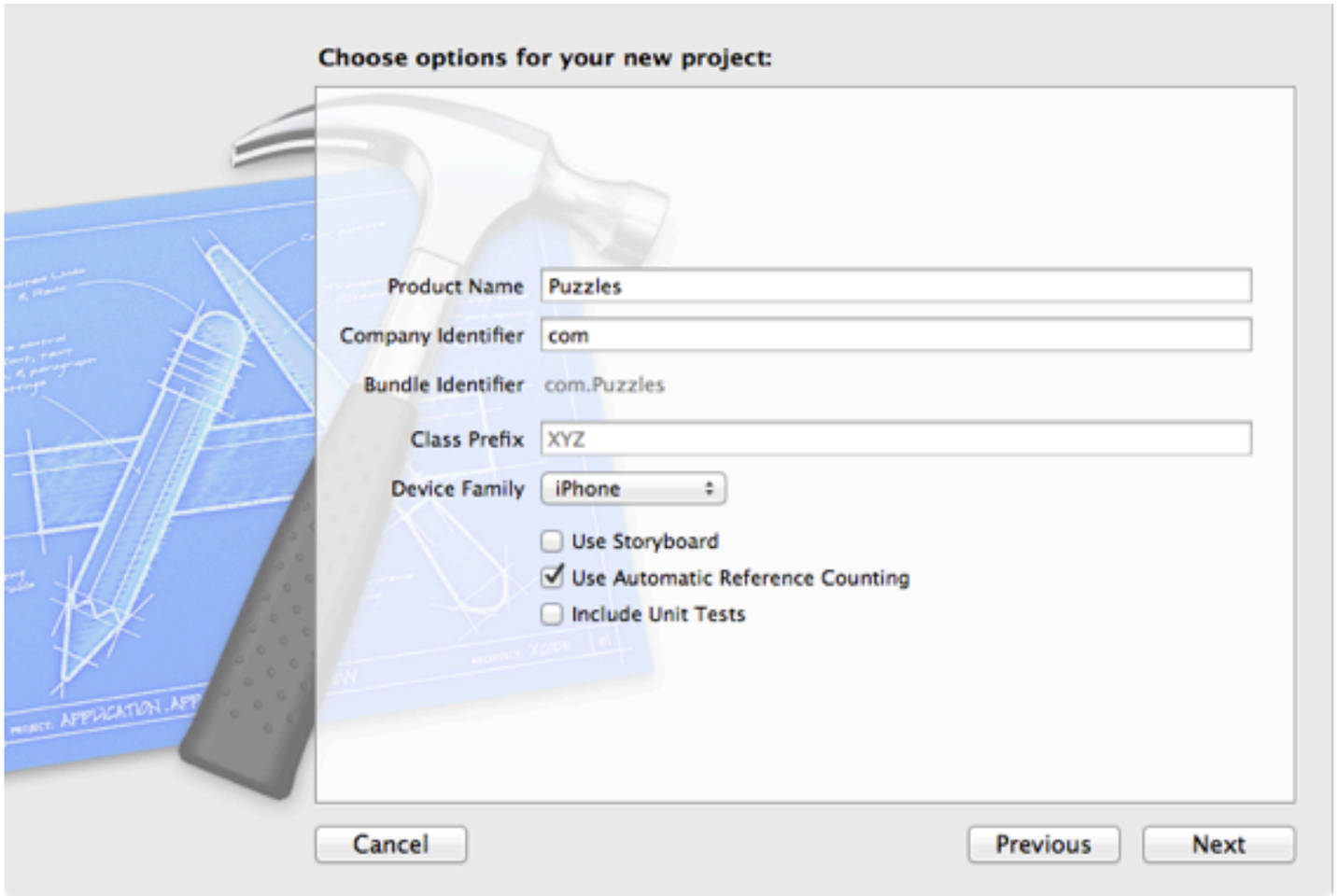

Figure 4 Creating a New Project

Click on the Next button. A new window will appear, to select the location for the project files. We will save ours on the Desktop. We are also provided with the opportunity to put the project under version control by selecting the Source Control checkbox at the bottom of the window. We will not do this for this project. Click on the Create button. In the new Xcode window ctrl-click on the Project name, Puzzles, in the Navigator area and then select *New Group* in the popup menu as shown in Figure 5.

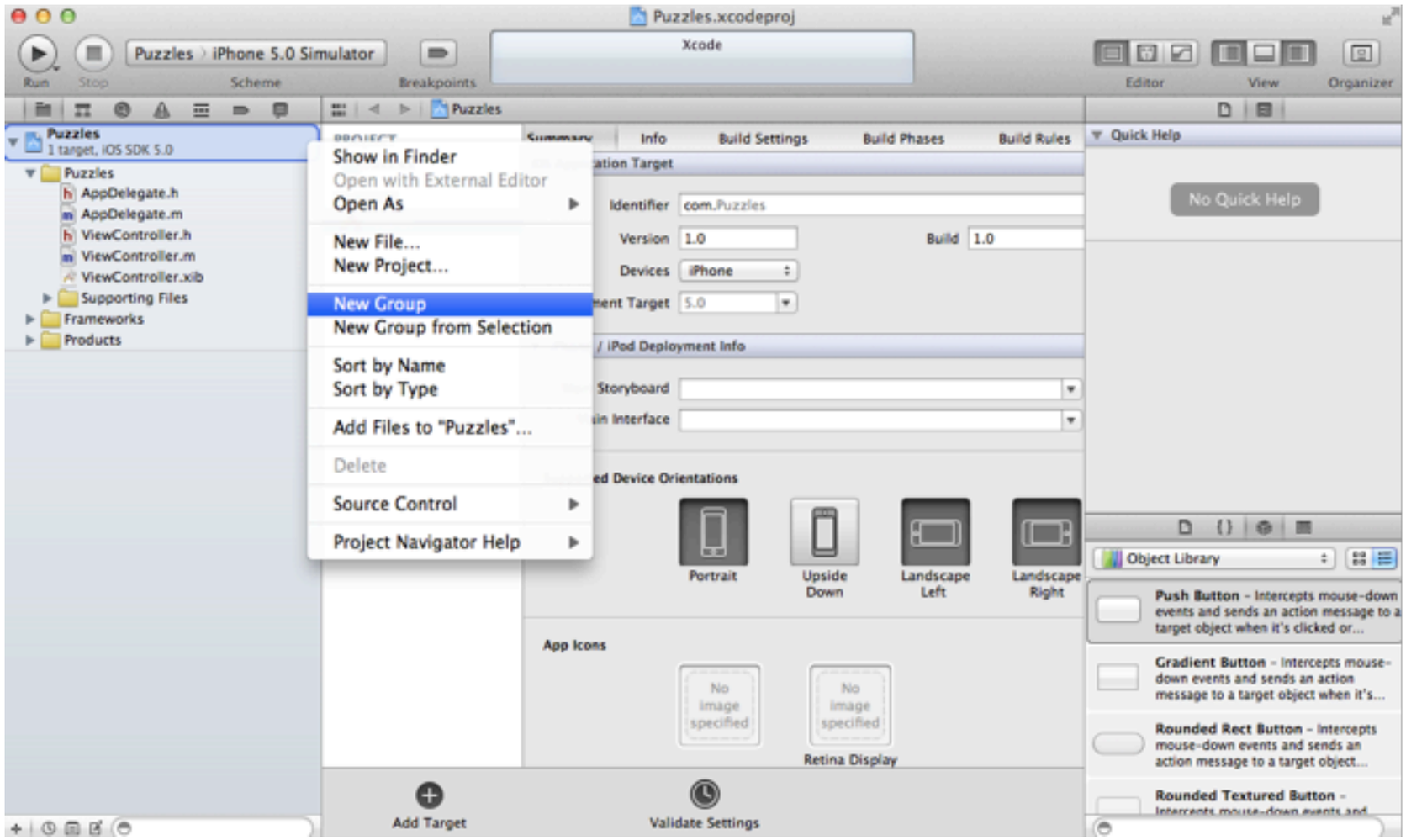

Figure 5 Creating a New Group

Click on the label for the *New Group* folder and change it's name to Resources. Open the Puzzle Images folder on the companion website for this book. Select all .png and .jpg images from the Puzzle Images folder and drag them to the newly created Resources group folder in Xcode--make sure that you have checked *Copy items into destination group's folder (if needed)* as shown in Figure 6. Click the Finish button, the image files should now be listed inside the Resources folder.

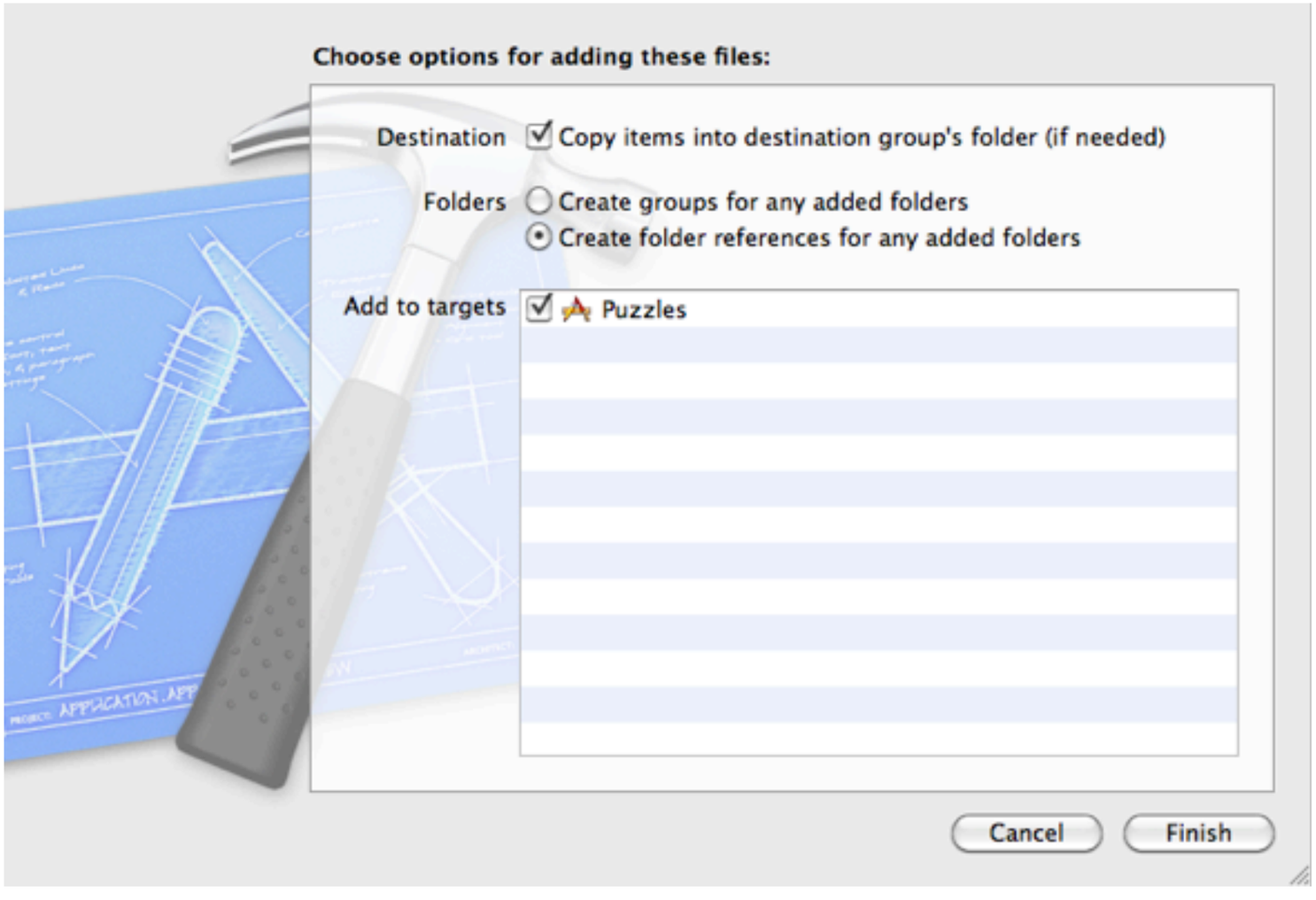

Figure 6 Setting Up the Puzzle Images

Creating a User Interface in Xcode

Now we will create a user interface for the Photo Puzzle app. First, select the ViewController.xib file in the Navigator area, and then navigate to View > Utilities > Show Attributes Inspector. Next, click on the large grey rectangle in the editor panel to select it. Your Xcode window should now appear similar to Figure 7.

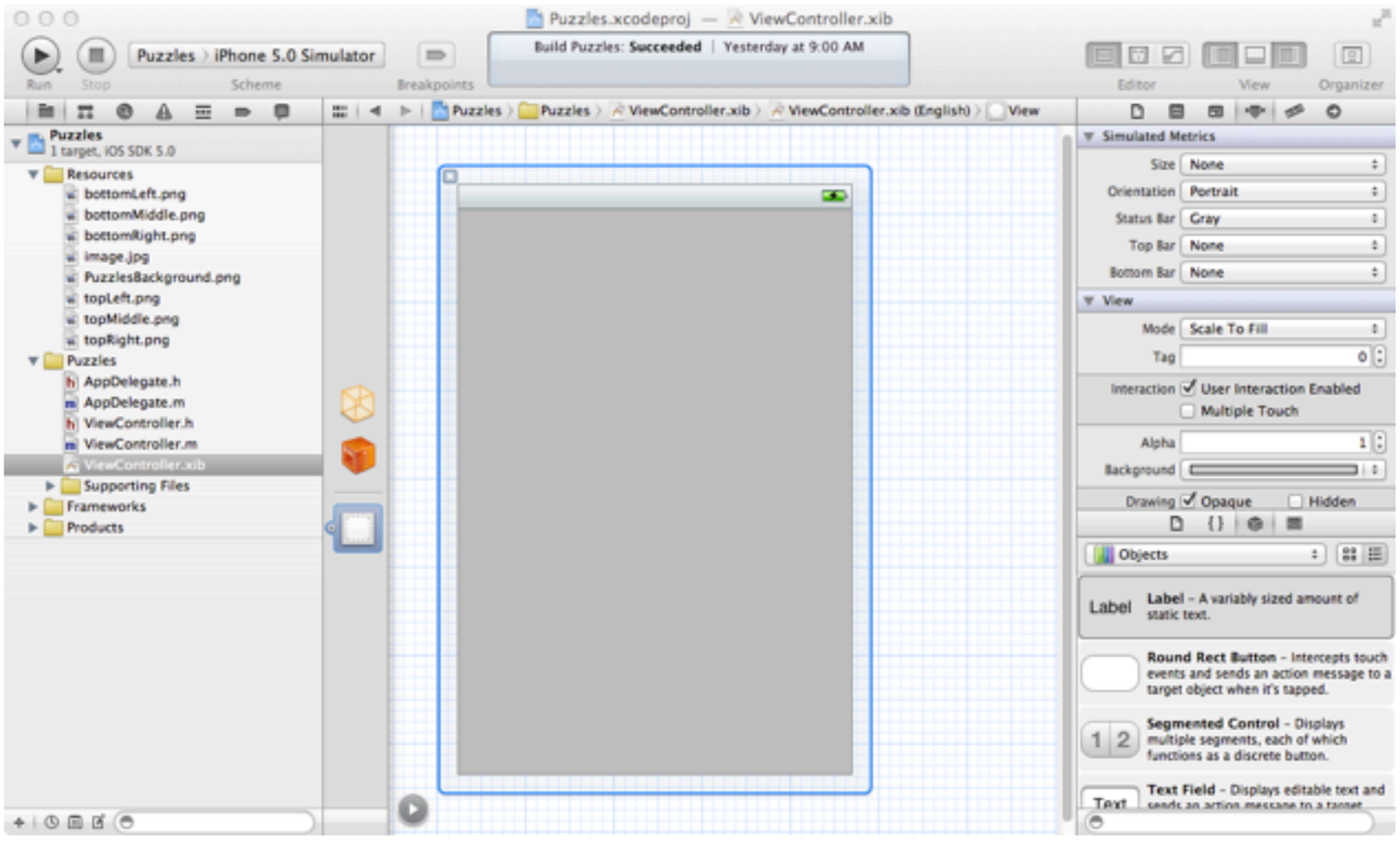

Figure 7 ViewController.xib File in the View Window

Look at the rightmost panel in Xcode, as shown in Figure 7 – this is the utilities panel. We will use the top part of this panel for editing the user interface elements' attributes. We will change the orientation of the View and add some user interface elements by using the bottom section that contains the library of objects for iOS applications. Notice the Simulated Metrics tab in the top of the Utilities panel as shown in Figure 7. The photo puzzle design used a landscape orientation, so click on the Orientation menu and then select landscape. Now the View should appear similar to Figure 8. Now we are ready to place user interface elements into the user interface. From the Library of objects toward the bottom of the Utilities area, find an Image View object as shown in Figure 9, and drag this object into the View window. Do this for a total of 7 times. Six of the image views will serve as puzzle pieces and one image view will serve as the background for the puzzle. Save your project.

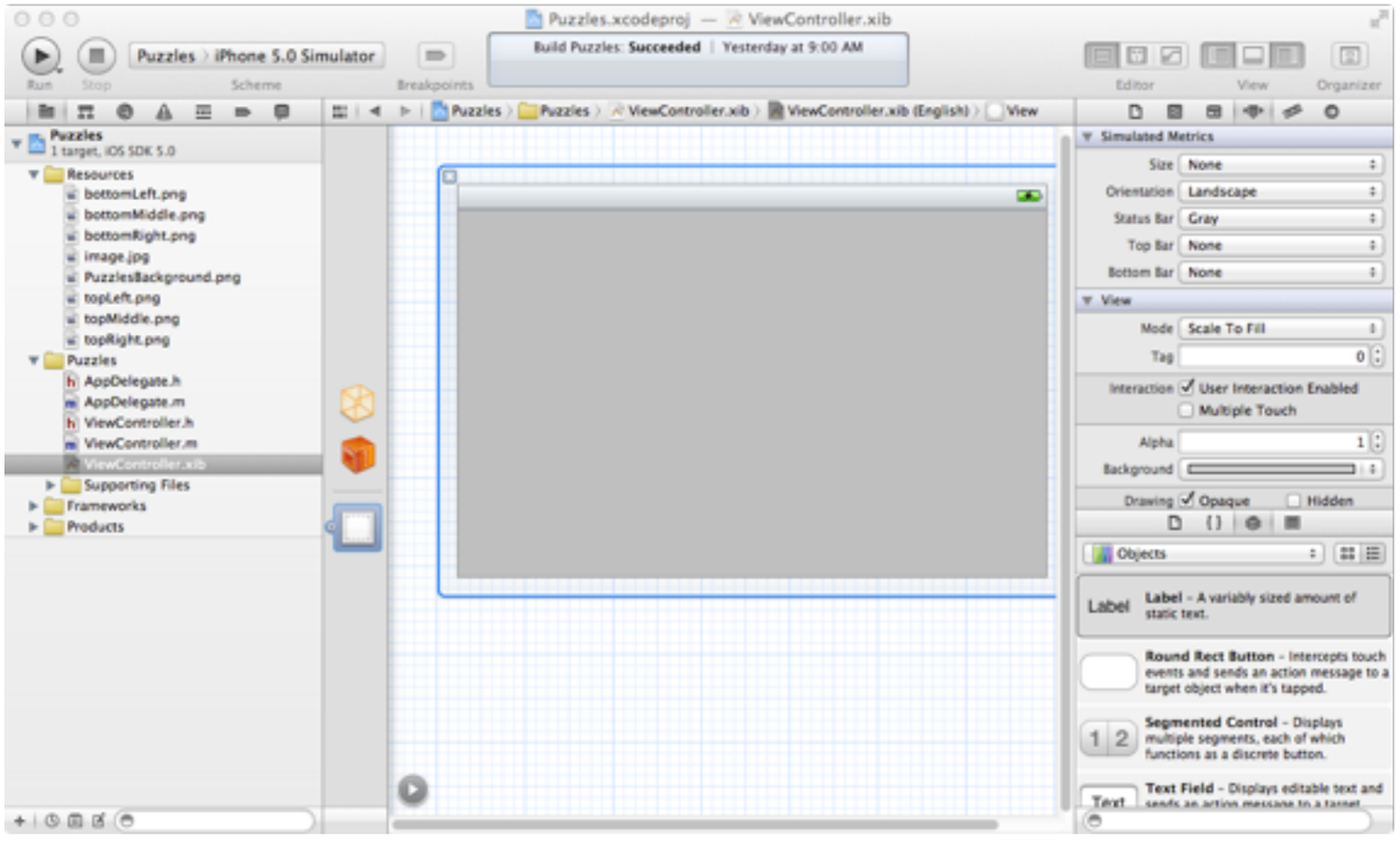

Figure 8 View in Landscape Orientation

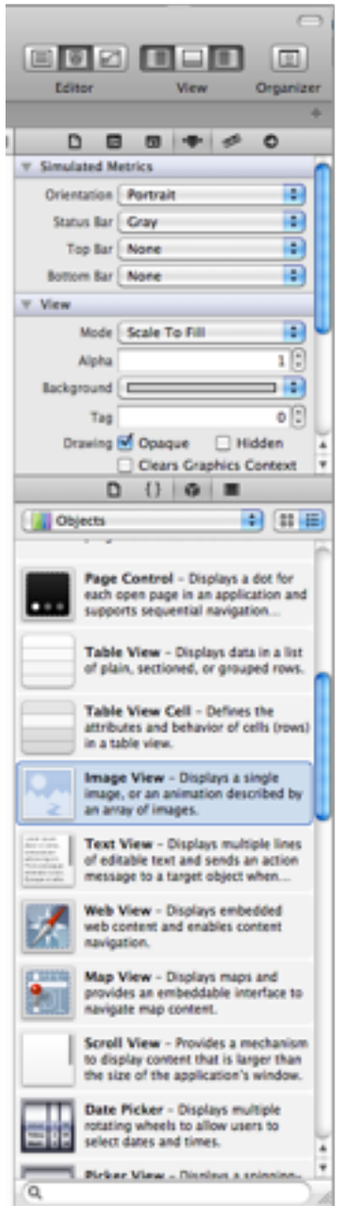

Figure 9 Utilities Pane

Resize the image views so that they all fit in the view window, the arrangement is not important at this point. The View window should appear similar to Figure 10. The label of the objects in the View window is the name of the class that these image view objects belong to, which is the UIImageView class.

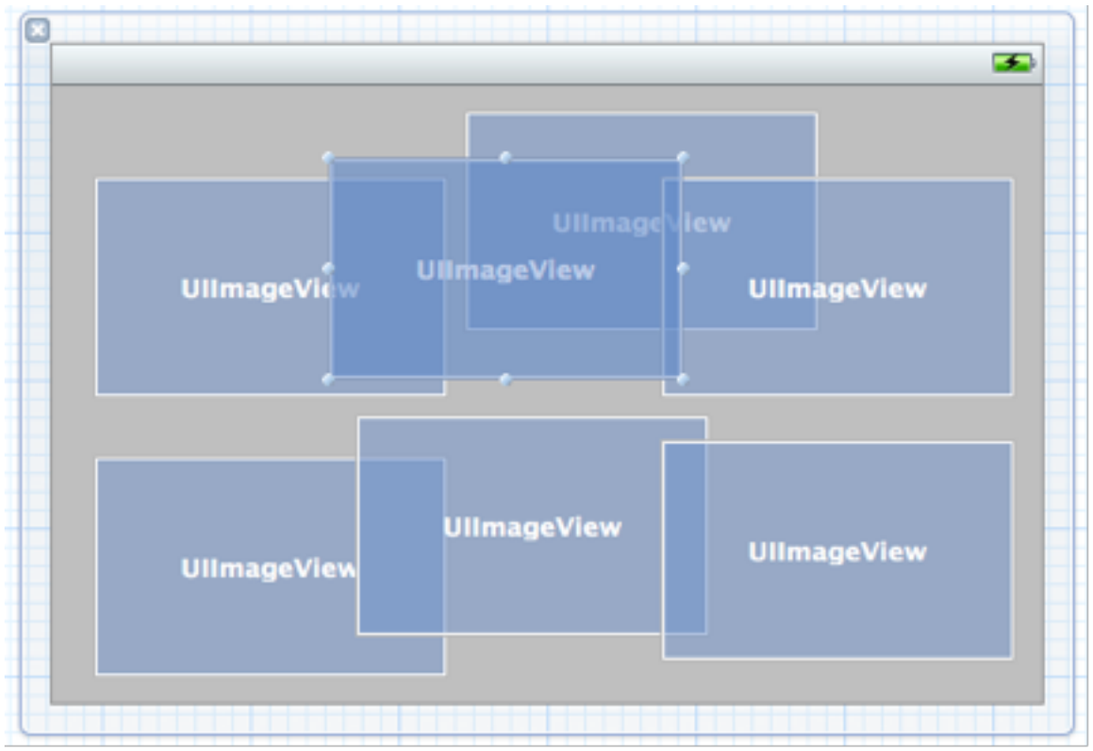

Figure 10 The View Window After Dragging 7 Image Views into it and Resizing the Views

Click on any one of the UIImageView objects in the View window. Now the top section of the Utilities area is the Image View Attributes section. Click on the *Image* drop down list and select topLeft.png, which is an image that will be the form the top left of the finished photo puzzle. Then within the View section of the Utilities panel change the mode attribute to Top Left. This will be essential later in our development. Also ensure that the User Interaction Enabled checkbox is checked, this is required so that users may interact with the image and be able to move it around the screen with their finger.

The Image View Attributes panel should appear similar to Figure 11 after the setting the attributes for the topLeft.png image.

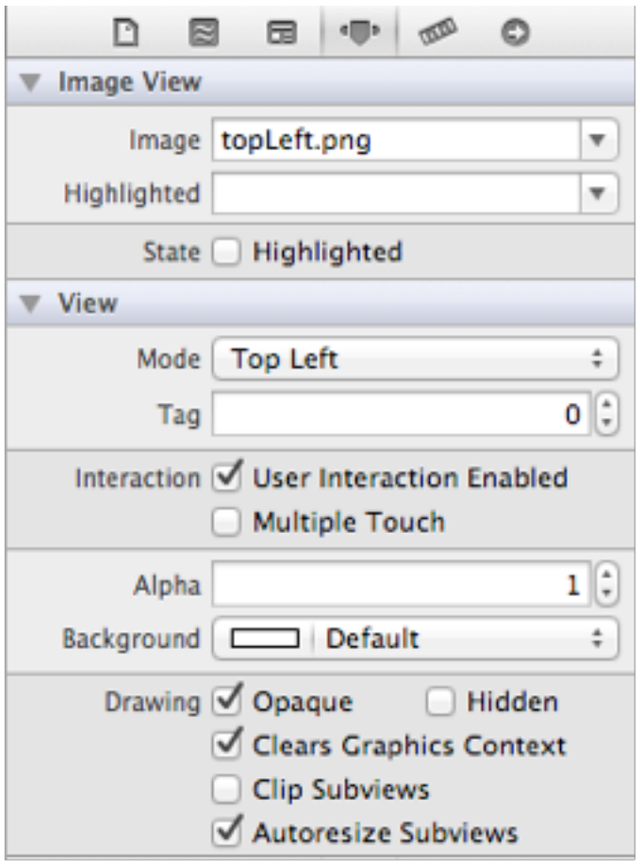

Figure 11 Image View Attributes

Select five different image views in the View window and set the image to topMiddle.png, topRight.png, bottomLeft.png, bottomMiddle.png, and bottomRight.png. Be sure to set the mode to Top Left and check the User Interaction Enabled checkbox for all of these images. Select the last unassigned image view and set it to the PuzzlesBackground.png image -- the mode for this image should be top left as was done with the other images, however, the User Interaction Enabled checkbox should not be checked since we will want to background to be stationary. Now the View should appear somewhat similar to Figure 12. Save your project.

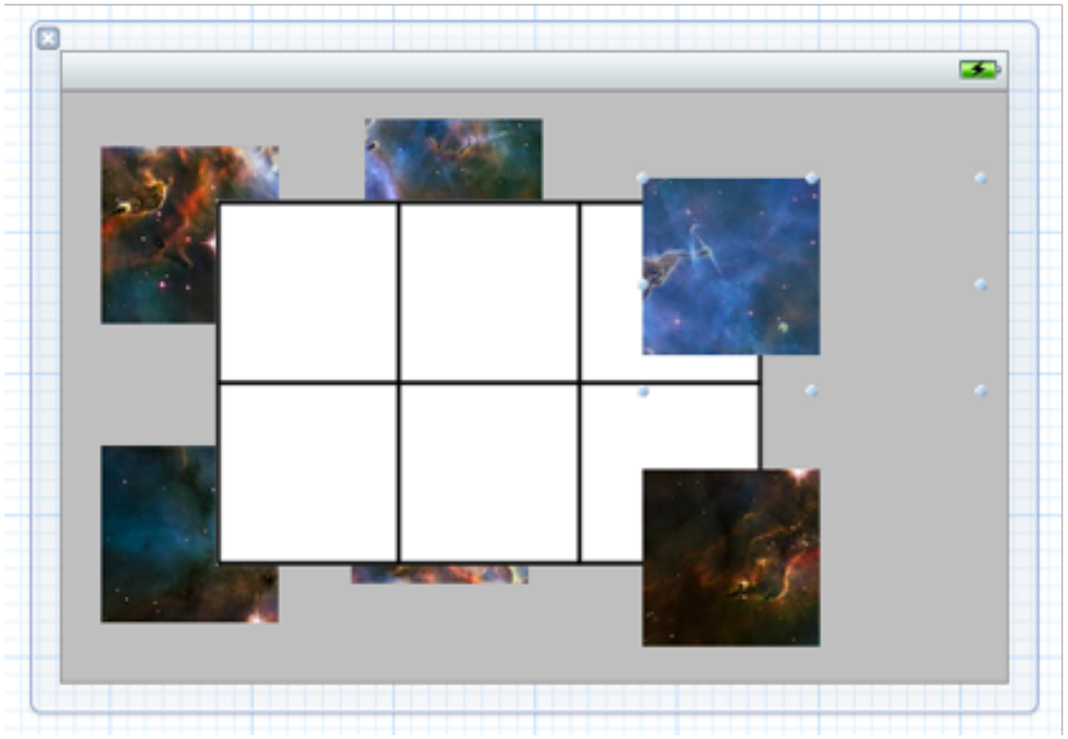

Figure 12 View with an Image Selected

Next, we will resize the Image Views. Select the View window, then open the Size Inspector by navigating to View > Utilities > Show Size Inspector in the Utilities area. First, click on the image that appears as a white grid – the background for the puzzle. Select the Frame Rectangle in the Show Menu. Enter coordinates of 100 and 50 into the X: and Y: text boxes respectively to position the background properly in the View, and also enter 278 for the width and 186 for the height into the Width and Height text boxes.

Then repeatedly click on the six puzzle pieces and change their sizes to 90 by 90; do not be concerned with the X: and Y: coordinates for the puzzle pieces at this time, we will set those coordinates for the puzzle pieces later. NOTE: It is very important to ensure that the Autosizing properties looks exactly as it does in Figure 13 with the top and left spacers in bright red and all the other spaces in very light red for each puzzle piece. Click on the autosizing spacers as needed to match the autosizing properties as shown in FIgure 13 for all of the puzzle pieces. When you have completed this task, then double check each puzzle piece again.

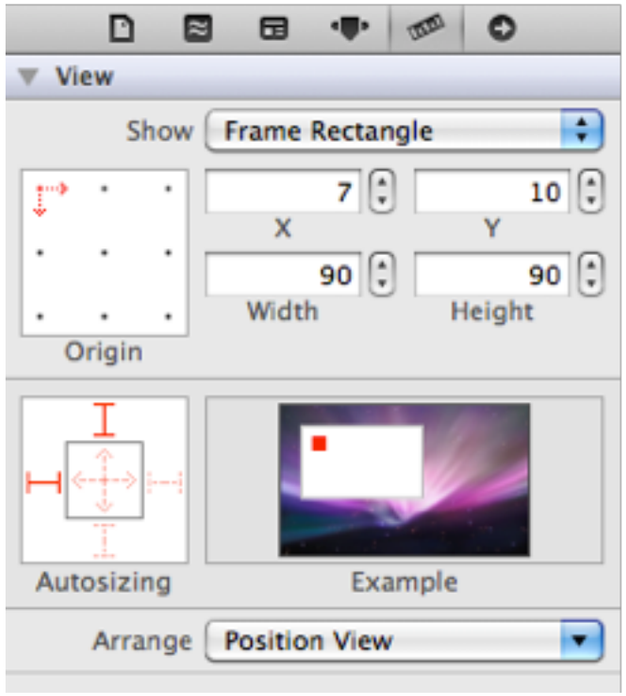

Figure 13 Image View Size Pane

Setting Up the View Hierarchy

Look again at the View window. Notice as shown in Figure 12 that some of the puzzle pieces might be partially hidden under the image that will serve as the background. If need be this can be fixed as follows. First change the view mode of the ViewController.xib—the window with the objects: File's Owner, First Responder, and View information by clicking on the gray Show Document Outline button in the bottom left of the Editor area as shown in Figure 8. In the new window that appears, expand the information for the View by clicking on the arrow to the left of the View button so that the expanded list elements within the View is visible as shown in Figure 14.

Notice that within Figure 14 that the file labeled *Image View – PuzzlesBackground.png* is the fifth element in the list. The order of elements is important. The order determines the topography or which elements are on top of other elements. The items at the top of the list are underneath in the View window and the items at the bottom of the list are on top of the others in the View. Since we want the puzzle background to be underneath

all of the other puzzle pieces we select it and drag it to the top of the list.Then all of the other images will be on the top of the PuzzlesBackground.png image view in our View window.

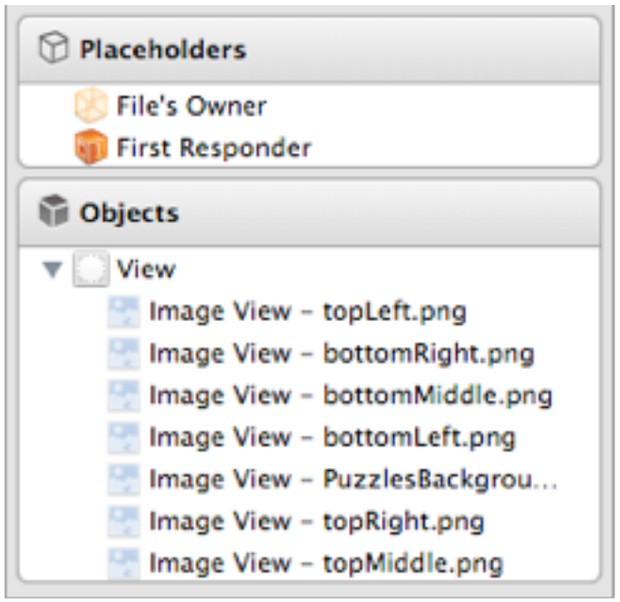

Figure 14 ViewController.xib File Contents

Adding Buttons and a Label

Recall that we also included two Buttons and a Score in the design for this app. We need a for *Play Again*, and *Preview* as described during the storyboard part of our design phase for this project. So drag two Round Rect Buttons and a Label from the Objects section of the Utilities area.

Double click on one of the Round Rect Buttons in the View window and type *Play Again*, then click on the other button in the View window and type *Preview*.

Double click on the Label object in the View window and replace the text with *Your Score*. The view window should now appear similar to Figure 15. This is a good time to save your work.

Good job, you have just created the user interface elements necessary to create the Photo Puzzle app.

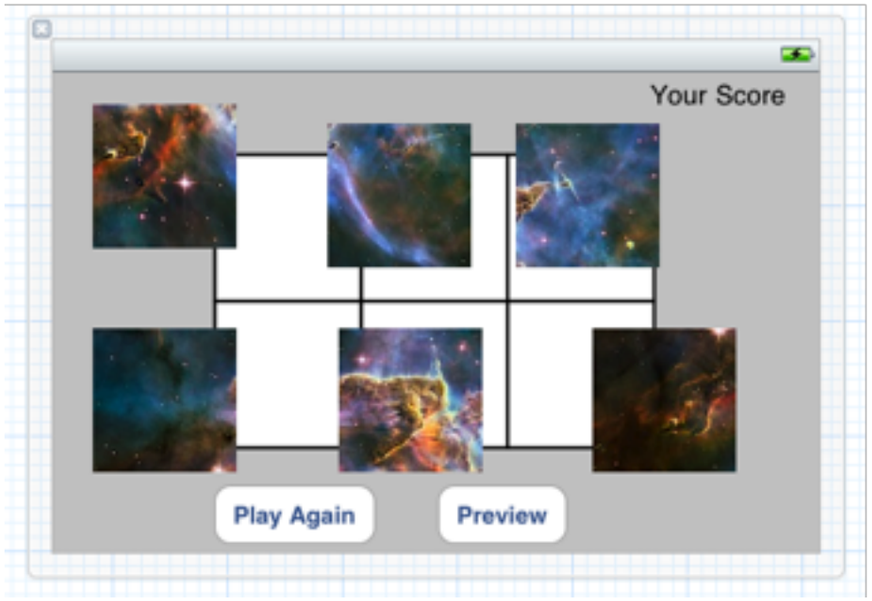

Figure 15 View Window

Creating Outlets

In order to use the images, buttons, and the label in the photo puzzle app, we need to establish connections between those elements in the user interface and the code that reacts to and/or manipulates those elements. We do this with outlets. IBOutlets are Xcode elements that implement these outlets. IBOutlet stands for Interface Builder Outlet. Recall that Interface Builder is integrated within Xcode.

Outlets can be created for many different kinds of user interface elements. The syntax for declaring IBOutlets for variables, labels, or images is slightly different.

For a variable: IBOutlet ClassOfVariable *nameOfTheVariable;

For a label: IBOutlet UILabel *scoreLabel;

For an image: IBOutlet UIImageView* imageView;

There are two ways of creating IBOutlets and making connections in Xcode. In earlier versions of Xcode the developer had to type the declaration of the IBOutlet in the interface part of the view controller and than make connections using Interface Builder. In Xcode 4.0 and later, IBOutlets can be created much more easily. Let's make a connection. Click on the ViewController.xib file in the Puzzles folder in Xcode; the editor should appear as shown in Figure 16

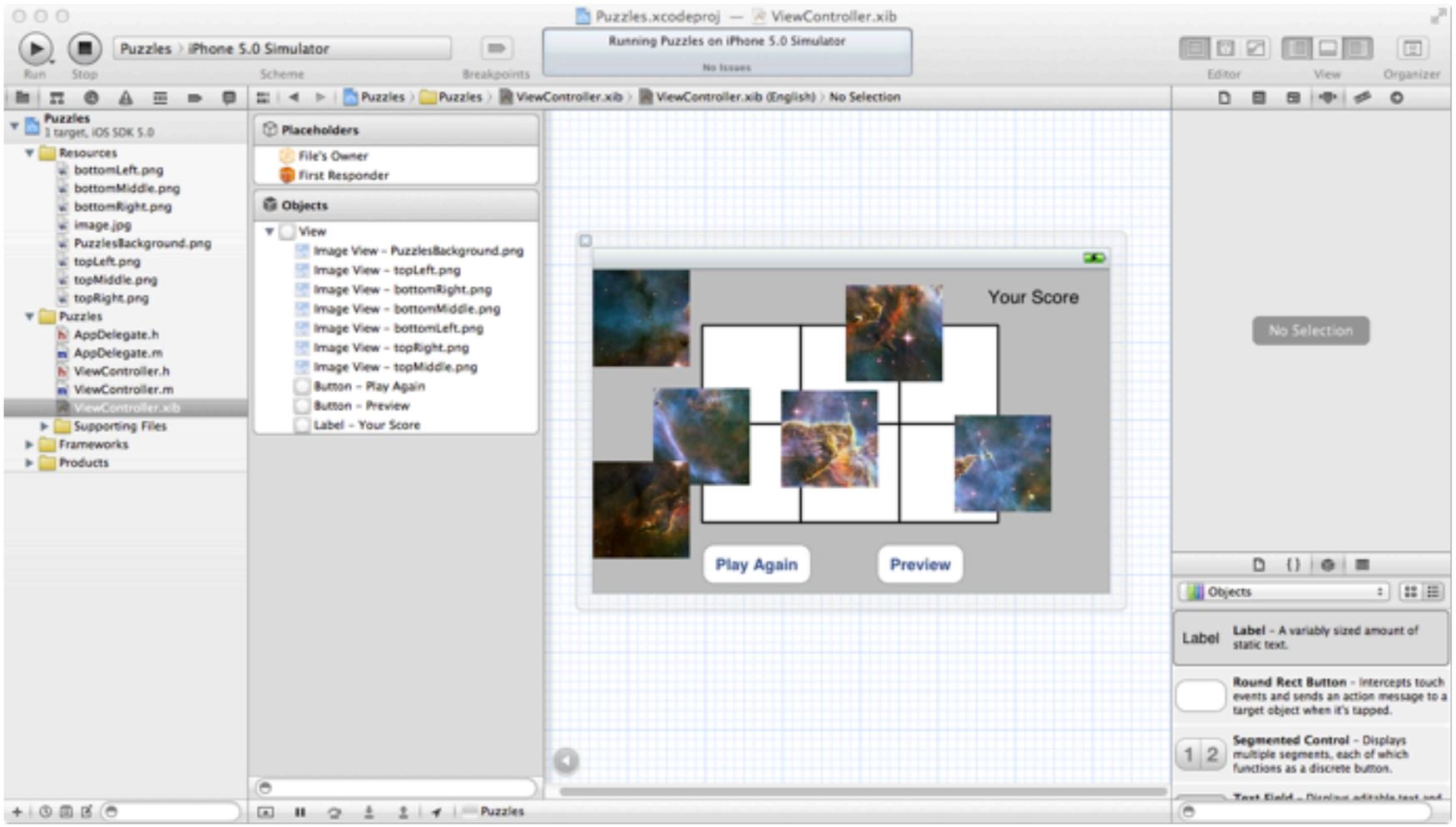

Figure 16 ViewController.xib File

Look at the vertical space between the Navigator area and the Editor area. Depending on the view mode you will see either a vertical panel with three icons as shown in Figure 8 and on the left of Figure 17 or the expanded view of the objects as shown in Figure 14 and on the right side of Figure 17. Whether we have the collapsed or the expanded view we see three primary objects. The first one is the File's Owner proxy. It represents the owner of the user interface. In this application, the File's Owner will be an instance of the ViewController class. Then we have the First Responder proxy. We will not use this proxy in the Photo Puzzle app, however, its purpose is to denote the first item in the responder chain for the app -- or simply which user interface element will respond to interactions on the screen. The third object in the window is the View proxy that contains the elements for the selected view., which we just examined in order to set the proper view hierarchy for the background of our app.

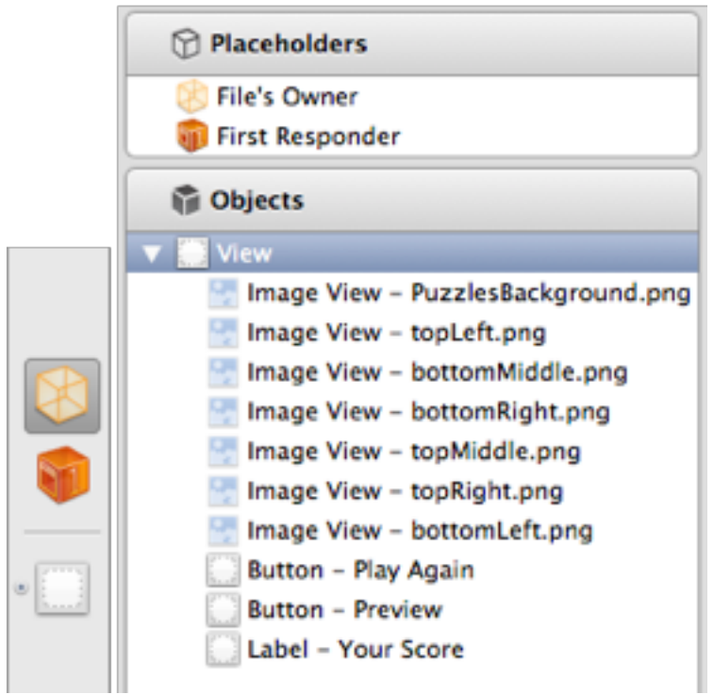

Figure 17 ViewController.xib File - Placeholders and Objects

To proceed with creating a connection, ctrl-click on the File's Owner icon. A File's Owner window as shown in Figure 18 with two Outlets—searchDisplayController and view—should appear. Notice that the View outlet is already connected to File's Owner by default. It informs the application that this is the view that will be displayed on the screen.

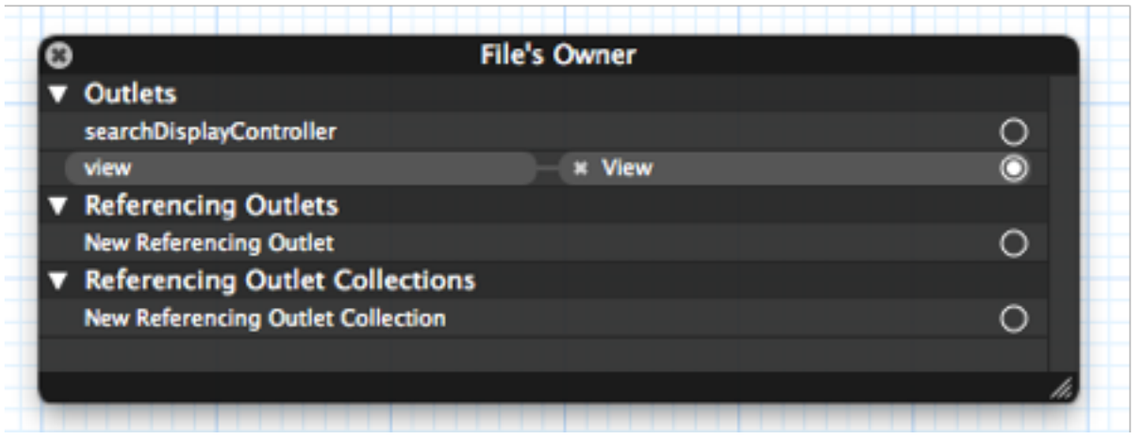

Figure 18 File's Owner Contents

Now click on the small gray button in the bottom left of the editor area to collapse the placeholders and view elements shown in Figure 17. We will also close the Utilities panel to provide more room for the Editor area. Navigate to View > Utilities > Hide Utilities. Then navigate to View > Assistant Editor > Show Assistant Editor to open a second editor window inside Xcode. As described previously in this book, when the Assistant Editor is opened it preloads the new editor window with a file related to the file that was already open in the Editor area. Most likely, the ViewController.h file will be displayed in the Assistant Editor as shown in Figure 19. If it is not, then select the ViewController.h. file in the left panel of Xcode.

Now we can start making our connections. Ctrl-click on one of the Image Views that contains a puzzle piece in the View window. A small black Image View window will popup as shown in Figure 20.

Notice the name of the image in the top bar of the window. In our case, we selected the topLeft.png image. Click on and drag from the little circle to the far right of the New Referencing Outlet so that the blue elastic band appears. Drag into the ViewController.h between the interface directive and the the end directive as shown in Figure 21.

When you release the mouse a new connection dialog popup will appear similar to that shown in Figure 22. Enter a name for the connection between the image and the code. We entered the first part of the file name for our connection, which in this case is topLeft, and then click on the Connect button.

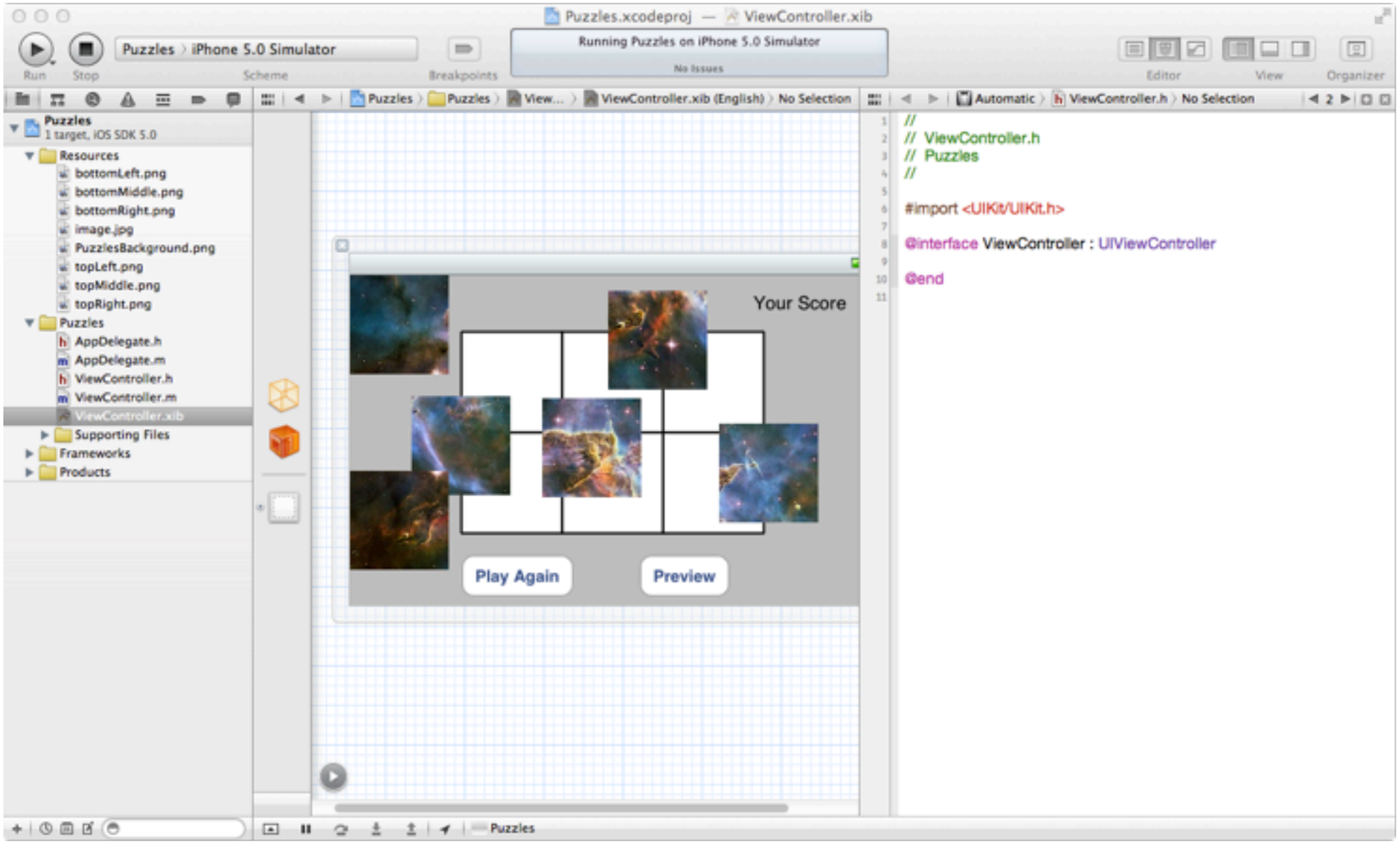

Figure 19 Preparing for Adding Outlets - xib and .h Files Open in Xcode Editor area

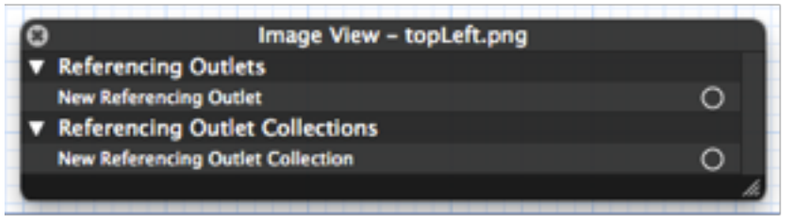

Figure 20 Image View Referencing Outlets

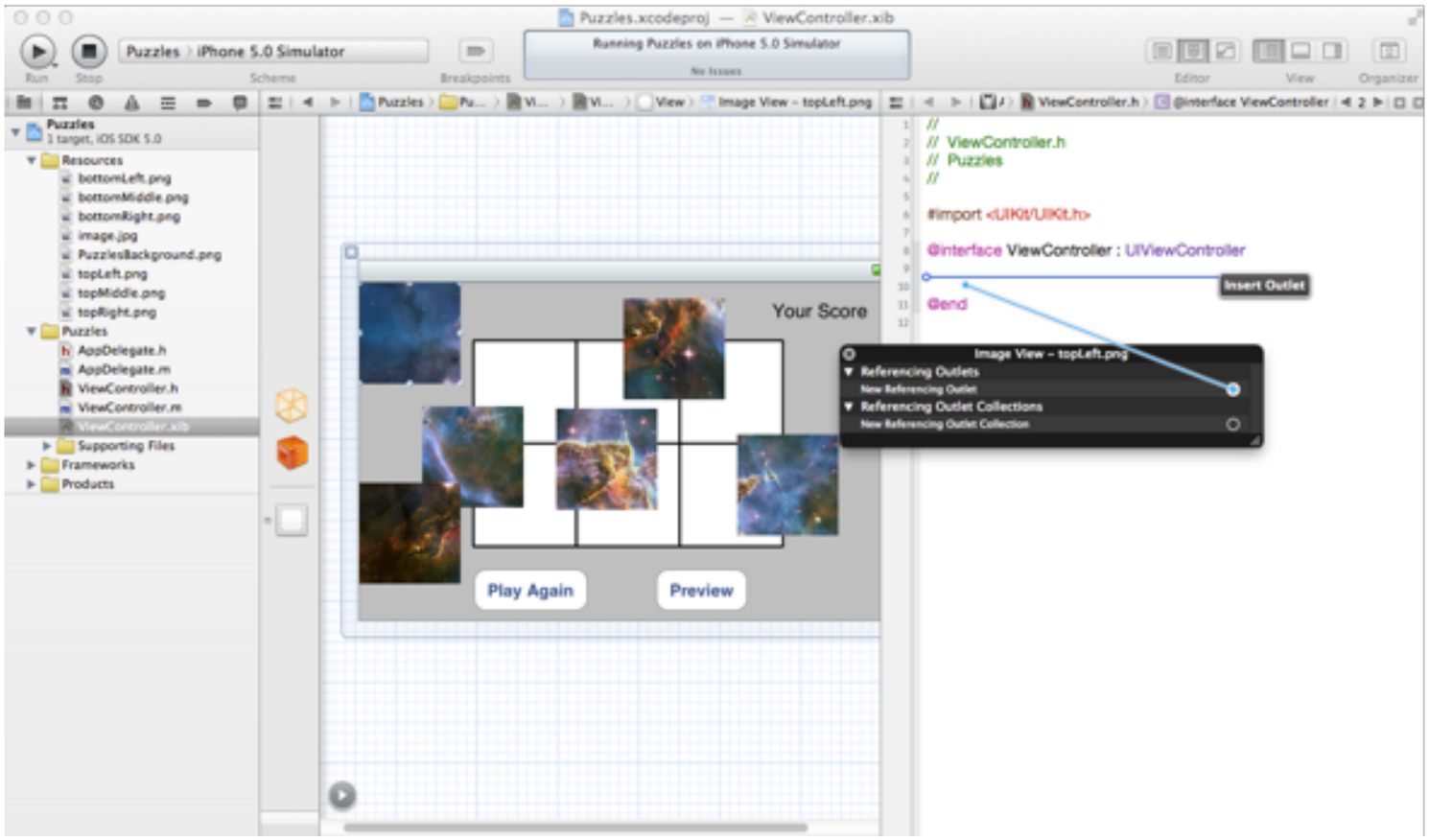

Figure 21 Connecting an Image View to the ViewController.h file

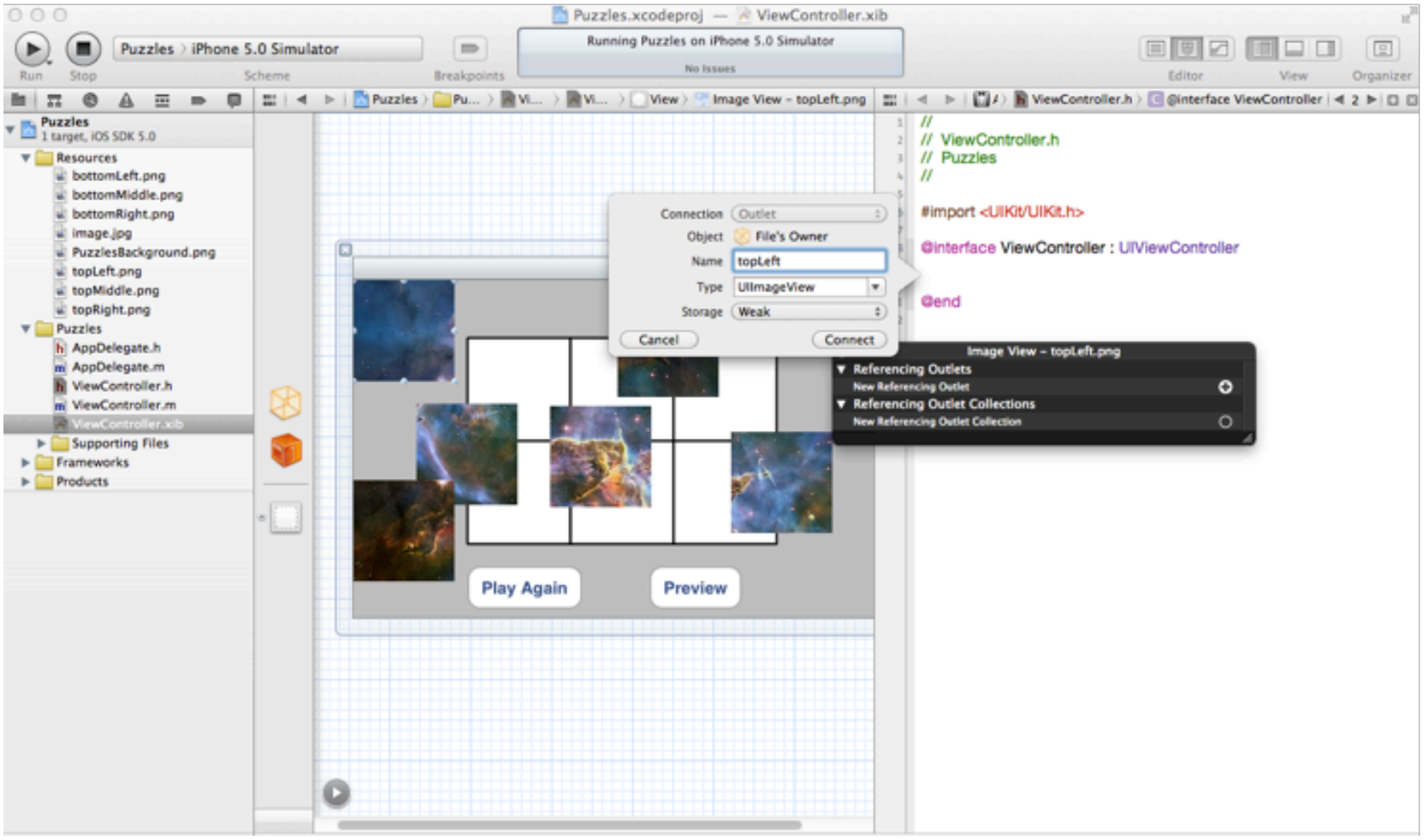

Figure 22 Naming the Image View Connection

Notice that the contents of the ViewController.h file has been modified. As a result of making the connection Xcode generated the code necessary to create an IBOutlet and its property for the image.

```
//
//  ViewController.h
//  Puzzles
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIImageView *topLeft;

@end
```

Now you might ask, what is a property? Properties are elements in Objective-C that are used when creating *accessor* and *mutator* methods to get the values from objects and to set the values in objects. In this case each puzzle piece is an image object with values that can be accessed or modified. In this app, properties are declared in the interface part of the ViewController class—the .h file. When properties are synthesized in the implementation part of the ViewController class--the .m file the accessor and mutator methods are created and code related to memory management is added to methods in the .m file. Since we selected the checkbox for Automatic Reference Counting when we created this project, the memory management will be handled automatically for us. Xcode automatically added the synthesize directive for the property at the top of the ViewController.m file and also set the property to nil in the viewDidUnload method in the ViewController.m file.

```
@synthesize topLeft;

[self setTopLeft:nil];
```

If the developer prefers, these statements can be manually inserted in the .h and .m files without using the elements in the xib file.

Repeat the process of creating and connecting outlets shown in Figures 20 through 22 for all of the rest of the puzzle piece images. That is, ctrl-click on each remaining Image View in the View window except the background image, then drag from the empty circle to the right of the New Referencing Outlet into the popup window just before the end directive. Enter a name for each new outlet connection that corresponds to the name of the each image such as topMiddle, topRight, bottomLeft, bottomMiddle, or bottomRight.

Next, create an outlet for the score label. Ctrl-click on the label in the View window and drag from the empty circle to the far right of the New Referencing Outlet to the ViewController.h file just above the end directive. Type scoreLabel for the name of this new Outlet Connection.

After making the connections look at both of the ViewController files. They should appear similar to the following code listings.

```
//
//  ViewController.h
//  Puzzles
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet UIImageView *topLeft;
@property (weak, nonatomic) IBOutlet UIImageView *topMiddle;
@property (weak, nonatomic) IBOutlet UIImageView *topRight;
@property (weak, nonatomic) IBOutlet UIImageView *bottomLeft;
@property (weak, nonatomic) IBOutlet UIImageView *bottomMiddle;
@property (weak, nonatomic) IBOutlet UIImageView *bottomRight;
@property (weak, nonatomic) IBOutlet UILabel *scoreLabel;
@end
```

```
//
//  ViewController.m
//  Puzzles
//

#import "ViewController.h"

@implementation ViewController
@synthesize topLeft;
@synthesize topMiddle;
@synthesize topRight;
@synthesize bottomLeft;
@synthesize bottomMiddle;
@synthesize bottomRight;
```

```
@synthesize scoreLabel;

- (void)didReceiveMemoryWarning
{
    [super didReceiveMemoryWarning];
    // Release any cached data, images, etc that aren't in use.
}

#pragma mark - View lifecycle

- (void)viewDidLoad
{
    [super viewDidLoad];
        // Do any additional setup after loading the view, typically from a nib.
}

- (void)viewDidUnload
{
    [self setTopLeft:nil];
    [self setTopMiddle:nil];
    [self setTopRight:nil];
    [self setBottomLeft:nil];
    [self setBottomMiddle:nil];
    [self setBottomRight:nil];
    [self setScoreLabel:nil];
    [super viewDidUnload];
    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}

- (void)viewWillAppear:(BOOL)animated
{
    [super viewWillAppear:animated];
}

- (void)viewDidAppear:(BOOL)animated
{
    [super viewDidAppear:animated];
}
```

```
- (void)viewWillDisappear:(BOOL)animated
{
        [super viewWillDisappear:animated];
}

- (void)viewDidDisappear:(BOOL)animated
{
        [super viewDidDisappear:animated];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}

@end
```

Checking Connections

Just to make sure that everything is correct, ctrl-click on the File's Owner button to the left of the Editor area to view all of the Outlet connections. The Outlets should appear similar to Figure 21. If your File's Owner outlets look just like Figure.23, good job! If not, redo the previous process for any Image Views that are not connected or are connected incorrectly. Before you make changes to any outlet connections, click on the little x icon to the left of the Image View – *name* to remove an incorrect connection before attempting to recreate the connection.

KEY POINT

The use of outlets is a very important concept in iOS Programming. An outlet is a variable that points to another object so that two objects in an application will be able to communicate at runtime. To use an outlet the following is needed:

1) Drag the user interface element from the Object Library to the View window

2) Ctrl-click on that element

3) Drag from the circle to the far right of the New Referencing Outlet to the just after the interface part of the class that is the owner of the view. If your view file is called className.xib you can create Outlets only in the className.h file not in someOtherClassName.h file.

4) Enter a name for the Outlet

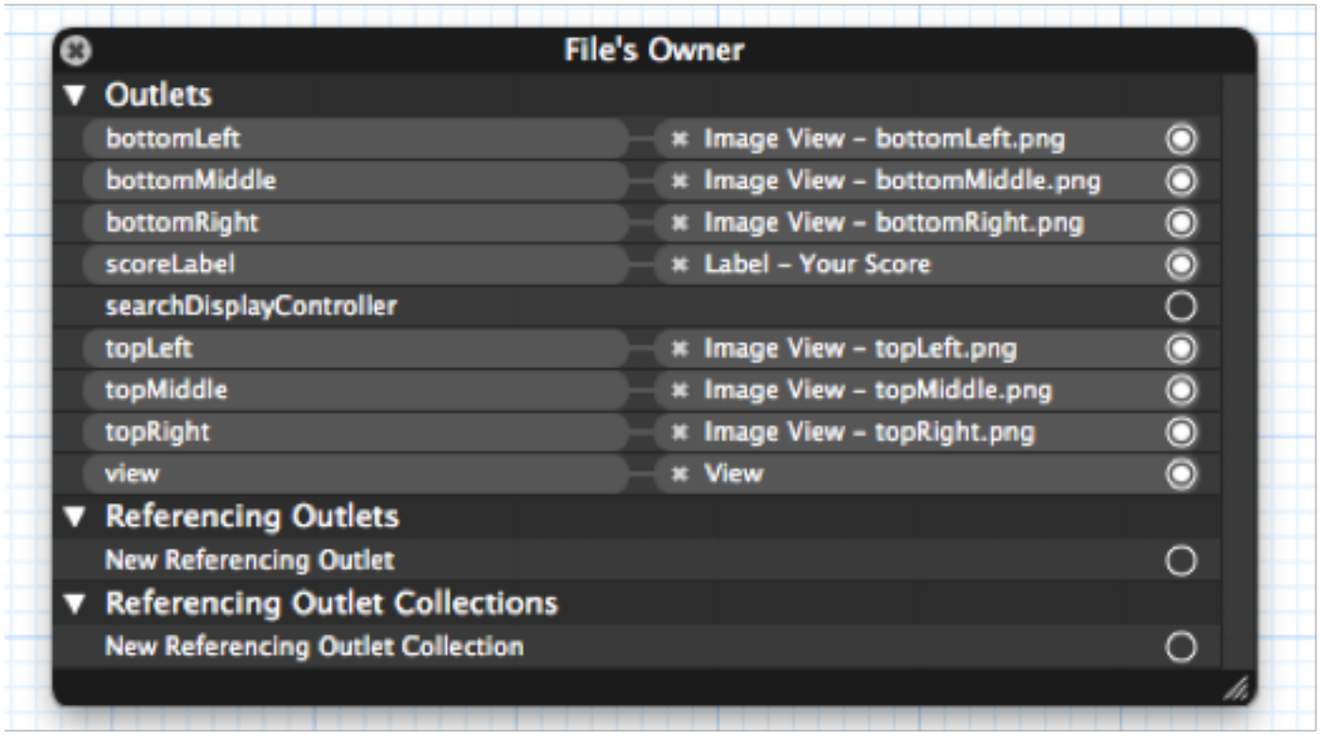

Figure 23 File's Owner Outlets

Congratulations! Be sure to save your project at this point, and click on the Run button in the upper right of your Xcode window to run this app in the iPhone simulator, as shown in Figure 24.

Figure 24 iPhone Simulation of the Photo Puzzle App

Something is wrong! We don't see the same arrangement for our user interface in the simulator as we did inside Xcode. The reason for this is that by default iPhone programs are displayed in Portrait Mode. Just as we had to specify the use of landscape mode for our View inside Xcode. The ViewController class created by Xcode contains several methods that are typical for projects using the Single View Application, that we selected when we created this project. One of those methods is the shouldAutorotateToInterfaceOrientation method. Scroll down in the ViewController.m file to find this method. Notice that the return type of the method is BOOL and that the method contains a return statement that supports all orientations except the portrait orientation upside down.

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}
```

To get desired interface orientations we change the right hand side of the equivalence operation that determines the value returned from this method by this statement. Presently the return value is YES only for Portrait. There are four four different orientations for iOS devices:

- UIInterfaceOrientationPortrait -- Vertical orientation with the Home button on the bottom
- UIInterfaceOrientationPortraitUpsideDown -- Vertical orientation with the Home button at the top
- UIInterfaceOrientationLandscapeLeft -- Horizontal orientation with the Home button on the left
- UIInterfaceOrientationLandscapeRight -- Horizontal orientation with the Home button on the right

For the photo puzzle app, we only want to display the photo puzzle in landscape orientation, so we will edit the return statement so that it uses the equivalence operator rather than the not equivalent to operator and so that the UIInterfaceOrientationLandscapeLeft is on the right side of the equivalence operation-- then the method appears as follows.

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
}
```

If we wanted to support two orientations such as LandscapeLeft and LandscapeRight we could use the logical OR operation with the appropriate values on the right side of the equivalence operator as in:

```
return ((interfaceOrientation == UIInterfaceOrientationLandscapeLeft) || (interfaceOrientation == UIInterfaceOrientationLandscapeRight));
```

If we wanted to apply all interface orientations simultaneously for an app, we could simply return YES as in:

```
 return YES;
```

Save your project and then click on the Run button to test your program in the Simulator again. Note that if you did not stop the simulation after the last time you ran the program a window similar to Figure 25 might appear. Simply click on Stop to close the window. You may wish to select the Do not show this message again checkbox so that this message won't continue to appear every time the Run button is clicked. Your app should now appear in the Simulator similar to Figure 26. Of course, the images cannot be moved yet but the user interface should now appear as expected.

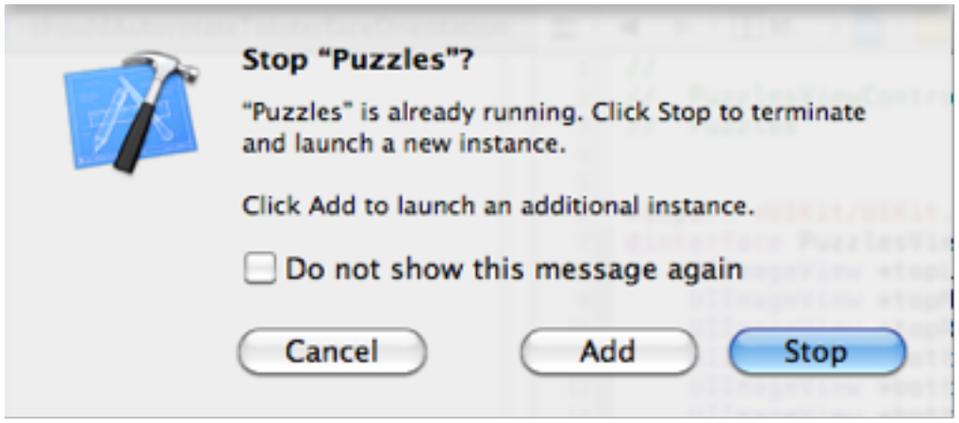

Figure 25 Xcode Stop Running Popup

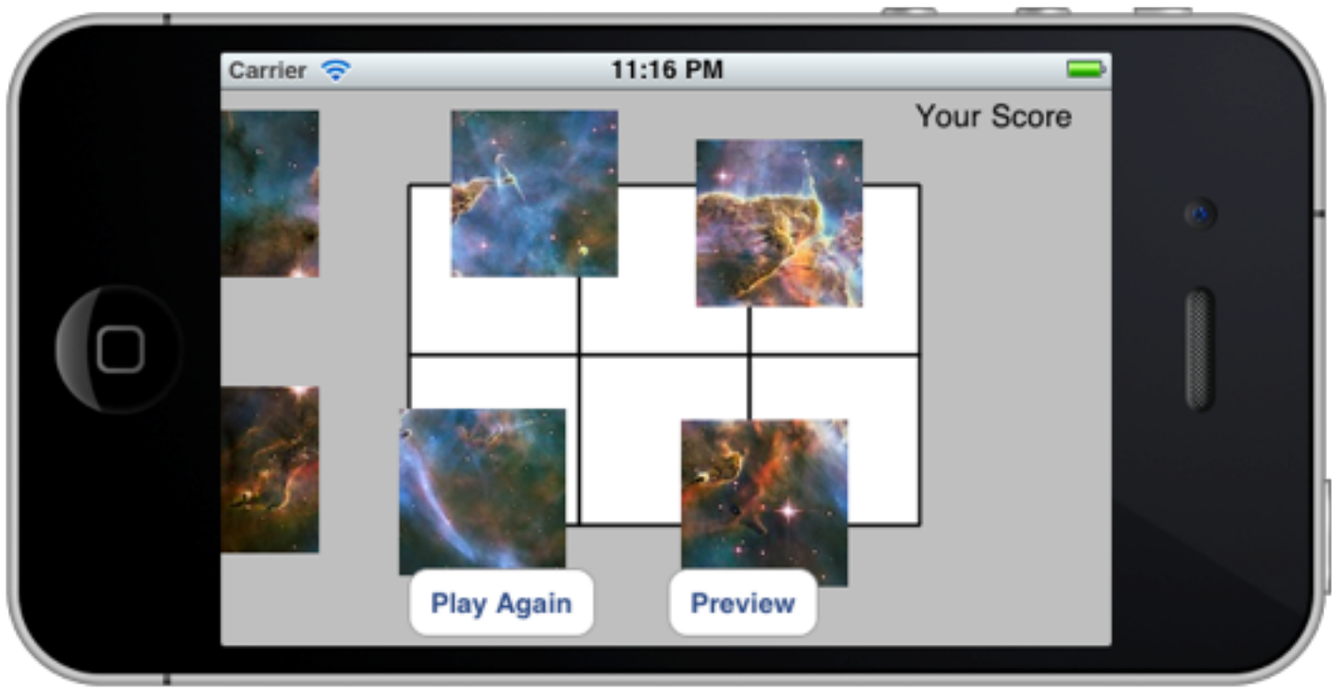

Figure 26 Photo Puzzle in the iPhone Simulator

Touch Events -- Moving Images on the Screen

In this phase of the implementation we will make the images move with our finger. To do this, it is important to understand the concept of touch events. A touch event is a type of event that is represented by a UIEvent object. Other instances of UIEvents can represent gestures and the shake motion. Events are instances of the UIEvent class that *store* information about an object can trigger an event—it stores information about all touches associated with the event. For example, when a user touches the screen, the system recognizes that as a UIEvent object of type UIEventTypeTouches and places it in the application event queue. Figure 27 illustrates the relationships between a user touching the iPhone and the underlying system that handles these events.

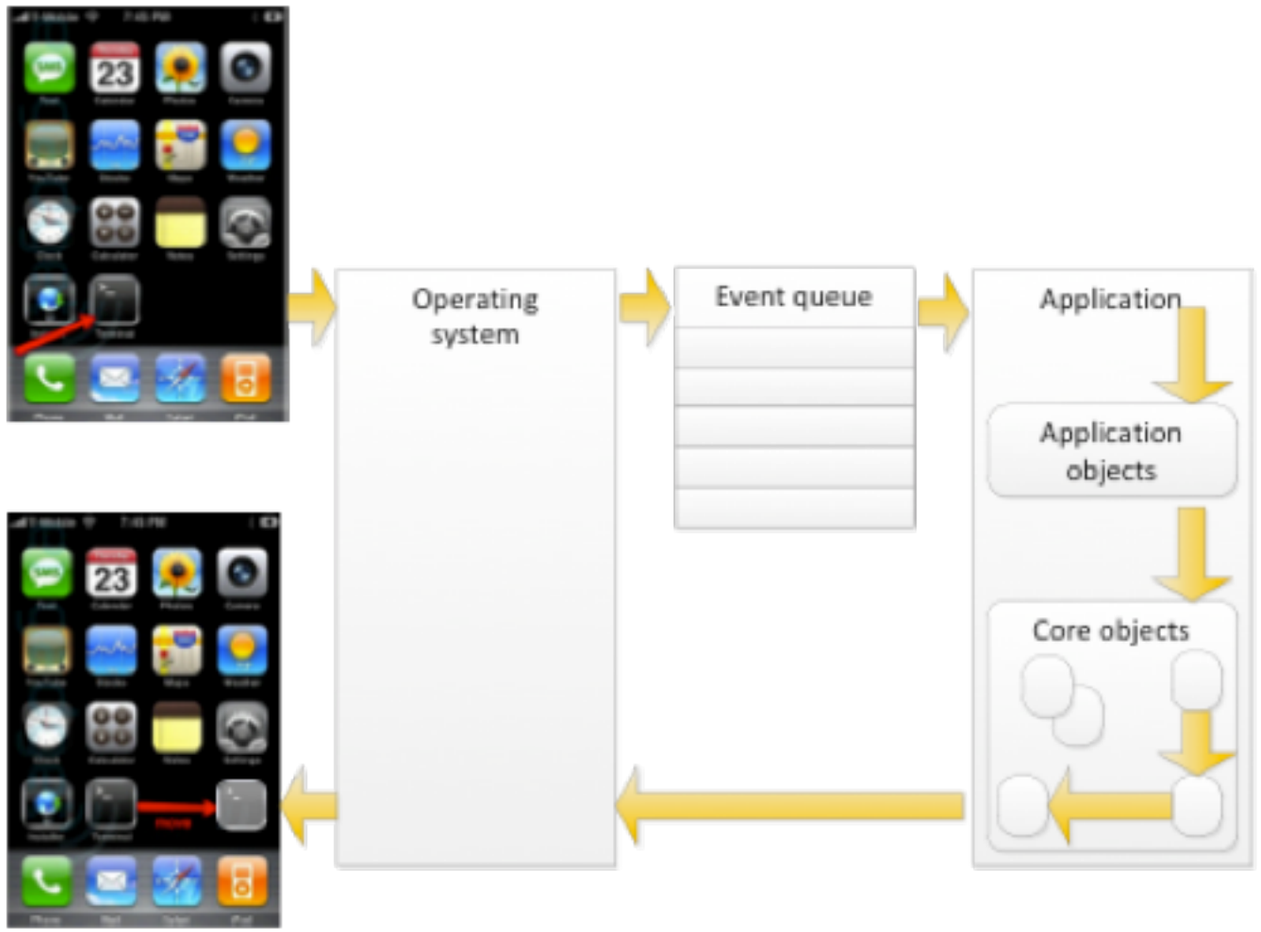

Figure 27 iPhone Touch Events

Events are stored in a queue until handled by an application program in the main run loop. Whenever we touch the screen an iOS device recognizes that as the beginning of a multi-touch sequence. Whenever all fingers or a finger that was touching the screen lift off the screen, this indicates the end of a multi-touch sequence. iOS stores various information about the multi-touch sequence such as its tap count, location of the finger/s and the time that the touch occurred.

An application program receives various event messages during a multi-touch sequence. Methods that handle these types of events include the following.

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;   // triggered when a user touch the screen with one or more fingers

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;   // triggered when the finger/s are moved

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;   // Triggered when the finger/s are lifted from the screen.

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event; // triggered when a touch sequence is cancelled by a system event, such as an incoming phone call
```

In our photo puzzle app we want to move puzzle pieces using only one finger. We don't really need to check what happens on touchesBegan, but we are definitely interested in the touchesMoved method so that the code can respond properly when puzzle pieces are moved. We are also interested in the touchesEnded method so that at that point we can check to see if a puzzle piece is in the right place. Instructions to implement these features follow.

Open the ViewController.m file in Xcode and add the following method. An explanation for each statement in the method follows.

```
- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event
{
    // Check for one finger touching the screen
    if([[event allTouches] count]==1) {
        UITouch * touch =[touches anyObject];

        // Get the touch location
        CGPoint touchLocation = [touch locationInView:self.view];

        // Check the image view
        if(touch.view!=self.view) {
          touch.view.center = touchLocation;
        }
    }
}
```

First, we need the count of touch objects. Inside the conditional for the if statement, we use the allTouches method of the UIEvent class that returns all Touch objects associated with the receiver. In our case the receiver is the ViewController view. Since we decided to have the photo puzzle pieces controlled by only one finger we compare the count to one. If only one finger is touching then the condition is *true*, and the rest of the code inside the if statement for this method will execute, otherwise this method will be done.

```
if([[event allTouches] count]==1)
```

The next line of code in this method is a declaration that gets a touch object from the collection of touches.

```
UITouch * touch =[touches anyObject];
```

Next, we check the location of the touch. This method returns the CGPoint that contains the location of the finger on the screen.

```
CGPoint touchLocation = [touch locationInView:self.view];
```

We use the UITouch class and its view property to prevent the whole puzzle from being moved on the screen. This condition in the if statement returns the view where the touches initially occurred. If the user is attempting to move a puzzle piece the condition will be true, and false otherwise.

```
if(touch.view!=self.view)
```

Finally, the touchLocation is assigned to the image UIView center property, which causes the image to move.

```
touch.view.center = touchLocation;
```

KEY POINT

Event objects store information about all touch events that occur during a multi–touch sequence. The information includes position, timestamp, and touch phase. To handle touch events, use one of the following methods:

- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event;

- (void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event;

Save your project and then click on the Run button. The simulator will open and appear similar to Figure 26. Now you will be able to drag your puzzle pieces with your mouse inside the simulator, or with a finger on a touch screen device. We will add the functionality to actually construct the puzzle and to animate the puzzle pieces in the next chapter.

Summary

This chapter provided instruction related to developing the first part of a Photo Puzzle app. The primary tasks addressed in this chapter were (1) designing the app, (2) creating a new project for the app, (3) selecting and customizing the attributes for the user interface elements, (4) making connections between the user interface elements and the code, (5) setting the orientation for the app, and (6) adding custom code to handle touch events.

We demonstrated the use of a storyline and a manually constructed storyboard to design the Photo Puzzle app. Then the steps necessary to create a new iOS Single View Application inside Xcode were described. Then we selected user interface elements for the Photo Puzzle's user interface. These elements consisted of a background grid for the puzzle, six puzzle pieces, two buttons, and a label. We assigned images to our background grid and puzzle piece user interface elements, and set the text for our buttons and label. We considered and selected various attributes for the different user interface elements. We illustrated how to make connections between the user interface elements and the ViewController class for this app. We also saw how the code was modified in the ViewController.h and ViewController.m files by Xcode as a result of the connections that we made. Then we described how to set different screen/device orientations for apps by modifying the code. Lastly, we described event handling and added a custom method to the View Controller class to handle touch events for the Photo Puzzle application. The development of the Photo Puzzle app will continue in the next chapter.

Review Questions

1. Which method is used to implement different orientations of a user interface.

2. How do you create a property?

3. What is the reason for synthesizing properties?

4. Which class is most commonly used for interacting with images?

5. Which property is used to determine if users can interact with a given user interface element?

6. Briefly describe the event handling cycle in iOS.

7. What types of methods are needed in order to use touch events?

8. How is the number of fingers touching the screen determined?

Exercises

1. Create an application that will display an image on the screen:

2. Create a simple application that displays in both LandscapeLeft and Portrait orientations.

3. Create an application that will display a tap count on the screen.

4. Create an application that allows a user to drag an image around the screen.

5. Create an application that allows a user to drag images only when two fingers are touching the screen.

Chapter 4

Photo Puzzle App - Part 2: Core Graphics and Animations

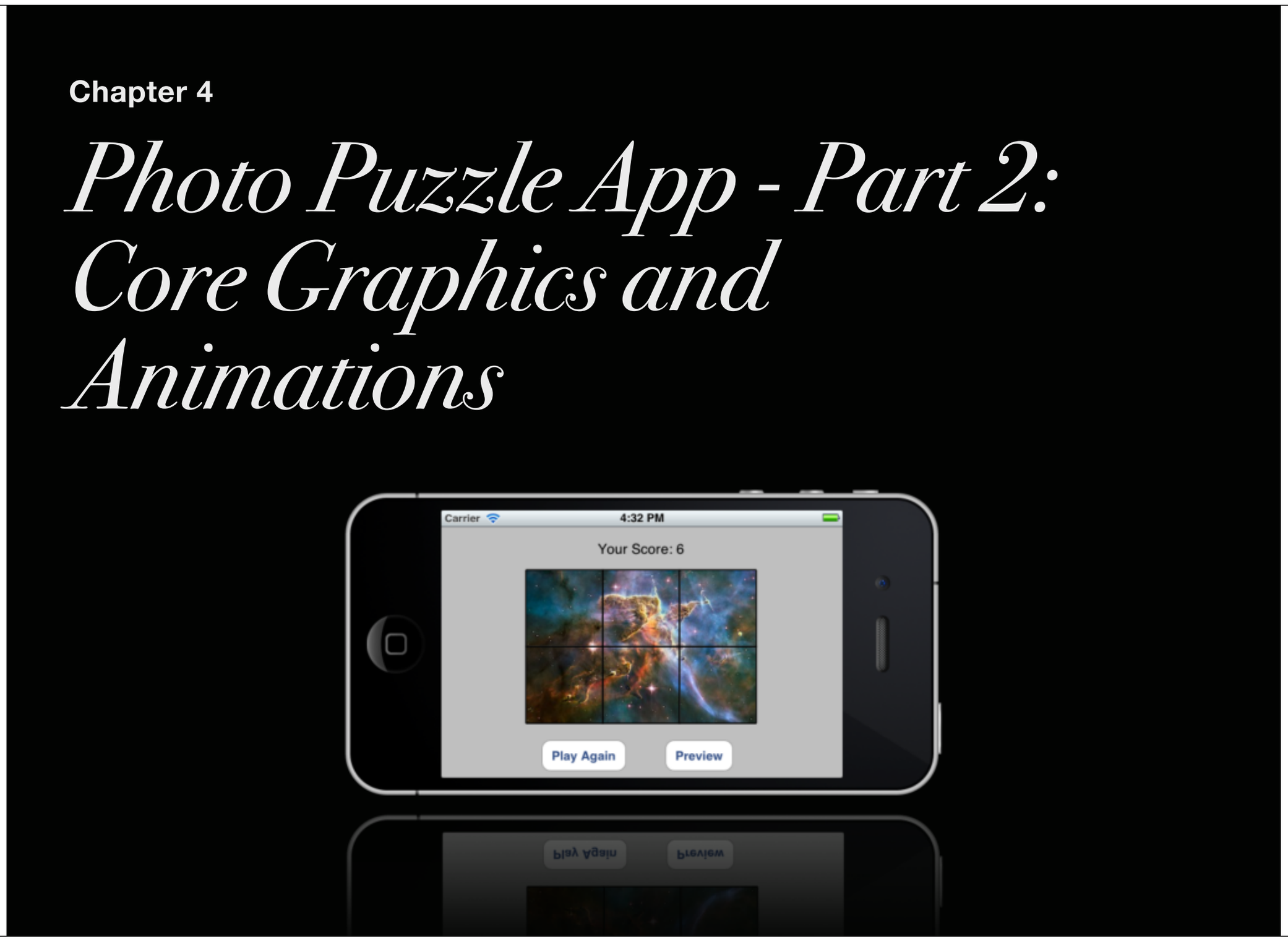

Photo Puzzle App - Part 2: Core Graphics and Animations

Concepts emphasized in this chapter:

- Basics of Core Graphics
- Building simple animations using UIView class methods
- Using a Model View Controller

Introduction

The emphases in this chapter are Core Graphics, building animations, and using a model view controller. These concepts are presented within the continuation of the development of the Photo Puzzle Project. Core graphics is one of the most important frameworks in iOS. We will use core graphics to define primitive shapes, and to position and track our puzzle pieces.

Implementation - Puzzle Pieces

In the first phase of implementation in this chapter we will create simple animations and implement the scoring system. In the second phase of implementation we will create the Preview functionality for the Photo Puzzle using a Modal View Controller.

Core Graphics

Most programming languages have methods for drawing paths, points, or rectangles. The Core Graphics framework is provided for basic drawing in iOS apps. In the first part of this section we will use Core Graphics (Quartz 2D) to manage the puzzle pieces and to extract the exact position of our finger on the screen using touch events and the CGPoint primitive.

Let's get started. Open the Photo Puzzle app project in Xcode and then expand the Frameworks folder in the Navigator area. This folder contains three frameworks, the UIKit framework, the Foundation framework, and the Core Graphics framework. The UIKit framework contains classes for all elements that can be incorporated into a user interface. These classes provide the means to create application objects, handle events, windows, views, and controls that have been specifically designed for touch screen interfaces. The Foundation Framework contains the superclass of all

objects, NSObject, and classes that are responsible for basic data types like NSString, NSInteger, collections like NSDictionary, NSArray, dates, and classes representing communication ports. The Core Graphics framework contains classes that relate to drawing, including color management, patterns, gradients, and creating and managing images using the Quartz advanced drawing engine. We will use methods within the Core Graphics CGGeometry class to check the intersections between rectangles, and to check membership of points in these rectangles for the puzzle pieces in the Photo Puzzle app.

In Objective-C Cocoa Touch, every object that appears on the screen has a frame. The frame is basically an instance of a CGRect, which is a structure that contains two variables that are also structures that contain the location and dimensions of a rectangle.

```
struct CGRect {
    CGPoint origin;
    CGSize size;
};
```

The origin of the CGRect is contained in a CGPoint structure object that contains the coordinates of the top-left corner of the rectangle.

```
struct CGPoint {
    CGFloat x;
    CGFloat y;
};
```

The size of the CGRect is contained in a CGSize structure that contains values representing the rectangle's width and height.

```
struct CGSize {
    CGFloat width;
    CGFloat height;
};
```

KEY POINT

Most of the examples in this book use Xcode to handle user interface elements created using Interface Builder which is integrated within Xcode. We don’t see it, but Xcode actually communicates with the UIKit framework in order to draw those elements on the iPhone screen. Because of this, it is important to realize that it is possible to build all those elements programmatically as well. All the subclasses of UIView such as UILabel, and UIButton have default constructors so to create a UILabel programmatically we could add code to the View

Controller class with a declaration such as UILabel *label=[[UILabel alloc] initWithFrame:CGrectMake(0, 0, 100, 20)]; **and then add this label to the screen using the addSubview method such as in** [self.view addSubview: label]; **instead of the way that we did it.**

Okay, now that we understand the underlying theory regarding how things are displayed on the screen of an iOS device we can now use the GCRect structures in the following sections of this chapter to control the positioning of our puzzle pieces.

Creating the Home Positions for the Puzzle Pieces

In the last chapter we set the position of the background grid for our puzzle pieces. Let's make sure this is correct since this step is critical to proper operation of the app. Select the ViewController.xib file in the Xcode navigator area and then select puzzle background grid inside the View window. Navigate to View > Utilities > Show Size Inspector. Double-check that the Show menu, Origin is set to the top left corner, that the X, Y, Width, and Height values, Autosizing and Example are as shown in Figure 1 before we set the home positions for the puzzle pieces.

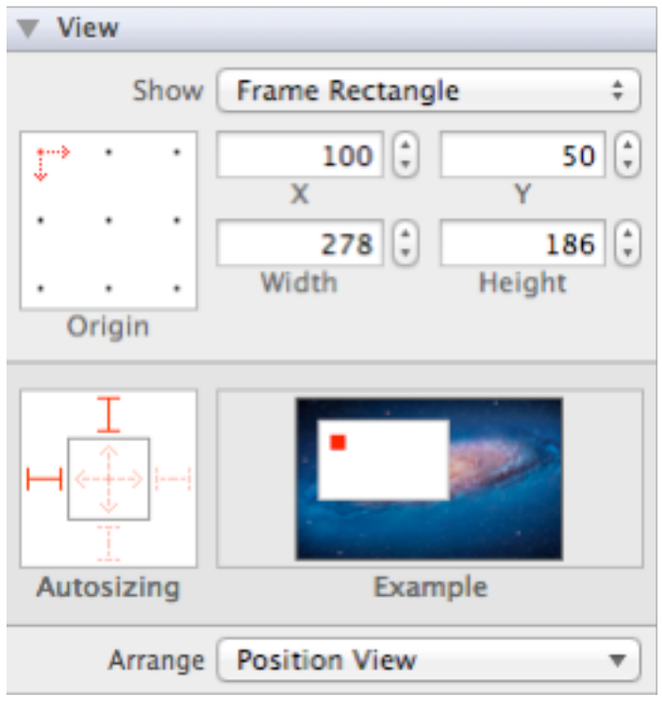

Figure 1 Puzzle Background Coordinates

We set the home positions for the puzzle pieces by dragging them to the desired positions within the View window. Notice that the X and Y coordinates within the Image View Size window automatically update as the pieces are moved within the View window. We could alternatively set the X and Y coordinates by entering the numerical values for the home positions. Drag all of the puzzle pieces within the View window until everything appears similar to Figure 2. Be sure that the puzzle pieces are not arranged in some order that would make it too easy to solve the puzzle. Save the project.

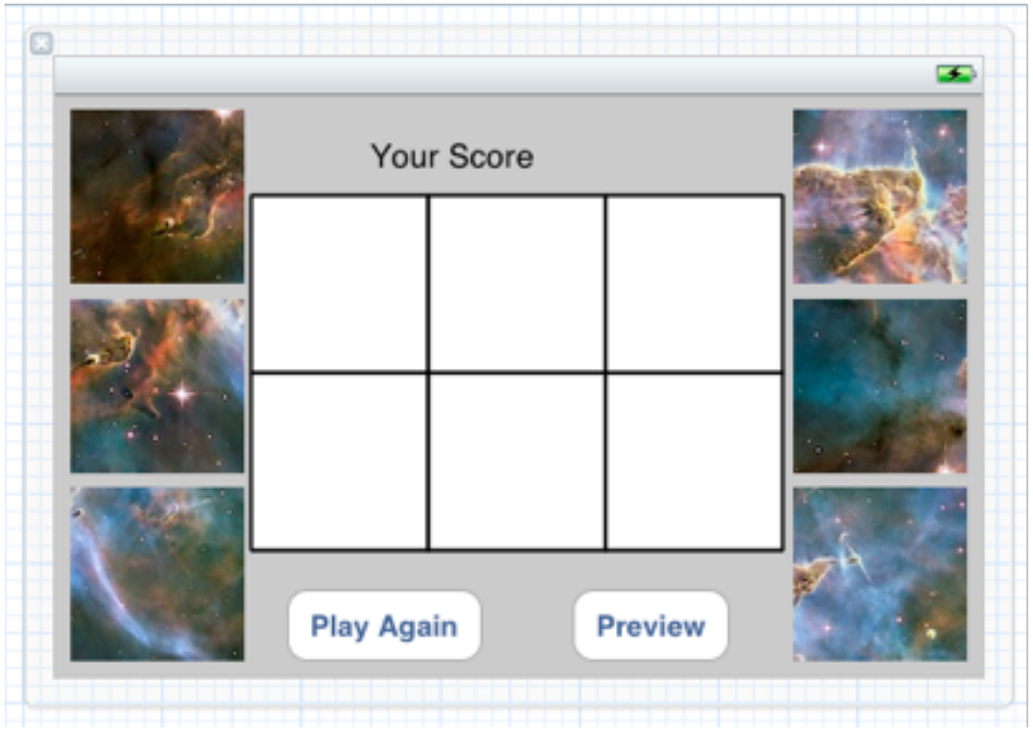

Figure 2 Home Positions for Puzzle Pieces

Recall that the coordinate system for the iPhone screen has its origin where X = 0, and Y = 0, in the upper left of the screen Examine the X and Y coordinates of your puzzle pieces to ensure that you understand the relationships of the puzzle pieces X and Y coordinates. This is an important concept with regard to the implementation of this app.

Storing the Home Positions for the Puzzle Pieces

In this app, if the user places a puzzle piece within the grid in the wrong place, it will be animated back to its home position outside of the grid. To implement this behavior in the app, the value of the coordinates of the home positions for the puzzle pieces will need to be stored within the code. Open the ViewController.m file and add the following NSValue declarations just after the synthesize directives.

```
NSValue *topLeftRectOrigin;
NSValue *topMiddleRectOrigin;
NSValue *topRightRectOrigin;
NSValue *bottomLeftRectOrigin;
NSValue *bottomMiddleRectOrigin;
NSValue *bottomRightRectOrigin;
```

We will use the NSValue objects to store the CGRect objects that will contain the information about the size and location of our puzzle pieces. Note that objects of primitive types like CGRect can not be stored directly within a collection object like an instance of a NSArray without first using a wrapper class such as the NSValue. Now open the ViewController.m file and add the following statements into the viewDidLoad method.We use the valueWithCGRect method from the NSValue class to store the CGRect structures into the NSValue objects. As we do this we include the UIIMageView frame property for each of the puzzle pieces that provides the position of the puzzle pieces that we previously defined.

```
// Define the home position for each of the puzzle pieces. If a piece is put into the wrong place inside the
// puzzle background grid it will animate back to its original home position
topLeftRectOrigin=[NSValue valueWithCGRect: topLeft.frame];
topMiddleRectOrigin=[NSValue valueWithCGRect:topMiddle.frame];
topRightRectOrigin=[NSValue valueWithCGRect:topRight.frame];
bottomLeftRectOrigin=[NSValue valueWithCGRect:bottomLeft.frame];
bottomMiddleRectOrigin=[NSValue valueWithCGRect:bottomMiddle.frame];
bottomRightRectOrigin=[NSValue valueWithCGRect:bottomRight.frame];
```

We now have a collection of puzzle pieces and coordinate values for the home position of those puzzle pieces, so we will create two arrays for these objects. Add the following declarations to the top of the ViewController.m file after the declarations for the NSValue objects.

```
NSArray *puzzlePieces;
NSArray *puzzleOrigins;
```

Next, add the following statements to the end of the viewDidLoad method in the ViewController.m file to load the arrays with the image objects and the origins for those objects.

```
// Arrays for the puzzle pieces and the objects that contain the coordinates of their origins on the screen
puzzlePieces=[[NSArray alloc]initWithObjects:topLeft, topMiddle, topRight, bottomLeft, bottomMiddle, bottomRight, nil];
puzzleOrigins=[[NSArray alloc]initWithObjects:topLeftRectOrigin, topMiddleRectOrigin,
topRightRectOrigin,bottomLeftRectOrigin,bottomMiddleRectOrigin, bottomRightRectOrigin, nil];
```

Defining the Positions for the Puzzle Pieces in the Background Grid

We have defined the home positions for our puzzle pieces which are their locations outside the puzzle background grid. We also need to define the final positions within the grid for where the puzzle pieces belong when the puzzle is formed correctly so that the photo is formed properly. These final positions for the topLeft, topMiddle, topRight, bottomLeft, bottomMiddle, and bottomRight images are labeled A through F respectively in Figure 3. Add the following declarations to the top of the ViewController.m file, just after the declarations for the NSValue objects to represent the final positions in the grid for the puzzle pieces.

```
CGRect topLeftRectGrid;
CGRect topMiddleRectGrid;
CGRect topRightRectGrid;
CGRect bottomLeftRectGrid;
CGRect bottomMiddleRectGrid;
CGRect bottomRightRectGrid;
```

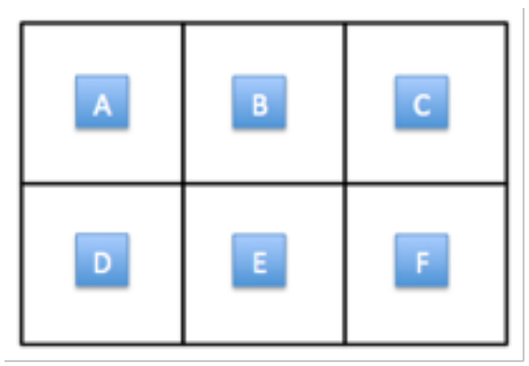

Figure 3 Final Positions in the Grid for the Puzzle Pieces

Notice in Figure 3 that the final positions A through F are smaller rectangles than the puzzle pieces, in fact they are one-third the size of the puzzle pieces. This will help prevent any overlap of pieces when they are locked into their final positions. To set the coordinate values of the final positions add the following code to the end of the viewDidLoad method in the ViewController.m file. Notice that we provide 4 values to the CGRectMake method as we define the rectangles within the grid. The first two parameters are the X and Y coordinate values for the center of the rectangle, and the next two parameters are the width and height of the rectangle. Recall that the origin, upper left corner of the grid is at X and Y coordinates of 100 and 50, we also defined the sizes of the puzzle pieces to be 90 points wide and 90 points high. So the area that will contain the top left puzzle piece has X coordinates of 100 and 190, and Y coordinates of 50 and 140. Examine the values for the topLeftRectGrid object. Do they make sense? The center of the topLeftRectGrid is 45 points to the right of 100 and 45 points down from 50 resulting in X and Y coordinates of 145 and 95 for the center of the top left square in the grid. So we provide those values for the X and Y coordinates of the new rectangle with width and height of 30 points. The rest of the rectangles for the definition of the final positions in the grid were determined in the same manner.

```
// Define the grid positions for the puzzle pieces
topLeftRectGrid=CGRectMake(145, 95, 30, 30);
topMiddleRectGrid=CGRectMake(235, 95, 30, 30);
topRightRectGrid=CGRectMake(325, 95, 30, 30);
bottomLeftRectGrid=CGRectMake(145, 185, 30, 30);
bottomMiddleRectGrid=CGRectMake(235, 185, 30, 30);
bottomRightRectGrid=CGRectMake(325, 185, 30, 30);
```

Creating the Grid Coordinates for the Final Positions of the Puzzle Pieces

Since it would be very tedious for a user to have to drag a puzzle piece to its exact points on the screen for its final position. So the way that we will determine if a user has dragged a puzzle piece pretty much in the correct place by checking if the defined center point for that piece is located within the bounds of the correct final position rectangle. If it is then the puzzle piece will snap into its exact final position. If it is not, then the puzzle piece will animate back to its home position outside the grid. Add the following declarations to the top of the ViewController.m file after the declarations for the CGRect objects in order to define the upper left coordinate values of the grid for the final positions of the puzzle pieces.

```
CGPoint topLeftGridFinalPoint;
CGPoint topMiddleGridFinalPoint;
CGPoint topRightGridFinalPoint;
CGPoint bottomLeftGridFinalPoint;
CGPoint bottomMiddleGridFinalPoint;
CGPoint bottomRightGridFinalPoint;
```

Next, add the following code to the end of the viewDidLoad method in the ViewController.m file to assign the X and Y coordinates for the center points of the puzzle pieces.We have added a few points of padding for the upper left corner of each puzzle piece in the grid.

```
// Define the upper-left coordinates for the puzzle pieces in their final position in the puzzle background grid
topLeftGridFinalPoint=CGPointMake(102, 52);
topMiddleGridFinalPoint=CGPointMake(194, 52);
topRightGridFinalPoint=CGPointMake(286, 52);
bottomLeftGridFinalPoint=CGPointMake(102, 144);
bottomMiddleGridFinalPoint=CGPointMake(194, 144);
bottomRightGridFinalPoint=CGPointMake(286, 144);
```

Save your project and then click on the Run button in Xcode to test the program. Note that your app will behave in the same way as it did at the end of chapter 3. Although we have added quite a few objects and assigned values to those objects in this chapter so far, we haven't yet provided any code to implement any new behaviors for those objects. What do we have left? We need to check if a puzzle piece is in the correct position. We need to either lock the puzzle piece into its final position or animate it back to its home position, and we need to update the score when a puzzle piece is placed in its correct position. Building three different modules of code for these operations is consistent with good software engineering practices and improves the quality of the software by defining independent modules for different functionalities which maximizes cohesion within the modules.

Checking the Position of the Puzzle Pieces

To check if puzzle pieces are positioned correctly we need to respond to the event when the user lifts his finger off the screen and invokes a method to find out where the puzzle piece is with respect to the puzzle background grid. Open the ViewController.m file and add the following code after the touchesMoved method. Do not be concerned about the error indicator that will appear in Xcode at this point, this is simply because the checkPuzzle method hasn't been declared yet. We will do that next.

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
    UITouch *touch =[touches anyObject];
    // Check the image view for the puzzle piece
    [self checkPuzzle:touch.view];
}
```

The touchesEnded method is called whenever the user is not touching an element on the screen anymore. Let’s imagine the situation that we are dragging one of the puzzle pieces on the screen using our finger; whenever we lift our finger off the screen, the touchesEnded method is invoked.

The implementation of the touchesEnded method is straightforward. We get the information available about the touched element and then we invoke the checkPuzzle method with the information about the puzzle piece. We will add more code to check the puzzle piece later. Now open the ViewController.h file and add the following declaration for our checkPuzzle method before the end directive.

```
- (void)checkPuzzle:(UIView *)puzzlePiece;
```

The implementation of the touchesEnded method is straightforward. We get the information available about the touched element and then we invoke the checkPuzzle method with the information about the puzzle piece. Add the following method to the ViewController.m file after the touchesEnded method.

```
- (void)checkPuzzle:(UIView *)puzzlePiece{
    CGPoint centerPoint= puzzlePiece.center;

    if([puzzlePiece isEqual: topLeft]){
        //Check if the piece is in the right place
        if(CGRectContainsPoint(topLeftRectGrid,  centerPoint)){
            //In the right place - lock in and increment score
        }
        else {
            //Not in the right place - animate back to home position
        } // End of topLeft
    }
}
```

The first thing we need in the checkPuzzle method is an object named centerPoint that stores the location of the center of the view representing the puzzle piece that is passed to this method by the touchesEnded method. The outer if statement checks if the element that was passed to the checkPuzzle method is the same element as topLeft. We need to know which element is which to apply the appropriate methods and values to it. Notice that we don't use the == operator, but rather we use the isEqual method since we are comparing the value of an object with component parts. If we used the equivalence operator it would only compare the addresses of the objects rather than the contents of the objects.

First, let's assume that the puzzlePiece is actually the topLeft element, then the condition for the if statement will be true and the nested if statement will execute. The nested if-else statement checks whether or not the center of the puzzlePiece is inside the proper rectangle for this puzzle piece in the background grid using the CGRectContainsPoint method. For now we simply place comments inside the body of the if statement. We will define these actions later.

We now need to add this same block of code with the two if statements five more times for each of the other five puzzle pieces, replacing the specific references to topLeft with the appropriate reference for each puzzle piece. You may wish to copy this block of code and paste it five times then carefully edit the references to topLeft with topMiddle, topRight, bottomLeft, bottomMiddle, and bottomRight as appropriate. Notice that there are two references to a specific puzzle piece in the block of code, once in each if condition. Save your project and click Run to test your program.

Implementation - Moving the Puzzle Pieces in the Code

In the previous section we developed the code that validates the final position of each of the puzzle pieces. Now it's time to implement the animations. If the center of a puzzle piece is within the rectangle for its final position in the background grid it will be moved slightly as needed to exactly fill the placeholder. This will allow the puzzle pieces to be accepted if they are placed very close, but not quite into the exact position. Otherwise the puzzle would require too much precision and could become very tedious. If the puzzle piece is not in the rectangle that represents its final position, then it will animate back to its home position outside of the grid.

For the implementation of these features we need two new methods. Add the following declarations to the ViewController.h file above the end directive. We will implement the placePuzzle method and then the animatePuzzle.

```
- (void) placePuzzle:(UIView *) puzzlePiece Into:(CGPoint) puzzlePlace;
- (void) animatePuzzle:(UIView *) puzzlePiece BackToOrigin: (NSValue *) puzzleOrigin;
```

Placing Puzzle Pieces in their Final Position

The placePuzzle method moves a puzzle piece, that has been moved by the user to a position that is close to its final position, so that it is then in exactly the correct position within the grid. This requires two arguments. The first argument is a UIView element that is one of the puzzle pieces, and the second argument is the CGPoint that contains the final position for the puzzle piece. Add the following method after the checkPuzzle method in the ViewController.m file. This method first copies the frame of the puzzle piece into a new object, then assigns the x and y values of the final position for the piece that was passed as the second argument, then copies the all of that back into the puzzle piece that was passed in so that it is now in its exact final position. Then the placePuzzle method sets the value of the user interaction enabled attribute to NO so that the puzzle piece is effectively locked in and cannot be moved. Setting this attribute is a good example of how user interface elements can be manipulated inside Xcode or programmatically in the code. This attribute was enabled through a checkbox when we created the puzzle pieces and now it is disabled within the code.

```
- (void) placePuzzle:(UIView *) puzzlePiece Into:(CGPoint) puzzlePlace{
    CGRect rect=puzzlePiece.frame;
    rect.origin.x=puzzlePlace.x;
    rect.origin.y=puzzlePlace.y;
    puzzlePiece.frame=rect;
    puzzlePiece.userInteractionEnabled=NO;
}
```

Animating Puzzle Pieces Back to their Home Positions

Manipulating view properties can be done to create animations in iOS programming. Animations are applied to view properties in animation blocks with definitions for the beginning and end of the changes that represent the animations. The animations occur when changes in property values cause a transition from the current state to a final state of the view properties. Two methods must always be paired when defining animations: beginAnimations:context and commitAnimations.

KEY POINT

Animation blocks are started by invoking the beginAnimations:context: class method of the UIView class. Then the view's property values are set to define the animation. Then the animation block is ended by invoking the commitAnimations class method.

Add the following method to the ViewController.m file after the placePuzzle method. This method is a bit more complex even though it only contains four lines of code. Our animatePuzzle method has two arguments. The first argument is a UIView element into which we pass a puzzle piece. The second argument is an NSValue that stores the information about the origin of the puzzle piece. Add the following highlighted method after the placePuzzle method in the ViewController.m file.

```
- (void) animatePuzzle:(UIView *) puzzlePiece BackToOrigin: (NSValue *)puzzleOrigin {
    [UIView beginAnimations:nil context:nil];
    [UIView setAnimationDuration:2];
    puzzlePiece.frame=[puzzleOrigin CGRectValue];
    [UIView commitAnimations];
}
```

The first statement inside the animatePuzzle method is an invocation of the beginAnimations method. The full declaration of the beginAnimations method follows.

```
+ (void)beginAnimations:(NSString *)animationID context:(void *)context;
```

The beginAnimations method has two arguments. The first argument is an animationID that determines the identity of an external animation. In our case we have only one animation so we will pass nil instead of a name. The next argument is the context for an external animation. Again, since we will only have one animation, in our case we pass nil.

We can set the values for many different properties of animations. In the second statement of our animatePuzzle method we set a value of two seconds for the AnimationDuration property using the following statement.

```
[UIView setAnimationDuration:2];
```

In the next statement we set the frame property of the puzzle piece back to its origin, with the following.

```
puzzleView.frame=[puzzleOrigin CGRectValue];
```

Recall that we must always pair invocations of beginAnimations with invocations to commitAnimations to define the end of the animation block, so the last statement we add to the animatePuzzle method is the following.

```
[UIView commitAnimations];
```

This completes the operations needed to implement our animation.

KEY POINT

Remember the simplest animation can be implemented in four steps:
1) Call beginAnimation using:
[UIView beginAnimations:nil context:nil];

2) Set the duration of the animation:
[UIView setAnimationDuration:2];

3) Change the properties that you want to animate:
puzzleView.frame=[puzzleOrigin CGRectValue];

4) Commit the animation:
[UIView commitAnimations];

Invoke placePuzzle and animatePuzzle methods in the checkPuzzle Method

Now that the placePuzzle and animatePuzzle methods have been implemented we can replace the comments in the checkPuzzle method with the appropriate invocations of the placePuzzle and animatePuzzle methods in the relevant blocks of code as in the following block of code for the top left puzzle piece.

```
if([puzzlePiece isEqual: topLeft]){
        //Check if the piece is in the right place
        if(CGRectContainsPoint(topLeftRectGrid, centerPoint)){
            [self placePuzzle:puzzlePiece Into:topLeftGridFinalPoint];
        }
        else {
            [self animatePuzzle: puzzlePiece BackToOrigin:topLeftRectOrigin];
        }
    }
```

Replace the comments in each of the nested if statements associated with each of the puzzle pieces in the checkPuzzle method, customizing each of the statements with the correct reference to the appropriate puzzle piece. Save and run your project to ensure that there are not any errors at this point.

Updating the Score

We will now create the scoring system for the Photo Puzzle app. First, create a variable to store the value of the score by adding the following declaration to the top of the ViewController.m file after the synthesize directives.

```
int score=0;
```

Each time the user places a puzzle piece in its correct final position the score will be increased by one. We created a label to display the score on the screen -- now we will define a method that will replace the default text we gave to the label with the actual score. Add the following method declaration to the ViewController.h file just above the end directive.

```
- (void) showTheScore:(int) scoreToShow;
```

Add the following method after the animatePuzzle method in the ViewController.m file.

```
- (void) showTheScore:(int) scoreToShow{
    NSString *text=[[NSString alloc] initWithFormat:@"Your Score: %d", scoreToShow];
    scoreLabel.text=text;
}
```

This method is relatively simple. First we create a variable named text, that is a NSString initialized with the library method, initWithFormat. The first part of the string is assigned the literal characters to display Your Score: then we use the format specifier %d to append a numerical value to be displayed as a decimal integer. We then assign this string to the text component of our score label in the second statement. Since we initialized the value of the score variable to 0 in its declaration its value will be 0 when the app first starts.

The last tasks we need to perform with regard to moving the puzzle pieces and updating the score are to increment the score label every time a puzzle piece is locked into its final position and to invoke the method we just created. We can perform both of these tasks in the placePuzzle method so add the following two statements to the end of the placePuzzle method in the ViewController.m file.

```
score++;
[self showTheScore:score];
```

Save your project and click on the Run button to compile and test your program. The animations and the scoring system should perform properly now.

Implementation - Add the Preview and Play Again Features

In this phase of the development we will create a new View Controller that will provide a preview of the puzzle image. To show the preview we will use a modal view controller. Most often we use modal view controllers to:

- Gather information from the user
- Temporarily present content
- Implement alternate interfaces for different device orientations

Creating and presenting a view controller modally is generally accomplished with four steps:

1. Create a new view controller

2. Set the transition style

3. Assign a delegate object

4. Invoke the presentModalViewController: animated method.

We will present each of these tasks in the next four subsections of this chapter.

Create a New View Controller

Creating a new view controller can be accomplished by following the next steps in this subsection.

Navigate to File > New > New File... to get a popup window for a New File as shown in Figure 4. Select Cocoa Touch Class in the left panel of this window, select UIViewController subclass in the right panel, and then click on the Next button.

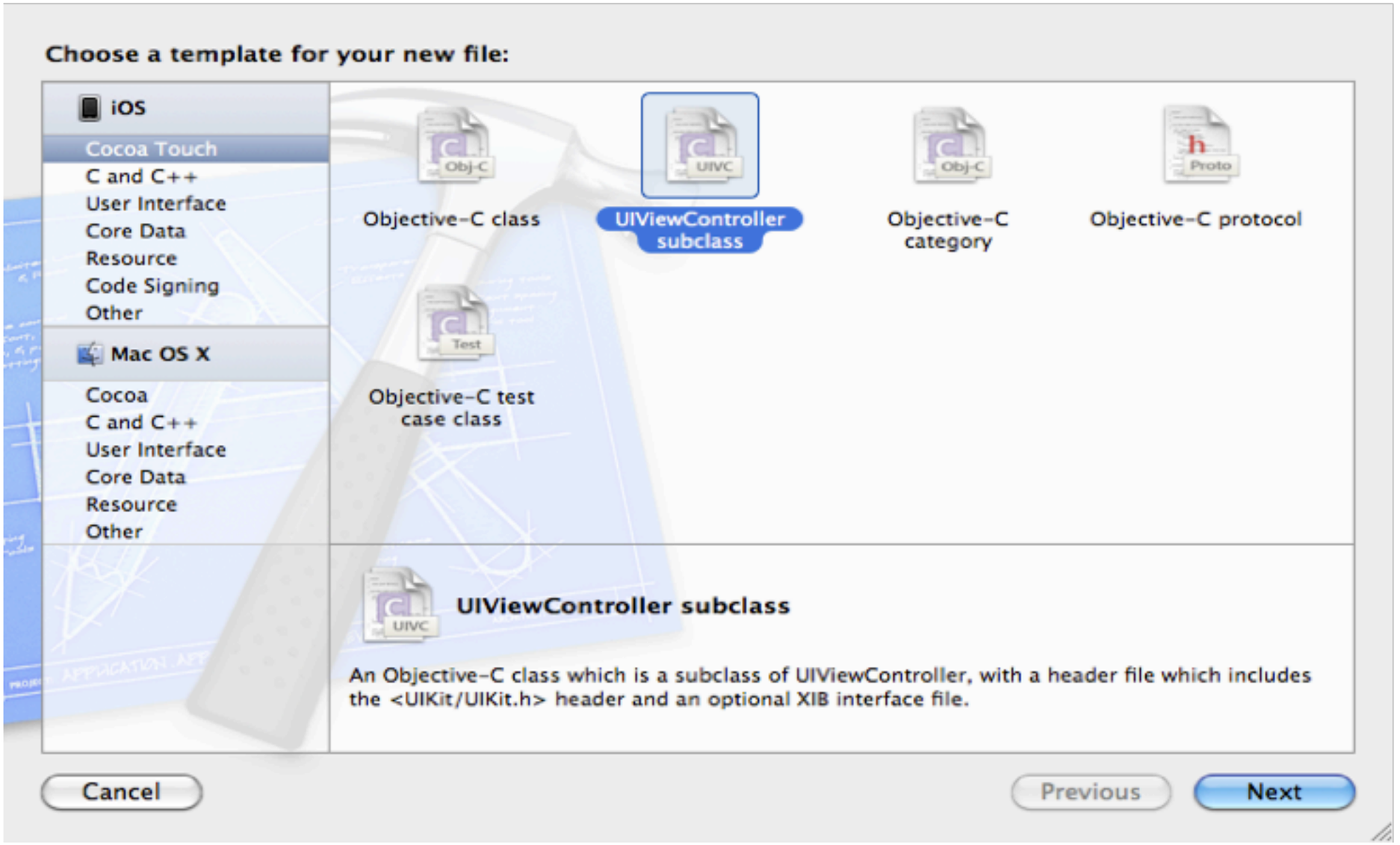

Figure 4 Cocoa Touch Templates

A new window will appear, enter Preview for the name of the new class. Make sure that the subclass is a UIViewController, that the Targeted for iPad checkbox is not selected, and that the With XIB for user interface checkbox is selected as shown in Figure 5. Then click on the Next button.

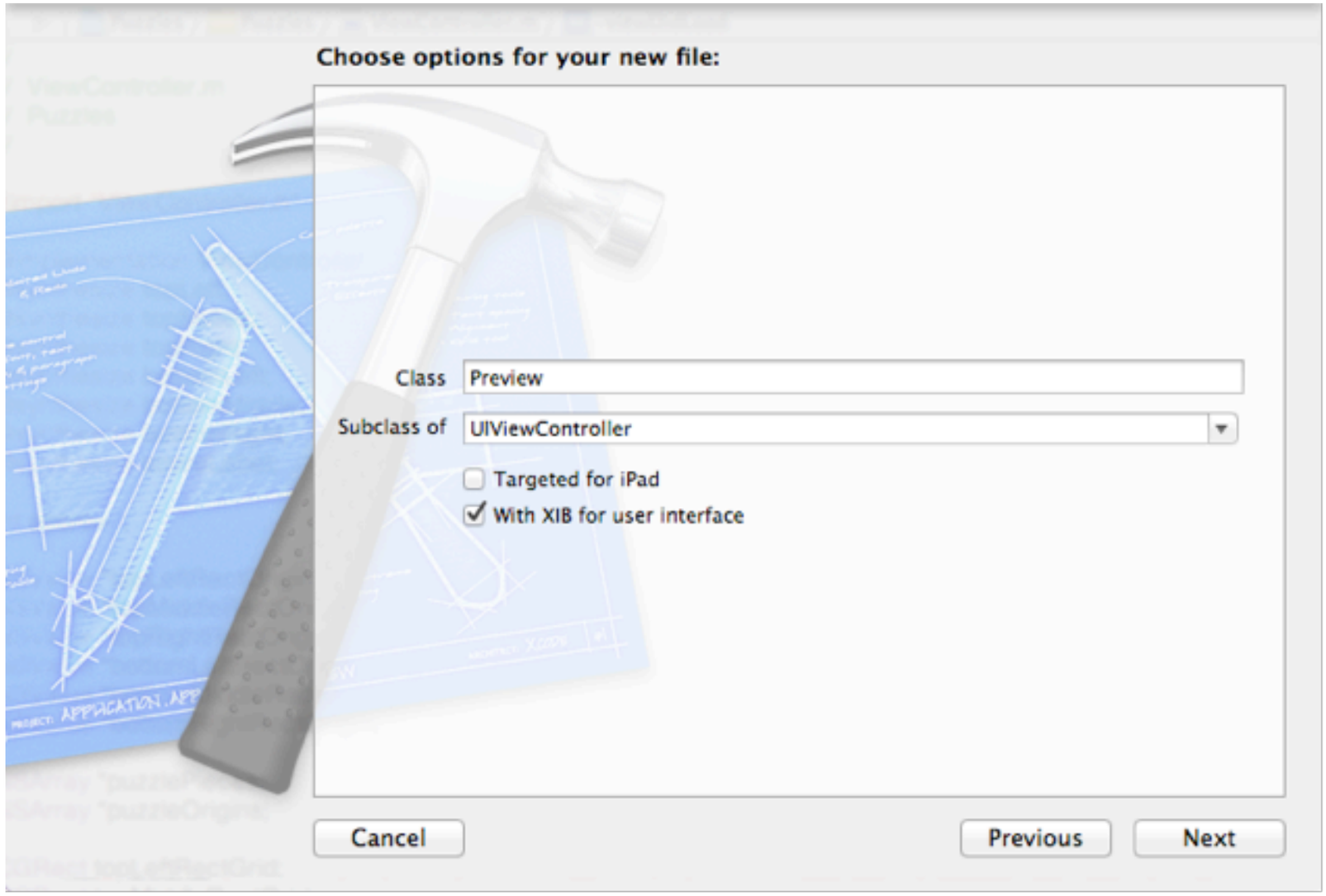

Figure 5 Preview View Controller

A new window will appear that will allow you to create the new view controller as shown in the previous figure. Be sure that the folder, Group, and Targets are selected as shown in Figure 6 then click on the Create button.

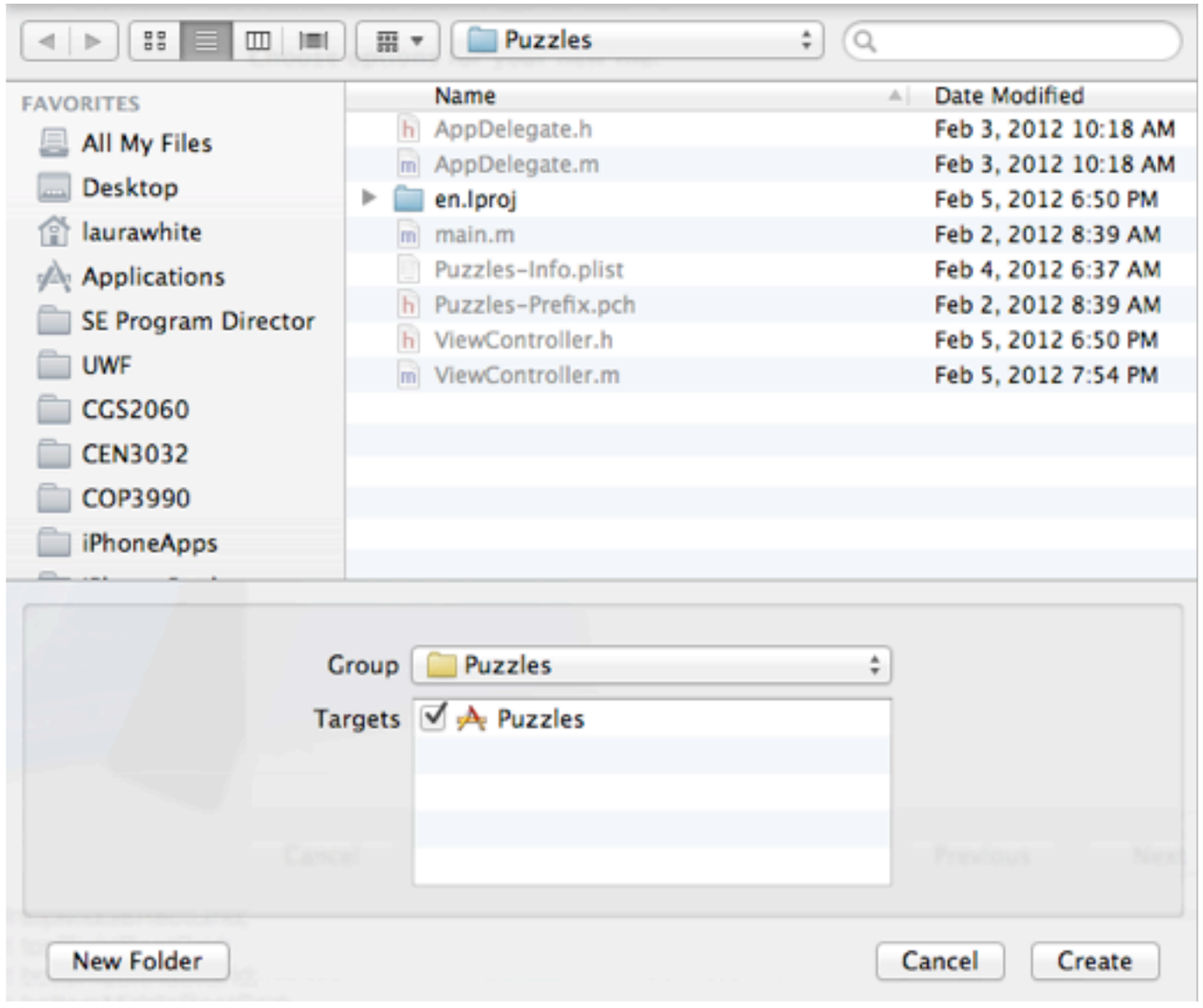

Figure 6 Create Preview View Controller

Three new files will appear in the Xcode Navigator Panel: Preview.h, Preview.m, and Preview.xib. Select the Preview.xib file in the Xcode Navigator Panel. Make sure that you can see View Attributes for the xib file by navigating to View > Utilities > Show Attributes Inspector.

Select the View in the Editor area by clicking on it and then rotate it to Landscape mode by changing the View's orientation to landscape in the Orientation menu in the Utilities panel.

Drag an Image View element from the Object Library into the View window. Select the Image View in the View window and then select the image.jpg file in the Image menu. It may be necessary to move and resize the image so that it is centered and fills the View window.

Drag an Round Rect Button element from the Object Library into the View window. Double click on the Button in the View window and change it's label to Go Back. The View window should now appear similar to Figure 7. Save your project.

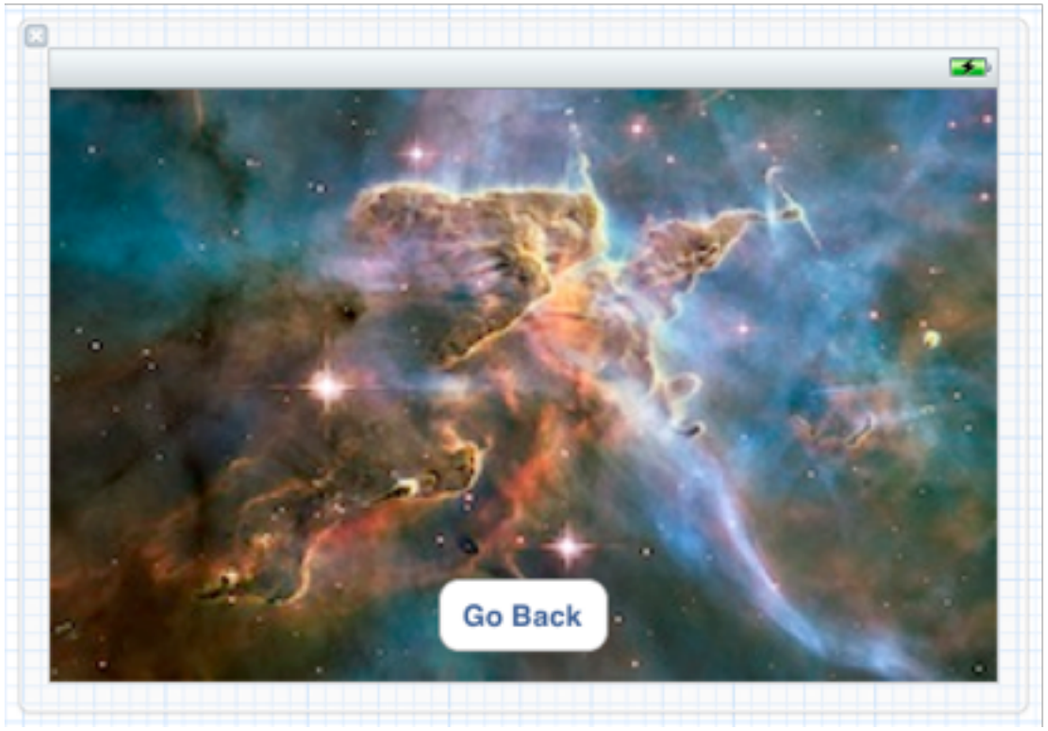

Figure 7 Puzzle Preview View

Next, we will add the functionality for the Go Back button by connecting an IBAction to the Go Back button. Display both the Preview.xib and the Preview.h files in the Xcode window by first selecting the Preview.xib file and then opening an Assistant editor in which to display the Preview.h file as shown in Figure 8. Note that we closed the Utilities area in Xcode to allow more space for editing. Then ctrl-click on the Go Back button in the View window. A new pop window with the list of the events that buttons can respond to will appear on the screen as shown in Figure 9.

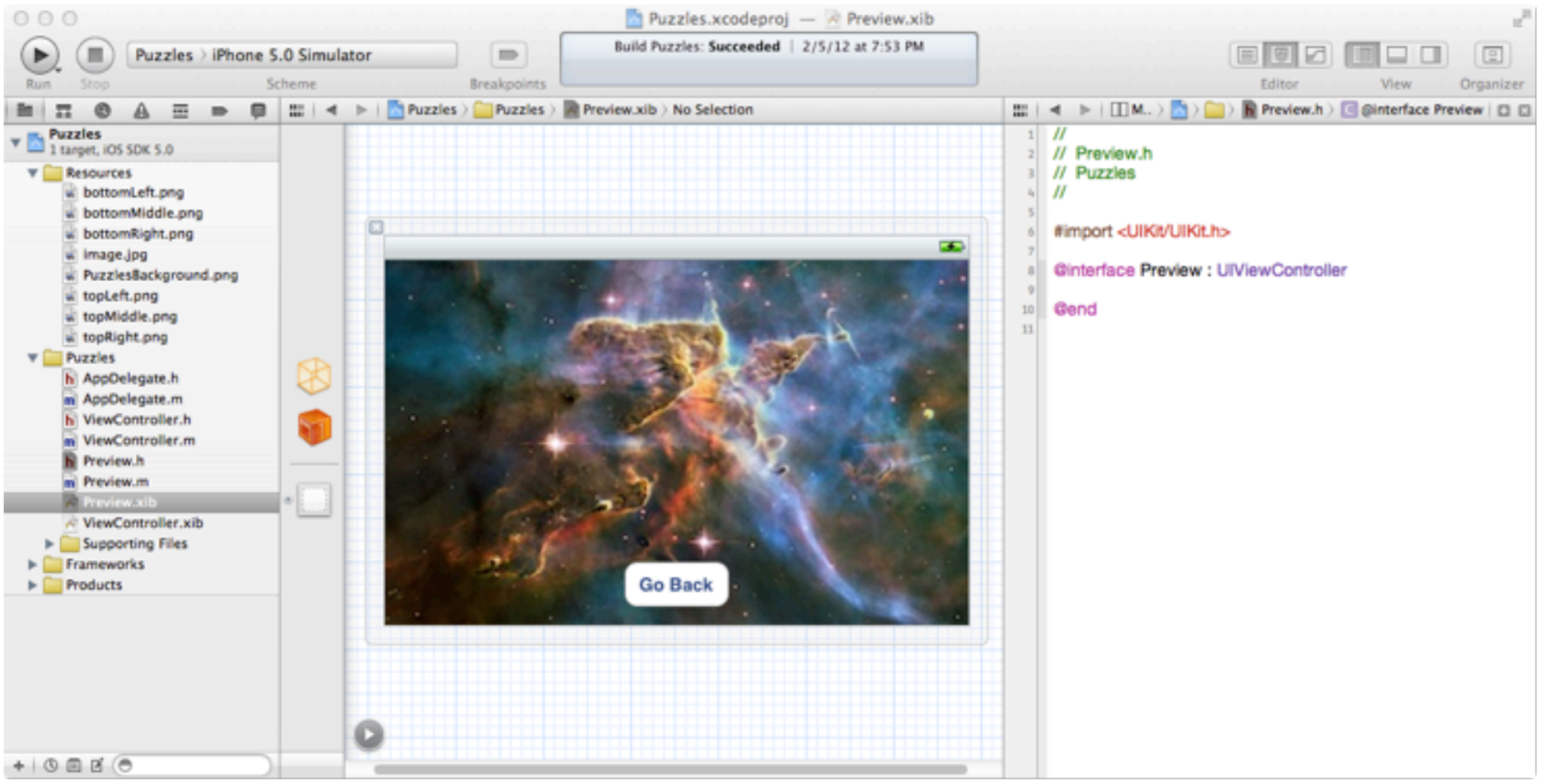

Figure 8 Xcode Displaying Preview.xib and Preview.h files

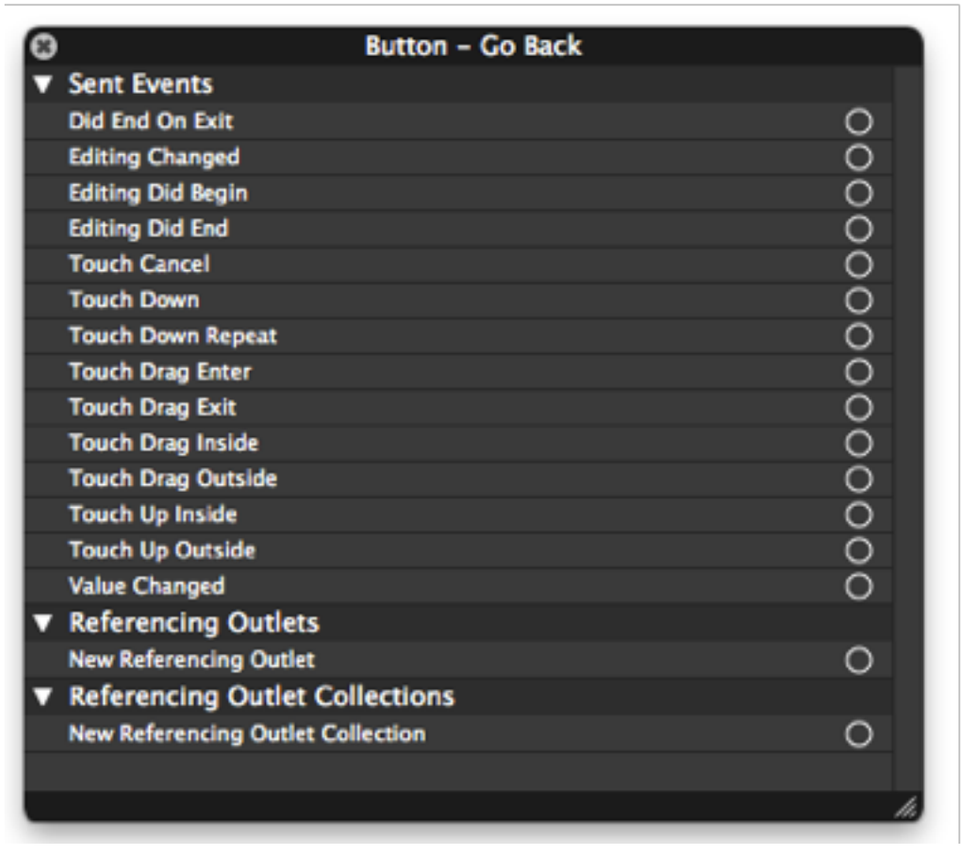

Figure 9 List of Events for Go Back Button

The Touch Up Inside event is most often used with buttons. Other user interface elements might use different events, for example the event Value Changed is often used with segmented controls. Click on the empty circle to the far right of the Touch Up Inside event shown in Figure 4.9 and drag the blue elastic line to the Preview.h file just above the end directive as shown in Figure 10. When you release the mouse another small window will appear in which you should enter dismiss as the name for the action connection and method that will implement this action as shown in Figure 11, and then click on the Connect button.

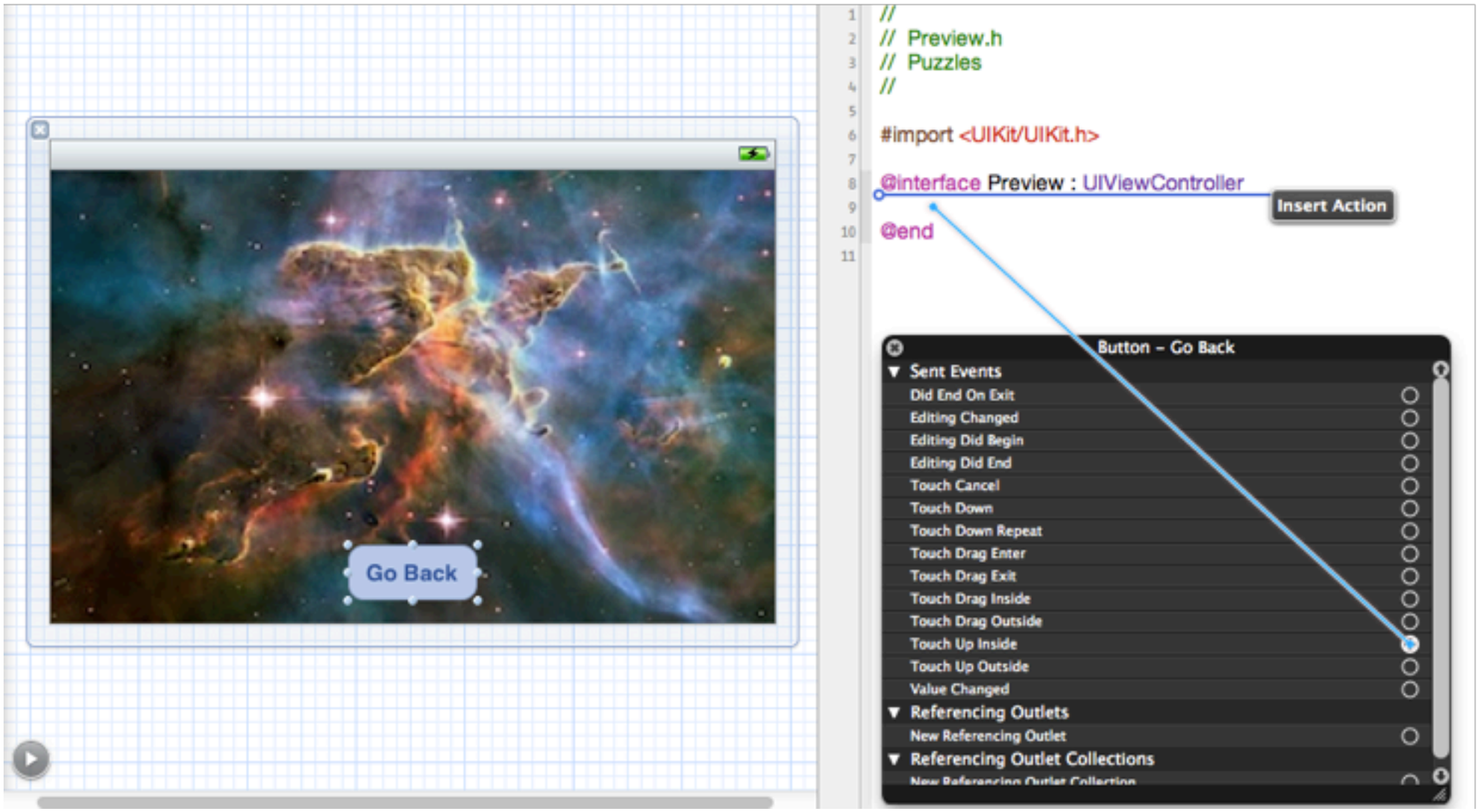

Figure 10 Creating an IBAction for the Go Back Method

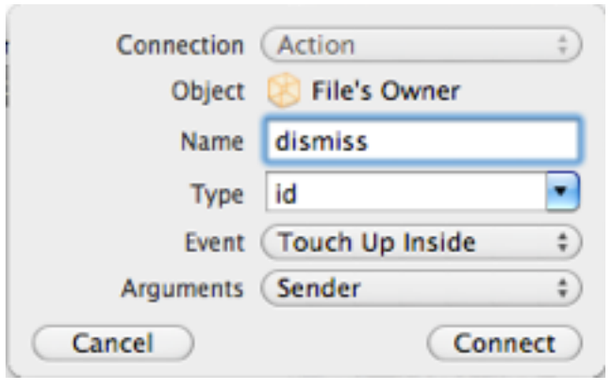

Figure 11 Setting Up an IBAction Method

Check the Preview.h file to see that the declaration of the IBAction dismiss method was created correctly as shown in the following code.

```
//
//  Preview.h
//  Puzzles
//

#import <UIKit/UIKit.h>

@interface Preview : UIViewController

- (IBAction)dismiss:(id)sender;

@end
```

Next toggle the assistant editor off to make room on the screen and then click on the Preview.m file in the Xcode Navigator pane to open the Preview.m file in the Editor area. We want to display the preview screen in landscape mode when the app runs on an iOS device, so just as we did in the previous chapter we once again we need to modify the shouldAutorotateToInterfaceOrientation method. Modify the return statement so that it appears as follows.

```
return (interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
```

Next, add the following statement to the dismiss method in the same file so that the method now appears as follows.

```
- (IBAction)dismiss:(id)sender {
    [self dismissModalViewControllerAnimated:YES];
}
```

Creating a modal view controller establishes a parent-child relationship. In our case the parent is the ViewController. By invoking the dismissModalViewControllerAnimated method we are removing the modal view controller for the Preview on the screen, and returning to its parent, the ViewController, which is the original screen for the Photo Puzzle. Save your project and click on Run to compile and test your program.

This completes the tasks necessary to create the new Preview view controller.

Set the Transition Style

The modalTransitionStyle is a defined constant that is used to specify the visual effect that occurs when a modal view controller is displayed or dismissed. There are three options for these effects:

- UIModalTransitionStyleCoverVertical – when the view is displayed it slides up from the bottom of the screen, and then slides back down when it is dismissed. This is the default value for the transition style.
- UIModalTransitionStyleFlipHorizontal – when the view is displayed it appears as a horizontal 3D flip from right to left so that the new view appears as if it were on the back of the previous view, and then flips back from left to right when the view is dismissed.
- UIModalTransitionStyleCrossDissolve - when the view is displayed the current view fades out as the new view fades in, and the same transition occurs when the view is dismissed.

For the Photo Puzzle program we choose the cover vertical style. Add the following highlighted to the viewDidLoad method in the Preview.m file.

```
self.modalTransitionStyle=UIModalTransitionStyleCoverVertical;
```

Assign a Delegate Object

Delegation is sometimes used in conjunction with modal view controllers. The modal view can delegate functionality by defining methods that are invoked by the parent view controller when certain events occur. Dismissing a modal view controller is commonly implemented through delegation so that the parent view controller that presents a modal view controller will also be responsible for dismissing the modal view controller. The

ViewController is the parent of the modal view controller we are constructing, however, delegation is not necessary in our application because we invoke the dismissModalViewControllerAnimated method inside the dismiss method implementation for the Go Back button of the Preview class.

Invoke the presentModalViewController Method

To invoke the presentModalViewController method will add a method that will be called when the Show Preview button is tapped. We will also add a method to start the puzzle over when a user taps on the Play Again button. To do this we will first create iBActions for both of the buttons. We will do this in exactly the same way as we did for the Go Back button on the Preview screen earlier in this chapter. So with the ViewController.xib file in the Xcode editor window, and the ViewController.h file in the Assistant Editor ctrl-click on the Play Again button and drag the elastic band from the empty circle to the far right of the Touch Up Inside event, to the ViewController.h file just above the end directive. Enter playAgain for the name of the action connection. Do the same for the Preview button but enter showPreview for the name of the action connection. Look at your ViewController.h file, Xcode should have added the following two declarations for methods to implement the IBActions associated with these buttons.

```
- (IBAction)playAgain:(id)sender;
- (IBAction)showPreview:(id)sender;
```

Now we need to add the code to the bodies of those methods in the ViewController.m file to implement the functionality for these buttons. Before we can invoke the view controller for the Preview from inside the ViewController class we need to import the interface for the Preview class by adding the following directive to the ViewController.m file after the #import ViewController.h directive.

```
#import "Preview.h"
```

Add the following code to the inside of the showPreview method in the ViewController.m file.

```
Preview *previewViewController=[[Preview alloc] initWithNibName:@"Preview" bundle:nil];
[self presentModalViewController:previewViewController animated:YES];
```

Next we add the implementation for the play Again button by adding the following code to the playAgain method in the ViewController.m file.

```
int numberOfViews=[puzzleOrigins count];
for (int i=0; i<numberOfViews; i++) {
    UIView *imageView= [puzzlePieces objectAtIndex:i];
    NSValue *rect=[puzzleOrigins objectAtIndex:i];
    [self animatePuzzle: imageView BackToOrigin:rect];
    imageView.userInteractionEnabled=YES;
}
score=0;
[self showTheScore:score];
```

Several tasks are performed in this method in order to start the photo puzzle again. First we create an integer variable that stores the number of elements in the array of puzzle pieces. We know that this photo puzzle has six elements, but using a variable instead of a literal is good practice so that if we want to change the number of elements we can do so by changing the value of the variable in one place rather than changing the literal value in many places which leads to errors if one of the occurrences is inadvertently not changed. Then we use a for loop to iterate through all the elements in the puzzleOrigins array to move each element back to its original home position and reenable user interaction so that the user can move the pieces on the screen. After the for loop, the score is reset to 0 and redisplayed using the showScoreMethod.

Congratulations! The photo puzzle application is complete! Click on the Run button to compile and enjoy your app!

Summary

A lot of programming resources were used in this chapter. We created a photo puzzle app, using core graphics, UIView animation methods, and developed a basic scoring system. We used primitive graphics types and an object wrapper in order to use primitive types in objects such as an NSArray. Another essential concept introduced in this chapter was the use of the CGRectContainsPoint method for determining if a rectangle contains a given point. This method is often used to detect the collision of objects in many types of applications. For example, this could also be used in a simple soccer game to detect if an object representing a soccer ball is inside the goal. We also described and showed how to use modal view controllers, which provides one way to create multi-view applications.

Review Questions

1. Each view has an important property that is responsible for its size and position. Which property is that?

2. Describe the primary reason for using instances of the NSValue class?

3. List all the properties that can be animated using the UIView class methods.

4. Describe the process of animating views.

5. In the Photo Puzzle project you have created and implemented a modal view controller. What are the primary reasons to use a modal view controller?

6. Describe the steps needed to create a modal view controller.

7. List and describe the modal transition styles.

8. Which method should be used to detect if the point CGPoint is inside a CGRect?

9. Write a code segment that will create a CGRect with origin (0,10) and size 20 by 40.

10. Describe the process of creating UI elements programmatically – provide examples.

Exercises

1. Create an application that contains one button that displays a modal view controller.

2. Create an application that stores primitive values like CGRect, CGPoint, in an NSArray and then extract those values. A user interface is not necessary for this exercise.

3. Create a photo puzzle application similar to the one developed in this project except that it should have eight puzzle pieces instead of six. Implement different orientations of the screen. Add an option for users to pick an image for the puzzle from their own photo gallery.

Chapter 5

Show Me App: Map Kit Framework

iBooks Author

Show Me App - Map Kit Framework

Concepts emphasized in this chapter

- *MapKit Framework*
- *Core Location Framework*
- *Annotations*
- *Callouts*
- *Displaying User Location*

Introduction

In this chapter you will be introduced to the MapKit framework. MapKit provides the tools that can be used to display interactive maps in iOS applications. The other framework used in this chapter is the Core Location Framework which provides the tools to determine the location of a device and track changes relative to location. Most often you need to use these two frameworks together if you want to display a fully functional map within an app.

In this chapter you will build a Show Me app. In this app we will create a map, and add two custom annotations for items of interest, and show the user's location. The custom annotations will be used to mark the Empire State Building, and the Statue of Liberty. Whenever the user clicks on an annotation, it will display additional information about the item of interest. You will also learn how to display and zoom to specific regions of a map. The plan for this project will be to:

1. Create a new project in Xcode
2. Link to the MapKit and Core Location Frameworks
3. Add images to the project

4. Build the user interface
5. Display the map
6. Display the user location
7. Add and customize annotations
8. Display the detailed view for each annotation
9. Run and test the application

Development

The development of the ShowMe application will be conducted in several phases.

Create a New Project in Xcode

Launch Xcode and create a new project as shown in Figure 1.

Choose an iOS Application -- Single View Application template as shown in Figure 2, then click on the Next button.

Figure 1 Xcode Window

Choose the Single View Application template as shown in Figure 2, then click on the Next button.

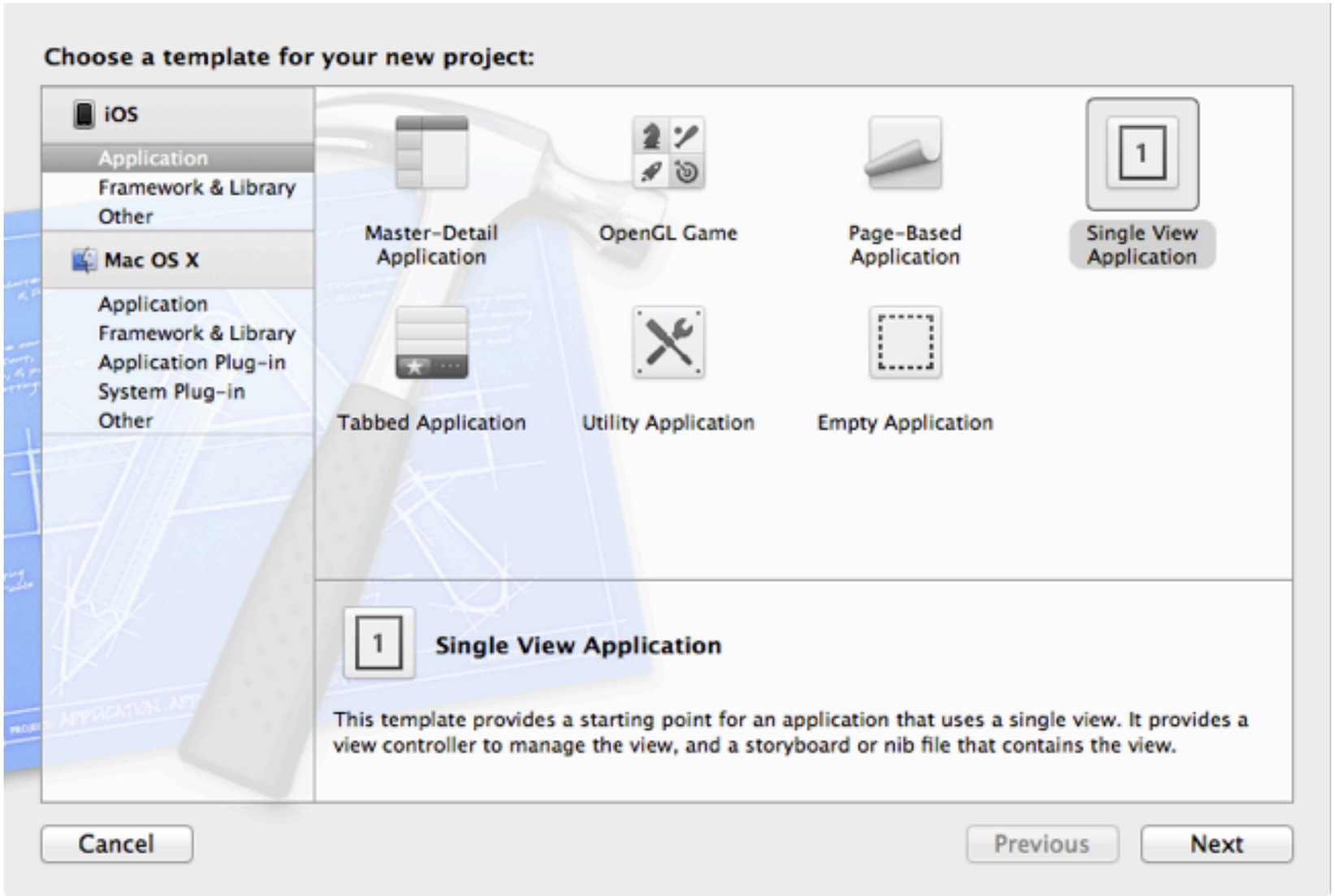

Figure 2 Creating a New Project – Selecting a Template

Enter ShowMe for the ***Product Name*** as shown in Figure 3. A different name can be used, but it will be easier to follow along with this project if you use the same name.

Enter *com* for the ***Company Identifier*** or leave it with the default value as shown in Figure 3.

Select iPhone in the Device-Family menu as shown in Figure 3.

Select the Use Automatic Reference Counting checkbox, but do not select the other checkboxes as shown in Figure 3.

Click on the Next button. A new window will appear as shown in Figure 4 where you select the location that you wish to save your project files. Do not select the Source Control checkbox at the bottom of the window, then click on the ***Create*** button

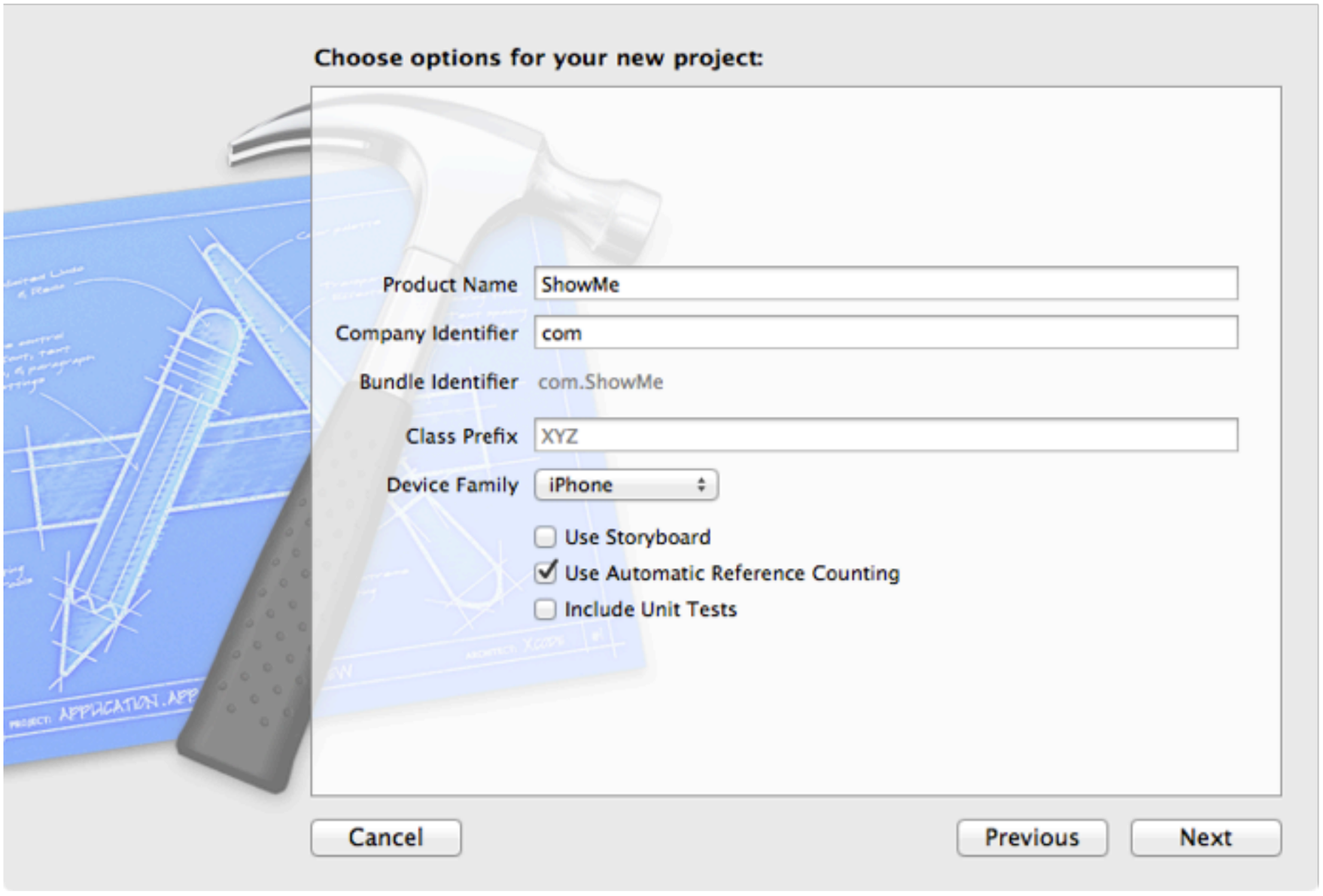

Figure 3 Creating a New Project – Naming the Project

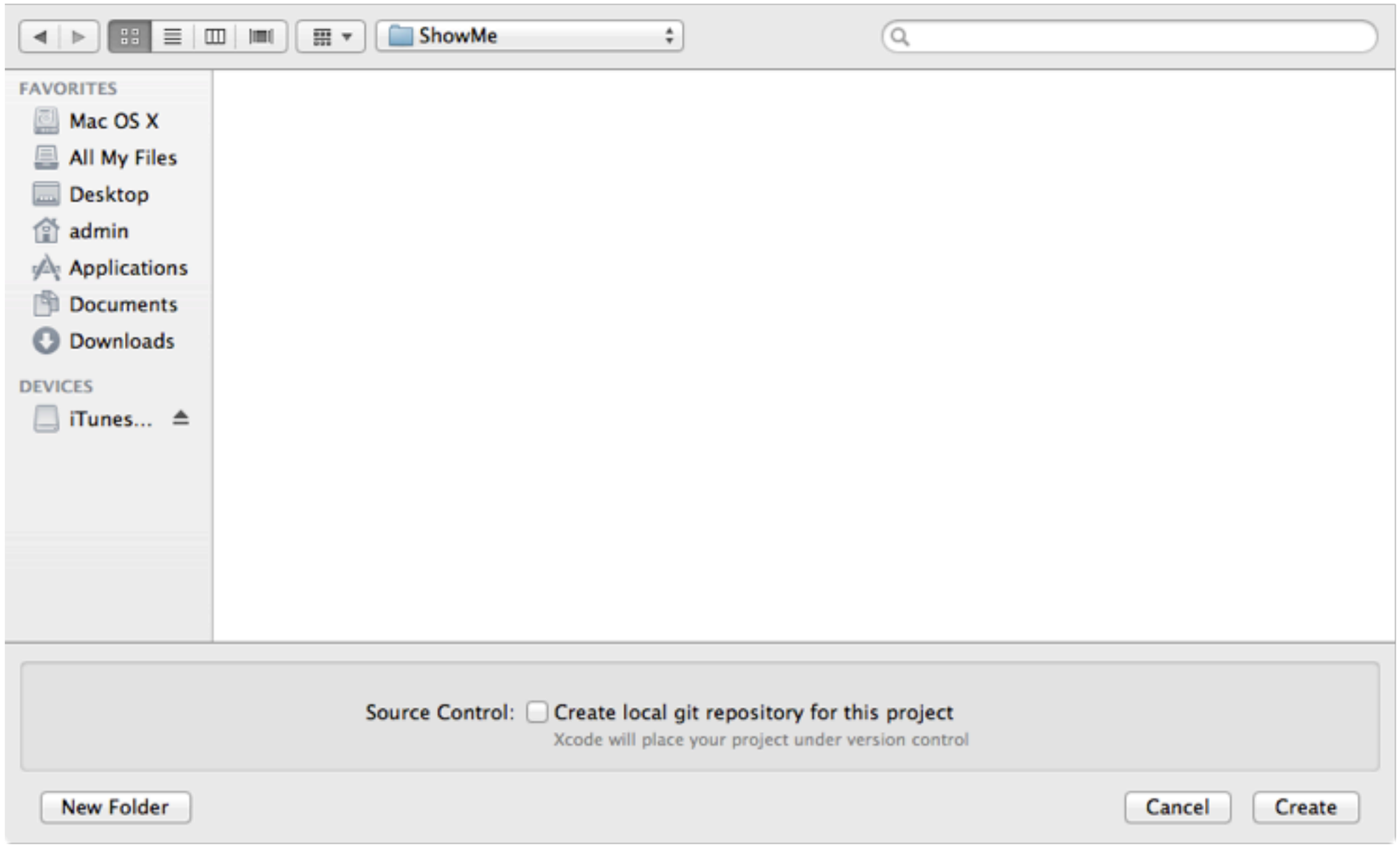

Figure 4 Creating a New Project – Selecting the Location for the Project

We will need a couple of images for this project, so go to the companion website for this book at http:\\epressplus.com and download the statue_icon.png and empire_icon.png files. Drag them into the Supporting File's folder within the Navigator area. Be sure to check the ***Copy items into destination group's folder (if needed)*** checkbox.

Displaying a Map

In this section you will learn how to display the map in your application. The process is simple, and consists of two simple steps.

First you need to add the appropriate frameworks to your project. Then you need to create an instance of the MapView class. That's all you need in order to create a fully functional map!

Click on the name of the project in the upper left part of the navigator panel. Click on the Build Phases tab in the new view. Your Xcode window should appear similar to Figure 5.

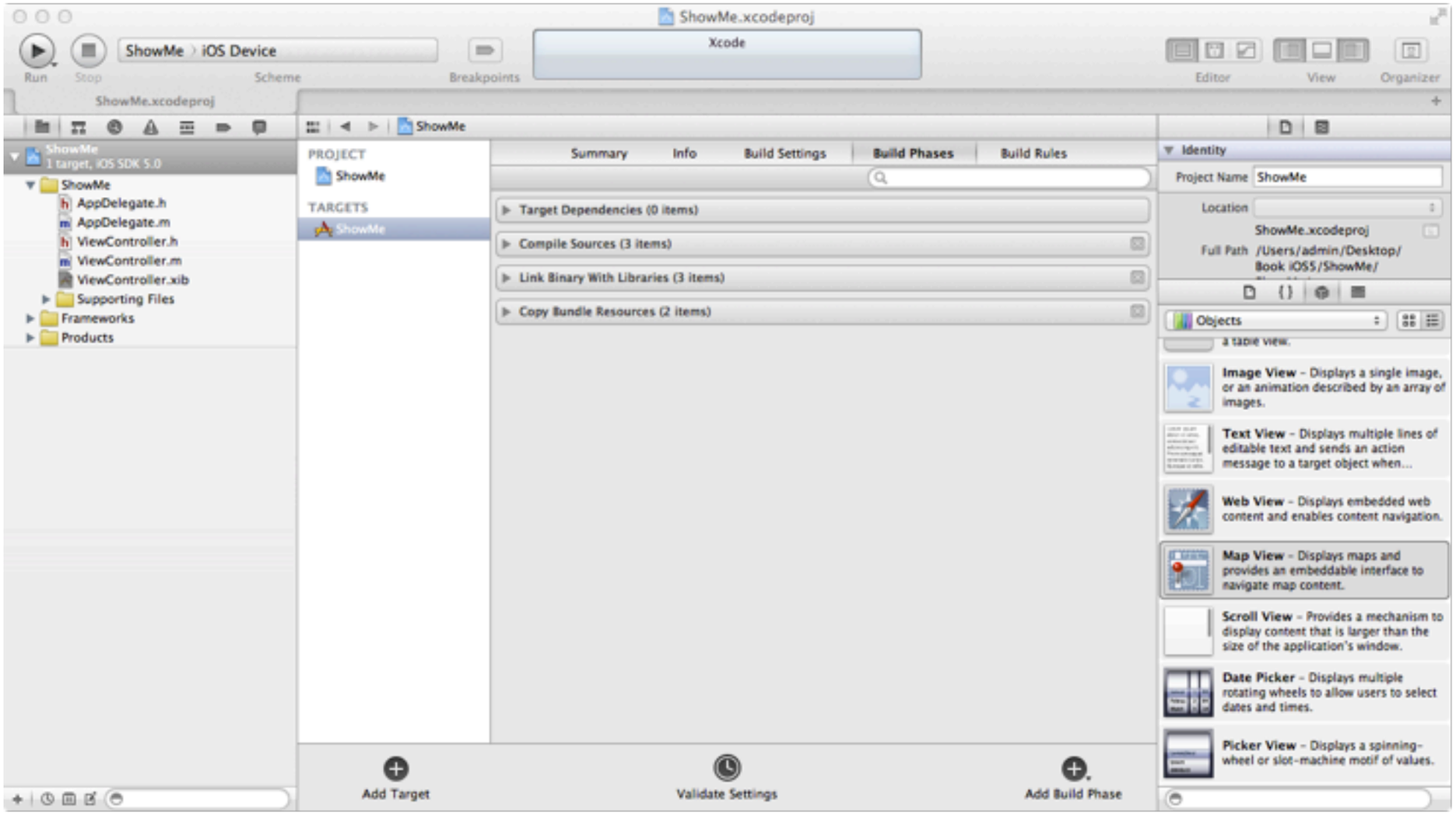

Figure 5 Build Phases in Xcode Window

Click on the arrow next to Link Binary With Libraries to expand the view of that information. The expanded view should appear similar to Figure 6.

Click on the '+' sign shown in Figure 6 and to access a list of additional frameworks and libraries that can be added to the project. Scroll down the list to find and select the CoreLocation.framework as shown in Figure 7 and then click on the Add button.

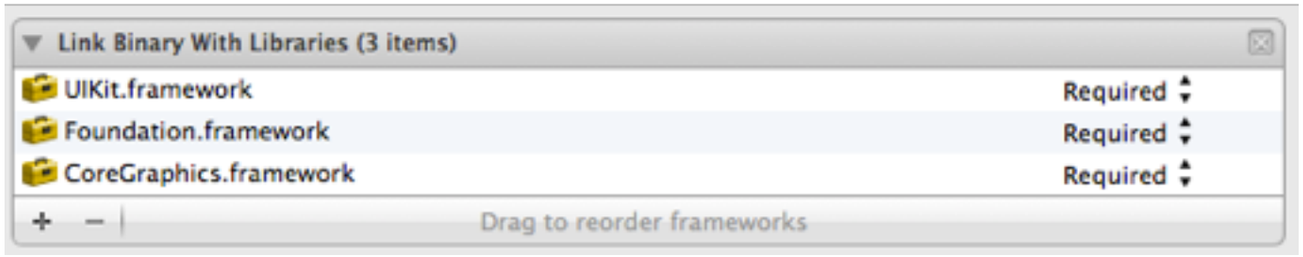

Figure 6 Linked Libraries

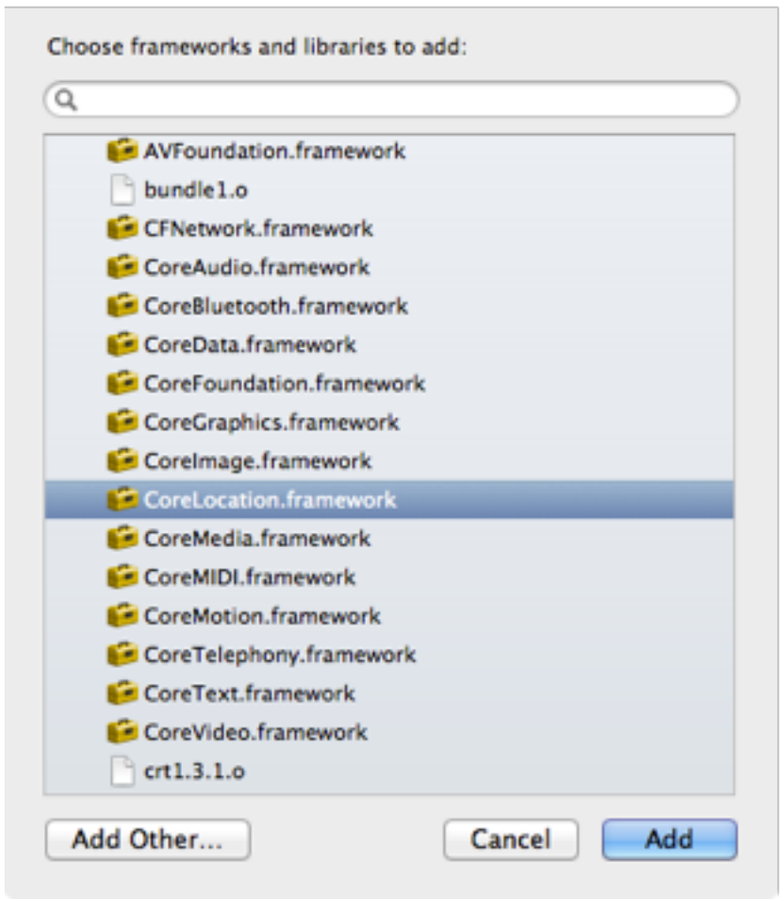

Figure 7 Adding Frameworks

Repeat this same process to add the MapKit.framework. Once this is completed, the Linked libraries should include five frameworks as shown in Figure 8.

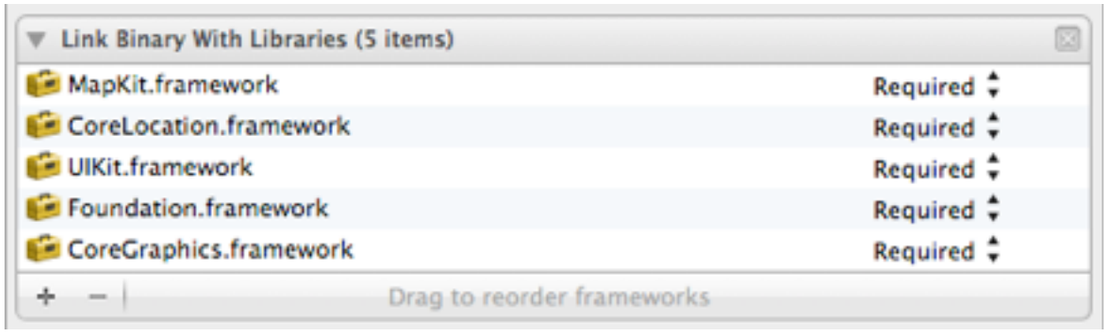

Figure 8 Linked Libraries with Additional Frameworks

Select the ViewController.xib in the Navigator area so that its View appears in the Editor area. From the Objects Library, drag a Map View into the View window. Then drag a Toolbar into the bottom of the View. The Toolbar will have one Bar Button Item in it. Drag two more Bar Button items into the Toolbar. Select each bar button item in the View window; name the first Bar button item Statue of Liberty, the second bar button item Empire State, and name the third bar button item Show All. Your View window should now appear similar to Figure 9.

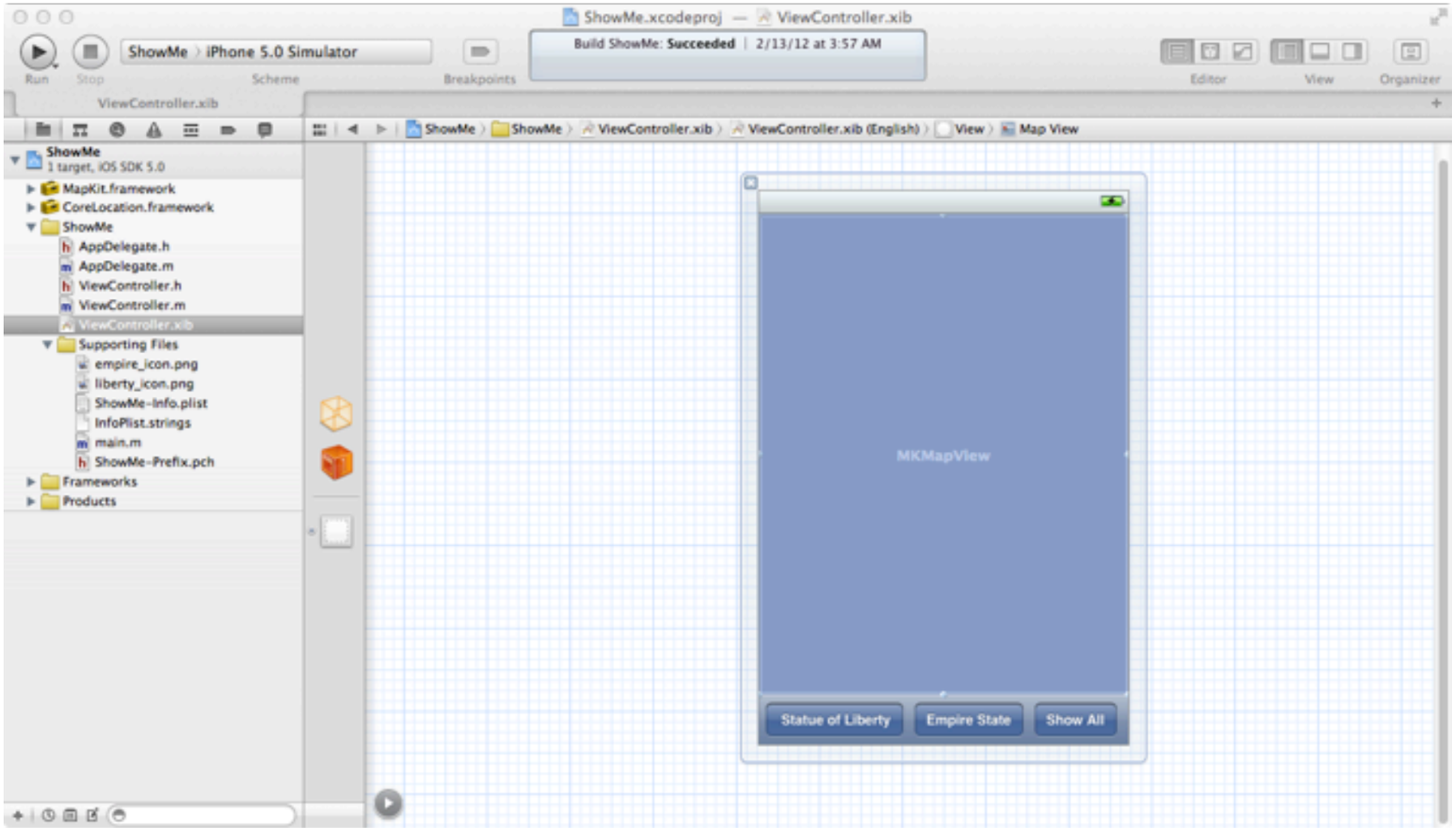

Figure 9 The View After Adding a Map View, Toolbar, and Two Additional Bar Button Items

Now drag a View from the Objects Library into the Editor area. This View will be used to display details about selected objects on the map. Add three elements from the Objects library to this new View. Add a Label, a Text View, and a Toolbar. Make sure that the Editable property of the Text View is NOT selected. Change the label on the Toolbar's button to Dismiss. Your View should appear similar to Figure 10.

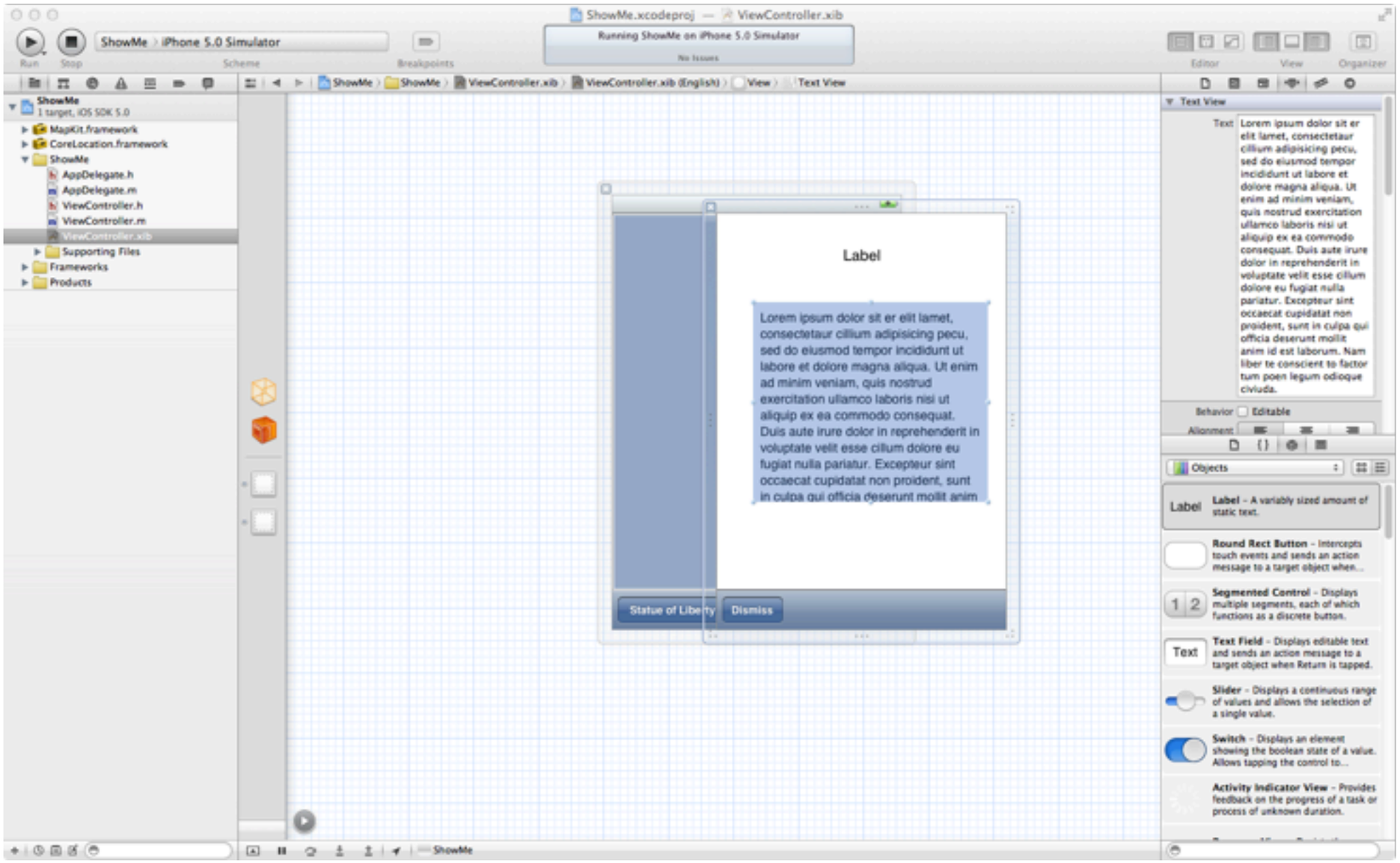

Figure 10 Newly Created View

Click on the Xcode Run button to run the app in the iPhone 5.0 Simulator. Your iPhone simulator should appear similar to Figure 11.

Figure 11 iPhone Simulator

That was simple! In this part of the chapter you have learned how to display a map in an iPhone app. In the next section you will learn how to customize the map.

Customizing the Map

The first part of customizing the code is to connect the map to the code. Open the Assistant Editor so that both the ViewController.xib file and the ViewController.h file are displayed in the Editor area. Ctrl-click on the Map View in the View window and drag from the circle to the right of New

Referencing Outlet in the popup window to the ViewController.h just above the end directive to create an instance of the map view. Call the newly created instance mapView. Then also drag from the circle next to delegate to the File's Owner icon to set the delegate of Map View as shown in Figure 12.

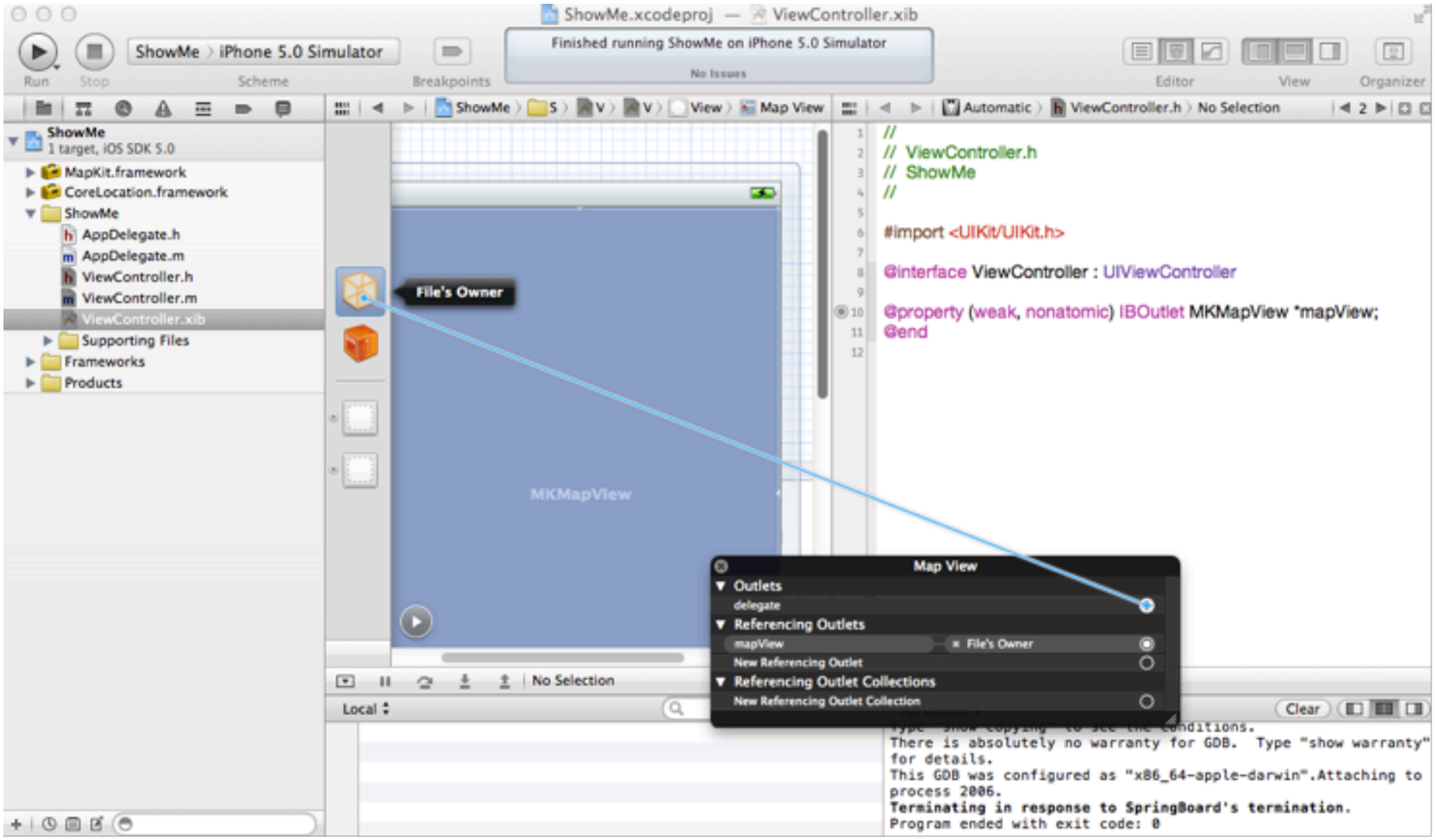

Figure 12 Connecting the Map View to the Code

Select the ViewController.h file and add the following import directive below the other import directive so that the ViewController.h file is as follows.

```
//
//  ViewController.h
//  ShowMe
//

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet MKMapView *mapView;
@end
```

Map Modes

There are quite a lot of things we can do with maps. One of the simplest tasks is to set its display mode. A Map View can be displayed in three different modes: Standard, Hybrid, and Satellite as shown respectively in Figure 13. To set the display mode of a map instance we need to assign the desired mode to the mapType property of an instance. The mapType property accepts the following values, MKMapTypStandard for standard view, MKMapTypeSatellite for Satellite mode and the Hybrid for the Hybrid mode.

```
mapView.mapType = MKMapTypeStandard;
mapView.mapType = MKMapTypeSatellite;
mapView.mapType = MKMapTypeHybrid;
```

We will display the map in the Show Me app in the Hybrid mode. Therefore, open the ViewController.m file and find the ViewDidLoad method. Add the following code to the ViewDidLoad method.

```
    self.mapView.mapType=MKMapTypeHybrid;
```

So your viewDidLoad method appears as follows.

```
- (void)viewDidLoad
{
    [super viewDidLoad];
    // Do any additional setup after loading the view, typically from a nib.
    self.mapView.mapType=MKMapTypeHybrid;
}
```

Figure 13 Different Modes of a Map View: Standard, Satellite and Hybrid

Showing User Location

Most of the apps that are using the map view display the user‘s location on the map. There are several different ways to do this. One way is to use the showsUserLocation property. This property can have values of YES or NO. Set the property to YES by adding the following statement to the end of the ViewDidLoad method, just after the statement just added.

```
self.mapView.showsUserLocation=YES;
```

Click on the Run button to start the application in the Simulator. This time you should see an Alert that checks if the user wishes to allow the app to show the user's current location. Select OK. The simulator should now appear similar to Figure 14. Notice the blue dot with the circle around it. By default, the simulator uses the location of Apple's headquarters in Cupertino, California as the current location. When the app is installed on an iOS device it will show the user's real location. You can override the default behavior in the simulator and select a different custom location by navigating to Debug > Location and then selecting your desired location.

Figure 14 Displaying the User's Location

Adding and Displaying Annotations on a Map

The Map kit framework provides the means to annotate maps with custom information. An annotation might be any object that conforms to the MKAnnotation protocol by implementing the coordinate property. Each annotation object contains information about the annotation's coordinates on the map. It can also contain additional information that can be displayed in a callout. Annotations only contain data, to visualize that data, a map view uses an annotation view which is instance of MKAnnotationView. The simplest MKAnnotationView subclass is the MKPinAnnotationView. MapView uses a queueing mechanism to display only the annotation views that are visible on the screen. The process of handling the annotations by map view is generally as follows.

1. Create an annotation.
2. Add the annotation to the map.
3. Customize the view in the map view delegate if needed.
4. Have the map view display the annotation view.

In our case, we first declare actions/methods that will connect the Toolbar button items to methods in the code so that the user will be able to to display annotations or not on the map. Click on the ViewController.xib file to open it in the Editor area. Select the Statue of Liberty bar button item, then ctrl-click on it. in the popup window that appears, click in the circle to the right of selector in the Sent Actions section and drag to the ViewController.h file just above the end directive as shown in Figure 15--enter showStatue for the name of the action. Do the same for the other two bar buttons naming them showEmpire and showAll. After doing this, the ViewController.h file should appear as follows.

```
//
//  ViewController.h
//  ShowMe
//

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

@interface ViewController : UIViewController

@property (weak, nonatomic) IBOutlet MKMapView *mapView;
- (IBAction)showStatue:(id)sender;
- (IBAction)showEmpire:(id)sender;
- (IBAction)showAll:(id)sender;
@end
```

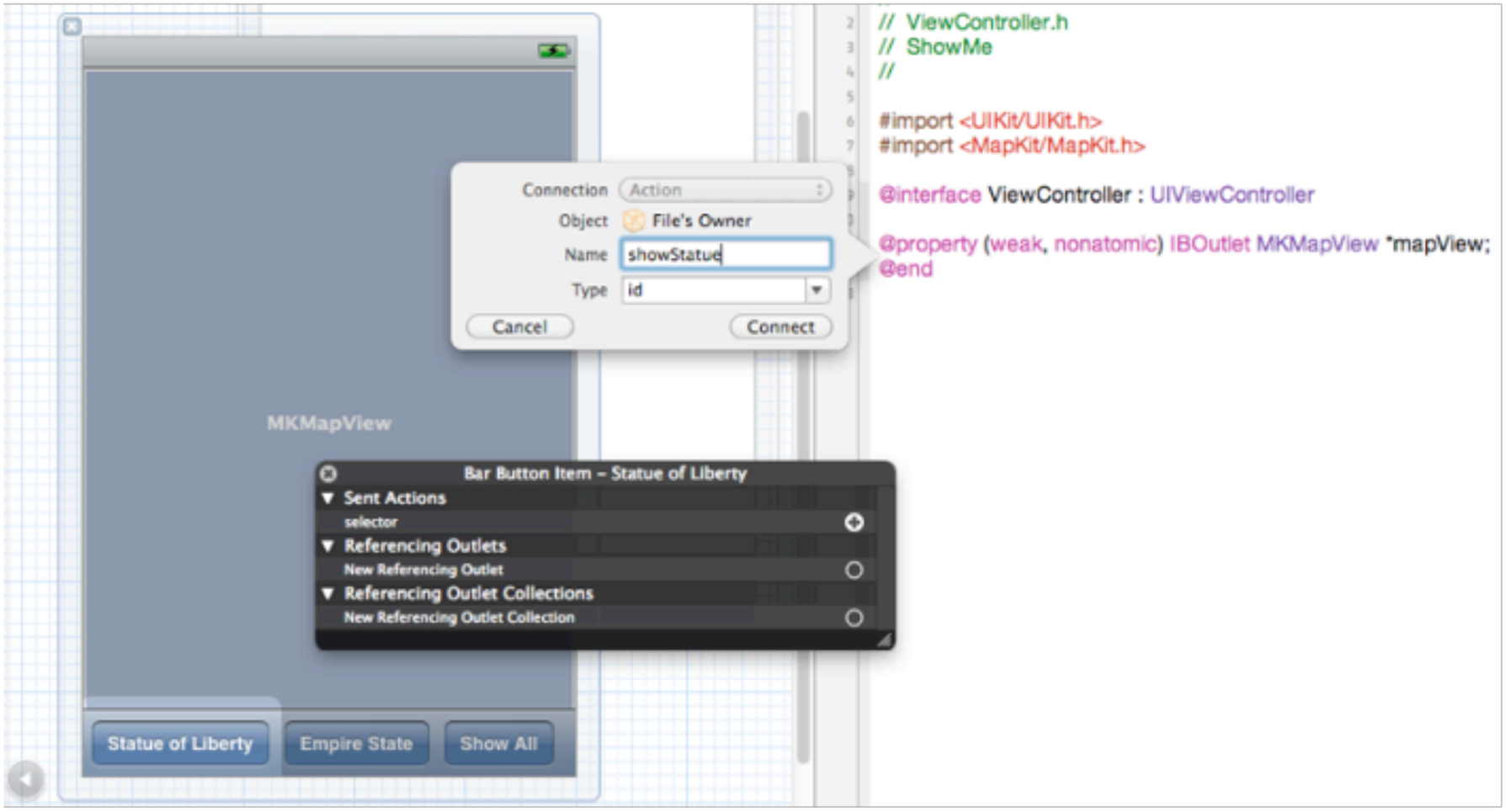

Figure 15 Adding an IBAction for the Statue of Liberty Bar Button

Add the following two declarations to the ViewController.h file.

```
- (void) zoomToNewYork;
- (void) showDetails:(id)sender;
```

Now, add the following two empty method bodies for those two declarations just before the end directive in the ViewController.m file. We will add the code inside these two methods later.

```
-(void) zoomToNewYork {
}

-(void)showDetails:(id)sender {
}
```

Ctrl-click on the showAll bar button item again and drag from the circle to the right of New Referencing Outlet to the ViewController.h file just above the end directive. Enter showOrRemoveAllButton for the name of the new IBOutlet. We need this outlet connection so that we can change the text on the label while the app is running. We don't need outlets for the other two bar button items.

Next ctrl-click on the second view in the ViewController.xib file and drag a New Referencing Outlet to the ViewController.h file just above the end directive, enter detailsView for the name of the new outlet connection. Also click and then ctrl-click on the Dismiss bar button item in this view and and drag from the circle to the right of selector to the ViewController.h file, enter dismissView for the name of the new action. Next create outlet connections for the text view and the label in the detailsView. Enter textView for the name of the text view, and titleLabel for the title of the label.
Add comments to indicate which actions and outlets belong to each view. Your ViewController.h file should now appears as follows.

```
//
//  ViewController.h
//  ShowMe
//

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

@interface ViewController : UIViewController

// Map View and its elements
@property (weak, nonatomic) IBOutlet MKMapView *mapView;
- (IBAction)showStatue:(id)sender;
- (IBAction)showEmpire:(id)sender;
- (IBAction)showAll:(id)sender;
- (void) zoomToNewYork;
- (void) showDetails:(id)sender;
@property (weak, nonatomic) IBOutlet UIBarButtonItem *showOrRemoveAllButton;

// Details View and its elements
@property (strong, nonatomic) IBOutlet UIView *detailsView;
```

```
- (IBAction)dismissView:(id)sender;
@property (weak, nonatomic) IBOutlet UITextView *textView;
@property (weak, nonatomic) IBOutlet UILabel *titleLabel;
@end
```

We have now created four IBActions that will be necessary to provide interaction with the application. One for displaying the Statue of Liberty, one for displaying the Empire State Building, one for removing or showing all annotations, and one for dismissing/hiding the detailsView. We also added a zoomToNewYork method that will change the displayed region of the map to the New York region and a showDetails method that will be used to display the detailsView. Furthermore, we created several outlet connections. The showOrRemoveAllButton will be used to show or remove all annotations from the map, detailsView is a reference to the second view of the application that will display information about an annotation. titleLabel will be used to display the information about title of the annotation, and textView will be used for displaying detailed information about the annotation objects. Click on the Run button in Xcode to make sure that the code builds successfully.

Now we are ready to add annotations objects to the class. Add the next four lines of code after the interface directive in the ViewController.h file, this means that all instances of this view controller class will contain these two annotations that will be used to display the annotation for the Statue of Liberty and the annotation for the Empire State Building.

```
{
    MKPointAnnotation *statueAnnotation;
    MKPointAnnotation *empireAnnotation;
}
```

Then, to allocate and initialize these objects, create coordinates using latitude and longitude values, assign those coordinates to the annotation objects, and then set the value of the title for the annotation objects, add the following code to the end of the viewDidLoad method in the ViewController.m file.

```
    statueAnnotation  = [[MKPointAnnotation alloc]init];
    empireAnnotation  = [[MKPointAnnotation alloc]init];
    CLLocationCoordinate2D cord = CLLocationCoordinate2DMake(40.689, -74.044);
    CLLocationCoordinate2D cord2 = CLLocationCoordinate2DMake(40.74, -73.98);
    statueAnnotation.coordinate = cord;
    empireAnnotation.coordinate = cord2;
    statueAnnotation.title=@"Statue of Liberty";
    empireAnnotation.title=@"Empire State Building";
```

We will also need callouts for our annotations so add the following directives to the top of the ViewController.m file, immediately following the #import directive.

```
#define STATUE_TAG 10
#define EMPIRE_TAG 20
```

Now we can add the code to the methods that put the annotations on the map and to zoom in on the region of the map where the annotation is located. We will add the code to the zoom method in the next section. Add the following code to the inside of the showStatue method in the ViewController.m file.

```
    [mapView addAnnotation:statueAnnotation];
    [self zoomToNewYork];
```

Add the following code to the inside of the showEmpire method in the ViewController.m file.

```
    [mapView addAnnotation:empireAnnotation];
    [self zoomToNewYork];
```

Then, add the following code to the inside of the showAll method in the ViewController.m file.

```
  if([showOrRemoveAllButton.title isEqualToString: @"Show All"]) {
      // Remove the annotations if there are any
      [mapView removeAnnotations:mapView.annotations];
      // Fill array with annotations
      NSArray * annotations =[NSArray arrayWithObjects:statueAnnotation, empireAnnotation, nil];
      // Add the array of annotations
      [mapView addAnnotations:annotations];
      [self zoomToNewYork];
      showOrRemoveAllButton.title = @"Hide All";
  }
   else {
      [mapView removeAnnotations:mapView.annotations];
      showOrRemoveAllButton.title = @"Show All";
  }
```

In the code just added, if the title is equal to ‘Show All’ then we are removing any annotations that were previously displayed on the map. Notice that we used the isEqualToString method to compare the string rather than the equivalence operator, since we want to compare the contents of the object--not the address of the objects. Strings should be always compared using isEqualToString method. We then create an array that contains our two annotations objects, and add the array to the map using the addAnnotations method provided by the MKMapView class. Next we use the

zoomToNewYork method to provide a closer view to the user, and finally the title of the button that shows and hides the annotations is changed to Hide All. If the title of the button is different than Show All the block of code in the else is executed--the annotations are removed from the map using the removeAnnotations method and the title of the button is changed to Show All.

Go ahead and see if it works. Click on the Run button to start the app in the simulator. Tap on the Statue of Liberty and the Empire State buttons. You should be able to see two red pins on the map, but they are so close together that in this scale we may be only able to distinguish one red pin as shown in Figure 16. You can zoom in or out the map just like you would on an actual iOS device. To zoom in double click on the map in the simulator. To zoom out hold the Option key and move your mouse, you can zoom in using this technique also if you prefer. Zoom in on the map. You should be able to see two pins clearly now as shown in the right side of Figure 16. Tap on the pins. The label should show up. You specified the text of the label by assigning the value to the title property of the annotation previously in the viewDidLoad method.

Figure 16 Map View with Annotations

KEY POINT

You can add the annotations to the map view using addAnnotation or addAnnotations method. At first you need to create the annotation object, and specify the latitude and longitude coordinates of the object. To remove annotations use either the removeAnnotation or removeAnnotations methods. Map view annotations are stored in an array, they can be retrieved by accessing the map view annotations property (mapView.annotations).

Changing the Visible Region of the Map

We can change the visible part of the map programmatically. For this app it makes sense to display the map initially with a closer view of New York. We will do this setting the visible region of the map to New York using the setRegion: animated: method. Add the the following code inside the zoomToNewYork method in the ViewController.m file.

```
CLLocationCoordinate2D center = CLLocationCoordinate2DMake(40.72, -74.01);
MKCoordinateSpan span = MKCoordinateSpanMake(0.1, 0.1);
MKCoordinateRegion region = MKCoordinateRegionMake(center, span);
[mapView setRegion:region animated:YES];
```

Region is a structure in the MKCoordinateRegion type that consists of two other structures, span and center. Span is responsible to for setting the zoom level of the map, and center determines the center of the region. Click on the Run button to test this new functionality in the simulator. Now when you click on any of the buttons in the simulator you should be able to see the how the map scrolls from the default location to the region set in the zoomToNewYork method.

Customizing the Annotations.

In order to customize the way annotations appear and/or function on the map we can customize mapView: viewForAnnotation: method which is called just before the map displays the annotation views on the map. We will construct this method incrementally via several steps. Add this method with the following custom code in it just before the end directive in the ViewController.m file. A few warnings will appear -- do not be alarmed, this is because the method isn't finished yet.

```
- (MKAnnotationView *)mapView:(MKMapView *)_mapView viewForAnnotation:(id < MKAnnotation >)annotation {
  static NSString *statueIdentifier=@"STATUE";
  static NSString *empireStateIdentifier=@"EMPIRE";
  if ([annotation isKindOfClass:[MKUserLocation class]]) {
     return nil;
```

```
    }
  return nil;
}
```

In the method just added, we created two static strings. The map view only displays annotations that are currently visible on the map view. Other annotations are stored in a queue, waiting for the map view to display them on the screen. In order to identify and dequee an annotation you need to use an unique identifier. And that's exactly what we created: a string that will serve as an unique identifier for each annotation. In the next step we check if the annotation being currently processed by the delegate is an annotation that displays our position (blue dot on the map). If the condition is satisfied we return nil, because further processing is not necessary. Now insert the following code just before the second return nil; in the viewForAnnotation method.

```
    if([annotation isEqual:statueAnnotation]) {
        MKAnnotationView *customAnnotationView =[mapView dequeueReusableAnnotationViewWithIdentifier:statueIdentifier];
        if(!customAnnotationView) {
            customAnnotationView=[[MKAnnotationView alloc] initWithAnnotation:annotation reuseIdentifier:statueIdentifier];
            UIImage *pinImage = [UIImage imageNamed:@"liberty_icon.png"];
            [customAnnotationView setImage:pinImage];
            customAnnotationView.canShowCallout = YES;
            UIButton *rightButton = [UIButton buttonWithType:UIButtonTypeDetailDisclosure];
            [rightButton addTarget:self action:@selector(showDetails:) forControlEvents:UIControlEventTouchUpInside];
            rightButton.tag=STATUE_TAG;
            customAnnotationView.rightCalloutAccessoryView = rightButton;
        }
        return customAnnotationView;
    }
```

If the condition for the if statement just added is satisfied, then we know that we are dealing with the annotation for the Statute of Liberty, so we use the dequeueReusableAnnotationViewWithIndentifier method to retrieve the instance of the annotation view with the statueIdentifier from memory. The condition in the next if statement just added checks if the instance of the annotation view already exists in memory. If yes, the only thing we need to do is to return the instance to the map view by calling the return customAnnotationView statement. If the customAnnotationView doesn't exist or it's equal to nil, it needs to be created. So we allocate and initialize the annotation view using the initWithAnnotation: reuseIdentifier method. Then we create an instance of the UIImage and set this image to be displayed instead of the default image of the red pin. The next statements set up the look and behavior of the callout. The simplest callout is a label that is displayed whenever a user taps on the annotation view on the map. In

our case we want to do something more interesting. We want to add a button that when tapped will display extra information about the annotated object. First we create an instance of a UIButton of type UIButtonTypeDetailDisclosure.
Then we use the addTarget: action: forControlEvents method to set the functionality of the button. The first argument sets the class that will be used for the target of the button. The second argument set's the action that will be executed whenever the button is used. Notice the @selector, it's syntax is the following @selector(nameOfTheMethod). Notice the semicolon after the showDetails method. It indicated that the showDetails method will take one argument. Finally we set the third argument as a UIControlEventTouchInside. This code is the programmatic way of creating an action instead of creating an action connection in the xib file. Then we associate the tag property with the STATUE_TAG value. In the last statement we set the rightCalloutAccessoryView to a button. Next, add equivalent code for the Empire State Building annotation immediately after the code just previously added.

```
if([annotation isEqual:empireAnnotation]) {
    MKAnnotationView *customAnnotationView =[mapView dequeueReusableAnnotationViewWithIdentifier:empireStateIdentifier];
    if(!customAnnotationView) {
       customAnnotationView=[[MKAnnotationView alloc] initWithAnnotation:annotation reuseIdentifier:empireStateIdentifier];
        UIImage *pinImage = [UIImage imageNamed:@"empire_icon.png"];
        [customAnnotationView setImage:pinImage];
        customAnnotationView.canShowCallout = YES;
        UIButton* rightButton = [UIButton buttonWithType:UIButtonTypeDetailDisclosure];
        [rightButton addTarget:self action:@selector(showDetails:) forControlEvents:UIControlEventTouchUpInside];
        customAnnotationView.rightCalloutAccessoryView = rightButton;
        rightButton.tag= EMPIRE_TAG;
      }
      return customAnnotationView;
  }
```

The complete viewForAnnotation method should now appear as follows.

```
- (MKAnnotationView *)mapView:(MKMapView *)_mapView viewForAnnotation:(id < MKAnnotation >)annotation{
    static NSString *statueIdentifier=@"STATUE";
    static NSString *empireStateIdentifier=@"EMPIRE";
    if ([annotation isKindOfClass:[MKUserLocation class]]) {
        return nil;
    }
    if([annotation isEqual:statueAnnotation]) {
        MKAnnotationView *customAnnotationView =[mapView dequeueReusableAnnotationViewWithIdentifier:statueIdentifier];
        if(!customAnnotationView) {
```

```
        customAnnotationView=[[MKAnnotationView alloc] initWithAnnotation:annotation reuseIdentifier:statueIdentifier];
        UIImage *pinImage = [UIImage imageNamed:@"liberty_icon.png"];
        [customAnnotationView setImage:pinImage];
        customAnnotationView.canShowCallout = YES;
        UIButton *rightButton = [UIButton buttonWithType:UIButtonTypeDetailDisclosure];
        [rightButton addTarget:self action:@selector(showDetails:) forControlEvents:UIControlEventTouchUpInside];
        rightButton.tag=STATUE_TAG;
        customAnnotationView.rightCalloutAccessoryView = rightButton;
    }
    return customAnnotationView;
}
if([annotation isEqual:empireAnnotation]) {
    MKAnnotationView *customAnnotationView =[mapView dequeueReusableAnnotationViewWithIdentifier:empireStateIdentifier];
    if(!customAnnotationView) {
        customAnnotationView=[[MKAnnotationView alloc] initWithAnnotation:annotation reuseIdentifier:empireStateIdentifier];
        UIImage *pinImage = [UIImage imageNamed:@"empire_icon.png"];
        [customAnnotationView setImage:pinImage];
        customAnnotationView.canShowCallout = YES;
        UIButton* rightButton = [UIButton buttonWithType:UIButtonTypeDetailDisclosure];
        [rightButton addTarget:self action:@selector(showDetails:) forControlEvents:UIControlEventTouchUpInside];
        customAnnotationView.rightCalloutAccessoryView = rightButton;
        rightButton.tag= EMPIRE_TAG;
      }
      return customAnnotationView;
   }
   return nil;
}
```

Click on the Xcode Run button to test the app now. Tap on each of the buttons to see what happens. You should be able to see the images for the Statute of Liberty and the Empire State Building now, as shown in Figure 17 instead of the red pins displayed previously. Tap on each annotation, the callout should now contain the title and the blue details disclosure button to the right of the title. At this point, nothing happens if you tap on the details disclosure button--we will add that functionality in the next section.

Figure 17 Callouts with Buttons.

Working with the Details View

In this section we will finish creating the detailed view for the annotations. Add the following code to the showDetails: method in the ViewController.m file.

```
if([sender tag]== STATUE_TAG)
{
    titleLabel.text=@"Statue of Liberty";
    textView.text=@"The Statue of Liberty  is a colossal neoclassical sculpture on Liberty Island in New York Harbor, designed by Frédéric
                    Bartholdi and dedicated on October 28, 1886.";
}
else if([sender tag] == EMPIRE_TAG)
{
    titleLabel.text=@"Empire State Building";
    textView.text=@"The Empire State Building is a 102 story landmark skyscraper and American cultural icon in New York City at the intersection
                    of Fifth Avenue and West 34th Street. ";
}
[self.view addSubview: detailsView];
```

This method is called when a user taps on the details disclosure button in the callout. The showDetails method has one argument, which is (id) sender. The iOS passes information about the object that invoked that method and stores it in the sender variable. Recall that each annotation in the iOS has it's own tag, and we can use this tag to identify the objects on the screen. In the showDetails method we use the tag property to determine which button was tapped by user. If the tag property is equal to the one stored in the STATUE_TAG, that means that user tapped on the button in the statueAnnotation callout, and if it is equal to EMPIRE_TAG, the user tapped on the empireAnnotation callout. Once we determine which annotation was tapped, we change the text property of the titleLabel and the textView so it displays the information related to annotation that was tapped. Then we display the detailsView with the addSubview method of the view.

Now we will provide the functionality for the Dismiss bar button item in the details view. The dismissView method is executed whenever user taps on the Dismiss button in the details view. The removeFromSuperview method removes the detailsView from the view, so the main view with map is displayed again Add the following statement to the dismissView method in the ViewController.m file.

```
[detailsView removeFromSuperview];
```

Click on the Xcode Run button to test your app. Tap on the buttons in the annotation callouts. You should see screen similar to the one in the Figure 18. Tap on the Dismiss button. You should go back to the main screen of the app with the map. You may want to make adjustments to the attributes for the label and the text view so that they appear as you wish.

Congratulations! You are done, and your app is ready!

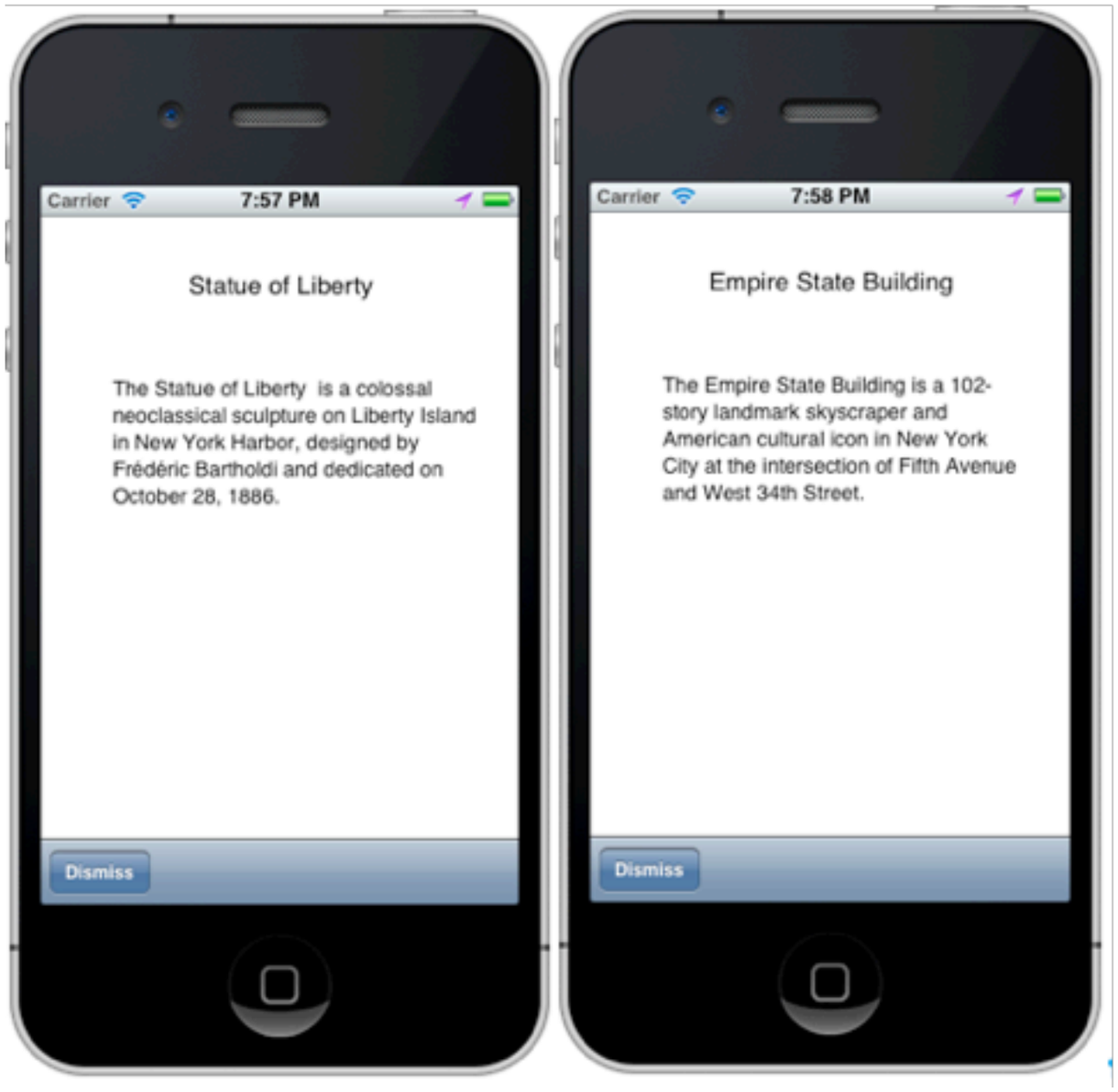

Figure 18 Details View

Summary

In this chapter you have learned about basic concepts related to the Map Kit framework and built an application that displays two annotations on a map. The MapKit framework works in conjunction with the CoreLocation framework and those frameworks need to be linked in your project to display an interactive map. The map is displayed by using an instance of the MKMapView, which can be added to a .xib file or programmatically. A Map view can display a user's location, change the visible region of the map, and display annotations. Annotations store information about the coordinates of the location. They can store also information about title and subtitle of the annotation callout which is displayed when user taps on the annotation. The visual representation of the annotation that we can see on the screen needs to conform tom MKAnnotationView protocol. To customize the look and behavior of the annotation view the map view delegate is used. In this chapter we used the delegate method to change the default 'pin' image of the annotation from the default red pin to a custom image, and we added a details disclosure button to the callout to display a view with details about the location of the annotation.

Review Questions

1. Describe the steps that need to be done in order to display a fully functional map in your app.

2. How do you display the user's location?

3. How do you change the default location settings in the iOS simulator?

4. Which method would you use to change the visible part of the map?

5. Which frameworks should be used in order to display a fully functional map?

6. Describe the steps that you need to make to display annotations on the map.

7. Which method do you use to display/remove an annotation, which method should be used to display/remove multiple annotations?

8. How do you customize the look of the annotation view and the callout?

9. What is the tag property, and for what purpose can it be used?

Exercises

1. Write a program that displays San Francisco, and New York annotations on the map.

2. Write a program that allows a user to add two annotations on the screen, and then calculate and display the distance between those two annotations.

Chapter 6

Treats! App - Part 1: Saving and Restoring Data

Treats! App - Part 1: Saving and Restoring Data

Concepts emphasized in this chapter:

- Creating user interface elements and setting attributes
- Integrating multiple views into an app

Introduction

Two new concepts that we will explore in this project involve saving and restoring data so that we can display high scores in a game. We need some sort of a mechanism to store the previous scores, which can persist even after players quit a game so that when playing the game again they can see what their prior best scores were and try to improve or beat their previous high score.

The iOS SDK provides us with two classes that we will use to implement the feature of saving and displaying the high scores for the Treats! app. These classes are NSUserDefaults and UIPickerView. Simple data objects such as NSString, NSArray, NSNumber, BOOL, and NSURL can be stored using the NSUserDefaults class. Generally, the contents of this class are used for storing user preferences such as login and password information, music preferences, and themes in a website. The NSUserDefaults class cannot be used to save images, music, multidimensional arrays, or any other complex data types.

The UIPickerView class implements picker views in user interfaces that use a spinning wheel to display a set of values. Picker views are incorporated in many applications and are fun to use. Dates and times are often displayed using objects implemented in the UIDatePicker which uses a custom subclass of the UIPickerView class. A picker view will be used in the high scores game to display the scores. We will see how to set the number of components and the number of rows in components during the implementation of the High Scores game. A use of a picker view user interface element also requires that the developer understand concepts related to data sources and delegates.

The development of the high scores game will involve the use of user interface elements for multiple views, images, text input, buttons, labels, and a picker view. The code for the high scores game will start the game, restart the game, move an element randomly around the screen, control the game timer, save scores display high scores, and clear the high scores.

Storyline

The storyline for this game is that the player collects as many dog bones as possible to feed the dog in a given time. The player collects the dog bones by touching the dog bone on the screen. The game flow for this storyline is described in Figure 1.

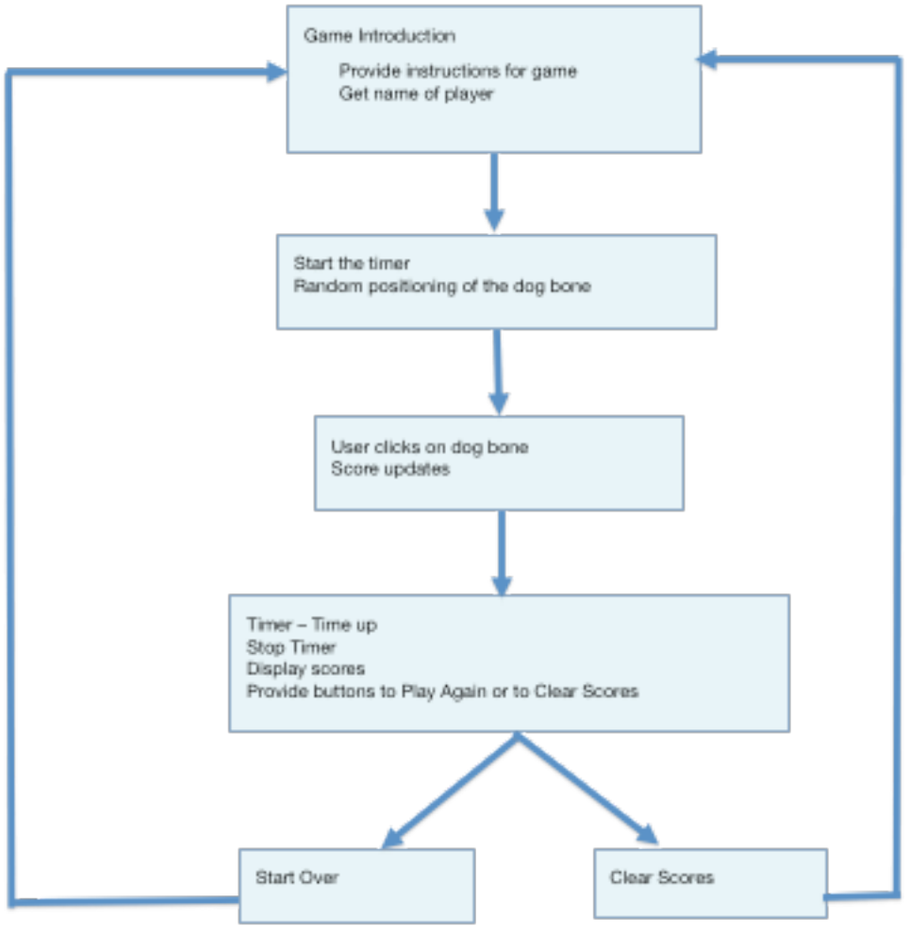

Figure 1 Storyline Flowchart

Game Development Strategy

The overall approach that we will use for creating this particular iPhone game app will consist of four primary phases. First we will design the screens or views that are needed for the game. Second, we will create the user interface elements for each screen using Interface Builder. Third, for each screen we will establish outlet connections from the view to the code. And fourth, we will write the code to provide the functionality for each of the user interface elements to achieve a completely functional application.

Create Treats! App in Xcode

The first thing we need to do is to create a new project in Xcode for our game. Follow the step-by-step instructions below to create a new app.

Open Xcode and click on the ***Create a new Xcode project*** icon to open the New Project window.

Select the ***Single View Application*** template as shown in Figure 2 and then click on the Next button.

On the next screen enter ***Treats!*** into the Product Name text box, enter or leave ***com*** in the Company Identifier, select ***iPhone*** in the Device Family menu. Ensure that the Use Storyboard checkbox is not checked, the Use Automatic Reference Counting checkbox is checked, and the Include Unit Tests checkbox is not checked and shown in Figure 3. Then click on the ***Next*** button.

A standard file save window will appear, select or navigate to the location where you wish to save your project, and ensure that the Source Control checkbox is not checked, and then click on the ***Create*** button. A new project window as shown in Figure 4 will appear.

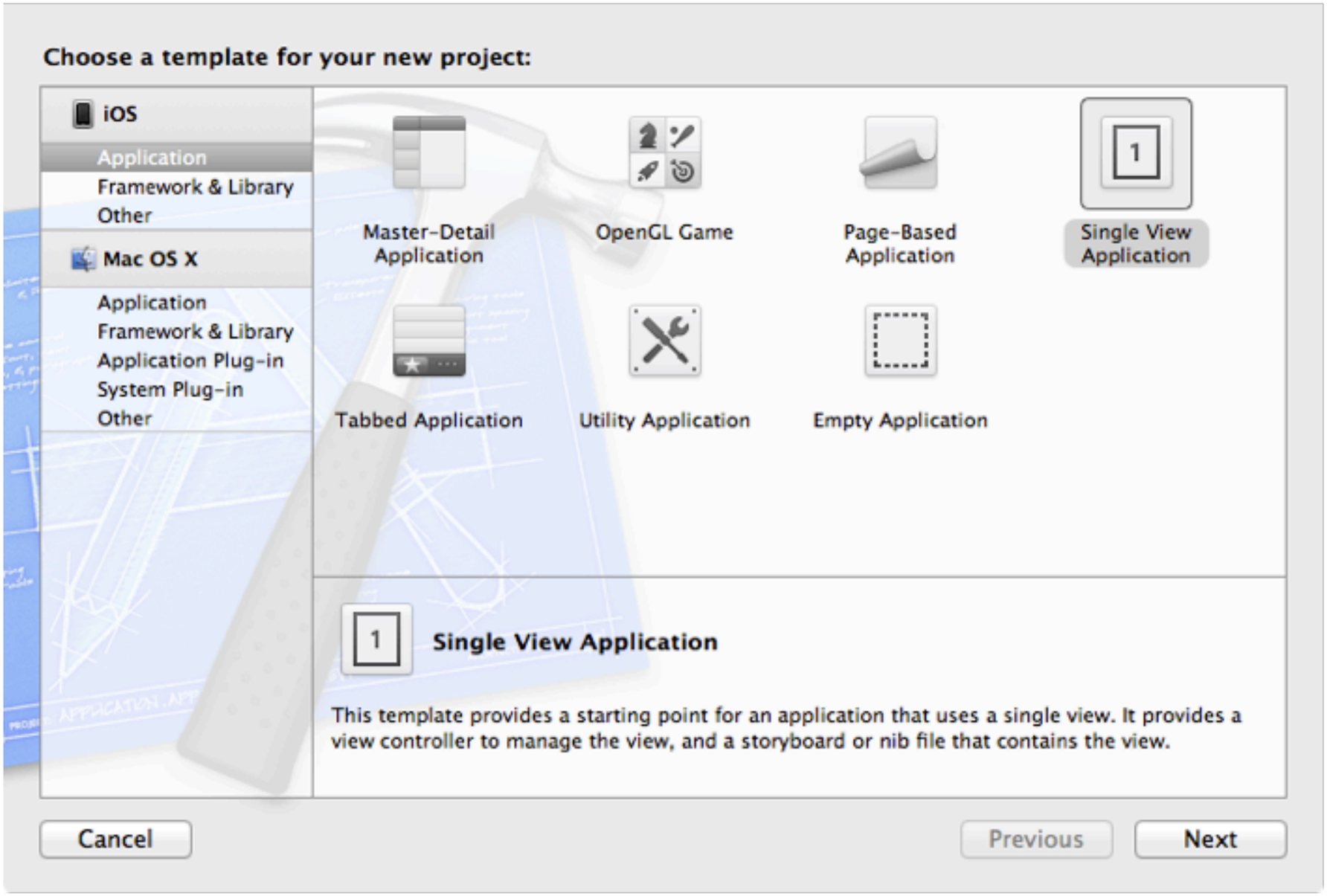

Figure 2 New Project Window

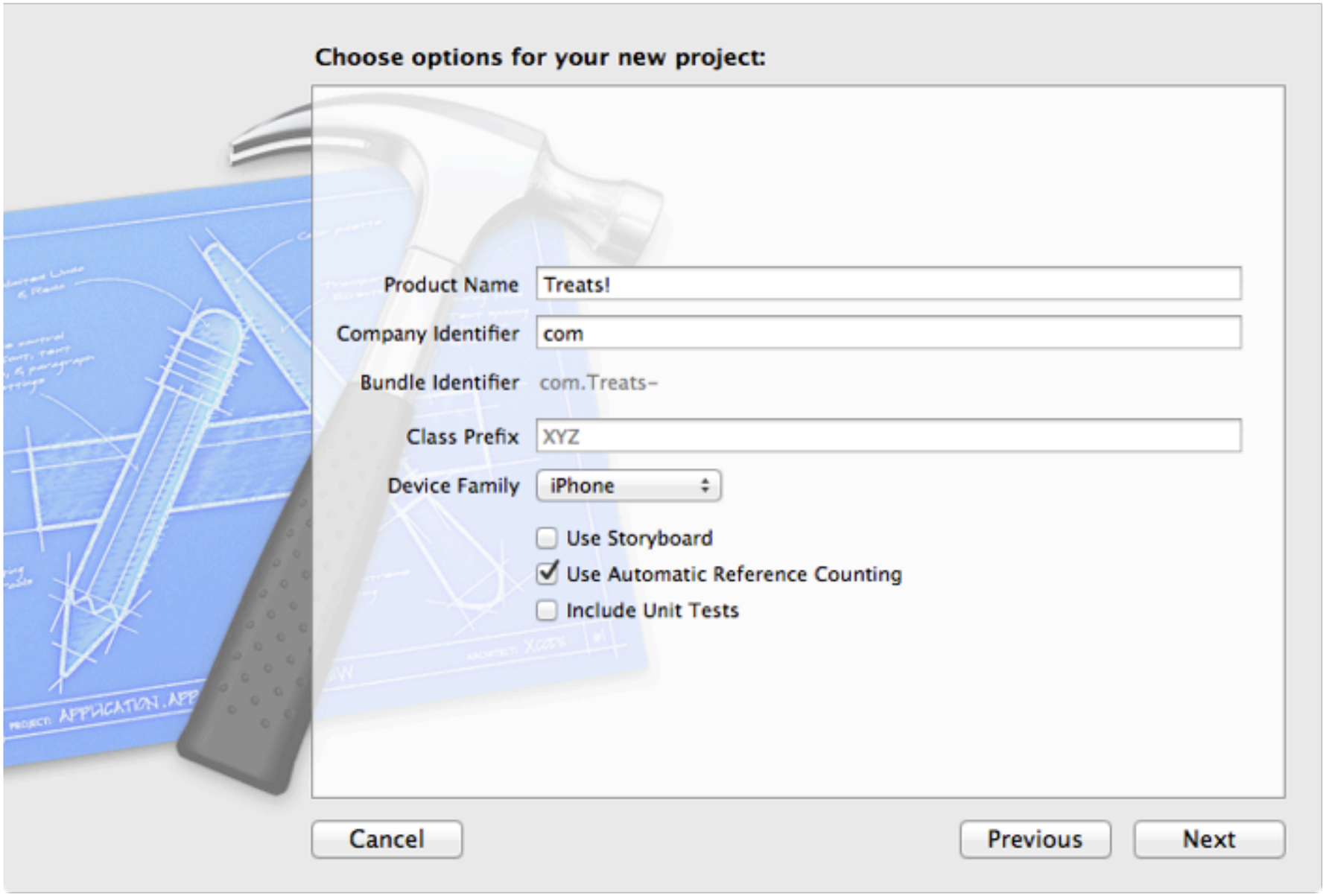

Figure 3 Choosing Options for a New Project

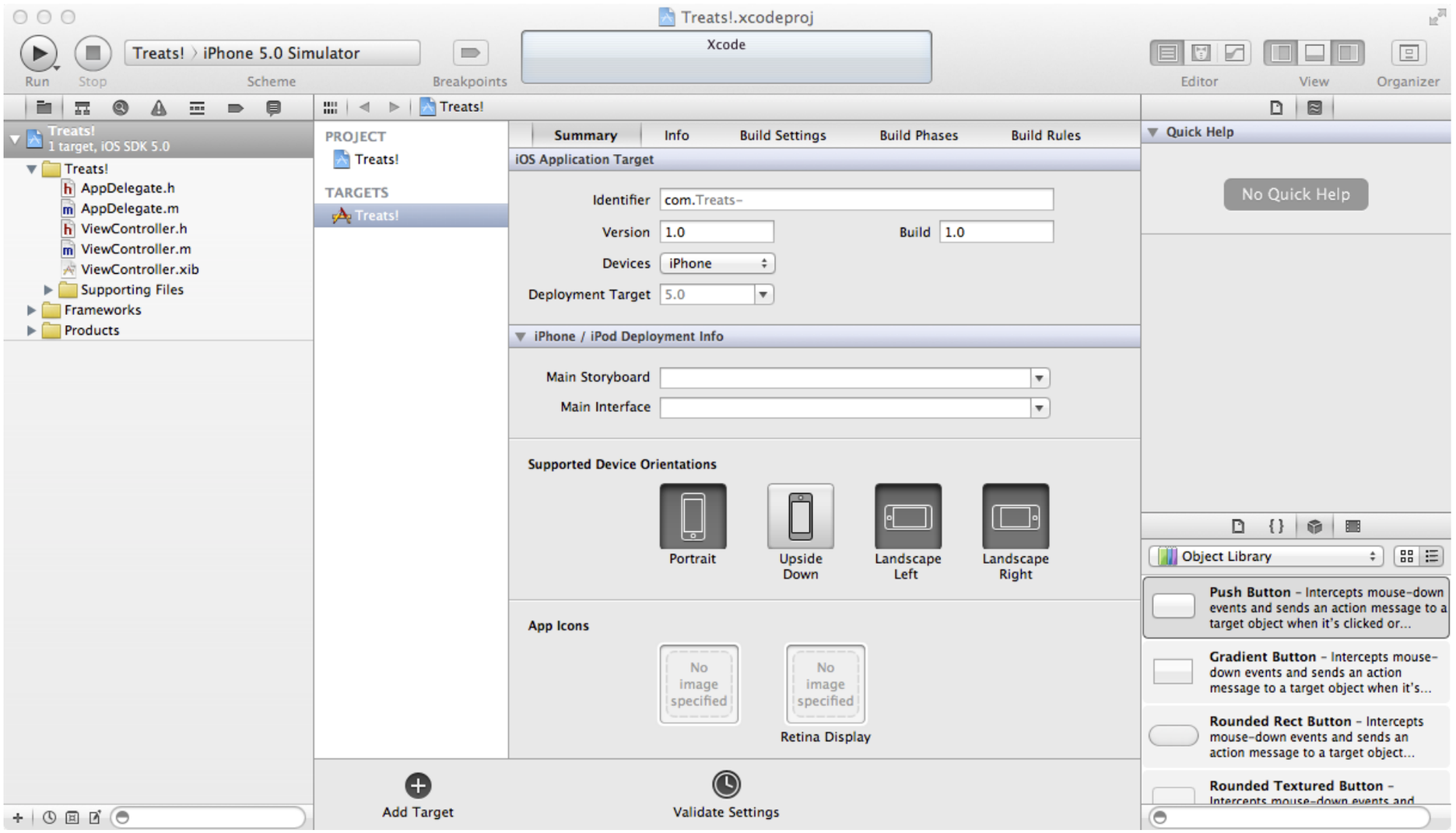

Figure 4 Treats! App Project Window

Look at the Navigator area in the leftmost panel in the Xcode window as shown in Figure 4. If it isn't visible then navigate to View > Navigators > Show Project Navigator.

Next, we will create a folder to the images needed for this app. Folders are referred to as Groups in Xcode. Ctrl-click on the name of the project inside the Project Navigator and then in the popup window select New Group as shown in Figure 5

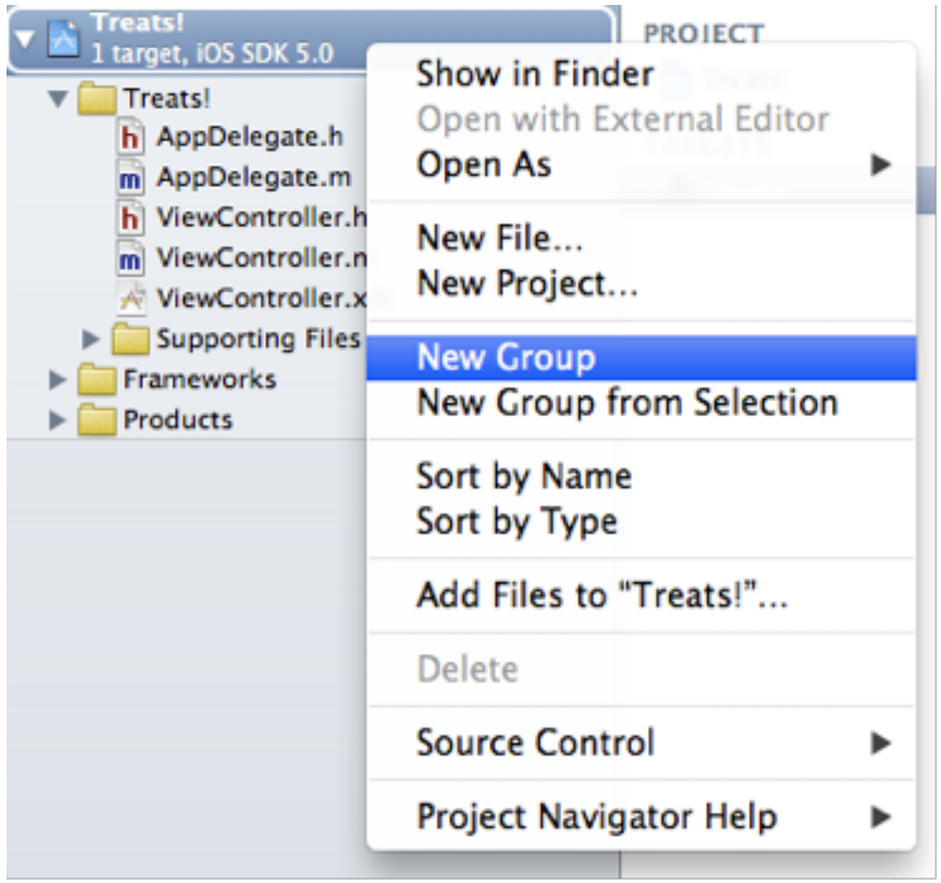

Figure 5 Creating a New Group

Select the new folder labeled New Group, then click on it again to edit its name, and change its name to ***Images***. Your Project Navigator area should now appear similar to Figure 6.

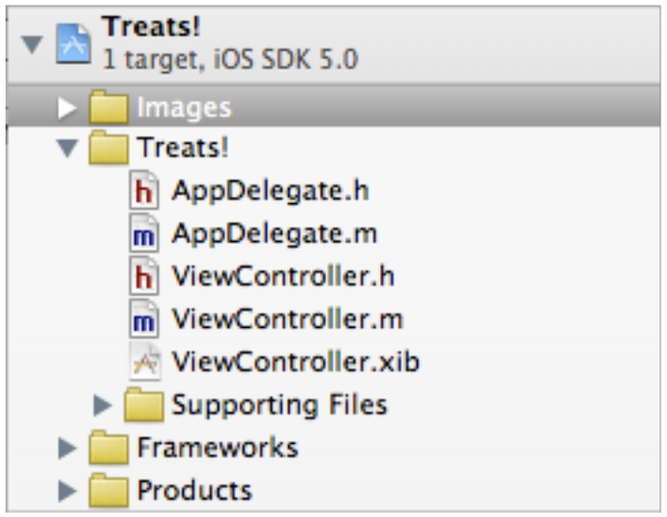

Figure 6 Project Navigator Area with new Images Group

The images needed for this app are available on the companion website for this book. Download the images and then drag them to the newly created Images group inside Xcode. Alternatively navigate to File > Add Files to "Treats!"... Be sure that the checkbox labeled Copy Items into destination group's folder (if needed) is selected as shown in Figure 7, and then click on the ***Finish*** button.

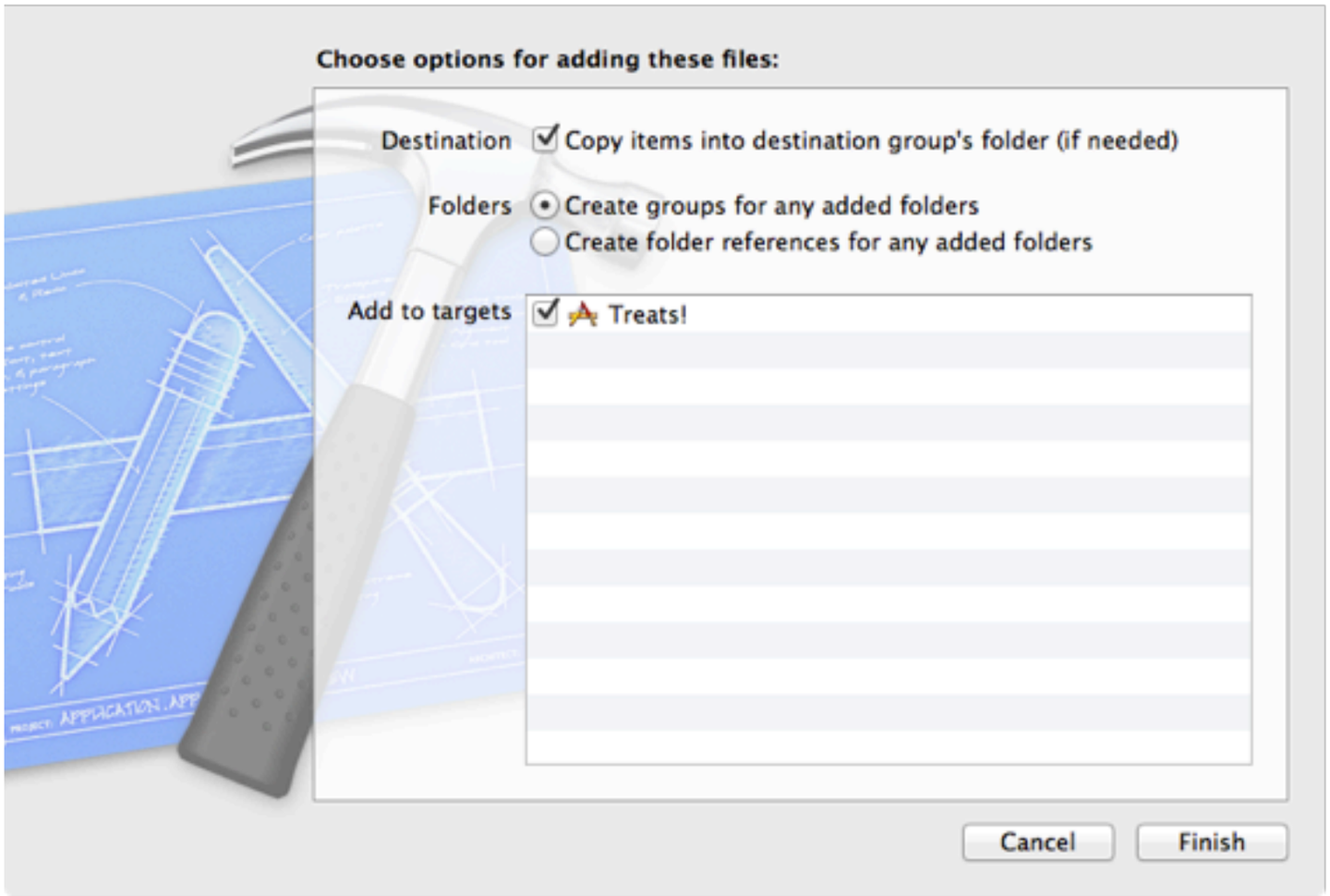

Figure 7 Copying Images into a Project

Setting up the User Interface

The user interface for this game will consist of three screens, each implemented as a separate view. Each view will provide different elements for the game. The first view will serve as the home screen for the game—what players see after launching the app on an iOS device. This view will contain

a background image, a text field for players to enter their name, a button for starting the game, and a text view that will be used to provide game play instructions.

The second view will serve as the interactive game screen—what the player sees while interacting with the game, and this view will consist of image views and labels; some labels will be static so they won't change during game play, and some of the labels will be dynamic which means that what is displayed as the value of the label will change during game play.

The third view will be the high scores screen used to display the high scores. A picker view will be used on this screen to display the top scores. This view will also contain two buttons; one to replay the game and one to clear the list of high scores. A label will also be displayed at the top of the screen to display the score that was most recently achieved.

Creating the First View – the Home Screen

The first view will serve as the home screen for the game. In the Xcode project navigator area, expand the Treats! folder if necessary and then click on the ViewController.xib file to display the View window in the Editor area.

The default View window has a portrait orientation. To change the orientation to landscape select the view by clicking on it, and then navigate to View > Utilities > Show Attributes Inspector. Set the value of the Orientation to Landscape as shown near the top of Figure 8. Then the View window will appear similar to Figure 9.

The first element for our view will be an image so click on View > Utilities > Show Object Library to open the Object Library panel. With the Objects tab selected, scroll down until the Image View element is visible as shown in Figure 10. Click and drag the Image View element into the View window and adjust the Image View element so that it fills the View window as shown in Figure 11.

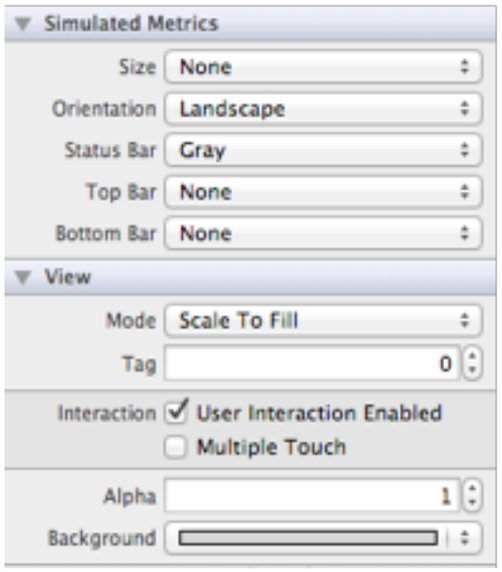

Figure 8 Setting the View Orientation to Landscape

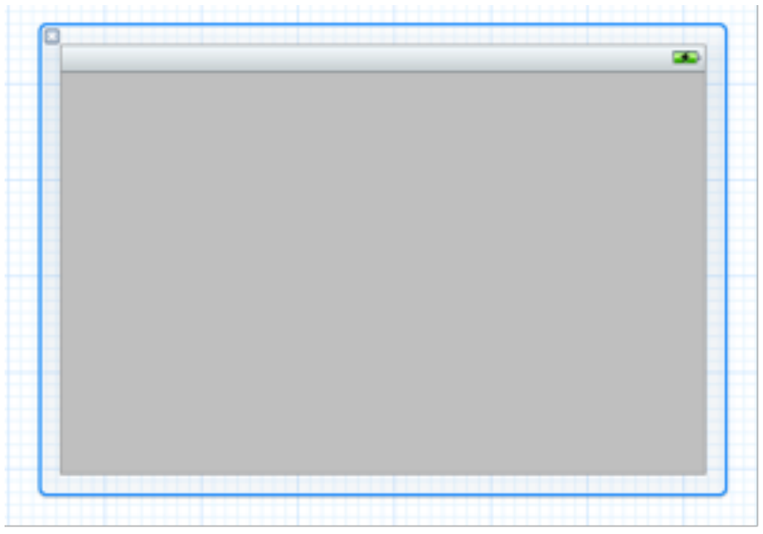

Figure 9 View Window in Landscape Mode

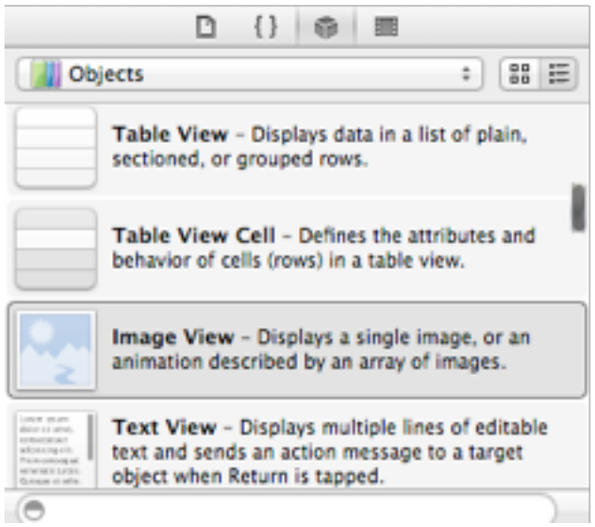

Figure 10 Objects Library Pane

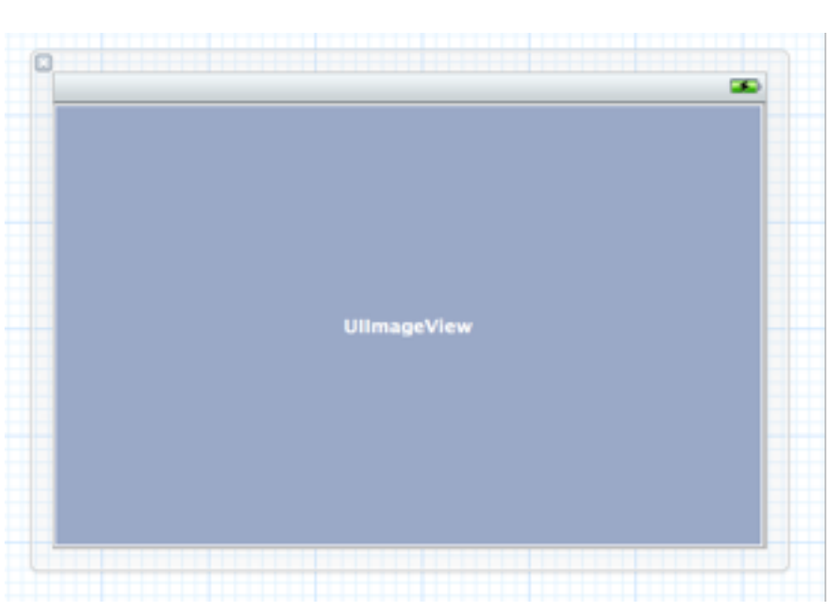

Figure 11 View Window

Click on the View > Utilities > Show Attributes Inspector to open the Image View Attributes tab in the Attributes Inspector Window as shown in Figure 12.

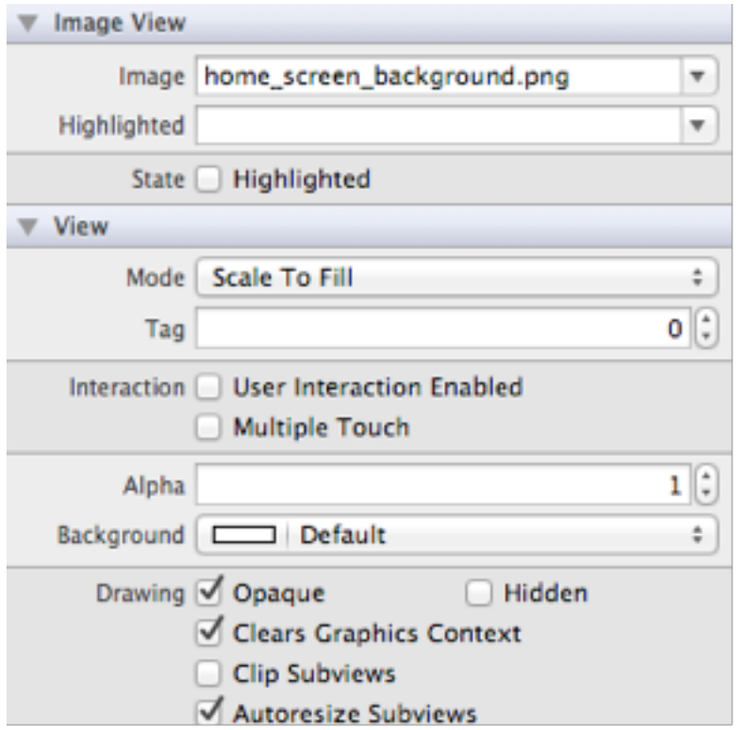

Figure 12 Image View Attributes Pane

Select the home_screen_background.png image in the Image menu. The View window should now contain the background image for the Treats! app so should appear similar to Figure 13.

Figure 13 View Window with the Background Image

Adding a Button, Text Field, and a Text View

In this section we will add the other elements for the Home screen's user interface, which are a Text Field, a Round Rect Button, and a Text View. Drag these elements from the Object Library into the View window and arrange them so that your View window appears similar to Figure 14.

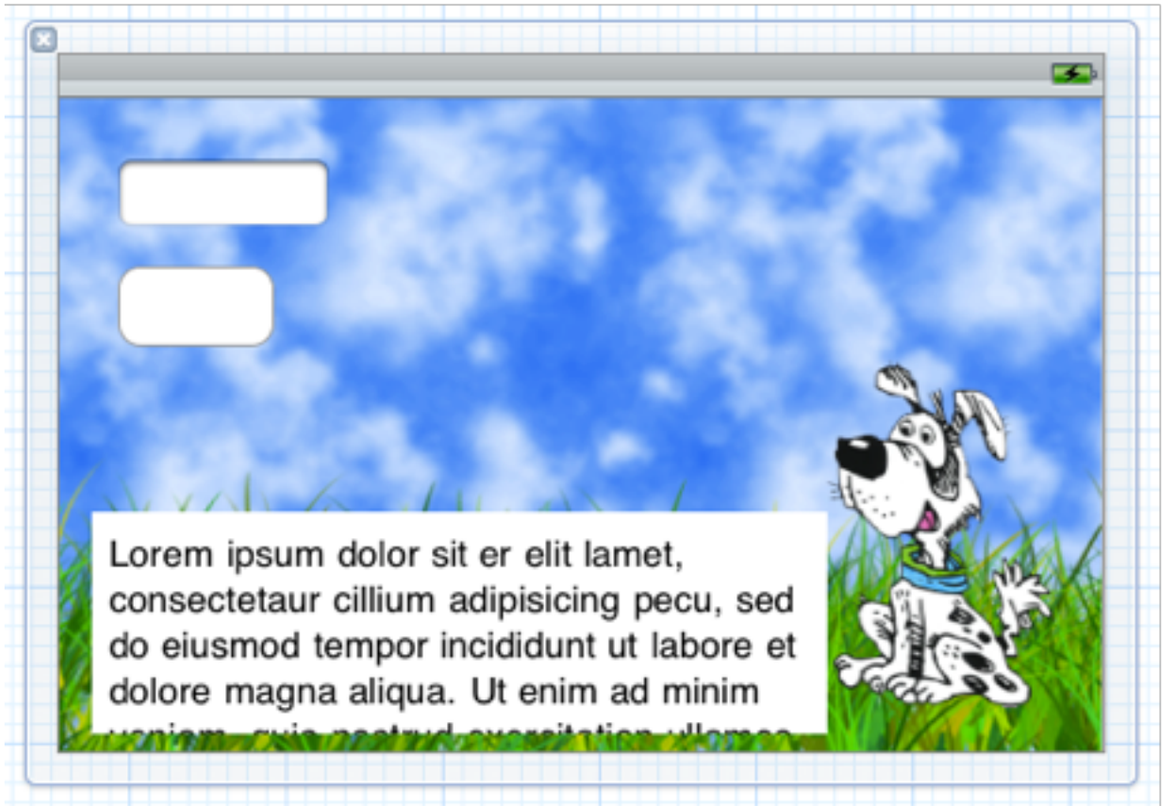

Figure 14 Home Screen View Window with User Interface Elements

In the next steps we will customize the appearance and size of the user interface elements. These steps don't have to be followed exactly. We encourage developers to experiment with the attributes and possibly end up with an app that is slightly different than the one presented in this chapter. Exploring and experimenting inside Xcode are an important part of learning iOS programming.

Customizing the Text Field

First we will customize the text field that will be used to get the user's name. Click on the text field in the View window. If the Text Field attributes are visible, similar to what is shown in Figure 15 then you are ready. Otherwise, navigate to View > Utilities > Show Attributes Inspector to open the attributes inspector in the Utilities area.

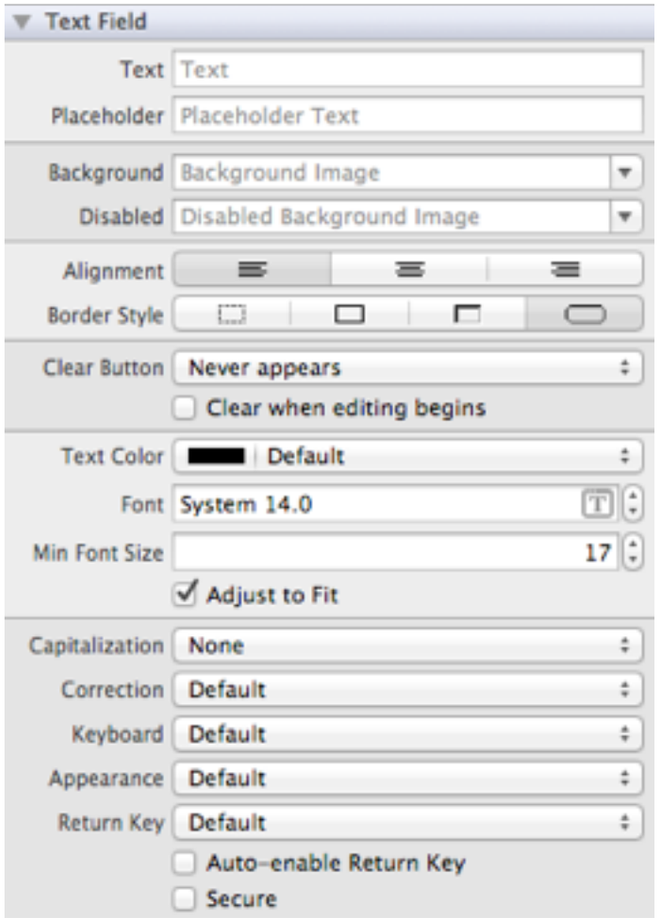

Figure 15 Text Field Attributes Pane

Enter ***Enter Your Name*** for the value of the Placeholder. Change the font by clicking on the Font menu and select Helvetica, Typeface Bold and Size 14 as shown in Figure 16. You may need to resize the text field to display all of the placeholder text.

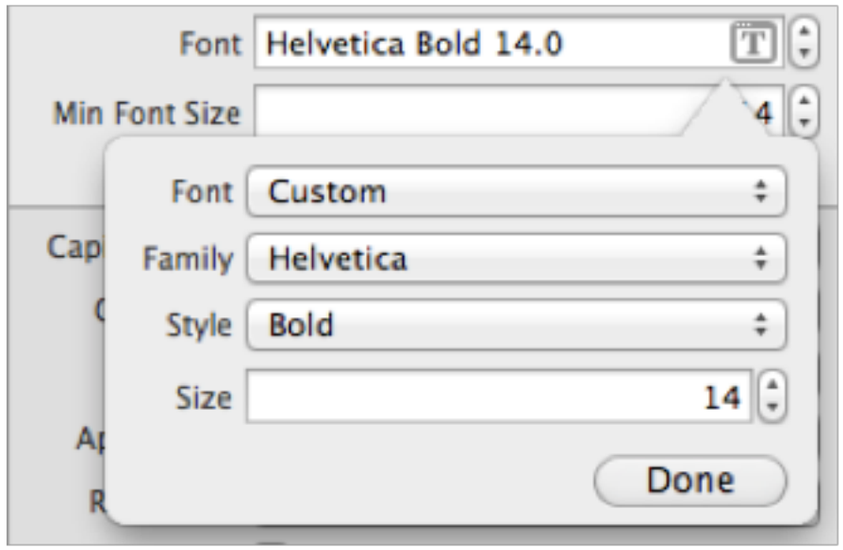

Figure 16 Selecting the Font for the Text Field

Next, scroll down in the Utilities area and then select the Clear when editing begins checkbox. Then change the Text Color of our placeholder text by clicking on the menu for the Text Color attribute and then selecting Dark Gray Color in the Menu. We also set the minimum font size to 14 as shown in Figure 17. Notice that the Adjust to Fit checkbox is already checked. This will ensure that the complete text is always displayed even if the text must be reduced in order to do so when the selected font size is greater than the minimum font size.

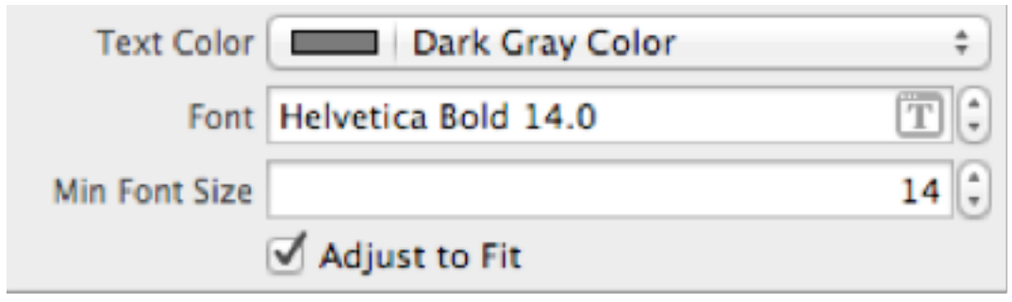

Figure 17 Changing the Text Color for the Placeholder Text

The next step is to change the alpha property of the text field. The alpha property sets the level of transparency for an element. Change the Alpha in the View section for the text field to 0.80 as shown in Figure 18. This will let the background image bleed though the text field just slightly to increase the visual appeal of the user interface.

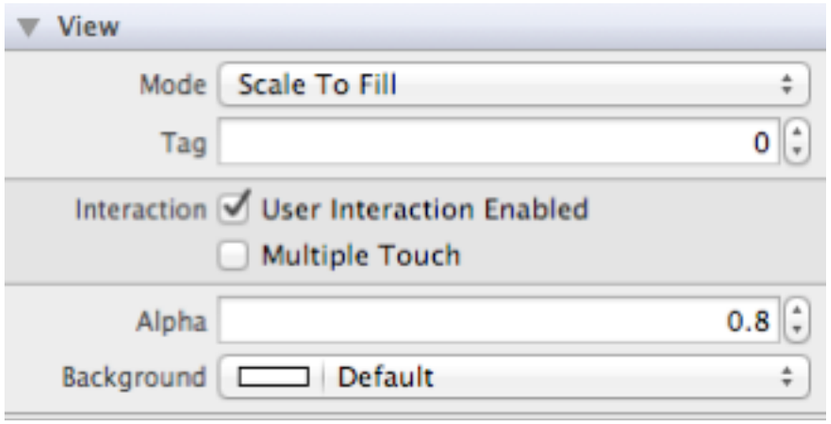

Figure 18 Changing the Transparency of the View

Changing the Button for Starting the Game on the Home Screen

The next steps are needed to set the attributes for the button on the user interface that will be used to start the game.

Select the button element in the View window. This will automatically redisplay the relevant attributes in the Utilities area. Change the button label by entering Start in the attributes text field for the Title.

Change the font to Helvetica 15 Bold or to whatever font is appealing to you.

Change the Text Color to Dark Gray Color as we did previously with the text field.

Scroll down to the View section and change the alpha property to 0.80.

Changing the Text View for Game Instructions on the Home Screen

The next steps are needed to set the attributes for the text view that will contain the game play instructions.

Click on the Text View in the View window to select it.

Enter the following text for the Text attribute: Tap on as many randomly appearing treats as possible to fill up the puppy and beat other players’ high scores.

Ensure that the Editable checkbox is NOT selected.

Change the Text Alignment attribute to center.

Change the Text Color to white.

Set the font to Helvetica 17.

Change the background color to Clear Color so that the box doesn't obscure the background image.

Resize the text view by stretching the handles so that the instructions are easy read on the bottom of the view window.

The Text view attributes should now appear similar to Figure 19 and the View window should appear similar to Figure 20.

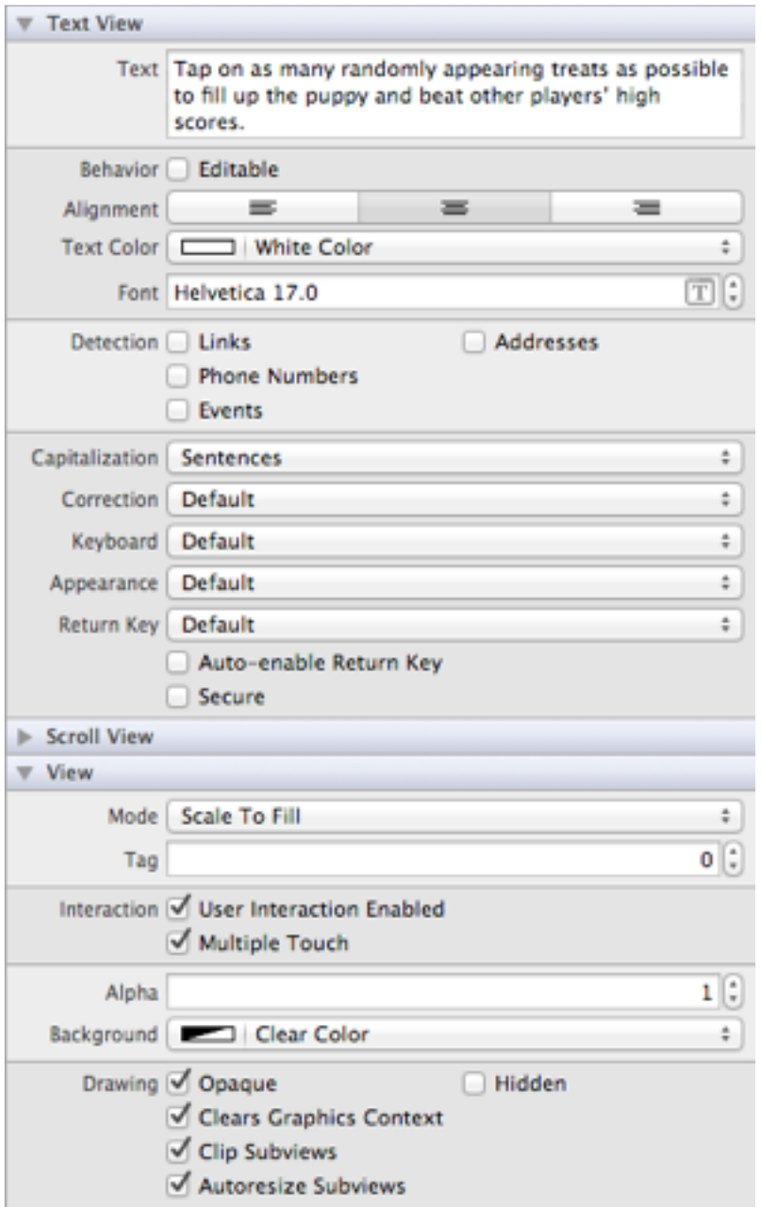

Figure19 Text View Attributes

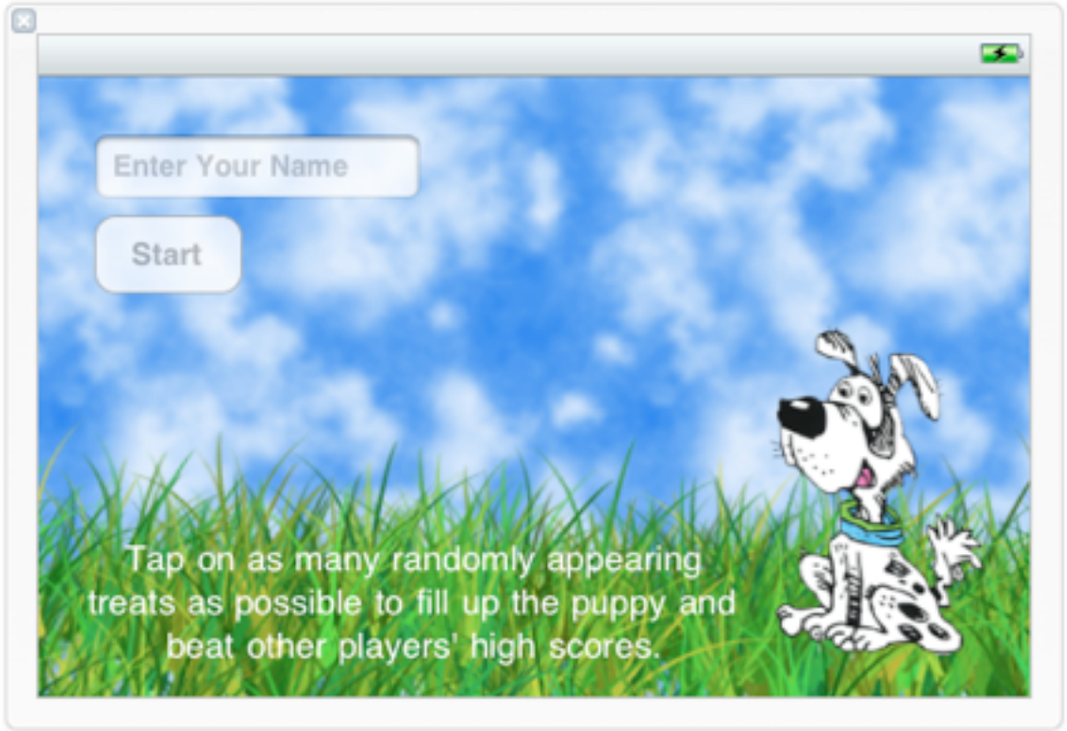

Figure 20 View Window with Background Image, Custom Button, Text Field, and Text View Elements

Voila! The first view contains all the elements for the game's home screen. In order for the elements to function as intended in the completed app, outlet connections need to be made between the user interface elements and the code.

Creating Outlets and Actions for the Elements on the Home Screen

The following steps will guide the developer through the process of creating outlets and actions for the user interface elements on the Home screen in order to provide functionality for those elements as needed. We do not need any additional functionality for the background image or the game play instructions in the Text View. However, we do need an IBOutlet connection in our code for the Text Field so that we can get the name a user enters, and we will need an IBAction in our code to create a method that starts the game play when a user taps on the Start button. We have

created IBOutlet connections and IBAction connections in prior projects in this book so will provide a concise set of instructions in this project. If more detailed instructions are desired, please refer back to these tasks in the previous projects.

Use the Assistant Editor so that you can display both the View window and the ViewController.h file in the Editor area. Ctrl-click on the Text Field and then select New Referencing Outlet in the popup window and drag the New Referencing Outlet to the ViewController.h file, just above the end directive. When you release the mouse and a new popup appears, enter yourNameTextField for the name of the outlet connection.

Next, ctrl–click on the button in the View window and then select the Touch Up Inside event in the popup window and drag that event to the ViewController.h file just above the end directive. When you release the mouse and a new popup appears, enter startGame for the name of the action connection. Now the ViewController.h file should appear similar to the following code.

```
//
//  ViewController.h
//  Treats!
//

#import <UIKit/UIKit.h>

@interface ViewController : UIViewController
@property (weak, nonatomic) IBOutlet UITextField *yourNameTextField;
- (IBAction)startGame:(id)sender;
@end
```

Let's now look at the ViewController.m file. A synthesize directive with the same name of the text field should have been added to the ViewController.m file near the top of the file, and a new method with the same name as the IBAction should have been also been added to the ViewController.m file. Examine the ViewController.m file to ensure that this new code was added. We want to develop the user interfaces for all of our screens before we add the custom code to implement the functionality. So, we will add the code for the startGame method and the code that gets the value of the text field in the next chapter.

Creating the Second View – the Interactive Game Screen

The second view that we create will provide the screen on which a user plays the game. This view will consist of two Image View elements and four Label elements. We will have one image for the background for the game play area, and one image for the dog bone. Two of the labels will be static so they will not change during the game, and two of the labels will be dynamic which means that their values will updated during game play.

To create a new view for the interactive game screen, navigate to File > New > New File…

In the New File window, select the Cocoa Touch in the iOS section in the leftmost panel of the window. Then, select the icon for a UIViewController subclass file as shown in Figure 25, then click the Next button.

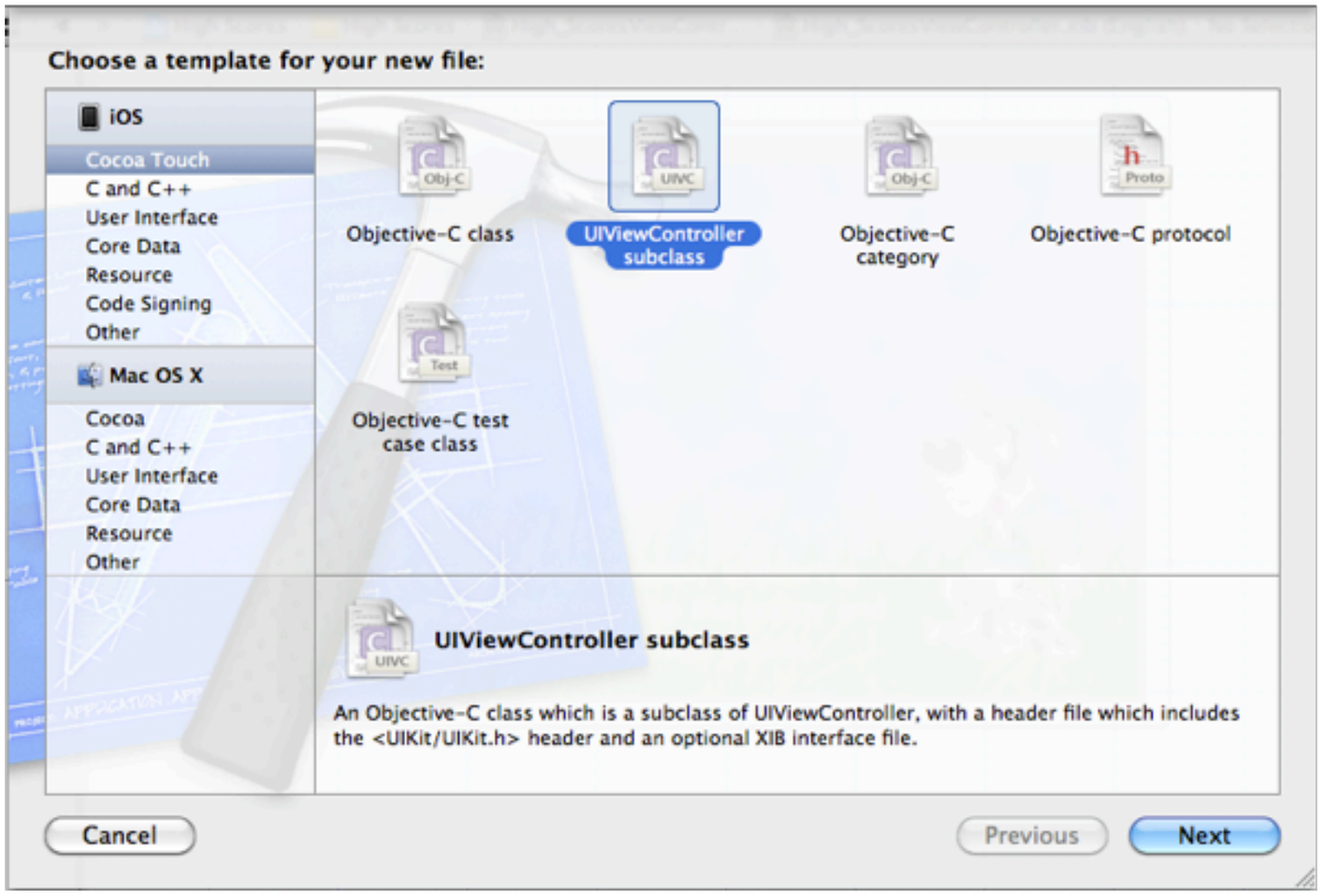

Figure 25 Creating a New View Controller

In the new window that appears, enter GameViewController for the name of the new class, ensure that the Targeted for iPad checkbox is not selected and the With Xib for user interface is selected as shown in Figure 26, then click Next.

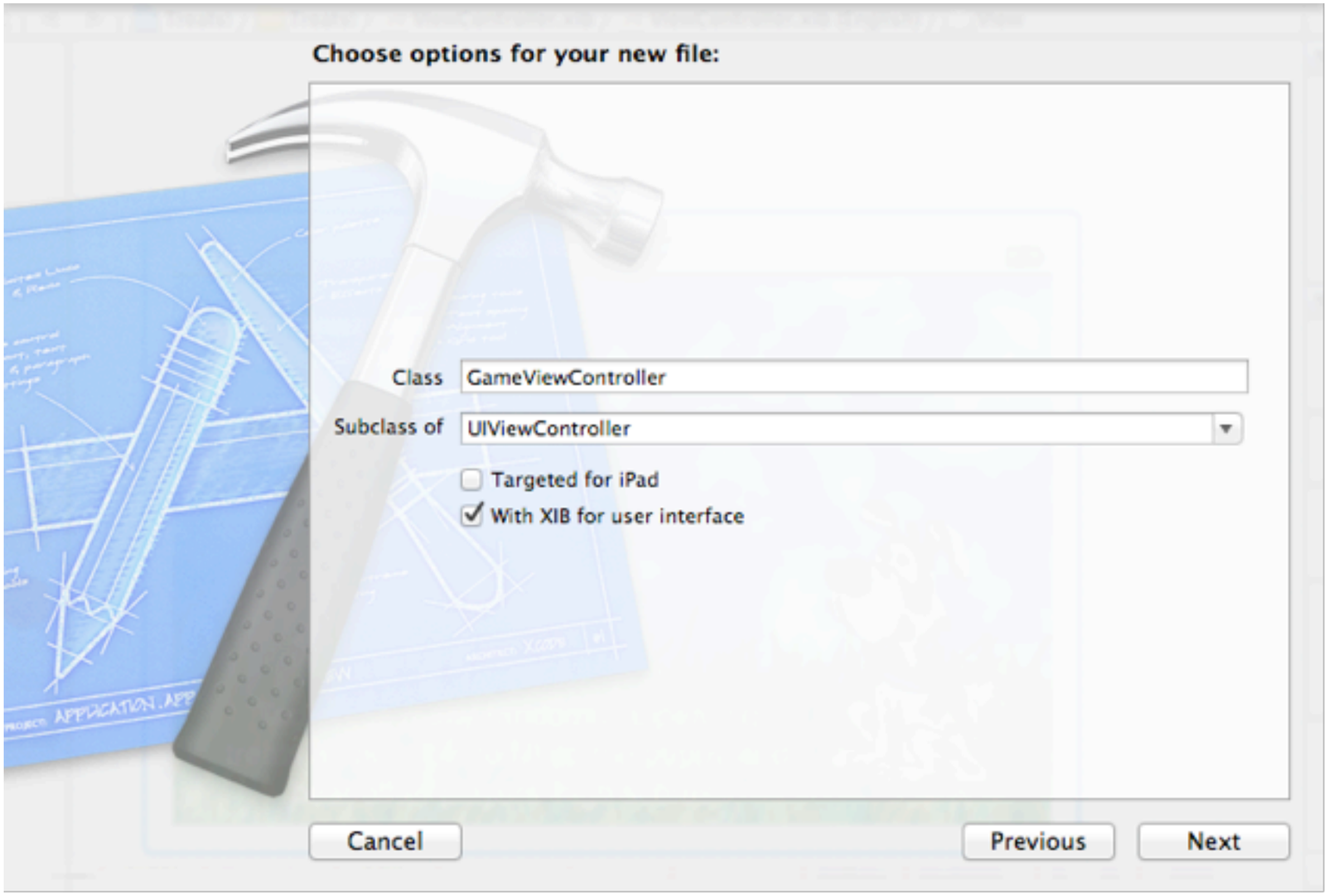

Figure 26 New File Window for a New UIViewController Subclass

In the next window, the project folder, Group locations, and Targets where the new view controller should be created may be selected as shown in Figure 27. Once reviewed or selected, click on the Create button.

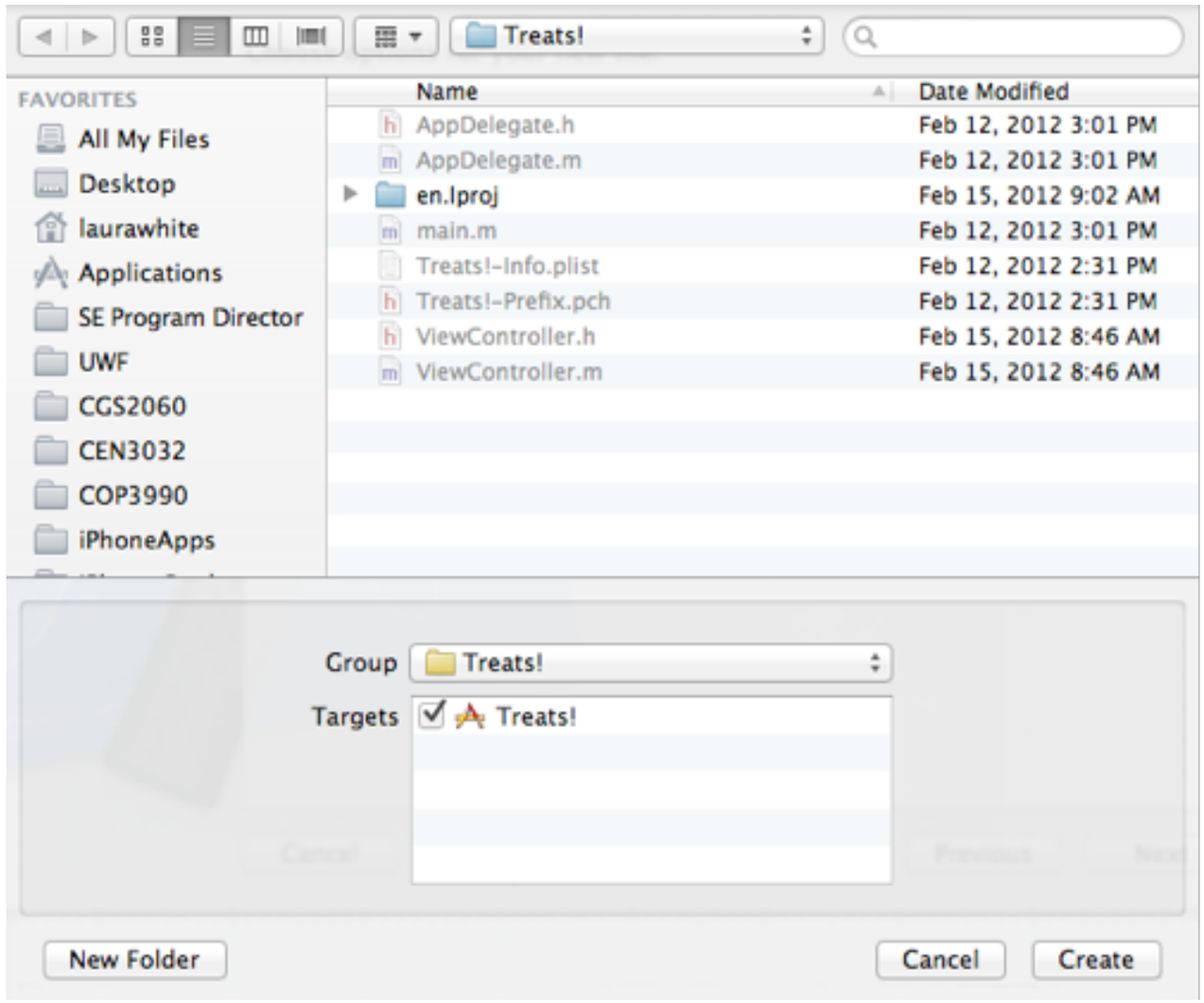

Figure 27 New File Window for a New UIViewController Subclass

Three new files, GameViewController.h, GameViewController.m, and GameViewController.xib are now included in the Navigator panel. Click on the GameViewController.xib file to open it in the Editor area. A new empty View window will appear. Click on the view in the Editor area and then navigate to View > Utilities > Show Attributes Inspector. Change the Orientation Property to Landscape as we did previously and is shown in Figure 8. Then drag an Image View element from the Object Library into the View window. Make sure that the Image View element fills the entire window. Select the Image View element in the View window. Use the Image menu in the Utilities area to select the gameBackgroundImage.png file. The View window should now appear similar to Figure 28.

Figure 28 Second View with an Image Background for the Interactive Game Screen

Adding labels for the Game Information Banner on the Interactive Game Screen

To add the labels for time and score, click on and drag four new Label elements from the Object Library in the rightmost panel to the View window. Arrange the labels across the top of the View window. Double-click on the first label from the left and change the text to Time:, and then click on the third label from the left and change the text to Score:. Notice that there are colons at the end of the Time and Score labels. Select all of the labels and change their fonts to Helvetica Bold 17 and the text color to Dark Gray Color. The View window should then appear similar to Figure 29.

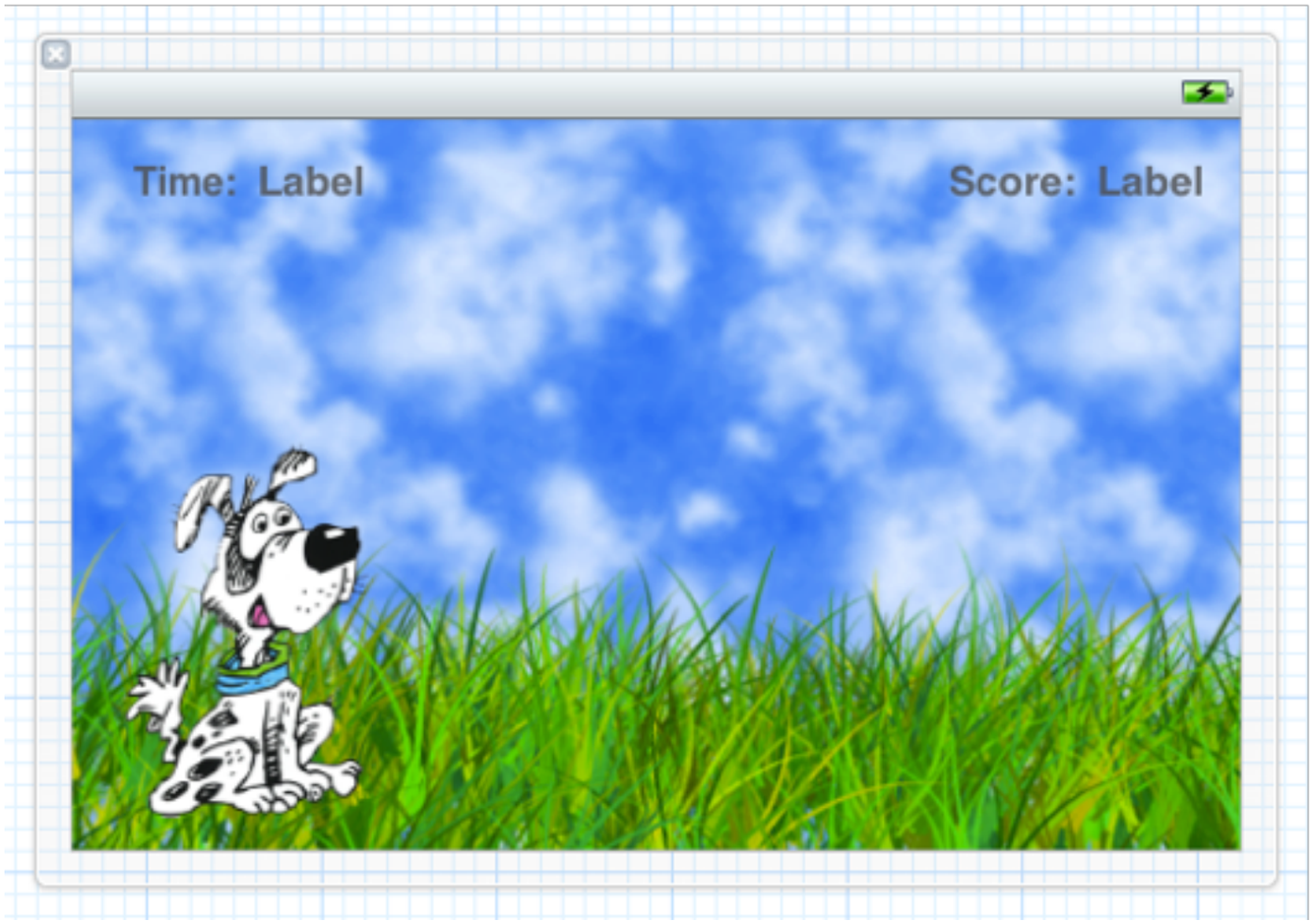

Figure 29 Second View with Labels for the Interactive Game Screen

Add a View for the Dog Bone on the Interactive Game Screen

To add the dog bone for the game, drag an Image View element from the Object Library to the View window. Use the Image menu to select dogbone.png as the contents of this image view.

Be sure to check the User Interaction Enabled checkbox in the Interaction properties section, and to uncheck the Opaque checkbox in the Drawing properties section of the Image View Attributes panel.

Select the dog bone in the View window and resize it so that it is a reasonable size. Your View window should now appear similar to Figure 30. Save the GameViewController.xib file.

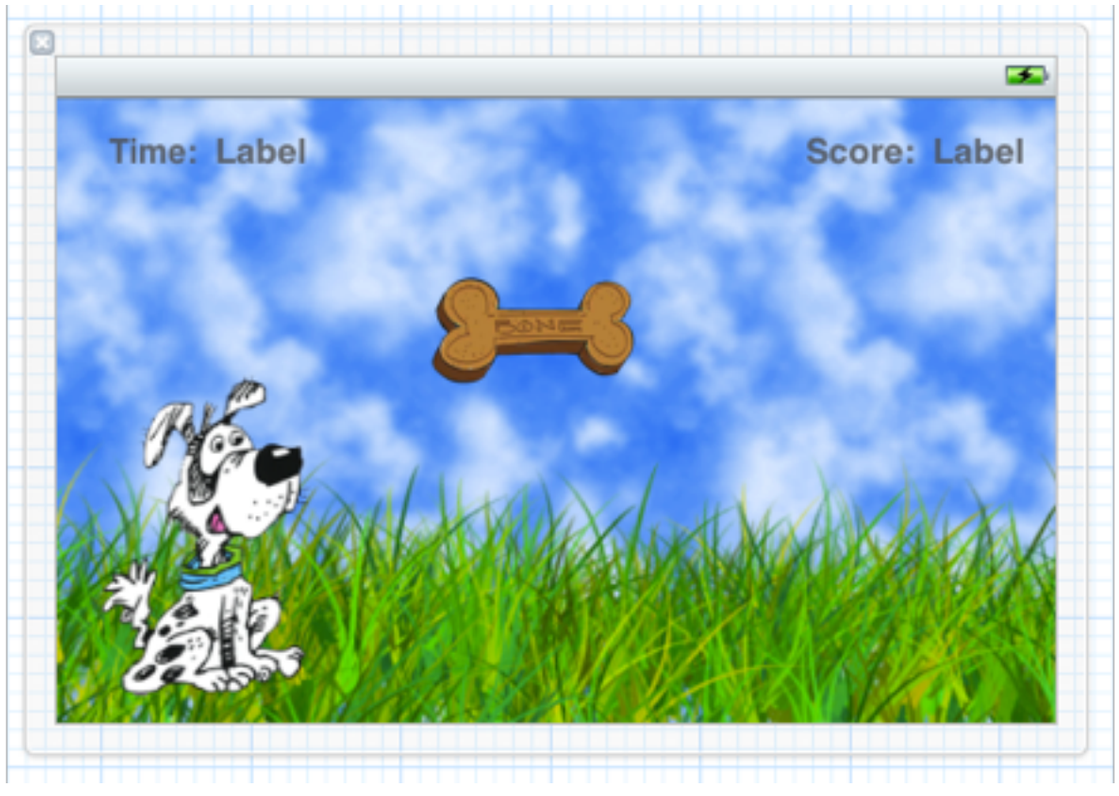

Figure 30 Completed Second View for the Interactive Game Screen

The user interface elements for the interactive game screen have now been created. Next, we need to create outlet connections between the user interface elements and the code that will implement the functionality for these elements.

Create Outlets for the Second View

In this step we will create IBOutlets for the GameViewController.xib file that contains the user interface elements for the second view for the interactive game screen.

Click on the GameViewController.xib file and then navigate to View > Assistant Editor > Show Assistant Editor. Make sure that the file contained in the Assistant Editor is the GameViewController.h file. Close the Utilities area to make more room for the files in the Editor area by toggling the third View button in the upper right of the Xcode window.

Next ctrl-click on the second label stilled named Label after the Time: label in the top left corner to access the outlets menu for that label similar to that shown previously in Figure 22. Drag from the circle to the right of New Referencing Outlet to just before the end directive in the GameViewController.h file. Release the mouse and enter timerLabel as a name for this IBOutlet and click on the Connect button.

Create another outlet connection for the label to the right of the Score: label in a top right corner of the view window but enter scoreLabel for the name of this IBOutlet.

Create another outlet connection for the dog bone image but name the IBOutlet dogBone.

The contents of the GameViewController.h file should now be as follows.

```
//
//  GameViewController.h
//  Treats!
//

#import <UIKit/UIKit.h>

@interface GameViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *timerLabel;
@property (weak, nonatomic) IBOutlet UILabel *scoreLabel;
@property (weak, nonatomic) IBOutlet UIImageView *dogBone;
@end
```

Creating the Third View – High Scores Display

The third view will be used to display the high scores. This screen will contain a label for the most recent score achieved, a spinning wheel to display the high scores, and two buttons; one to play the game again, and one to clear the high scores. Follow the steps below to create this view.

With the GameViewController.xib file selected and displayed in the Editor area, drag a View element from the Object Library into the View window.

Click on the new View and then in the Attributes Inspector change the orientation to Landscape.

For this view's background, select the Scroll View Textured Background Color from the Background menu.

Navigate to View > Utilities > Show Size Inspector and set the size of the new View to 480 by 300 points by entering 480 for the value of the W: property, and 300 for the value of the H: property.

Drag two Round Rect Buttons from the Object Library into the upper left and upper right of the new View window. We will use the first button to restart the game, and the second button will be used to clear the high scores.

Double click on one of the buttons in the new View window and then change its text to Play Again.

Double-click on the other button in the new View window and then change its text to Clear Scores.

Drag a label from the Object Library into the top center of the View window. We will use this label to dynamically show the most recent score achieved in the game.

Drag a Picker View element from the Object Library into the lower part of the new View window. We will use this element to show the high scores on the screen. We resized this slightly to allow borders between the picker view and the view.

Set the value of the label font, size, and text color for the label and the buttons as you wish. We used Helvetica 15, bold for the fonts and sizes of the text on the buttons, and Dark Gray Color for the text color on the buttons. We kept the default value of System 17 and White for the Label.

Now that all of the user interface elements have been added for the third view, click on the elements in the View window and move them around so that the View window appears similar to Figure 31. Notice that the default values are still shown for the Picker View, we will fix this later on. Save the GameViewController.xib file. The next step is to create the outlet connections for this view.

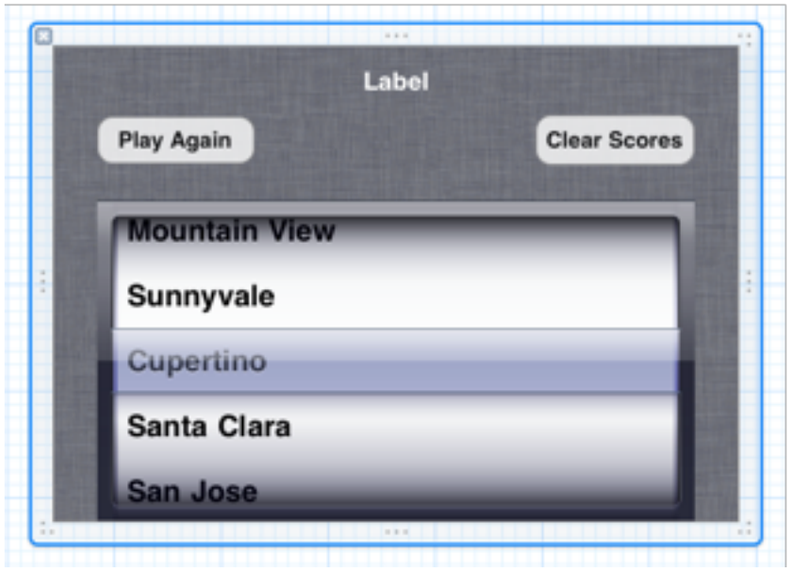

Figure 31 View Window with a Label, Buttons, and a Picker View

With the GameViewController.h file in the Editor Area, display the GameViewController.h file in the Assistant Editor. Ctrl-click on the new View (be careful that you are selecting the view and not any elements within the view and then drag from the circle to the right of a New Referencing Outlet to just above the end directive in the GameViewController.h file. Enter highScoresView as the name for the IBOutlet connection.

Ctrl-click on the picker view and drag a new referencing outlet to just above the end directive in the GameViewController.h file. Enter picker for the name of this new IBOutlet connection.

Ctrl-click on the label and then click on the circle to the far right of New Referencing Outlet and drag just above the end directive in the GameViewController.h file. Enter finalScoreLabel for the name of this IBOutlet connection.

Ctrl-click on the Play Again button and then click on the Touch Up Inside event and drag to just above the end directive in the GameViewController.h file. Enter playAgain for the name of this IBAction.

Ctrl-click on the Clear Scores button and then click on the Touch Up Inside event and drag just below the interface declaration. Enter clearScores for the name of this IBAction.

The code in the GameViewController.h file should now be as follows.

```
//
//  GameViewController.h
//  Treats!
//

#import <UIKit/UIKit.h>

@interface GameViewController : UIViewController
@property (weak, nonatomic) IBOutlet UILabel *timerLabel;
@property (weak, nonatomic) IBOutlet UILabel *scoreLabel;
@property (weak, nonatomic) IBOutlet UIImageView *dogBone;
@property (strong, nonatomic) IBOutlet UIView *highScoresView;
@property (weak, nonatomic) IBOutlet UIPickerView *picker;
@property (weak, nonatomic) IBOutlet UILabel *finalScoreLabel;
- (IBAction)playAgain:(id)sender;
- (IBAction)clearScores:(id)sender;
@end
```

As a result of the connections that we made, Xcode will have automatically added code to the GameViewController.m file. Select and review the contents of the GameViewController.m file. Xcode will have added synthesize directives at the top of the file, and added statements to the ViewDidUnload method to clear the memory for these objects. Xcode also created empty methods at the end of the file for the playAgain and clearScores actions -- we will add the code to these messages in the next chapter.

Now let's test our app. Make sure that the iPhone 5.0 Simulator is selected as the scheme in the upper left of the Xcode window and then click on the Run button. If no mistakes were made during the development of this project, the app will be open in the iPhone Simulator. Remember that the app will not behave as desired until we add the code necessary to implement the functionality—which we will do this in the next chapter. Also notice that the first screen does not display properly yet either. At this point, it is valuable to Run the app just to ensure that there are no errors detected by Xcode.

Congratulations! You have now created three views for the Treats! user interface and created IBOutlets and IBActions so that the code can access or respond the user interface elements as appropriate. Save your project.

Summary

The objective of the chapter was to create an User Interface for the Treats! app. Before starting the development of the game the storyline was represented with a flowchart and an overall approach for the development was defined. The decision was made to create three views for each screen in the app. For each view, the user interface elements were dragged from the Object Library to the View window, the attributes were set, and then the appropriate outlet connections or action connections were created. The outlets and actions establish the bridge between the user interface elements and the custom code that will provide the functionality for the user interface elements. Upon completion of this chapter the reader will have an application template that is ready to fill-in with custom code in the next chapter.

Review Questions

1. What is the name of a variable that enables connection between your custom code and a user interface element so you can manipulate it with properties of the UI element from the code?

2. What are IBActions?

3. Describe the process of creating IBOutlets and IBActions.

4. What are the purpose of the @property and @synthesize directives? What benefits are derived from their use?

5. Describe the steps of adding a View Controller to your project.

6. What steps are required to display a view in landscape mode?

7. Which event is typically sent to a method that will be invoked whenever a user taps on a button?

Chapter 7

Treats! App - Part 2: Saving and Restoring Data

Treats! App - Part 2: Saving and Restoring Data

Concepts emphasized in this chapter:

- NSUserDefaults
- NSTimer
- Touch Events
- UIPickerView
- Setting the User Interface orientation programmatically
- Transitioning between multiple views in an app

Introduction

In the previous chapter we set up the user interface elements necessary for our Treats! app. We also created outlets and actions without writing a single line of code. Adding the code is what will make the game come to life-- what will allow users to interact with the app. So, now we will write the code to fill in the methods that control the user interface elements. The functionality for the elements in the app's user interface will be created in Xcode by adding Objective-C programming language code. We will step through the process of adding this code in the same order that we created each view for each of our screens; first for the home screen, then the interactive game screen, and then the high scores screen.

Writing the Code for the First View – the Home Screen

The first code we add will be to make the GameViewController class visible and accessible to the ViewController. This is accomplished by clicking on the ViewController.m file and adding the following import directive at the top of the file, just after the existing import directive.

```
#import "GameViewController.h"
```

Now the contents of the view defined in the GameViewController class can be displayed whenever a user taps the Start Game button which is contained within the ViewController class.

Set the Orientation for the Home Screen

Since iPhone apps are oriented in Portrait mode by default and our game is designed for a landscape layout we need to modify the code for the orientation inside the shouldAutorotateToInterfaceOrientation method inside the ViewController.m file so that the method appears as follows.

```
- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
}
```

The value on the right-hand side of the equivalence operation is what makes the orientation of the user interface display in landscape mode with the Home button on the left. Developers can also specify portrait, portrait upside down, and landscape right in this method.

Add the Code to the startGame Method

The next step is to add the implementation of the startGame method so that the game will start when a user taps the Start Game button on the first screen. Find the empty startGame method in the ViewController.m file, and add the following code inside the body of the method so that the method appears as follows.

```
- (IBAction)startGame:(id)sender {
    if(yourNameTextField.text.length>0) {
        GameViewController *game= [[GameViewController alloc]initWithNibName:@"GameViewController" bundle:nil];
        //game.yourName=yourNameTextField.text;
        [self presentModalViewController:game animated:YES];
    }
    else {
        UIAlertView *alert=[[UIAlertView alloc]initWithTitle:@"Message" message:@"You need to enter your name first!" delegate:nil
                            cancelButtonTitle:@"OK" otherButtonTitles:nil];
        [alert show];
    }
    [yourNameTextField resignFirstResponder];
}
```

First this method checks if a user entered a name by checking the number of characters in the text property of the name text field. If the number of characters is greater than zero then it is presumed that a name was entered and an instance of the GameViewController class is created using the initWithNibName method of the UIViewController class.

The commented line in this section of code will be used to store the value of the name that users enter into the text box for their name. The next line of code displays the second view, the interactive game screen using the [self presentModalViewController:game animated:YES] message. If the user didn't provide a name when this method is called then an alert message is created and displayed on the screen. This is accomplished by creating an instance of the UIAlertView class and using the show method to display that alert message. After this the text field releases its privilege as first responder so that the keyboard will lower off the screen.

Select the iPhone 5.0 Simulator in the Scheme as shown in Figure 1 and then click on the Run button to build and run the app. The iPhone simulator should appear as shown in Figure 2.

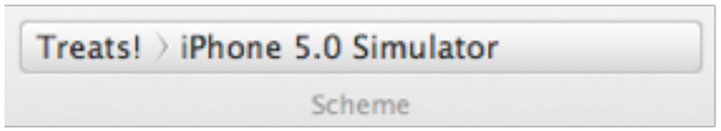

Figure 1 Active scheme as shown in Xcode

First tap on the Start button without entering a name to verify that the error checking for the text field in the startGame method functions properly as shown in Figure 3.

Next tap on the Enter Your Name text field and enter your name. Since this is a text field a keyboard will appear as shown in Figure 4. Enter your name.

Figure 2 The Treats! App Running in the iPhone Simulator

Figure 3 Alert Displayed on the Home Screen

Figure 4 Home Screen with Keyboard for the Enter Your Name Text Field

Writing the Code for the Second View – the Interactive Game Screen

The code needed for the interactive game screen will be added to set the orientation for the interactive game screen, create variables, manage memory, load the view for the game onto the game screen, create a timer for the game, handle touch events, and save persistent score data.

Find the shouldAutorotateToInterfaceOrientation method in the GameViewController.m file. The game screen is not displaying correctly because the orientation is in portrait mode. Change the user interface orientation of the game screen to landscape mode by changing the value of the interface orientation by replacing the return statement in the shouldAutorotateToInterfaceOrientation to the following.

```
return (interfaceOrientation == UIInterfaceOrientationLandscapeLeft);
```

This modification implements the landscape orientation for the interactive game screen by returning a YES when the orientation should be rotated to landscape mode displayed right-side up when the iPhone is held so that the iPhone home button is physically located to the left of the screen.

Add Code for Variables, and Add Property and Synthesize Directives

The first thing we need for the game screen is to create variables that store values related to game play. After we declare these variables we can then define properties and methods that use these variables when the game is played.

Click on the GameViewController.m file inside Xcode to open the file in the editor.

Add the following declarations at the top of the GameViewController.m file, just below the synthesize directive. Notice that the names of the variables (e.g., score, timer) are defined so that the name describes the purpose of the variable, which is an important programming practice.

```
int score;
int timerCount;
NSMutableArray *scoreArray;
NSMutableArray *nameArray;
NSString *yourName;
NSTimer *timer;
NSUserDefaults *defaults;
```

Add the following property directive to the GameViewController.h file for the yourName object.

```
@property (nonatomic, retain) NSString *yourName;
```

Add the following corresponding synthesize directive to the GameViewController.m file.

```
@synthesize yourName;
```

Now go back and uncomment the assignment to the yourName property in the startGame method in the ViewController.m file.

Add Custom Code

Add the following declarations for custom methods that will mange the location of the dog bone, manage the timer, and manage the score, to the GameViewController.h file.

```
- (void) moveBone;
- (void) startTimer;
- (void) updateTimer:(NSTimer*) theTimer;
- (void) displayScoreView;
- (void) saveScore;
```

Next we need to add method body skeletons for these methods at the end of the GameViewController.m file.

```
-(void) moveBone{
}

-(void) startTimer{
}

-(void) updateTimer:(NSTimer*)theTimer{
}

-(void) displayScoreView{
}

-(void) saveScore{
}
```

Now that we have created these methods we can add messages to the viewDidLoad method that start the timer and move the dog bone on the screen. Click on the GameViewController.m file to open it in Xcode and add the following code to the end of the viewDidLoad method.

```
[self startTimer];
[self moveBone];
self.scoreLabel.text=@"0";
```

The viewDidLoad method is called by the operating system when the view is loaded into memory. So, this is a great place to setup the game. Since the game is a timer based game we should be able to start the timer as soon as the player is ready to start the game. The startTimer method will contain the code necessary to precisely start the code and is called by executing the [self startTimer] message. The code for the startTimer is explained later in this chapter. The [self moveBone] message causes the moveBone method to execute once when the game view is loaded into memory. The self.scoreLabel.text statement initializes the text for the score label to 0, and the self.yourNameLabel.text statement loads the player's name entered into the text box into the game.

Now is a good time to test your app. Click on the Run button in Xcode. Enter your name in a text field and then tap on the Start button. The game should now launch interactive game screen as shown in Figure 5. Notice that the value of 0 is now displayed for the score as a result of the statement we just added to the viewDidLoad method.

Figure 5 Interactive Game Screen

Add Code for the Timer on the Interactive Game Screen

The next task we need to accomplish is to set up the timer. Add the following code to the inside of the startTimer method near the end of the GameViewController.m file.

```
timerCount=30;
timer=[NSTimer timerWithTimeInterval:1 target:self selector:@selector(updateTimer:) userInfo:nil repeats:YES];
NSRunLoop * myRunLoop = [NSRunLoop currentRunLoop];
[myRunLoop addTimer:timer forMode:NSDefaultRunLoopMode];
```

First this code sets the time for playing the game. In this context it is set to 30 seconds, you can change that to whatever duration you wish for your game. The next statement dictates that the updateTimer method should be called every second. We will add this method in the next step. The next two statements cause the timer to loop as needed during the game. Now add the following code to the inside of the updateTimer method near the end of the GameViewController.m file.

```
timerCount--;
if(timerCount>=0){
    self.timerLabel.text=[NSString stringWithFormat:@"%d",timerCount];
}
else{
    [timer invalidate];
    self.dogBone.userInteractionEnabled=FALSE;
    [self saveScore];
}
```

As soon as the timer starts it gets decremented. A conditional statement checks if the timer is greater than or equal to zero. If the timer's value is greater than or equal to zero, the label showing the timer value on the screen is updated to the new value, if the timer's value is zero, then the timer is stopped by the invalidate message, and the next statement sets the enabled property for the bone to false so that it will not respond to taps anymore. At this point the game is ended.

Add Code to Move the Bone Around the Interactive Game Screen

In this section we will implement the method that will move the bone to random places on the screen. Let's first look at the iPhone coordinate system, as shown in Figure 6. Notice that the origin of the coordinate system is in the upper left corner of the screen. As you move to the right on the screen the x value increases, and as you move down the screen the y value increases. One of the available iOS methods to draw at a random x, y location on the screen is the arc4random() method. By default this method returns a value between 0 and 1. To generate a number between 0 and 9 inclusive, we use the format arc4random%10, or arc4random%100 to generate a number between 0 and 99 inclusive. Add the following code to the inside of the moveBone method in the GameControllerView.m file.

```
int x=arc4random()%370+5; // moves bone horizontally
int y=arc4random()%175+65; //moves bone vertically
dogBone.frame=CGRectMake(x, y, 100, 60);
```

We want the bone to appear in random places inside the screen boundaries but not too close to the edge of the screen so we use the arc4random method to derive pixel values for the x coordinate from 5 to 375 and pixel values for the y coordinate from 65 to 240 which are assigned to the integer variables x and y. The last statement in the method defines the frame for the dog bone. We use the CGRectMake function with four arguments, the x coordinate of the upper left corner, the y coordinate of the upper left corner, the width, and the height to define the frame. We use the x and y positions generated by the arc4random methods to our CGRectMake and constants of 100 and 60 for the width and height of the rectangle that contains the dog bone.

Figure 6 iPhone Coordinate System

Go Ahead and run the game. Enter your name on the home screen and tap on the Start button. If the game goes to the interactive game screen and the timer counts down from 30 as shown in Figure 7 then everything works like it should at this point. Good job!

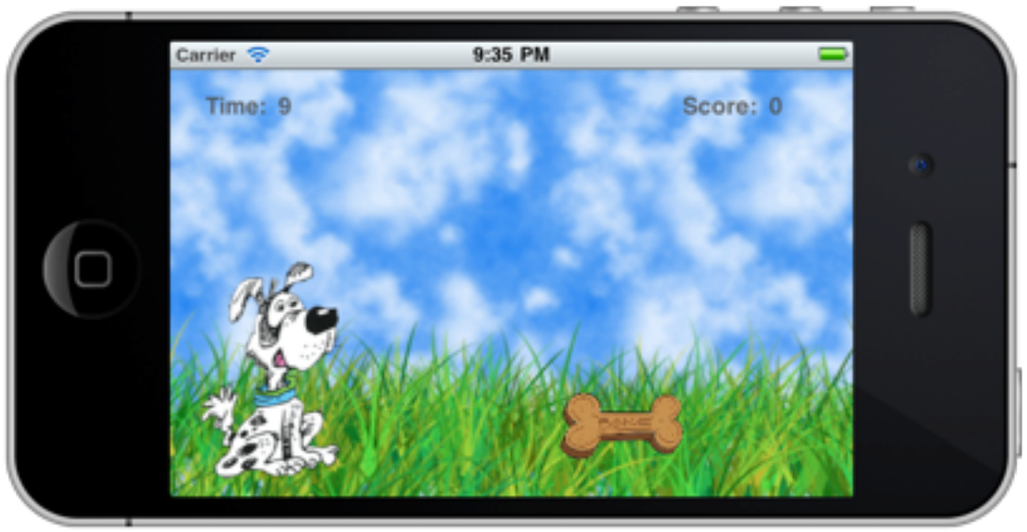

Figure 7 Game View with Running Timer

Add Code to Handle the Touch Events

The next thing we need to do is to add the code that handles the touch events needed to play the game. We will use touch events to credit the player with points whenever he/she touches the bone. Touch events are triggered whenever the user touches the iPhone screen. These events can either be handled or disregarded. In our case we use the handlers in the UITouch class to handle the touch events. The touch handlers require two parameters, NSSet touches which represents the collection of touches, and a UIEvent, which represents the event to which the touch belongs. The touch handlers are described below:

-(void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event; // triggered when the user touch the screen with one or more fingers

-(void)touchesMoved:(NSSet *)touches withEvent:(UIEvent *)event;   // triggered when finger/s move on the screen

-(void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event; // triggered when the finger/s are lifted from the screen.

-(void)touchesCancelled:(NSSet *)touches withEvent:(UIEvent *)event; // triggered when a touch sequence is cancelled by a system event

In our app we want to check if the player taps the bone, so we will use the touchesBegan method. Add the following method to the end of the GameViewController.m file.

```
- (void)touchesBegan:(NSSet *) touches withEvent:(UIEvent *)event {
    UITouch *touch=[touches anyObject];
    if([touch.view isEqual: dogBone]) {
        [self moveBone];
        score++;
        scoreLabel.text=[NSString stringWithFormat:@"%d",score];
    }
}.
```

The touchesBegan handler is invoked when a user taps anywhere on the screen. The handler creates an instance of the UITouch class that is then checked to determine whether the view that belongs to the touch matches the view of the bone. If this condition is true then the moveBone method is called so that the bone will then move to a new random place on the screen, the score is incremented and the score label is updated.

Click on the Xcode Run button. You may notice that we very frequently run our app in the simulator. This is important since the more frequently you test your code, the less code has to be examined to find a mistake if one should occur. It is always best to find your mistakes early.

Add Code to Save Persistent Data Generated on the Interactive Game Screen

The NSUserDefaults class contains methods that can be used to save persistent data such as a user login, password, or some kind of application state. To save data using the NSUserDefaults class we need to create an instance of the class, and then use one of its methods specific to different types of data appropriate for our game data. To retrieve data an instance of the class is created and a method specific to the type of data is used to retrieve the data. We will use this class to save the player's score in our game. Add the following code to the inside of the saveScore method in the GameViewController.m file.

```
defaults=[NSUserDefaults standardUserDefaults];
NSMutableArray *highScoresArray=[[NSMutableArray alloc]initWithCapacity:0];
NSMutableArray *highScoresNamesArray=[[NSMutableArray alloc]initWithCapacity:0];
[highScoresArray addObjectsFromArray: [defaults arrayForKey:@"highScores"]];
[highScoresNamesArray addObjectsFromArray: [defaults arrayForKey:@"highScoresNames"]];
NSString *currentScore=[NSString stringWithFormat: @"%d", score];

int highScoreCount;
highScoreCount=[highScoresArray count];
if(highScoreCount>10) {
    highScoreCount=10;
}
```

```
// Hide the scores
if(highScoreCount>0) {
    BOOL scoreFound=FALSE;
    for(int i=0;i<highScoreCount;i++) {
                if(score>=[[highScoresArray objectAtIndex:i] intValue]&&!scoreFound) {
            if(i<=10){
                [highScoresArray insertObject:currentScore atIndex:i];
                [highScoresNamesArray insertObject:self.yourName atIndex: i];
                scoreFound=TRUE;
            }
                }
    }// End of for loop
    if(!scoreFound&&highScoreCount<10) {
                [highScoresArray addObject:currentScore];
                [highScoresNamesArray addObject:self.yourName];
    }
}
else{
    [highScoresArray addObject: currentScore];
    [highScoresNamesArray addObject: self.yourName];
}
    [defaults setObject:highScoresArray forKey:@"highScores"];
    [defaults setObject:highScoresNamesArray forKey:@"highScoresNames"];
    [defaults synchronize];
    // [self displayScoreView];
```

First the NSDefaultsObject instance is initialized and two mutable arrays are created. The objects from the NSDefaults are copied to the newly created arrays. The NSString object is created to hold the current score. The high scores array is checked for the number of elements; if the number is greater than 0, we loop through all the elements and check if the current score is higher than any of the existing scores in the array. If the current score is higher, it is added to the array. The last thing that happens is that the arrays are released and the displayScoreView is called to display the list of scores. Notice that the call to displayScoreView is commented out for now. We will uncomment this later when we have finished the code that is needed to display the score view.

We have now provided the code for the second view—the interactive game screen of our app. Be sure to save your files and run the app in the simulator to ensure that it behaves as expected at this point. We will now implement the functionality for the High Scores screen.

Writing the Code for the Third View – the High Scores Screen

Previously we added a picker view user interface element—a spinning wheel that displays a set of values as shown in Figure 8, to our third view. We are going to use this user interface element to display high scores in our app. We also need to define the action for the two buttons on this screen for play again and to clear the high scores.

Understanding Picker Views

There two kinds of pickers in iOS programming. The two kinds are Date Pickers that are typically used for displaying date and time related information, and Picker Views that are typically used for displaying custom content. The next section contains a brief description of both of these user interface elements.

Date Pickers

The date picker is an instance of the UIDatePicker class. The user interface for a date picker presents multiple rotating wheels with which users can select dates and times. Examples that use instances of the UIDatePicker class are the Timer and Alarm panes within the built-in iOS Clock application. A date picker may take several forms depending on the selected datePickerMode property. The default datePickerMode property value displays pickers for both date and time. There are four possible values for this Mode. The user interfaces that are displayed as a result of these modes are shown in Figures 8 through 11. Figure 8 shows the date picker for displaying both date and time, Figure 9 shows the date picker for displaying time, Figure 10 shows the date picker for displaying date, and Figure 11 shows the date picker for displaying a timer.

A date picker is not used in the application developed in this chapter, but the process for using one is straightforward. In general, to use a date picker in an application, a Date Picker object must be dragged from the Xcode objects library into a View, and then its properties can be customized. Additionally, outlets should be created to connect the picker with the setDate:animated method. A picker cannot provide a timer on its own even though it displays time. An instance of the NSTimer class is needed to associate a timer with an app.

Figure 8 DatePicker in UIDatePickerModeDateAndTime Mode

Figure 9 DatePicker in UIDatePickerModeTime Mode

Figure 10 DatePicker in UIDatePickerModeDate Mode

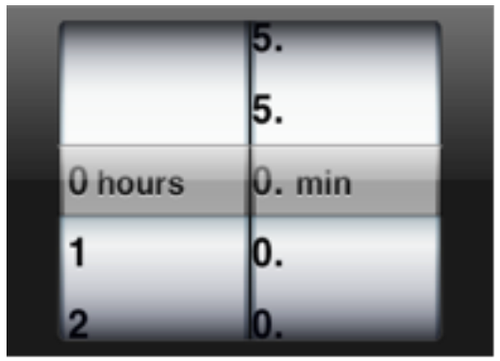

Figure 11 DatePicker in UIDatePickerModeCountDownTimer Mode

Picker Views

Pickers are instances of the UIPickerView class. A picker has more control features than a date picker. The number of columns, number of rows in the columns, size of the columns, and appearance of the items displayed in the picker can all be customized in a picker. An example of a picker with one column is shown in Figure 12.

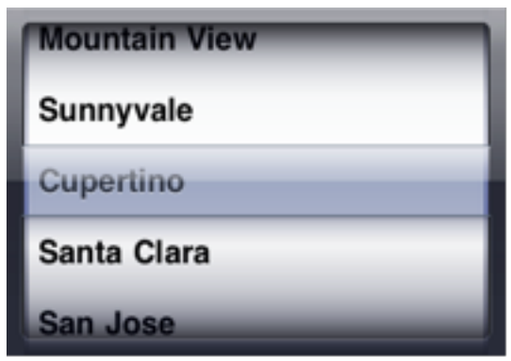

Figure 12 Example of a Picker View with One Column

The implementation of the picker is a bit more difficult than for the date picker. In general, first a Picker view is dragged from the Object Library to a View window, then a data source and a delegate for the picker must be specified, and then the methods must be defined that implement the desired functionality for the picker.

The relationship between the picker, the data source, and the delegate is represented in Figure 13.

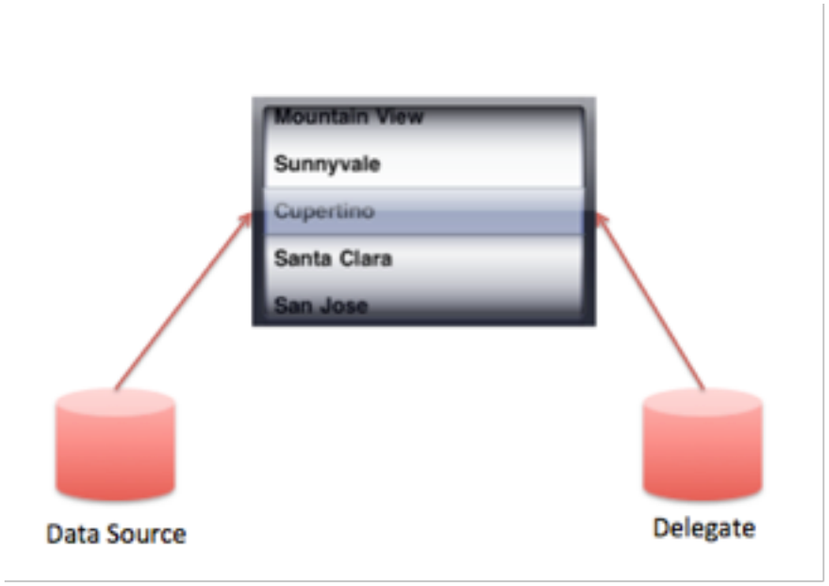

Figure 13 Picker View: Delegate and Data Source

The picker data source provides the data about the number of rows and components to the picker view, this is implemented in the UIPickerViewDataSource protocol and specifically by adding methods, such as the following examples, to a class that serves as a data source for a picker:

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 1;
}
```

In this method the desired number of components (can also be thought of as columns) inside the picker view is specified. The number of desired components should be returned by this method. The next method could be used to specify the number of rows desired in a picker view.

```
- (NSInteger)pickerView:(UIPickerView *)pickerView  numberOfRowsInComponent: (NSInteger)component{
    return [array count];
}
```

In most of the cases data to be displayed in the picker view is stored in an array, therefore it is useful to return the number of rows can be returned by the count method for arrays.

The delegate is used to construct the components with the content for each row for the picker view, this is implemented in the UIPickerViewDelegate protocol. This can be done by adding a method, similar to the following, to a class that serves as a picker view delegate.

```
- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:(NSInteger)row forComponent:(NSInteger)component {
    return [array objectAtIndex: indexPath.row];
}
```

This method is responsible for displaying the content of the rows in a picker view. Use syntax similar to the code above to display information from an array inside a picker view. The picker used in our app will be described in more detail in the next section.

The general process for incorporating a picker view in an app is to (a) drag a picker view from the Xcode objects library into the View window, (b) set a data source that provides the number of components and rows for the picker view, and a delegate for the picker view that provides the methods that implement the rows and contain the content for the rows, (c) declare the protocols for the data source and delegate, and (d) implement the methods for the protocols.

Set the Data Source and Delegate for the Picker View on the High Scores Display Screen

Click on the GameViewController.xib file in Xcode to open it in the editor area. Click on the View icons in the panel to the left of the editor area to open the view that contains our picker view.

To set the data source for the picker view, ctrl-click on the picker view item in the GameViewController.xib window, which will open a new outlet window. Click on the open circle to the right of the Outlets dataSource and drag the elastic blue line to the File's Owner element in the GameViewController.xib window, as shown in Figure 14. Then click on the open circle to the right of the Outlets delegate and drag the elastic blue line to the File's Owner element in the GameViewController.xib window as you just did to set the datasource. Save the file.

Add Code for the Picker View on the High Scores Display Screen

To indicate that we have a picker view datasource and delegate in the interface directive for the GameViewController class by adding the following code to the interface directive in the GameViewController.h file.

<UIPickerViewDelegate, UIPickerViewDataSource>

The interface directive will now appear as follows.

```
@interface GameViewController : UIViewController<UIPickerViewDelegate, UIPickerViewDataSource>
```

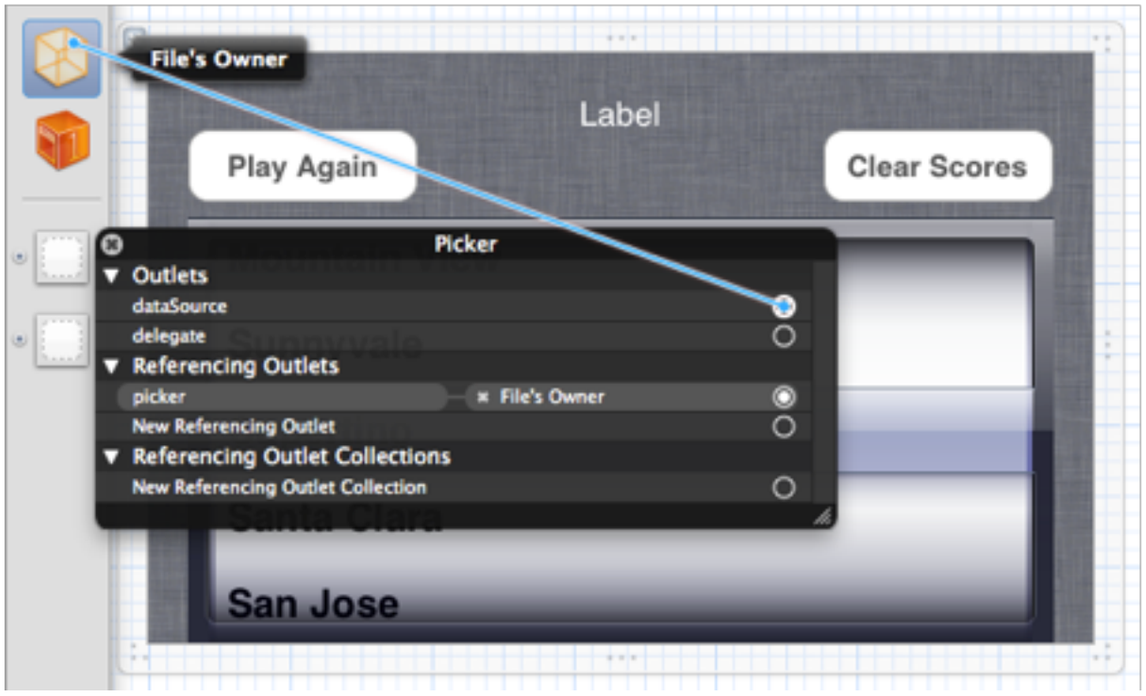

Figure 14 Setting the Data Source for the Picker View Element

Next, add the following method near the end of the GameViewController.m file.

```
- (NSString *)pickerView:(UIPickerView *)pickerView titleForRow:(NSInteger)row forComponent:(NSInteger)component {
    NSString *nameString=[[defaults arrayForKey:@"highScoresNames"]objectAtIndex:row];
    NSString *scoreString=[[defaults arrayForKey:@"highScores"]objectAtIndex:row];
    NSString *boneText=[NSString stringWithFormat:@"%@ dog bone for %@",scoreString, nameString];
    NSString *bonesText=[NSString stringWithFormat:@"%@ dog bones for %@",scoreString, nameString];
    int numberOfBones = [scoreString intValue];
```

```
    if (numberOfBones==1)
        return boneText;
    else
        return bonesText;
}
```

This method is called every time when the operating system needs to draw a new row in the picker view on the screen. So if our picker has 10 rows, then the method is called 10 times. The first thing we do in this method is create four strings. The first will contain the name, and the second will contain the score, and the third and fourth merge that score and name into a message that will be displayed in the picker view. To ensure proper grammar in our game the third string refers to a single dog bone, and the fourth string refers to zero or multiple dog bones. Note that we used the row property of the method to access the correct object in the array. Next, we will create a method that specifies how many components (or columns) there are in our picker. Add the following method near the end of the GameViewController.m file. We only need one component which displays a string so we simply return a value of 1.

```
- (NSInteger)numberOfComponentsInPickerView:(UIPickerView *)pickerView {
    return 1;
}
```

The next step is to specify the number of rows in our picker view. Add the following method near the end of the GameViewController.m file.

```
- (NSInteger)pickerView:(UIPickerView *)pickerView numberOfRowsInComponent:(NSInteger)component {
    return [[defaults arrayForKey:@"highScores"]count];
}
```

This method sets the number of rows of the components in the picker view. In our case we will return the count of the highScores array taken from the defaults array.

Add Code to Display the View for the High Scores Display Screen

Now that the view for the High Scores screen is partially implemented we can add the code to the method we previously created to display the view. Add the following code to the inside of the displayScoreView method in the GameViewController.m file.

```
    [self.view addSubview:highScoresView];
    highScoresView.frame=CGRectMake(0,0,480,300);
    [picker reloadAllComponents];
    NSString *boneText=[NSString stringWithFormat:@"Congratulations %@, you collected %d dog bone", self.yourName, score];
```

```
NSString *bonesText=[NSString stringWithFormat:@"Congratulations %@, you collected %d dog bones", self.yourName, score];
if(score==1) {
    finalScoreLabel.text=boneText;
}
else {
    finalScoreLabel.text=bonesText;
}
```

This method defines the highScoreView, sets the position of the frame to the iPhone screen origin, and sets the size of the frame to 480 by 300 points, and then loads the elements into the picker view. The last statements are added to display the information about the final score for the user. Now we can uncomment the last statement in the saveScore method that we added previously. Find the saveScore method in theGameViewController.m file that calls the displayHighScores method. Remove the comment symbol so that the statement appears as follows.

```
[self displayScoreView];
```

Save your files, and click on the Run button in Xcode to test your app in the iPhone simulator. The high scores display is functional at this point. We will provide the functionality for the buttons on the high scores screen next. We already declared the methods, and added body skeletons for the methods that will provide the functionality for the buttons on the high scores screen.

Add Code for the Play Again Button on the High Scores Display Screen

To play again, we simply need to dismiss our current view so that the previous view that contains the Home Screen is displayed and reset the score. Add the following statementsto the inside of the playAgain method in the GameViewController.m file.

```
[self dismissModalViewControllerAnimated:YES];
score = 0;
```

Add Code for the Clear Scores Button on the High Scores Display Screen

To provide the functionality for the Clear Scores button. We need to clear the defaults information, synchronize the array, and then invoke the playAgain method to redisplay the interactive game screen. Add the following code to the clearScores method in the GameViewController.m file.

```
[defaults removeObjectForKey:@"highScores"];
[defaults removeObjectForKey:@"highScoresNames"];
[defaults synchronize];
[highScoresView removeFromSuperview];
[self playAgain:nil];
```

Play Your Game!

The Treats! app is now complete. Save all files then click on the Run button in Xcode to test your app. The last thing to do is to now consider if there are any minor tweaks needed for the appearance of the app. We noticed that there wasn't room for players' names that were more than a few characters long on the high scores screen so we lengthened the size of the label for the player's name. The final screens are shown in Figures 15 through 17.

Figure 15 Completed Home Screen

Figure 16 Final Interactive Game Screen

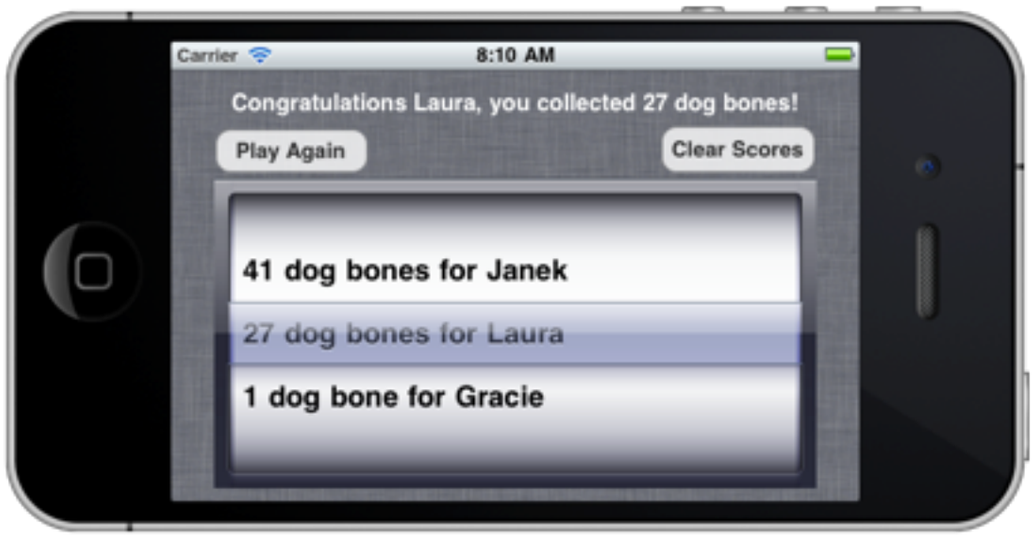

Figure 17 Final Scores Screen

Summary

This chapter presented the development of a high scores game that introduced the use of the NSUserDefaults class to save persistent data and a picker view user interface element to display persistent data within the context of a high scores game. Before starting the development of the game a storyline was constructed with a flowchart and an overall approach for the development was defined. The approach was to create user interface elements for multiple screens of the game, make connections between the user interface elements and the code, and then to provide the code for the functionality of the user interface elements in the view controller classes.

The objective of the game is for players to tap as many randomly appearing dog bones on the screen as possible within a set time frame. This game consists of three screens, each implemented within a separate View.

1. The Home Screen presents a background image, labels to provide the instructions for the game, a text box used to acquire a player's name for the current game, and a button to start the game play by launching the Interactive Game Screen.

2. The Interactive Game Screen Game presents a background image, a label the score, and a countdown timer for the time left in the game, and a bone that randomly moves around the screen.

3. The High Scores Screen presents a picker view to display the scores and two buttons to restart the game or to clear the scores.

In addition to the introduction of two new classes – NSUserDefaults and UIPickerView, the implementation of this game also incorporated the use the iPhone coordinate system for defining the location and size of interface elements in points, managing the orientation of application screens, and the implementation of touch events.

Review Questions

1. Which class would you use to display a set of values using the spinning wheel?

2. Can you save images using the NSUserDefaults? Explain why.

3. What does the synchronize method do?

4. List the touch handlers. When are they triggered?

5. Which method is responsible for controlling the number of components that will be displayed in a picker view?

Exercises

1. Develop an app that requires the user to tap some items and double tap other items.

2. Develop an app that displays high scores and contains items that should be tapped and items that should not be tapped on. Increase the score for items that should be tapped and decrease the score for items that should not be tapped.

Chapter 8

Flash Cards App: Storyboards

Flash Cards App - Storyboards

Concepts emphasized in this chapter

- *Storyboards*
- *Navigation Controllers*

Introduction

In this chapter you will be introduced to a new feature in iOS programming with Xcode 4.3 and iOS 5, which is storyboards. Storyboards provide a way to represent scenes within an app and to define relationships and transitions between those scenes. A storyboard is shown within the Editor area of Xcode and consists of scenes and seques. Scenes within a storyboard are managed by view controllers that represent different screens within an app. Segues are used to define transitions between scenes. In this chapter you will create a sample flash card application using the Xcode storyboard feature. This app will contain six view controllers with questions, answers, and explanations that helps a user learn about Star Wars. Let's get started!

Development

The development of the Flash Cards app will be conducted in several phases.

1. Create a new project that uses a storyboard

2. Add view controllers to the storyboard

3. Set the layout of the views and add user interface controls

4. Add a navigation controller and create segues between the scenes

5. Provide custom code to implement a little more functionality for the app

Create a New Project in Xcode

Launch Xcode and create a new project -- choose a Single View Application template for your project.

Figure 1 Welcome to Xcode Window

Choose the Single View Application template as shown in Figure 2, then click on the Next button.

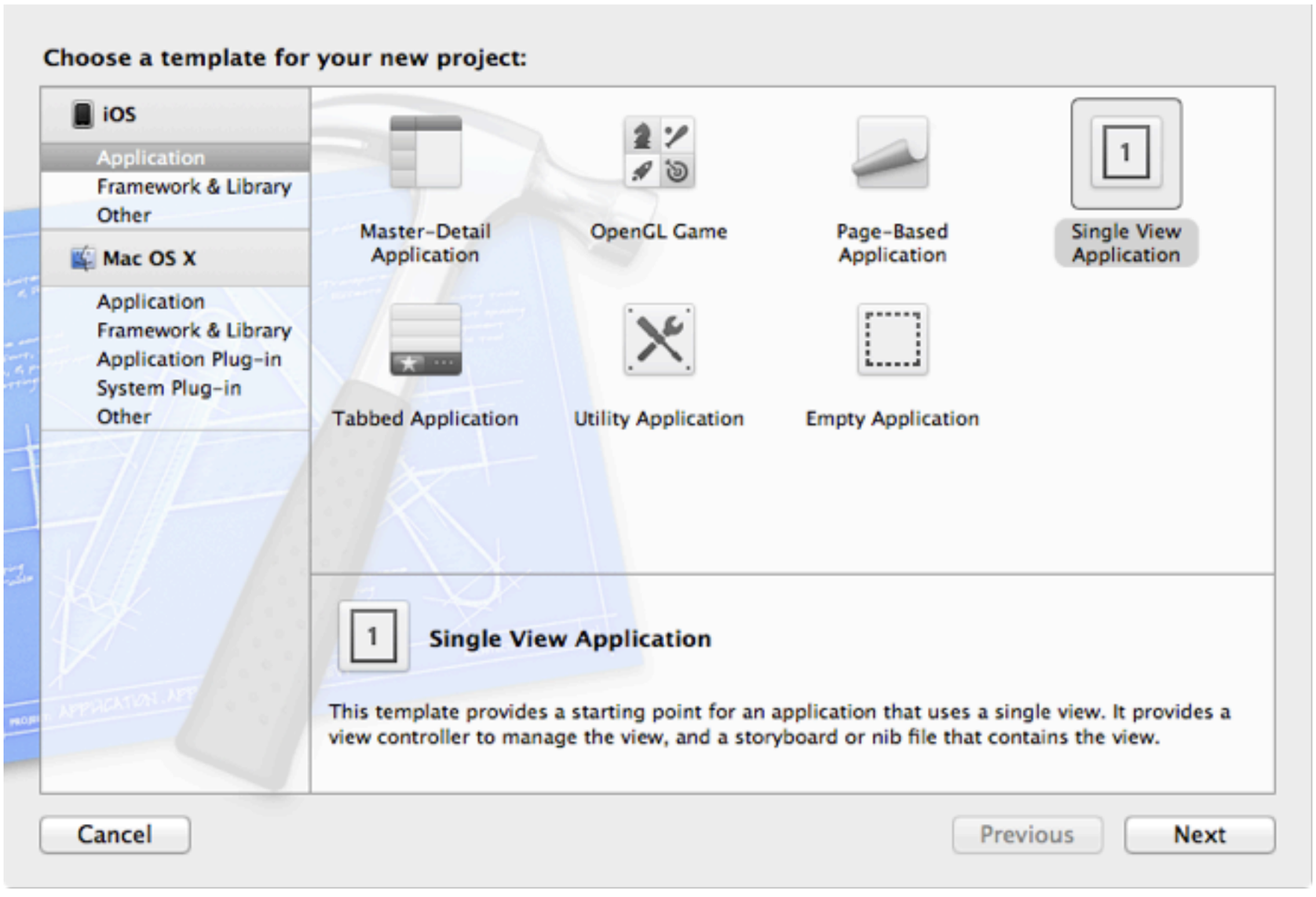

Figure 2 Creating a New Project – Selecting a Template

Enter Flash Cards for the ***Product Name*** as shown in Figure 3. A different name can be used, but it will be easier to follow along with this project if you use the same name.

Enter *com* for the ***Company Identifier*** or leave it with the default value as shown in Figure 3.

Select iPhone in the Device-Family menu as shown in Figure 3.

Select the ***Use Storyboard*** checkbox and the ***Use Automatic Reference Counting*** checkbox as shown in Figure 3. Click on the Next button.

Choose a location for your project, do not select the Source Control checkbox at the bottom of the window, then click on the ***Create*** button.

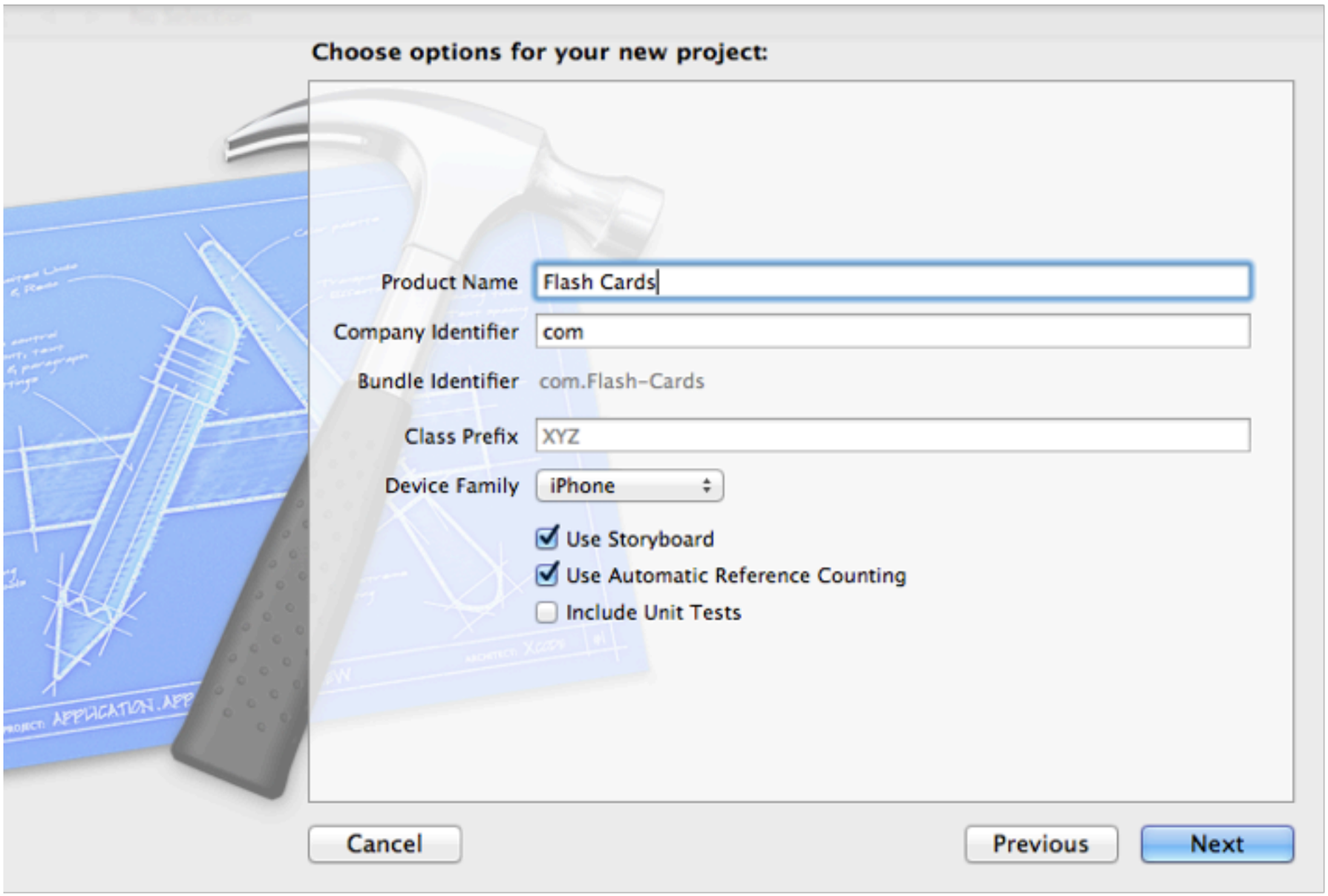

Figure 3 Creating a New project – Naming the Project

Storyboard Basics

As a result of selecting the Use Storyboard checkbox when creating this project, we find a MainStoryboard.storyboard file in the Flash Cards folder in the Navigator area of Xcode. Files with a storyboard extension store information about user interfaces and connections between the view controllers. Select this file, your editor area should appear similar to Figure 4.

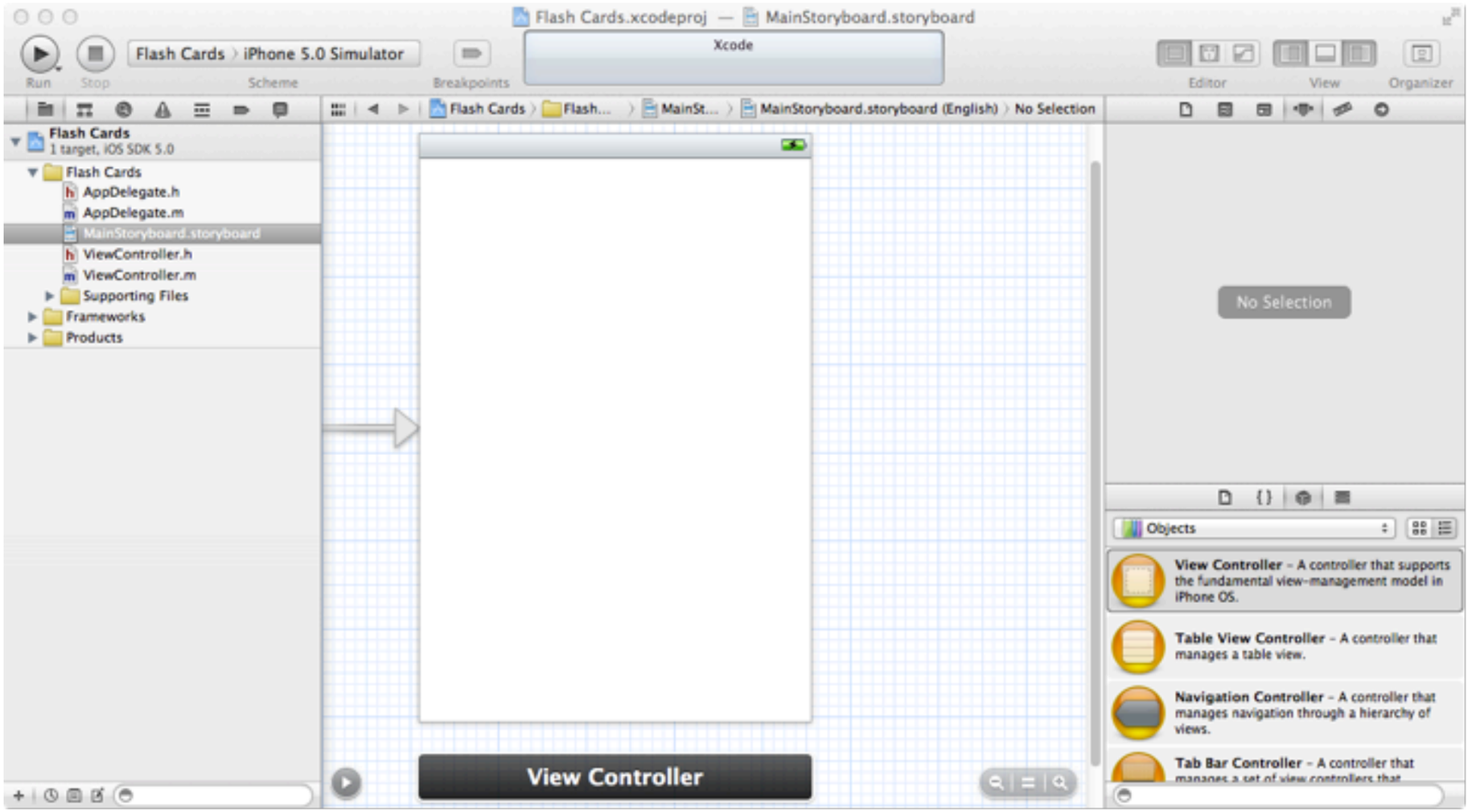

Figure 4 Storyboard

The big rectangle in the center of the screen represents the View Controller. Notice the arrow as shown in Figure 5 on the left of the View. The arrow denotes the start of the sequence of the storyboard. What does it mean? The start of the sequence is what will be displayed on an iOS device first.

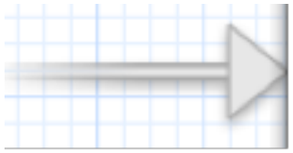

Figure 5 Start of Sequence Arrow

Notice the small set of buttons, as shown in Figure 6, in the bottom right corner. of the editor window You can use these buttons to zoom in or zoom out of the content of the storyboard window. This is very helpful when multiple views are displayed in the editor area.

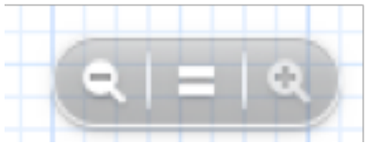

Figure 6 Zoom Buttons

Create the Home Screen

We can edit view controllers, including adding user interface elements to them, in the storyboard editor. To add user interface elements, simply double click on the view controller to zoom in to it, and then drag the desired user interface element from the objects library. Let's do this now. Double click on the view controller, and then drag a label, text view, and button into the view. Change the name of the label to ***StarWars Expert.*** Change the text of the text view to ***Welcome here human. Are you ready to become a Jedi Warrior?*** and then make sure that the Editable attribute for this text view as shown in Figure 7 -- and all text views used in this app-- is NOT checked. Change the text on the button to ***Begin***. Your view should now appear similar to Figure 8.

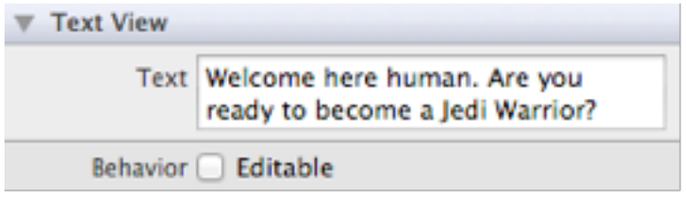

Figure 7 Editable Attribute for the Text View

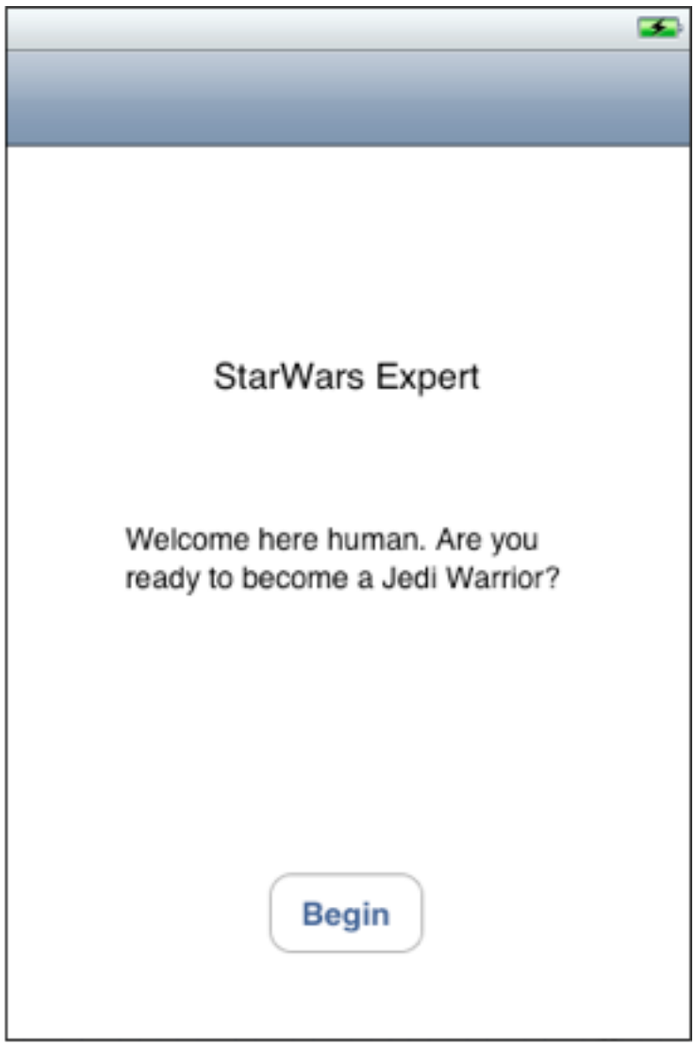

Figure 8 Customized View Controller

Click on the Run button in Xcode to test the application in the iPhone simulator. Your simulator should look similar to Figure 9.

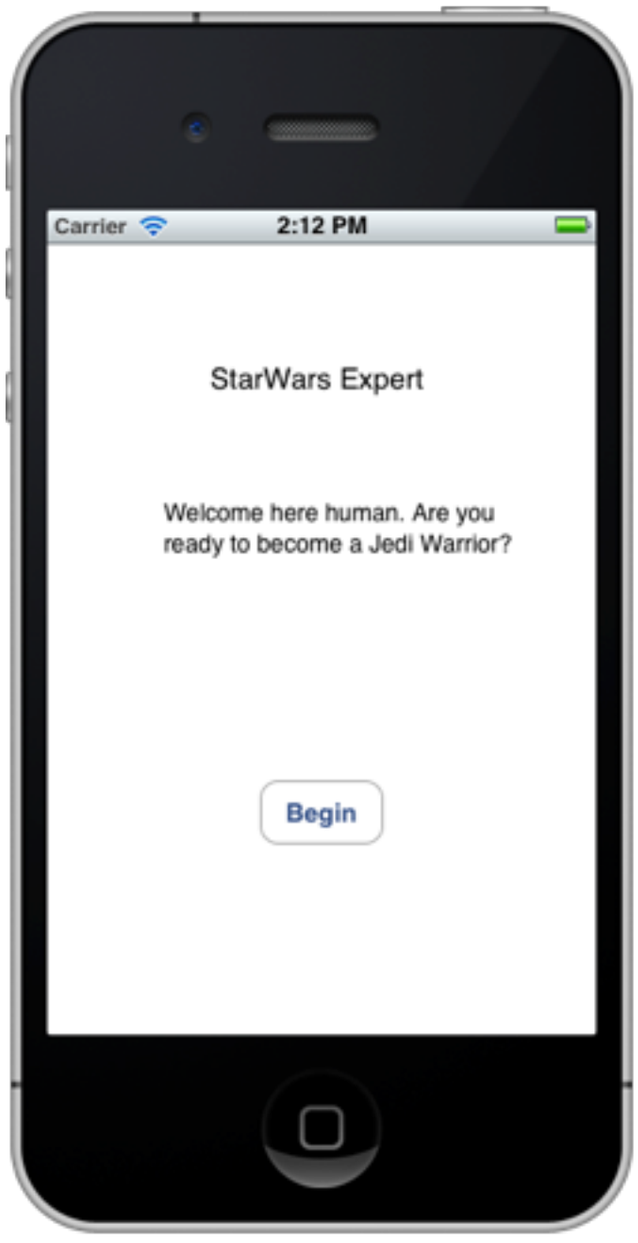

Figure 9 Flash Cards App Running in the iPhone Simulator

Our sample Flash Cards app will consist of two questions test a user's knowledge about the Star Wars movies.

Once a user taps on the Begin button on the Home screen the screen with question 1 is presented. The question screens have two buttons--one b button is labeled with the correct answer and one button is labeled with an incorrect answer. If the user taps on the button that contains the correct answer then the screen with the next question is displayed. If the user answers incorrectly the screen with an explanation is displayed that also contains a Continue button. When the user continues the screen with the next question is presented. Once the last question is answered correctly or the continue button on the associated explanation is tapped, then the final screen is presented.

For these six screens we will need six view controllers--one view controller for each screen.

We just created the first home screen view. Now we need to create more views for:

- Question 1, this view will display the first question.
- Explanation 1, this view will display information about the first question.
- Question 2, this view will display the second question.
- Explanation 2, this view will display instructional information about the second question.displaying explanation of the answer the second question
- Congratulations, this view will display the final screen.

So let's get started. Go ahead and drag five View Controllers from the Objects library into the storyboard editor window. Your editor should then appear similar to Figure 10.

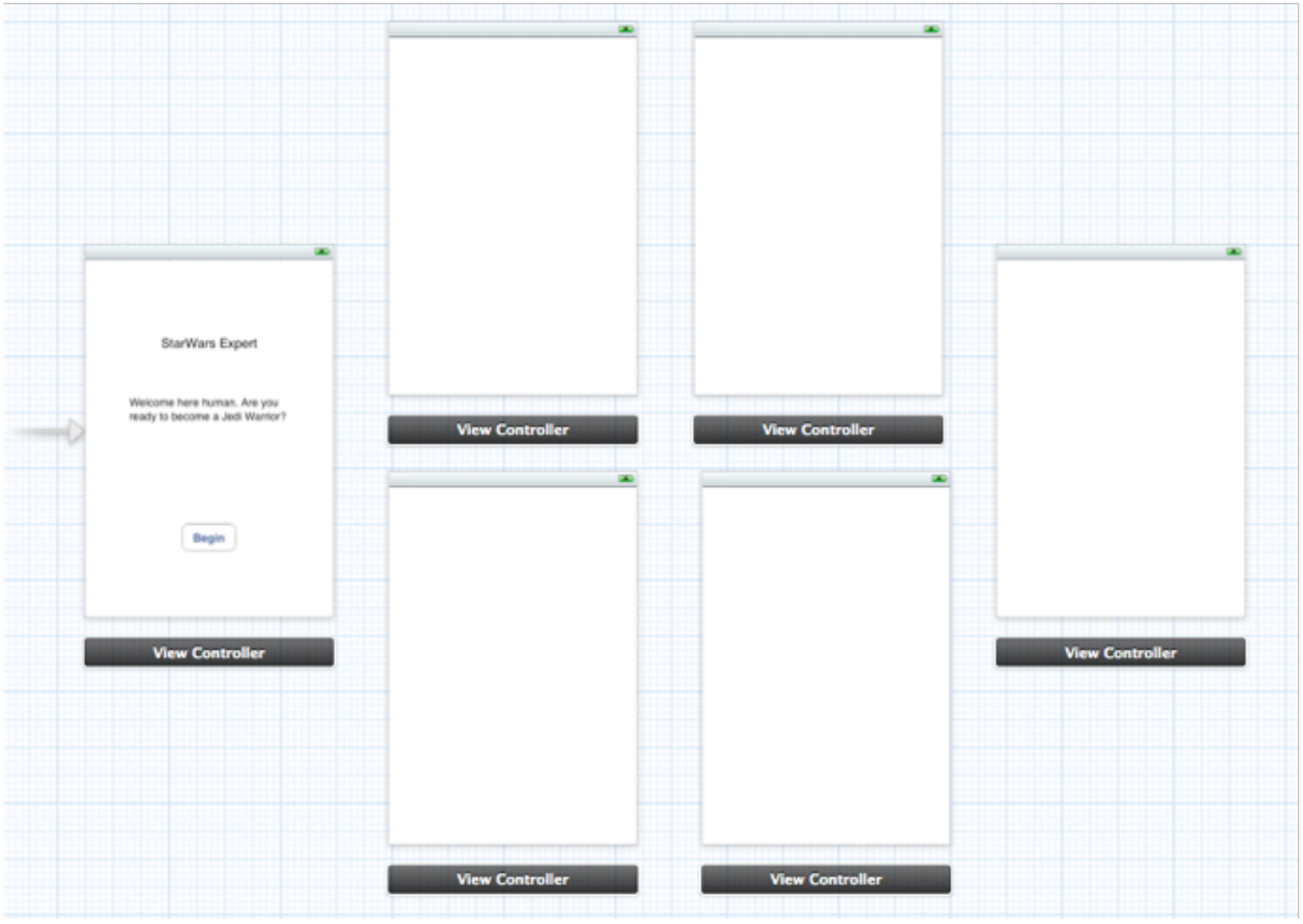

Figure 10 Storyboard Editor with Six Views

Create the Question 1 Screen

Now that we have all the views that we need for our app, we need to add the user interface elements that we for our app to the additional views. Double click on any of the empty views, and drag a label, text view, and two buttons from the objects library into the view. Change the title of the label to ***Question 1***. Change the text in the text view to ***What rank does Jar Jar attain?*** Change the title of the buttons to ***General*** and ***Admiral***. Don't forget to uncheck the Editable attribute for the text view. Resize and reposition the elements, and select attributes as you wish. This view should appear similar to Figure 11 when you are done.

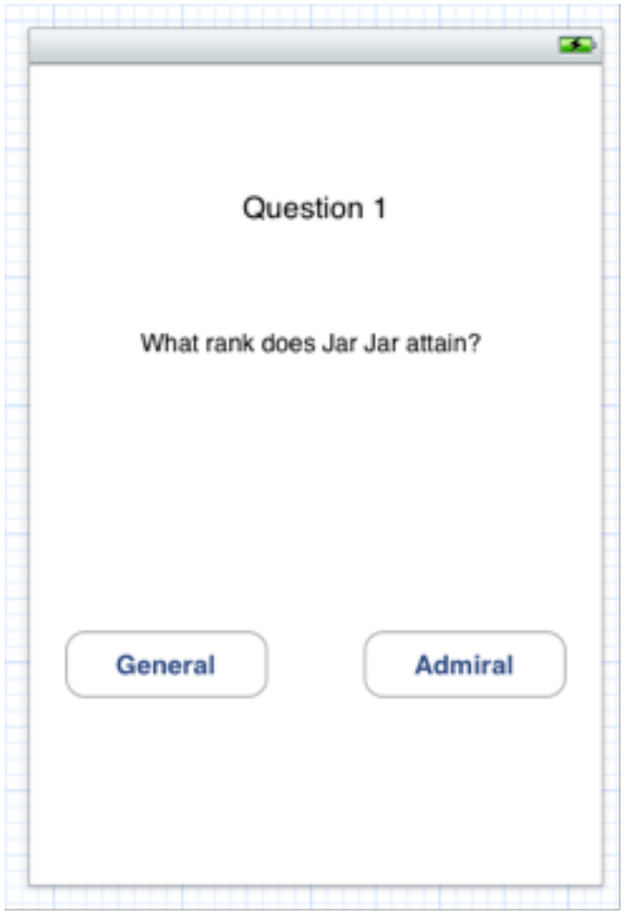

Figure 11 Question 1 View

Create the Explanation 1 Screen

Double click on any of the empty views, and drag a label, text view, and one button from the objects library into the view. Change the title of the label to ***Explanation***. Change the text in the text view to ***Jar Jar becomes General Binks and helps lead the attack on The Trade Federation Army.*** Change the title of the button to ***Continue***. Don't forget to uncheck the Editable attribute for the text view. Resize and reposition the elements, and select attributes as you wish. This view should appear similar to Figure 12 when you are done.

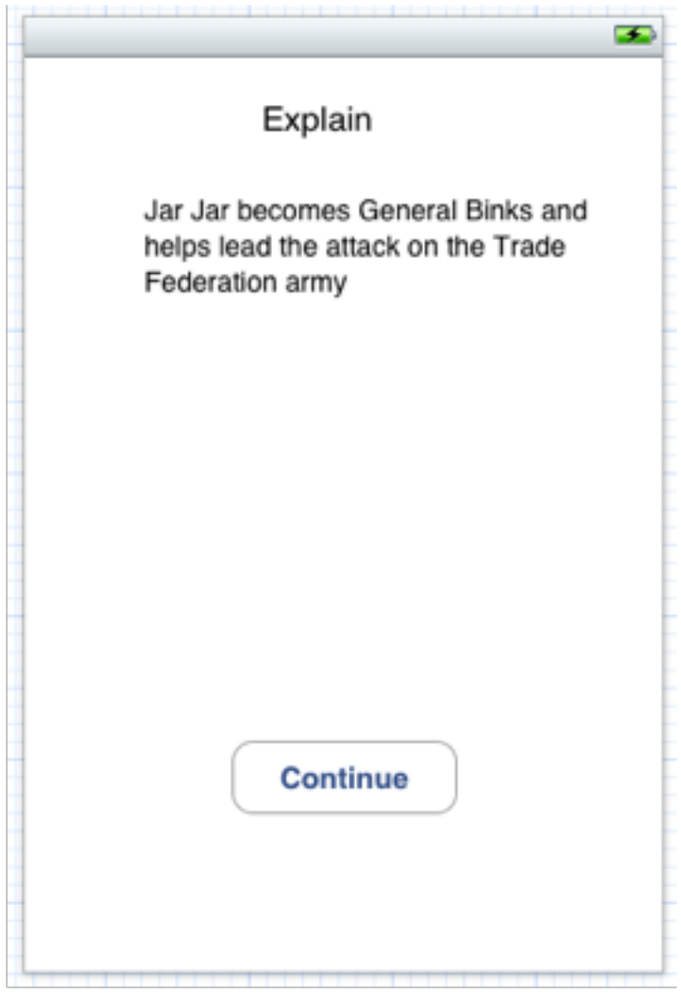

Figure 12 Question 1 Explanation View

Create the Question 2 Screen

Double click on any of the empty views, and drag a label, text view, and two buttons from the objects library into the view. Change the title of the label to ***Question 2***. Change the text in the text view to ***What is the last word spoken in Star Wars: Episode 1 - The Phantom Menace?*** Change the title of the buttons to ***War*** and ***Peace***. Don't forget to uncheck the Editable attribute for the text view. Resize and reposition the elements, and select attributes as you wish. This view should appear similar to Figure 13 when you are done.

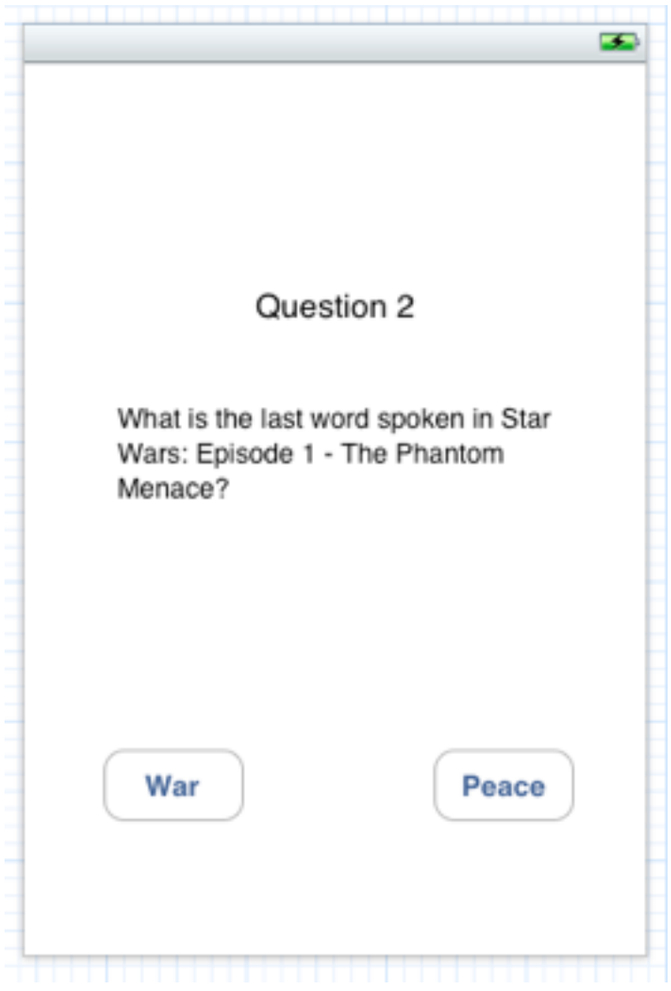

Figure 13 Question 2 View

Create the Explanation 2 Screen

Double click on any of the empty views, and drag a label, text view, and one button from the objects library into the view. Change the title of the label to ***Explanation***. Change the text in the text view to ***The Gargans and the people of Naboo have a new relationship and there is no more war in Naboo.*** Change the title of the button to ***Continue***. Don't forget to uncheck the Editable attribute for the text view. Resize and reposition the elements, and select attributes as you wish. This view should appear similar to Figure 14 when you are done.

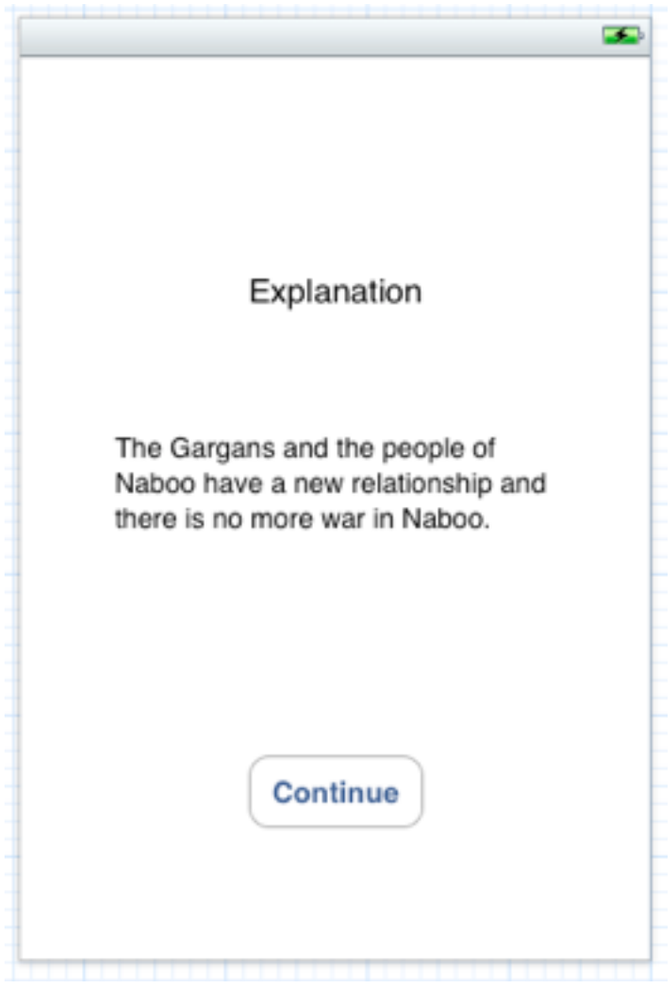

Figure 14 Question 2 Explanation View

Create the Congratulations Screen

Double-click on the last empty view and drag a label, text view, and a button from the objects library into the view. Change the title of the label to ***Congratulations!*** Change the text view to ***You are now a StarWars Expert!*** Change the title of the button to ***Home***. Don't forget to uncheck the Editable attribute for the text view. Resize and reposition the elements, and select attributes as you wish. This view should appear similar to Figure 15 when you are done.

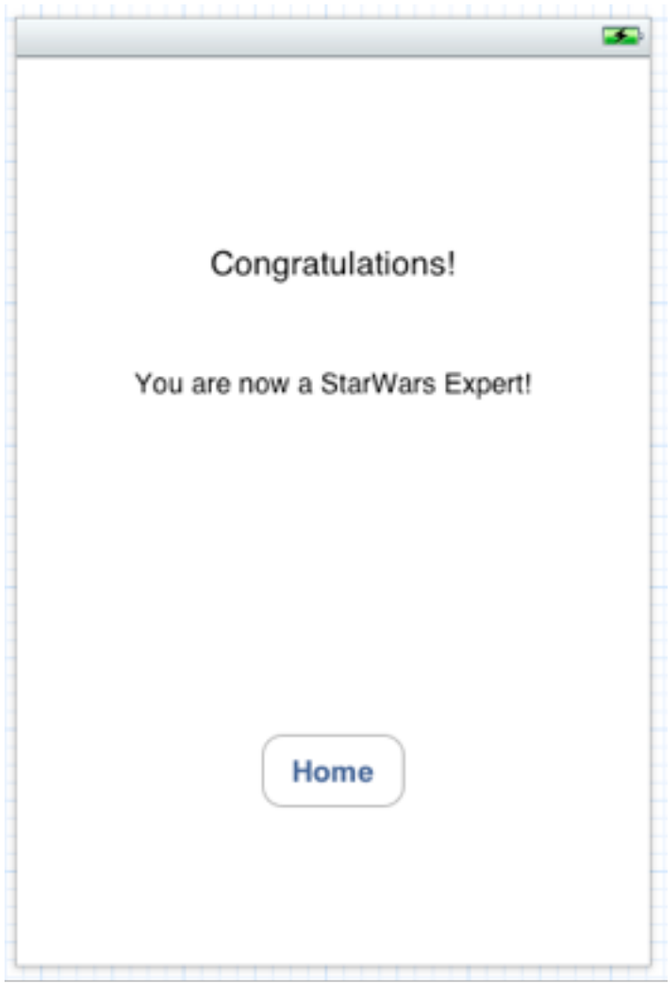

Figure 15 Final View

Now all of your views should contain user interface elements. Click on the Run button to build and test your app. The app doesn't et have the functionality to change the views so you should only see the first screen and the Begin button doesn't do anything yet either. However, successfully building the app at this point indicates that there are no faults in what you have done so far.

Working with Segues and a Navigation Controller

A segue represents the transition from one scene/screen called the source to the next scene/screen called the destination. For example in our application the Home screen is a source and the view for the first question is a destination. There are several different kinds of segues: Custom, Modal and Push, and each of them has its advantages.

Custom Segue - Can be used when a transition other than that provided by the Modal or Push Segues is desired. To implement a custom segue the developer should overwrite the perform method in a class that is using the segue.

Modal Segue - Should be used to display some information momentarily. For example it might be used to present the view that displays contact information, or the Help screen for an application.

Push Segue - Must be used with the navigation controller, and it is used to ***push*** the contents of a destination's view controller on top of the source's view controller. The push segue should be used in complex applications comprised from multiple scenes/screens. We will use the the push segue in our application.

Creating a segue is simple. To create a segue in our app, ctrl-click on a button in one of the view controllers, select the desired type of segue, and then drag to the view controller that should be displayed when a user taps on the button.

Let's do this now. Select the MainStoryboard.storyboard file to open it in the editor area if it isn't already selected. Drag Interface Builder editor, if it's not open yet.

Drag a Navigation Controller from the objects library to the editor area.Your storyboard should then look similar to Figure 16. We added the Navigation Controller into the left side of the editor area.

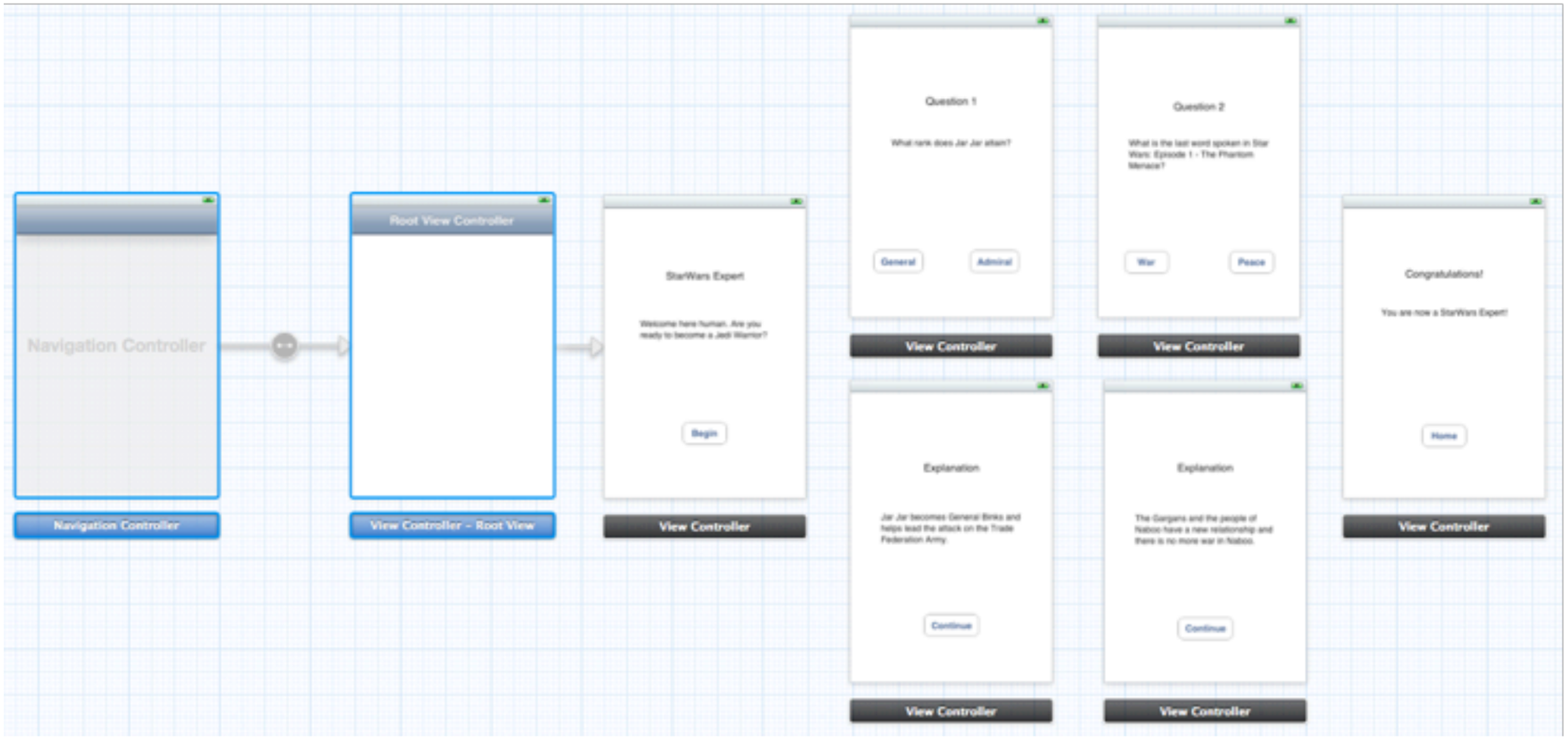

Figure 16 Storyboard with the Navigation Controller

Adding the Navigation Controller to our Storyboard resulted in the addition of a Root View Controller also. that you just dragged to window is represented by two view controllers in the storyboard. Each navigation controller needs to have a root view controller, that will contain the content that should be displayed when the navigation controller is launched. Since we already created a home screen for our app we don't need this additional controller so deselect the Navigation Controller--make sure that only the Root View Controller is selected-- then press the delete key.

Now ctrl click on the Navigation Controller, in the popup window drag from the circle next to Relationship - rootViewController to the View Controller that contains the home screen of our app as shown in Figure 17.

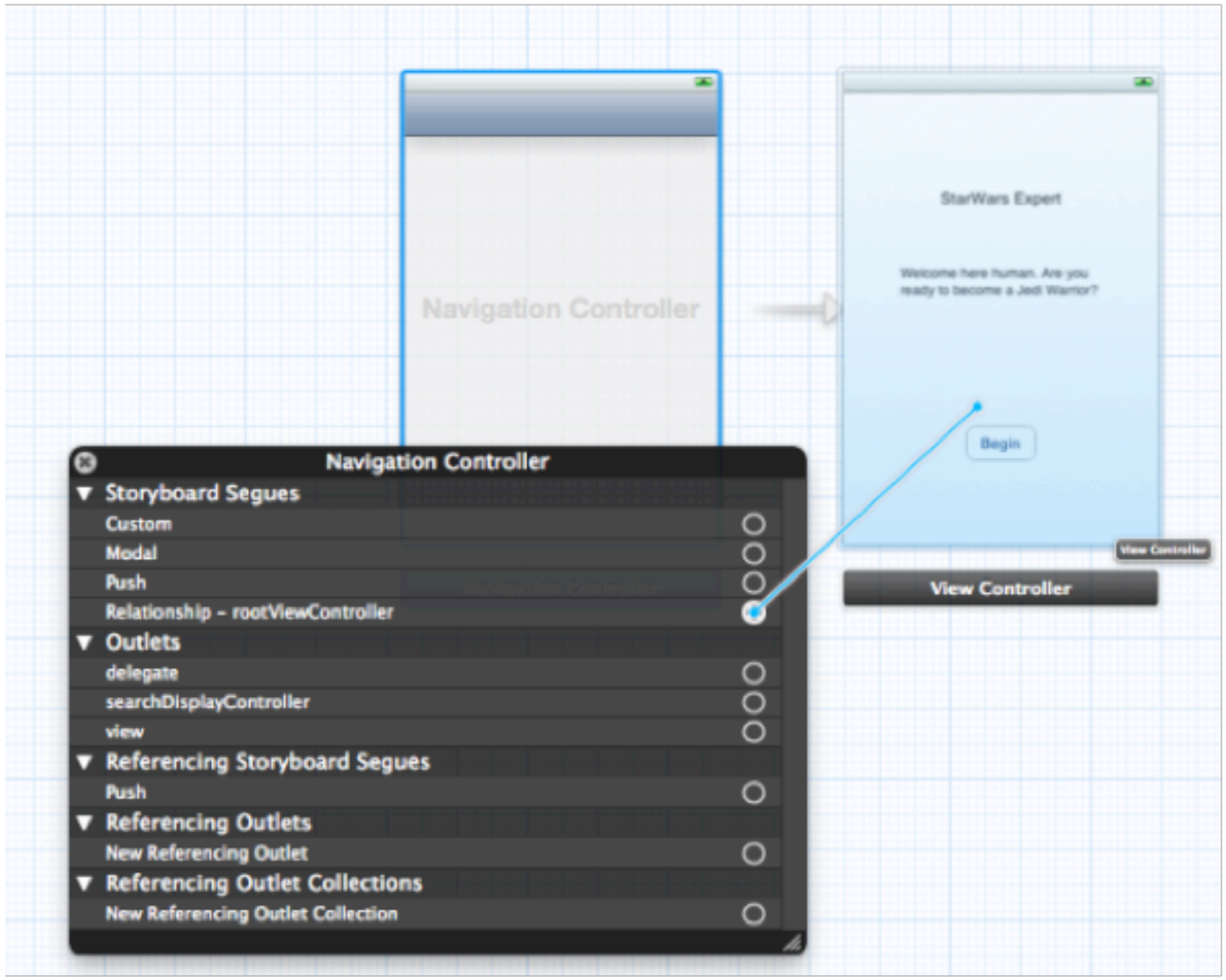

Figure 17 Setting the New Root View Controller for the Navigation Controller

Since we want the Navigation Controller to be responsible for managing our views and to be first in the view hierarchy click and drag the arrow shown in Figure 18 so that instead of pointing to the home screen it points to the Navigation Controller. Your view should now appear similar to Figure 19.

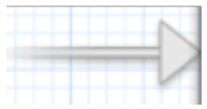

Figure 18 Arrow that Indicates the First View

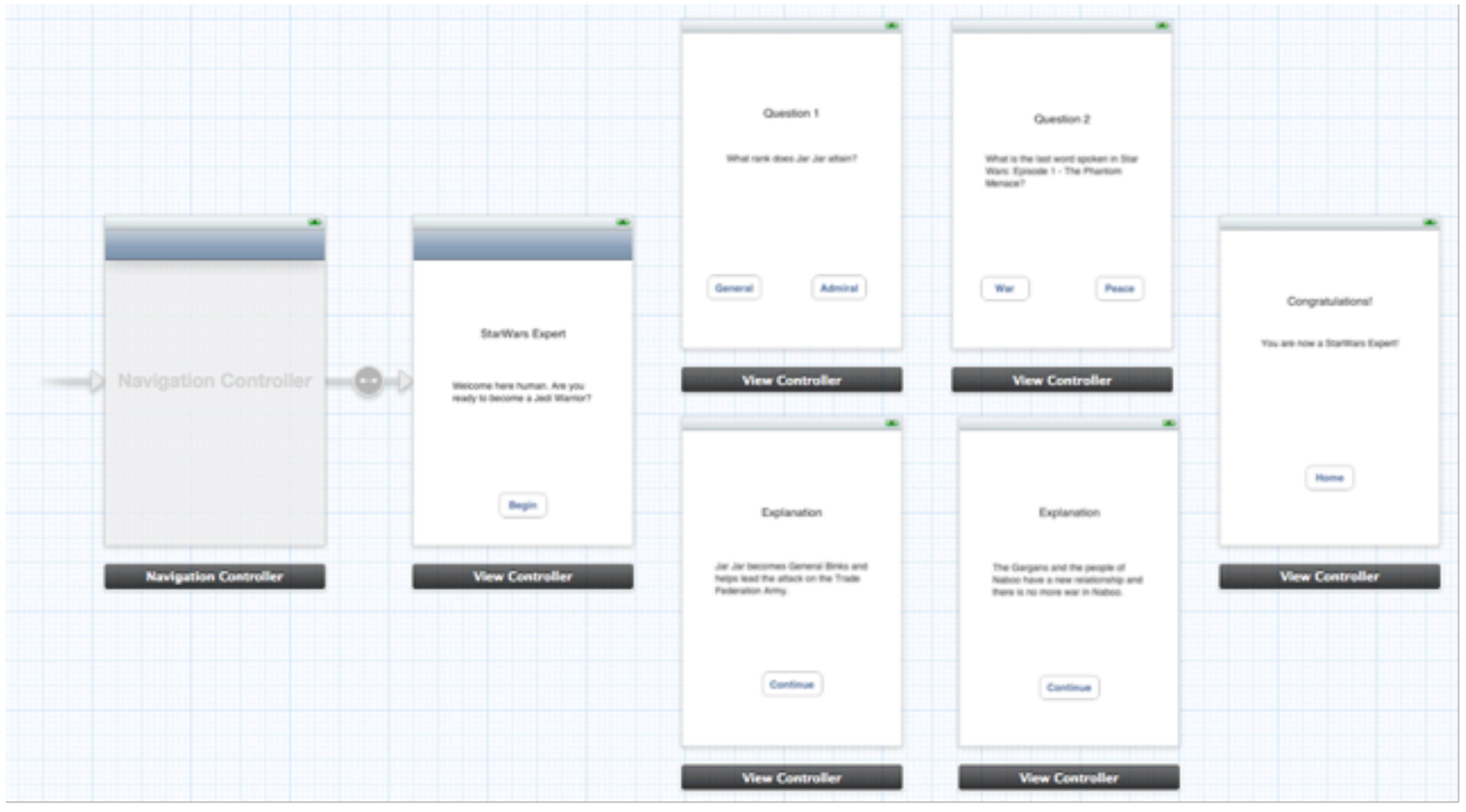

Figure 19 Changing the First Display Item in the Storyboard to the Navigation Controller

Now we need to create the segues between the View Controllers for the screens with our flash card home, question, explanation, and congratulations screens. Select the view controller for our home screen then ctrl-click on the button inside that view. Look at the popup window that appears and make sure that it is for the button and not the view. The. top section of the popup is titled Storyboard Segues and it contains the three types of segues that we described previously. Recall that we want push segues so drag from the circle to the right of Push to the Question 1 view controller as shown in Figure 20.

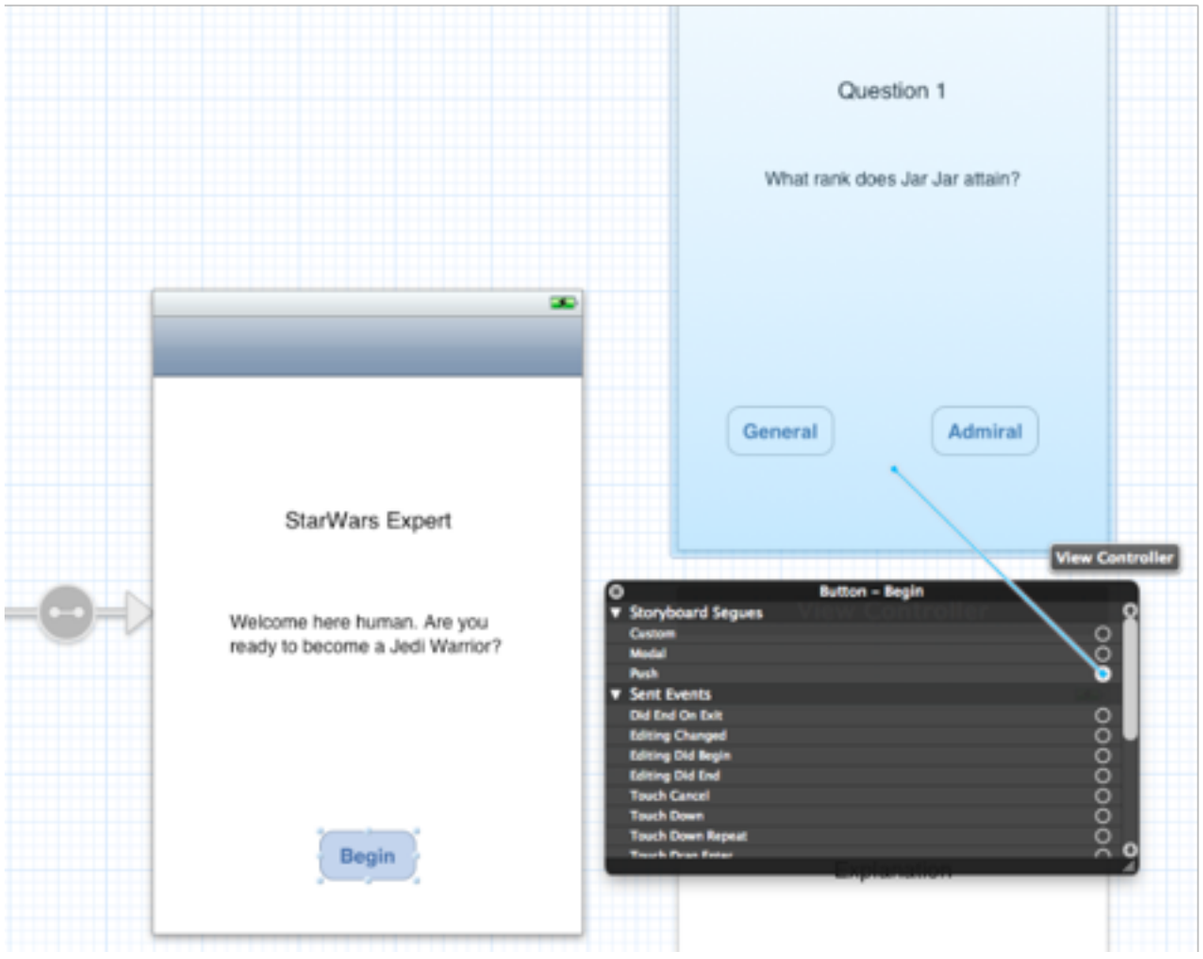

Figure 20 Creating a Push Segue Between the Home Screen and the Question 1 Screen

If the segue was created successfully you should see a new arrow between the view for the home scene and the view for the Question 1 scene as shown in Figure 21.

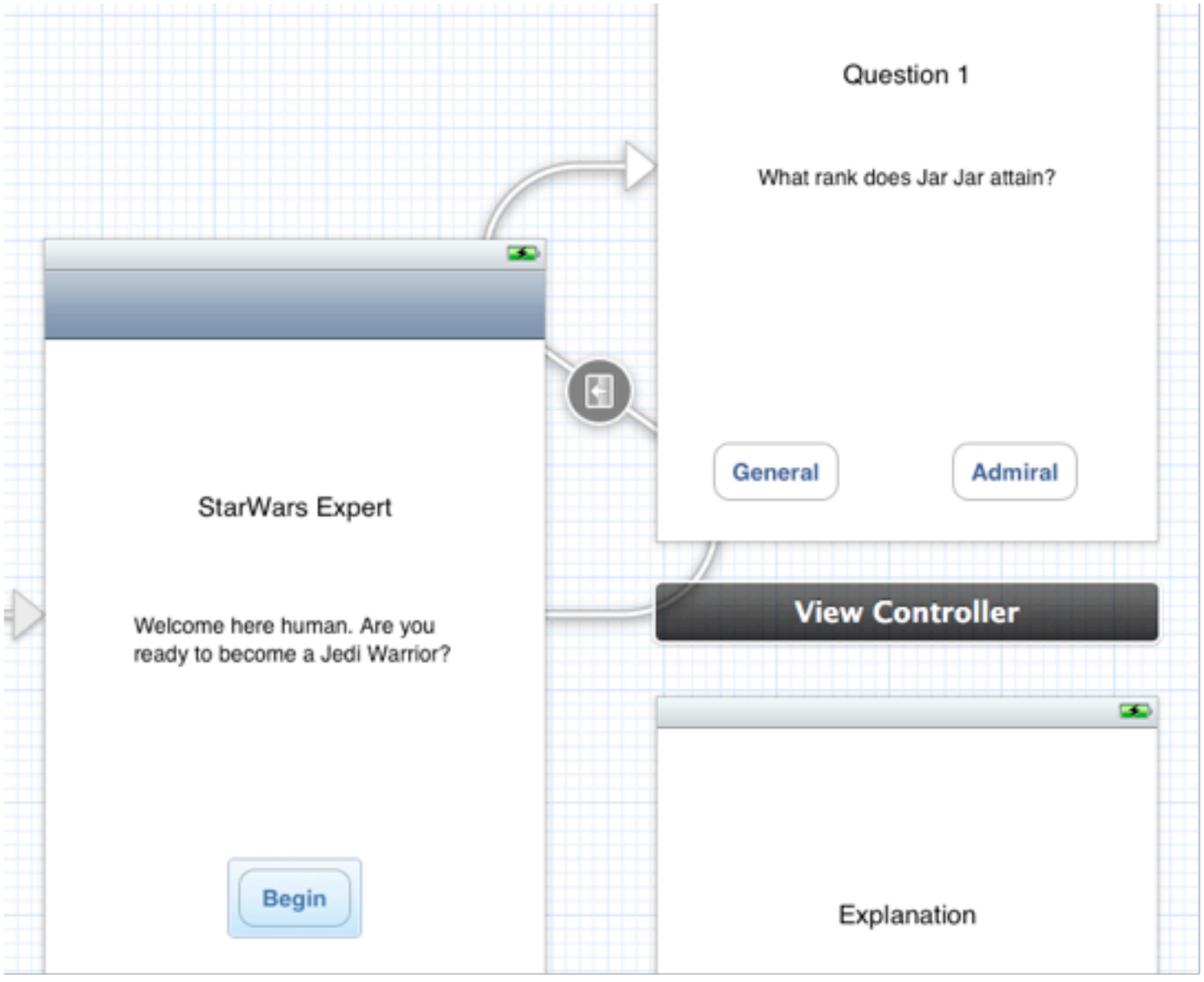

Figure 21 Push Segue Between the Home Screen and the Question 1 Screen

Now repeat this process to create push segues from the:

- Question 1 screen's General button to the Question 2 screen.
- Question 1 screen's Admiral button to the first Explanation screen we created.
- First Explanation screen's Continue button to the Question 2 screen.
- Question 2 screen's War Button to the second Explanation screen that we created. with the Explain 2 Controller
- Peace Button with the Congratulations Controller
- Continue button in the Explain 2 view to Congratulations controller.

Once all these push segues have been added, your storyboard should appear similar to Figure 22.

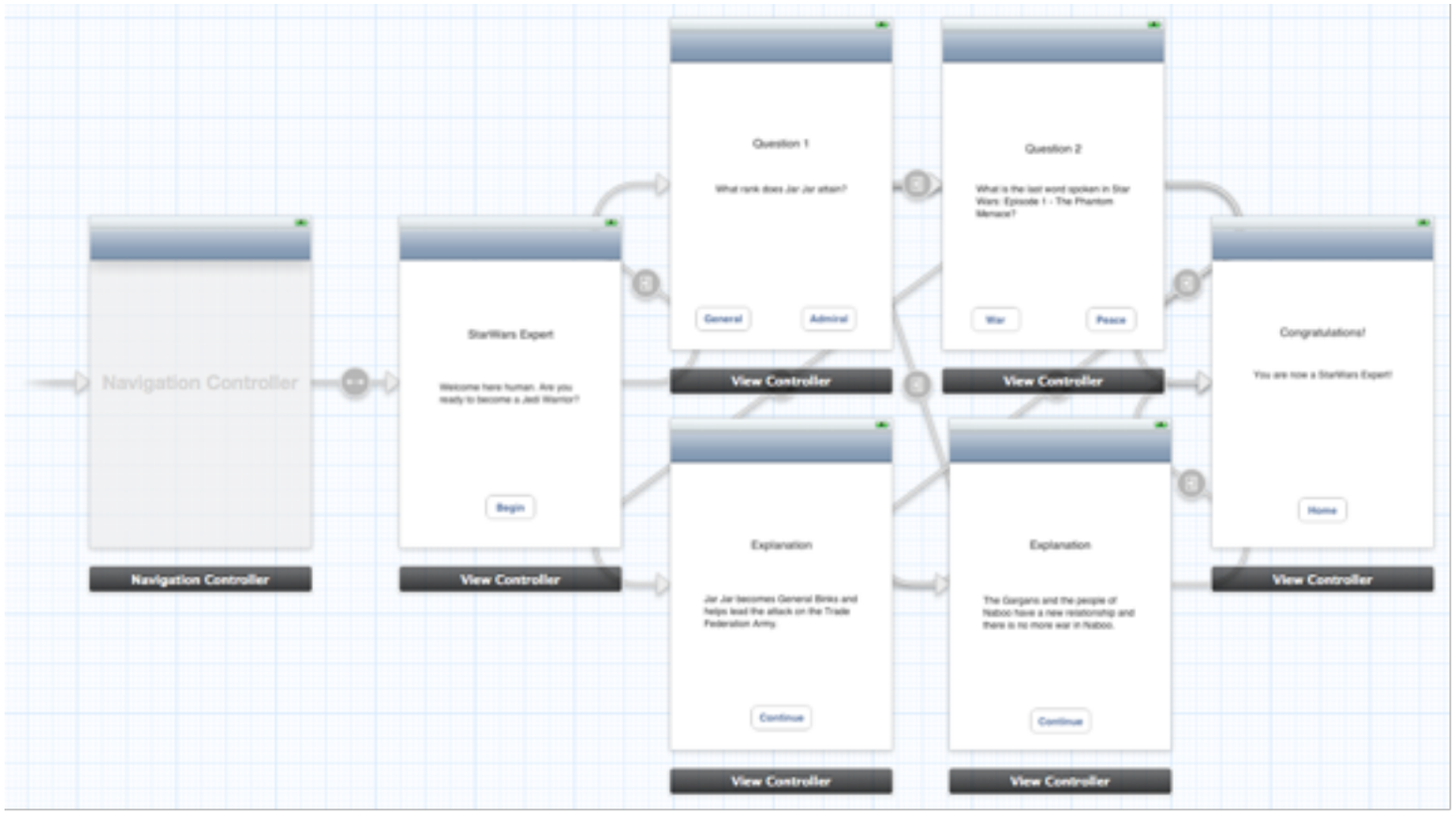

Figure 22 Storyboard with Push Segues

Click on the Run button to test your app. You should be able to navigate between the view controllers by tapping on the buttons. All of the buttons should function as expected except for the Home button on the Congratulations screen. Additionally, the navigation bar at the top of the simulator will contain a Back button on every screen, as shown in Figure 23, except the Home screen since there are no screens before the Home screen.

Figure 23 Navigation Bar in the iPhone Simulator

The gray bar at the top of the screen in the simulator is the navigation bar which is displayed by default in view controllers that use a navigation controller hierarchy. Notice the Back button. The navigation controller stores the hierarchy of the view controllers in the stack. Whenever a user taps on one of the buttons on the screen the storyboard segue calls the navigation controller's pushViewController: animated: method to present a new view controller and to push it to the top of the stack. When the Back button is tapped, the popViewController: animated: method is called and the view controller is removed from the stack and the previous view controller in the hierarchy is displayed on the screen.

To move to the first object in the hierarchy, our Home screen which serves as the root of the hierarchy, we need to use the popToRootViewControllerAnimated: method. We will do this in the next section.

KEY POINT

A Navigation Controller stores the information about the view controllers that are used in a app within a stack. One of the following methods can be used to navigate between these view controllers.

The popToRootViewController: animated: method is used to go back one view controller.

The pushViewController: animated: method is used to display a new view controller.t

The popToRootViewControllerAnimated: method is used to display the first view controller in the view hierarchy.

The storyboard's push segues call the pushViewController and popVIewController methods automatically so the developer doesn't need to provide any code for these transitions. However, to move to the root view controller the developer must provide some custom code

Write the Custom Code

In order to add the additional functionality to this app we need to add custom code to a another class. Navigate to File > New > New File... and select the UIViewController subclass template as shown in Figure 24. Click on the Next button.

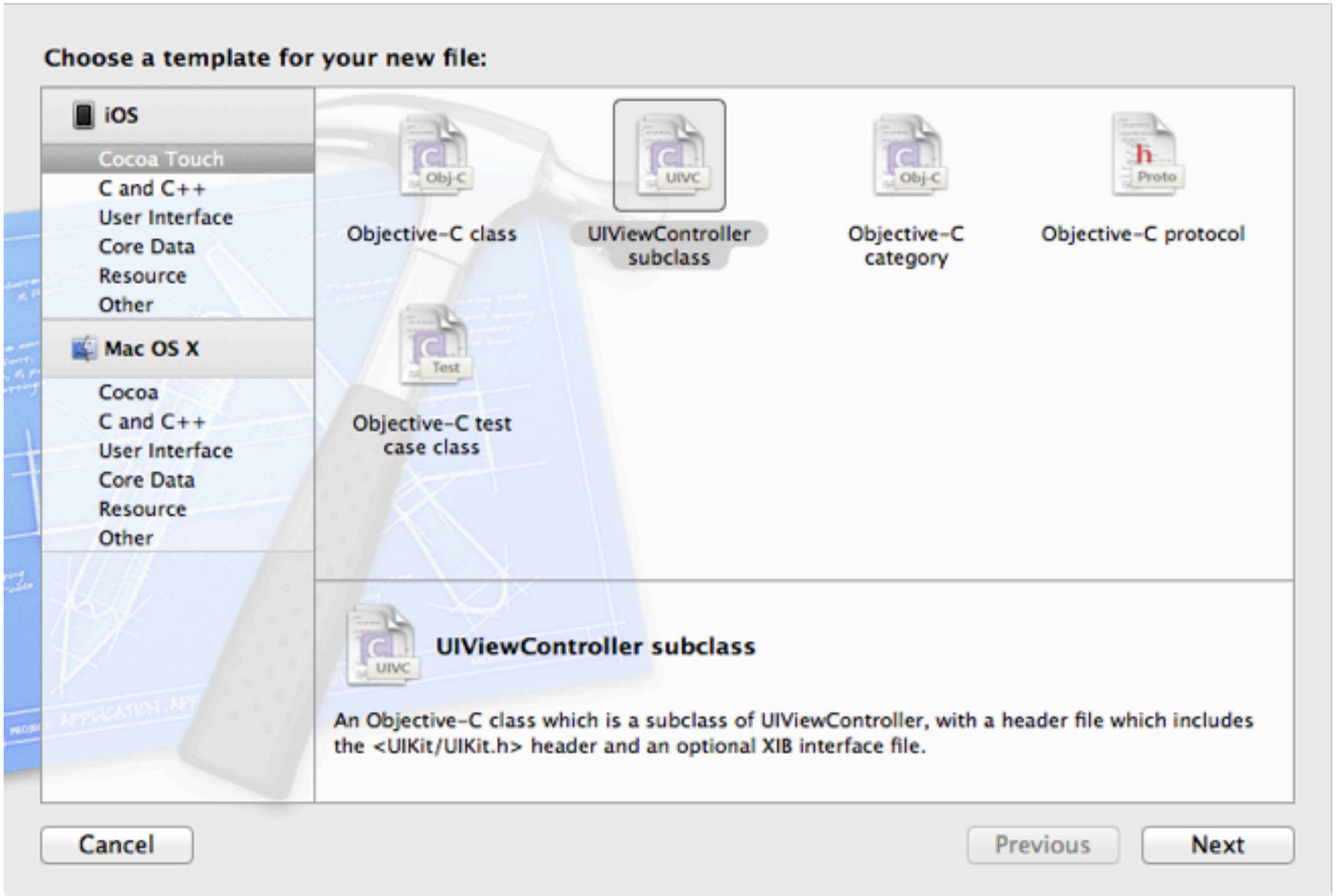

Figure 24 Adding a View Controller Subclass

Enter FinalViewController in the Class text box and make sure that the Targeted for iPad checkbox and the WIth XIB for user interface checkbox are not selected, as shown in Figure 25, then click on the Next button.

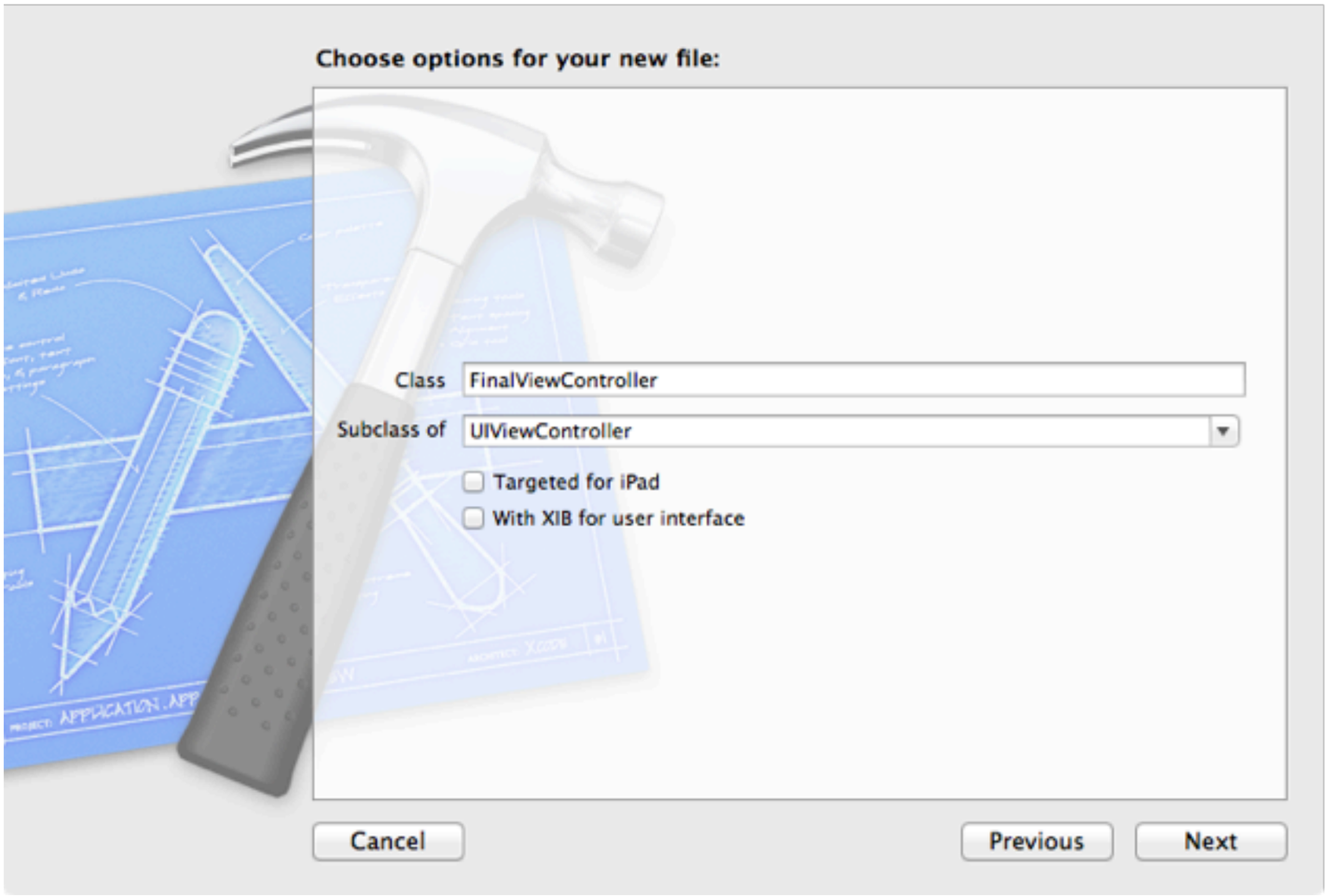

Figure 25 Adding a View Controller's Subclass - Specifying the Name of the Class

In the window that appears, select the location to save the class and then and click on the Create button. Two new files, FinalViewController.h and FinalViewController.m will appear in the Navigator area. Now, select the MainStoryboard.storyboard file in the navigator area to redisplay the storyboard in the editor area and then select the view controller for the Congratulations screen., and then. Navigate to View > Utilities > Show

Identity Inspector to open the Custom Class properties for this screen at the top of the Utilities area. Change the value of the Class property to the FinalViewController as shown in Figure 26 to change the identity of this view controller.

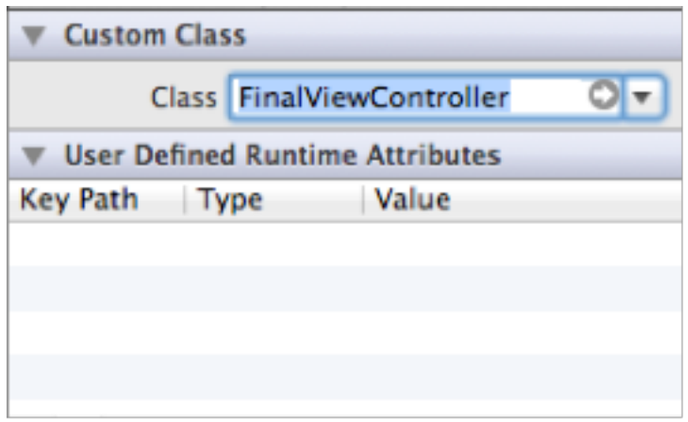

Figure 26 Specifying the Identity of the View Controller

Now we need to add custom code. Open the Assistant editor and make sure that the storyboard is displayed in one part of the editor area and the FinalViewController.h file is displayed in the other part of the editor area. Double click on the view controller for the Congratulations screen in the storyboard to zoom in on it. Ctrl-click on the Home button, then drag from the circle to the right of Touch Up Inside Sent Event to just above the end directive in the FinalViewController.h file as shown in Figure 27. Enter dismiss for the name of the action connection.

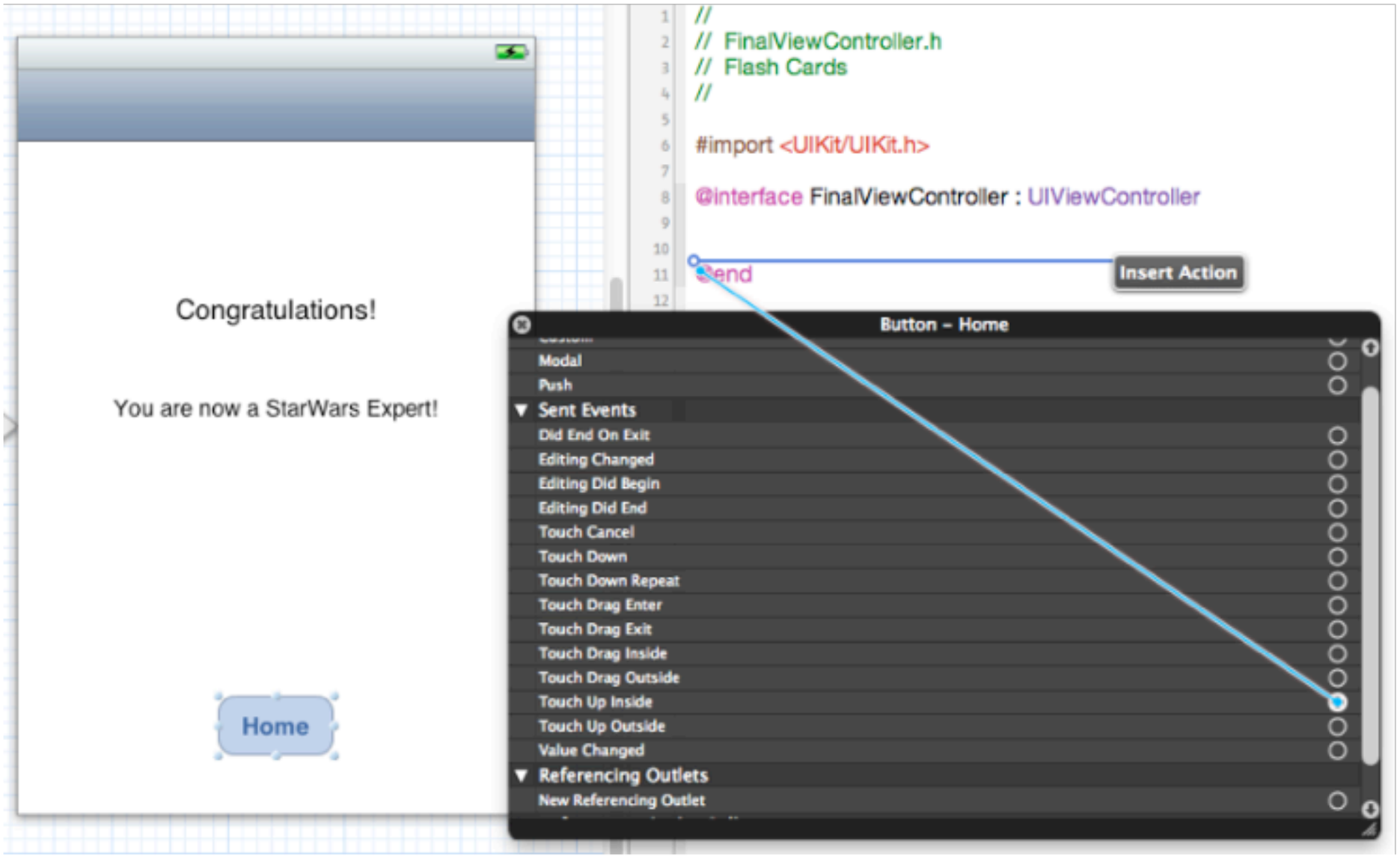

Figure 27 Creating a New Action Connection

Let's check the code. Look at your FinalViewController.h file - it should look similar to the following.

```
//
//  FinalViewController.h
//  Flash Cards
//

#import <UIKit/UIKit.h>

@interface FinalViewController : UIViewController

- (IBAction)dismiss:(id)sender;
@end
```

Look at your FinalViewController.m file and find the implementation for the dismiss method. Add the following line of code to the inside of this method.

```
 [self.navigationController popToRootViewControllerAnimated:YES];
```

The dismiss method should now appear as follows.

```
- (IBAction)dismiss:(id)sender {
    [self.navigationController popToRootViewControllerAnimated:YES];
}
```

Recall that the popToRootViewControllerAnimated method will display the root view controller, which in our case is our Home screen. The dismiss method will be called whenever user taps on the Home button on the Congratulations. Click on Run button to test your app. Congratulations! You are done!

Summary

In this chapter you have learned basic concepts regarding use of the storyboard and the navigation controller. Storyboards can be used to define navigation flow between view controllers in your application using segues. There are three kinds of segues, modal, push, and custom. The Modal segue is used to display some content modally, for a limited time. The Push segue must be used within the navigation controller to push or pop the view controllers. Finally The Custom segue can be used to define your own type of transition between view controllers. Segues can be also used to define data that is passed between the view controllers.

A Navigation Controller is used to manage hierarchical content. Currently displayed controllers are stored in a navigation stack. At the bottom of the stack is a root view controller and on the top of the stack is the view controller that is currently displayed. For all but the root view, the navigation controller provides a back button to allow the user to move back up the hierarchy. You can navigate between the view controllers in the navigation stack using navigation controller methods such as the (a) pushViewController: animated: to present a new view controller on top of an existing view controller, (b) popViewController: animated to display the previously displayed view controller--tapping on the navigation bar's back button provides the same behavior, and (c) popToRootViewControllerAnimated: to go back to the root view controller. One familiar example of the use of a navigation controller is the native iOS Settings app.

Review Questions

1. What is at the bottom of a navigation controller's hierarchy?

2. What is a segue? What kinds of segues are provided in storyboards?

3. Which kind of segue should be used to present content momentarily?

4. Which method of a navigation controller should be used to display a new view controller?

5. Which method of a navigation controller should be used to display the previous view controller in a hierarchy?

6. Which method of a navigation controller should be used to display the root view controller?

7. All but which view controller has a back button in the navigation bar?

8. When is the use of storyboards beneficial? Describe the overall process of using storyboards.

Exercises

1. Enhance the Flash Cards app so that it is more comprehensive regarding some topic of your choice. Improve the visual appeal in your app.

2. Write an app that presents some content modally, using a storyboard.