National University of Singapore
School of Computing
CS3217: Software Engineering on Modern Application Platforms
AY2010/2011, Semester 2

Problem Set 2: **Objective-C & Coding to Specifications**

Issue date: 17 January 2011
Due date: **23 January 2011**

Introduction

This is the first proper assignment where you start building a real application. The application that you will build over the next 5 weeks is a clone of the AngryBirds game. This is a puzzle game where the objective is to shoot projectiles to destroy a number of targets. The targets of often blocked by obstacles and you have to figure out how to get around them.

A screenshot of what your app might look like is attached at the Appendix. Note that while the grading of the problem sets will be done separately, you will eventually have to integrate all the parts together. You should put effort into ensuring that each problem set is fully debugged before you move on or you will run into a lot of problems when you try to integrate all the parts in a few weeks.

Reminder:	Please read the entire assignment before starting.

This assignment is organized as two parts. In the first part, you will learn how to create a basic interface for the app. In the second part, you will practice reading and interpreting specifications, and writing Objective-C code that satisfies our specifications. In addition, you will be given an introduction to using `checkRep` methods and testing strategies. You will implement three classes and link them with the code provided to complete the implementation of a graphing polynomial calculator. You will be required to answer some questions about both the code you are given and the code you need to write.

There are two zip files that come with this problem set:

- `images.zip`, which contains the image files you will need for the interface; and
- `RatPolyCalculator.zip`, a template project for the second part of the problem set.

Please make sure you have both of them.

Deploying Applications on your Development iPad

You should now have received your distributed iPad. It's time to try installing an application onto your iPad for testing. To do so, you must ensure your computer and iPad device is configured for iOS development. First, download "`cs3217 student profile.developerprofile`" from IVLE and double click the file to run it. When prompted by XCode Organizer, enter the password "`cs3217`" used to secure the Developer Profile. If successful, you will find the installed entries under Developer Profile and Provisioning Profiles in Xcode Organizer, see See Figure 1. (You can open this window in the `Window` drop-down menu of Xcode).

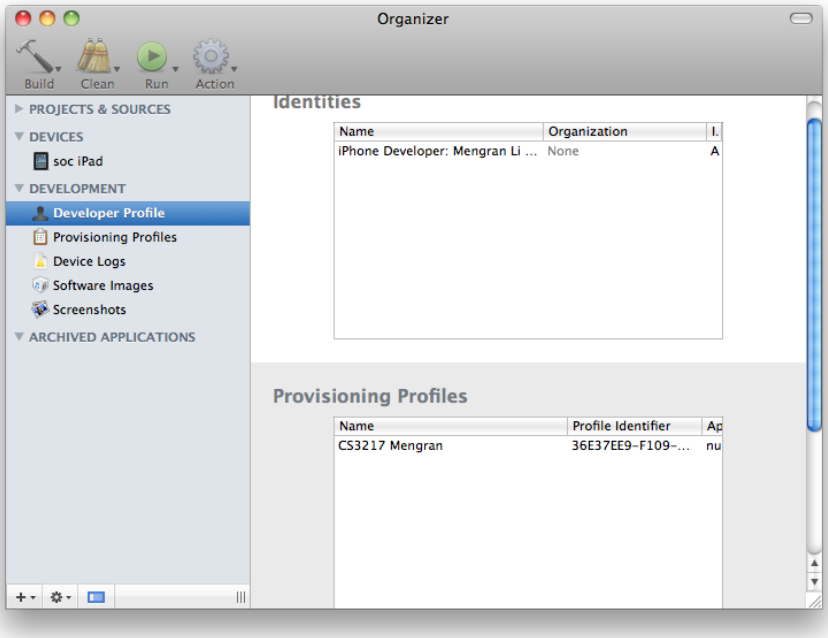

Figure 1: Certificate and profile.

Once that's done, let's try to install a very simple application, in fact, just a blank application. Create a view-based iPad application, named `DeviceTest`. While this blank view-based application can be run in the simulator, it needs to be signed with it can run on your device. Before your code can be signed, the application name needs to conform to what's specified in the provisioning profile. Expand the group of resources in the `Groups & Files` section your Xcode window. Look for a file named `DeviceTest-info.plist`. Open it and look for a field called "Bundle identifier", as shown in Figure 2. Change

```
com.yourcompany.${PRODUCT_NAME:rfc1034identifier}
```

to

```
nus.cs3217.${PRODUCT_NAME:rfc1034identifier}.
```

Note that the code identifier is case-sensitive.

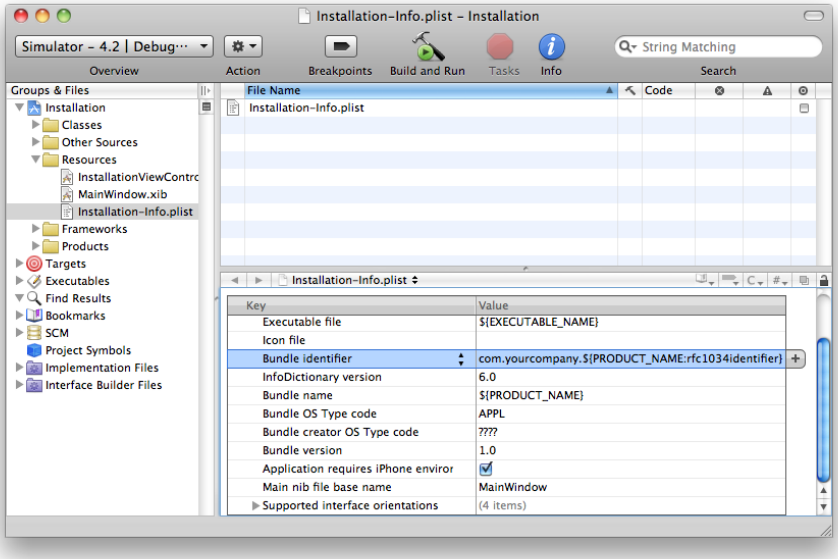

Figure 2: Bundle identifier.

Connect your iPad to your apple PC by the USB cable. If this is the first time you are configuring your iPad, click yes when prompted to set up the device for development.

Change the active SDK in Xcode from simulator to device. by going to `Project` ⟶ `Set Active SDK` ⟶ `Device`.

Click `Build and Run` now, you should see the blank application loaded up in your iPad and running. Now that you are able to successfully deploy an application on the iPad, you can do the same for your future applications by applying the similar steps.

Part 1: Interface (5 points)

In this section, you will follow a detailed walkthrough on how to create a basic interface to your game.

1. Follow the same steps described in the last problem set and create a view-based ipad application called ”Game” (or whatever else you desire).

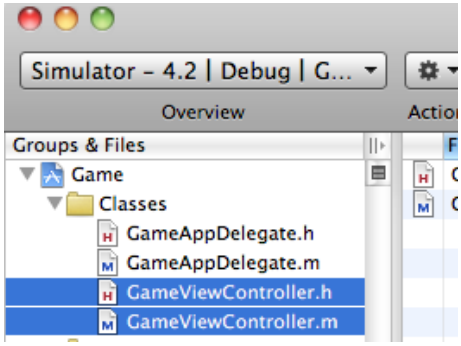

Figure 3: Controller class.

In the Xcode IDE, notice the highlighted `GameViewController.h` and `GameViewController.m` in the classes group, as shown in Figure 3. These are the interface and implementation of the controller class of our application. Now open `GameViewController.h`. Add one **IBOutlet** and one **IBAction** handler as shown below.

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

@interface GameViewController : UIViewController {
      IBOutlet UIScrollView *gamearea;
}

- (IBAction)buttonPressed:(UIButton *)sender;

@end
```

The **IBOutlet** and **IBAction** keywords do nothing but to inform the interface builder to pay attention to the variables or methods they prefix. So our `buttonPressed` method’s return type is just void.

An **IBOutlet** is a reference to an object through which the controller does its output. In this case, we are going to draw to a scrollview later. We’ll use the **IBOutlet** gamearea to reference it.

An **IBAction** method handles an event generated by user interaction with the view. In this case, we are going to handle button press events in the buttonPressed method.

2. Expand the list of resources. Notice the highlighted `GameViewController.xib`, as shown in Figure 4, this is the file that stores the interface of your application. It is possible to create an interface programmatically, instead of drawing in the interface builder, but that is not in the scope of this problem set. You can learn how to do this on your own later.

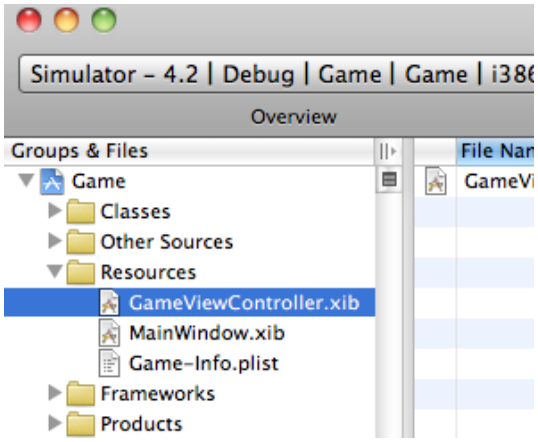

Figure 4: Selecting the interface.

3. Now it's the time to draw our interface. Double click on `GameViewController.xib`. This will start the interface builder. Go to the `Interface Builder` menu and click `Hide Others`, as shown in Figure 5. This will hide windows opened from other applications and give you a clear view. Now notice the components of the interface builder.

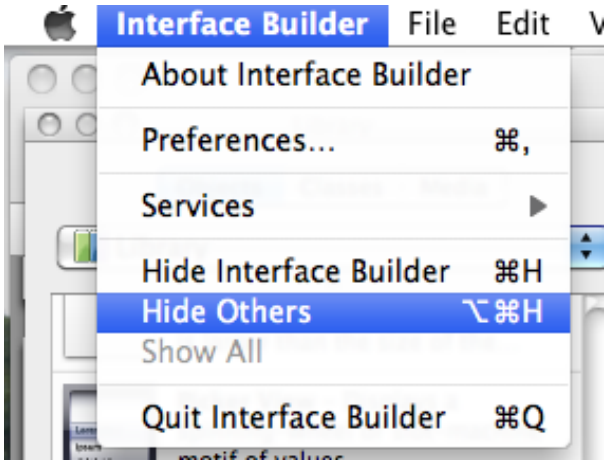

Figure 5: Hide others.

There is a window titled `View`, as shown in Figure 6. This is where we draw our interface in a WYSIWYG manner. It is currently empty.

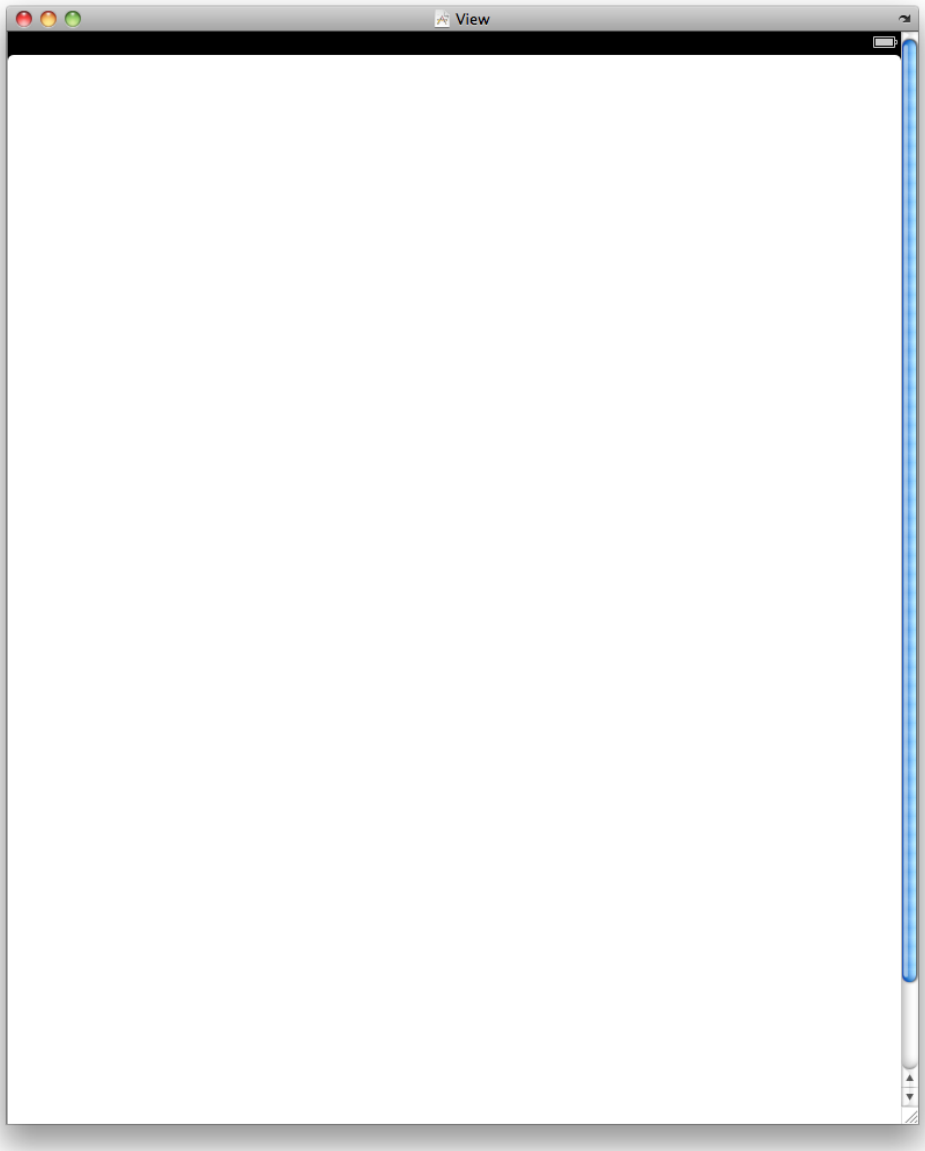

Figure 6: Empty view window.

There is a library window as shown in Figure 7. It contains all the basic building blocks of a view. We can drag and drop them into our view.

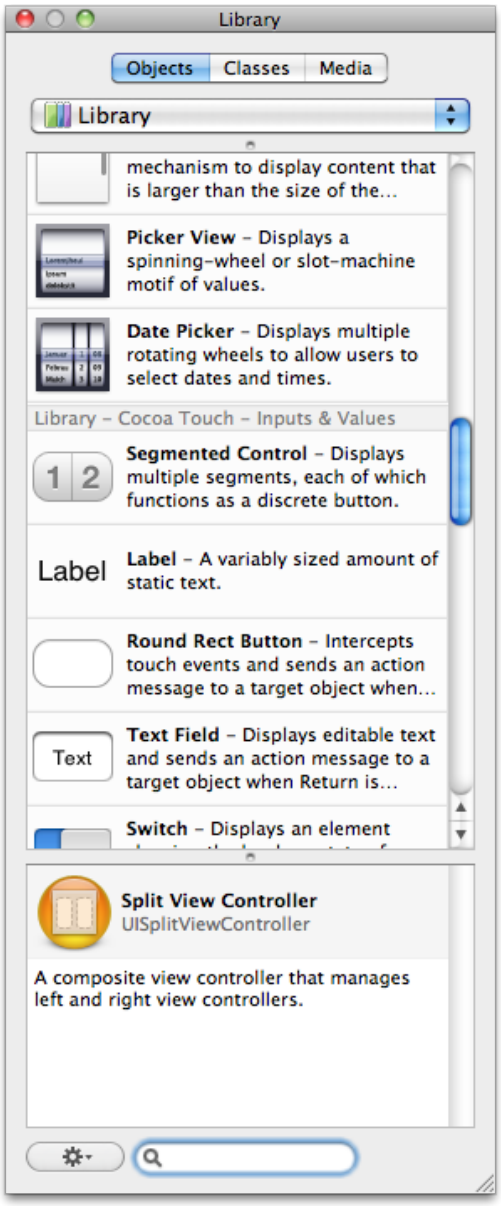

Figure 7: Library window.

There is a `main window` that shows all of the objects in our `GameViewController.xib`, as shown in Figure 8. The `File's Owner` is our controller class. This is where we wire things up to our controller's **IBOutlet** and **IBAction**. The `View` is our top level UIView in our view hierarchy. All UIView objects are arranged in a hierarchy in which each has a superview and each may have any number of subviews. the `View` represents the superview of all these views.

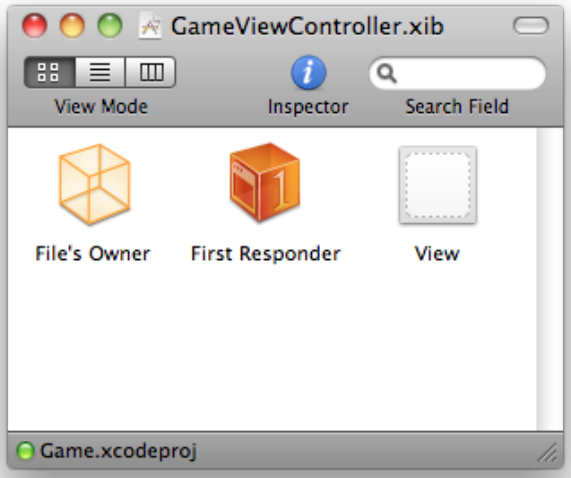

Figure 8: Interface builder main window.

There is an inspector window that shows an object's attributes, and allows us to modify them, as shown in Figure 9. Since no object is selected now, the inspector window is empty.

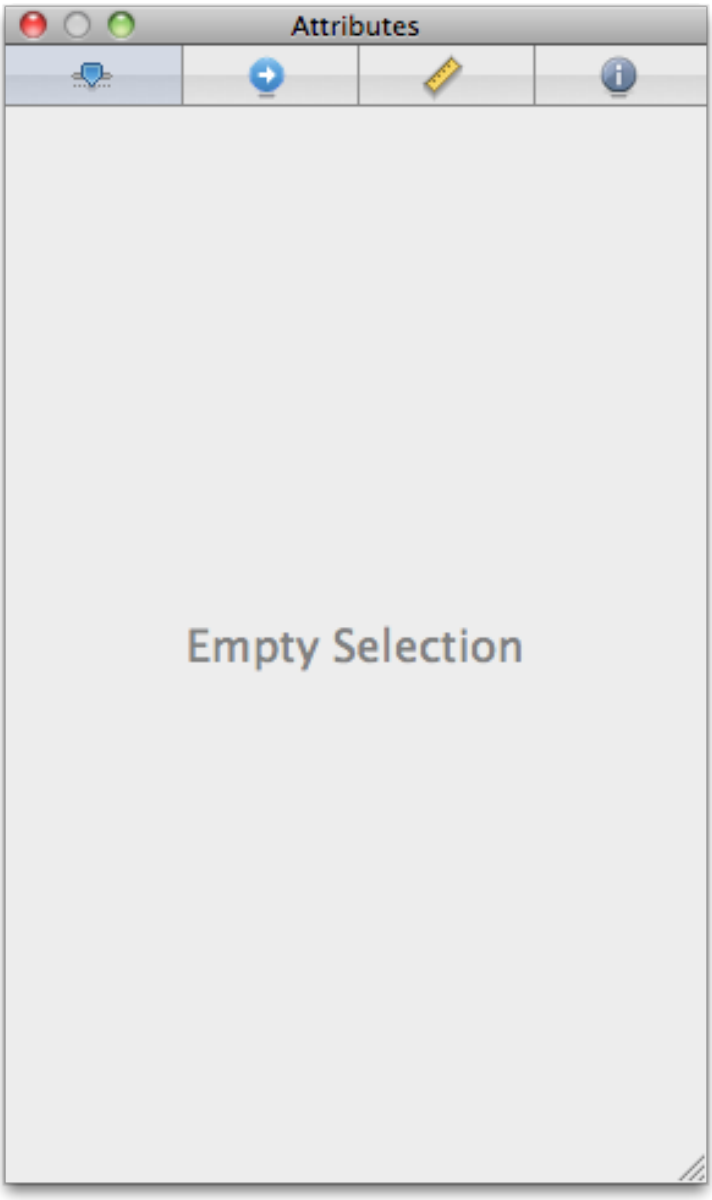

Figure 9: Inspector.

4. Now let's look at how we can change an object's attributes. Click on the `View` in the `main window`. This will select the top level view. Notice that the inspector window is now showing the attributes of the top level view, as shown in Figure 10. Change the orientation from `portrait` to `landscape`. Notice that the view window is automatically updated to reflect this change.

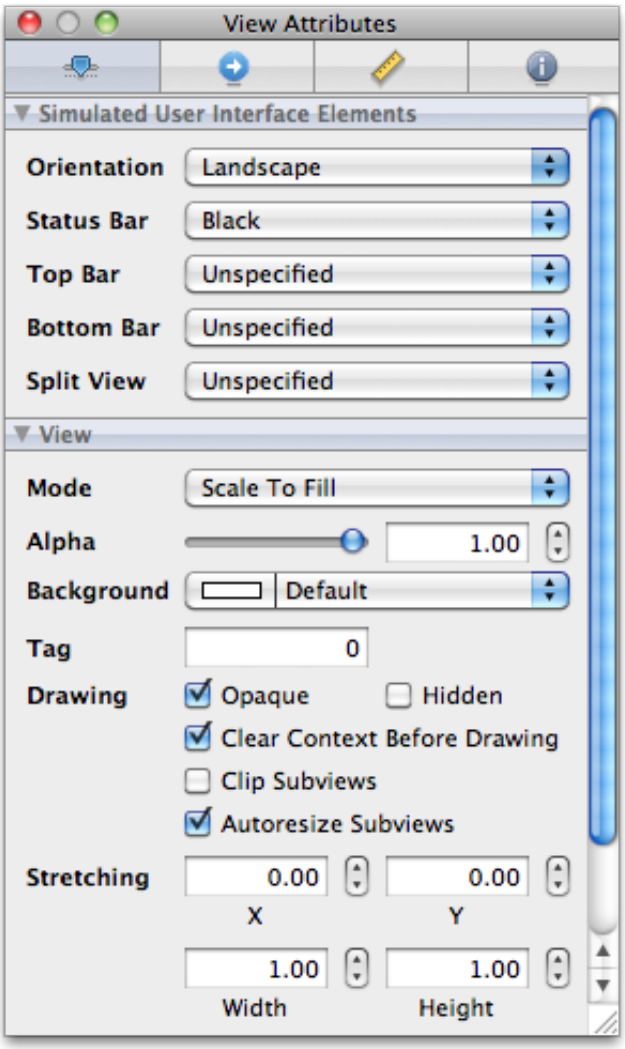

Figure 10: Toplevel inspector.

5. Drag and drop a `Round Rect Button` from the library window to our view.

6. Now we wire the button press event from this button to our **IBAction** handler `buttonPressed`. Click on the button to select it. Then, while holding the control key, drag the button to `File's Owner` in the `main window`. Notice the line protruding from the button as we drag it, as shown in Figure 11.

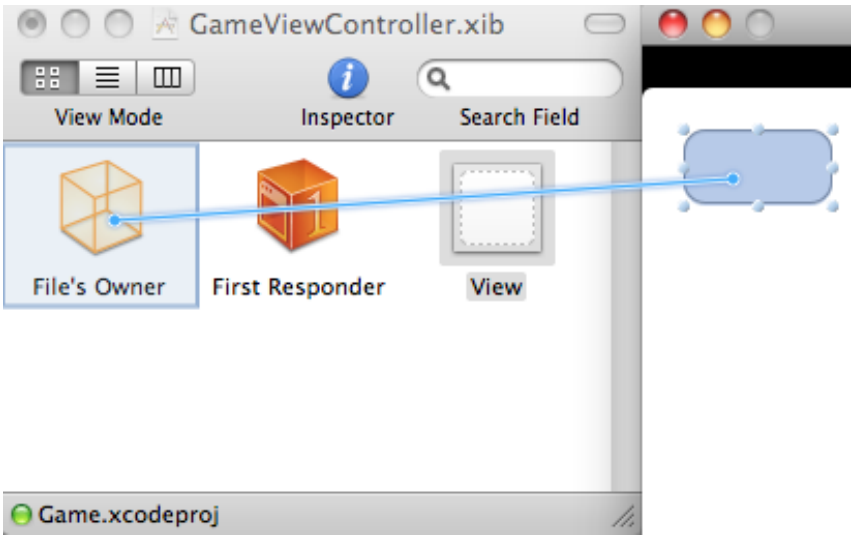

Figure 11: Wiring button 1.

After dragging, a small window will popup, showing the list of event handlers that we have already declared in the controller class. At the moment, there is only one, `buttonPressed`, as shown in Figure 12. Click on it. There we go. The button is successfully wired to `buttonPressed`.

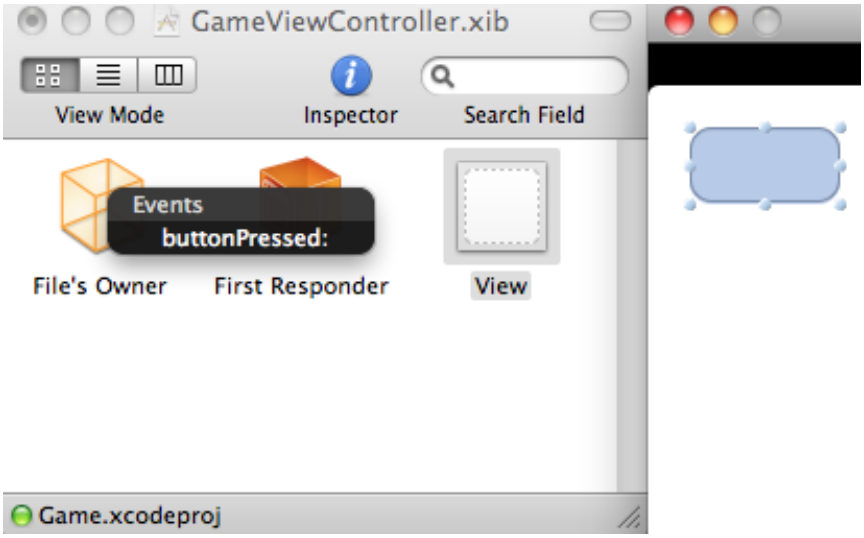

Figure 12: Wiring button 2.

7. Now copy and paste this button to create three more buttons. Note that when we copy a button, we also copy its wirings. That means, the button press events generated by the three new buttons are also handled by `buttonPressed`. Align these buttons nicely at the top of our view.

8. Time to add labels to our buttons. Click on the leftmost button. Go to the inspector window and change the title of the button to **SAVE**, as shown in Figure 13. Add the titles **LOAD**, **START** and **RESET** to the other buttons.

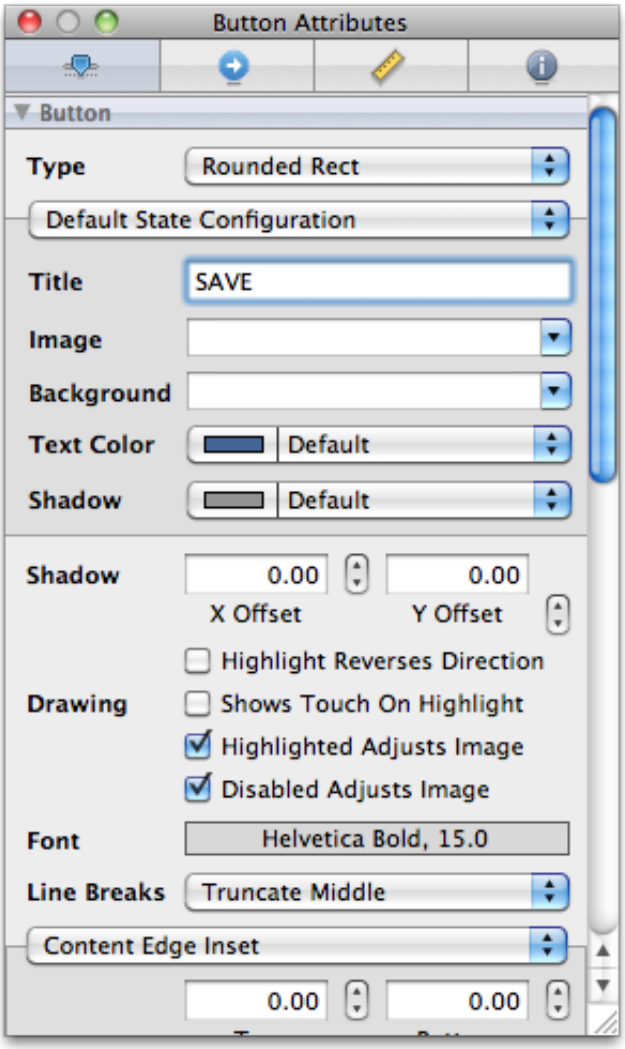

Figure 13: Button title.

9. Drag and drop a `Scroll View` to our view. Stretch it to cover the entire space below the buttons. Next we need to wire it to our **IBOutlet** gamearea. Remember how we dragged the button to `File's Owner`? This time we need to do it the other way around. Drag `File's Owner` to the new scroll view added, and a small window appears. It contains all the **IBOutlet** declared in our controller class. Currently there is only one, `gamearea`, as shown in Figure 14. Click on `gamearea`. Now the scroll view is wired.

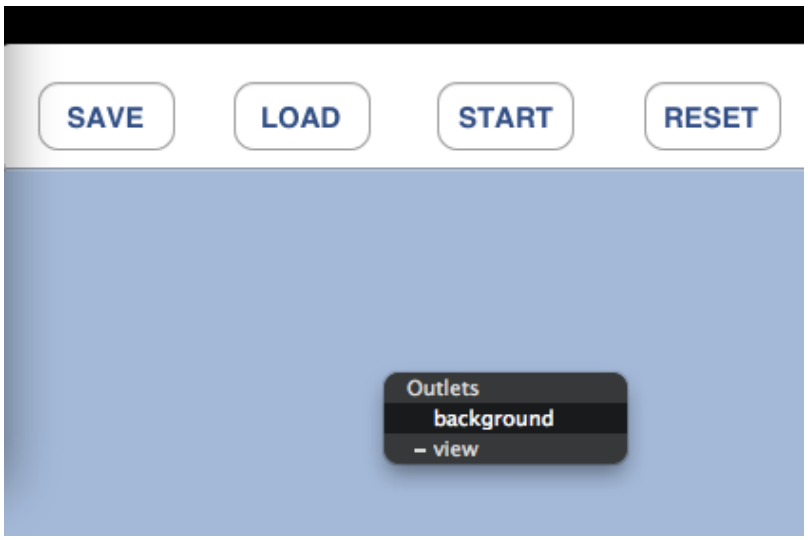

Figure 14: Wire scrollview.

10. Click on the added scroll. Uncheck `Vertical` and `Bounce Scroll` in the inspector window, as shown in Figure 15.

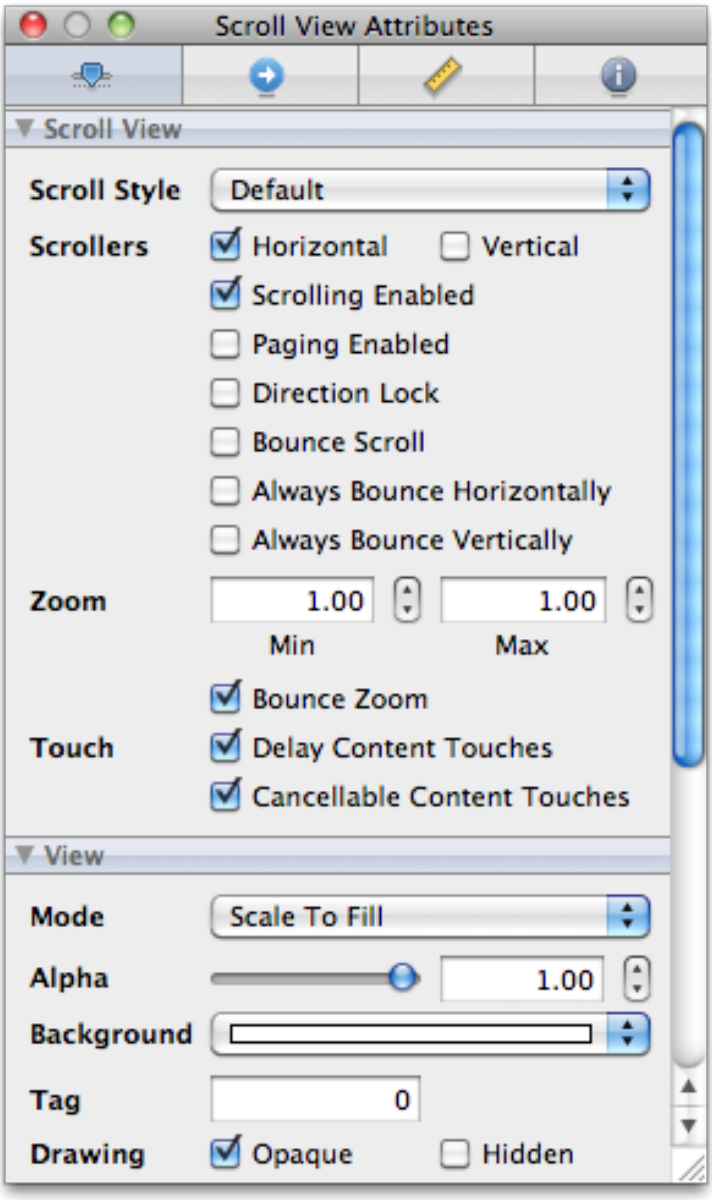

Figure 15: Scrollview attributes.

11. We are done with the interface building for now. You should now have an interface like this Figure 16.

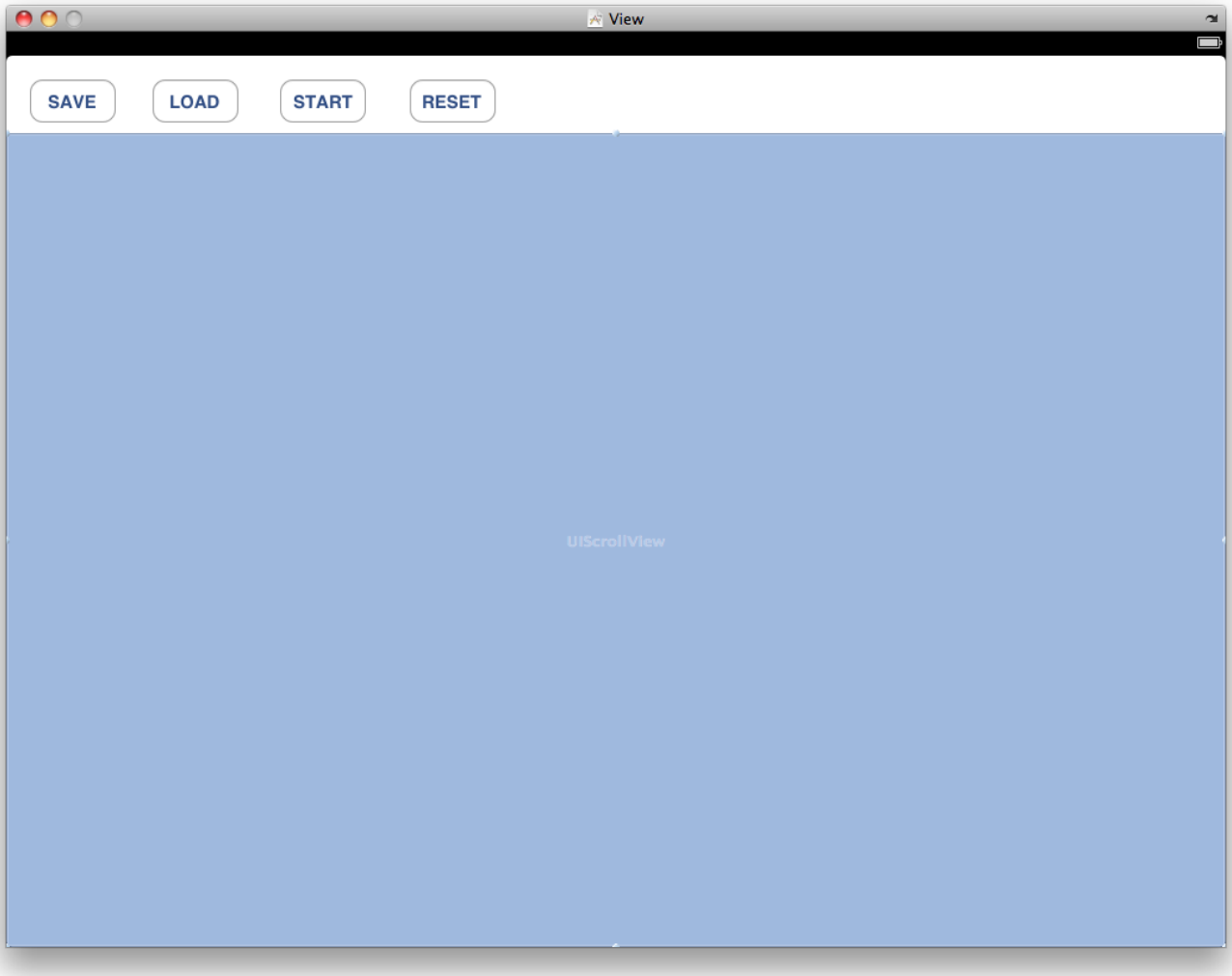

Figure 16: First interface.

12. Save the interface and quit the interface builder.

13. Now, build and run the application. You will see something like Figure 17.

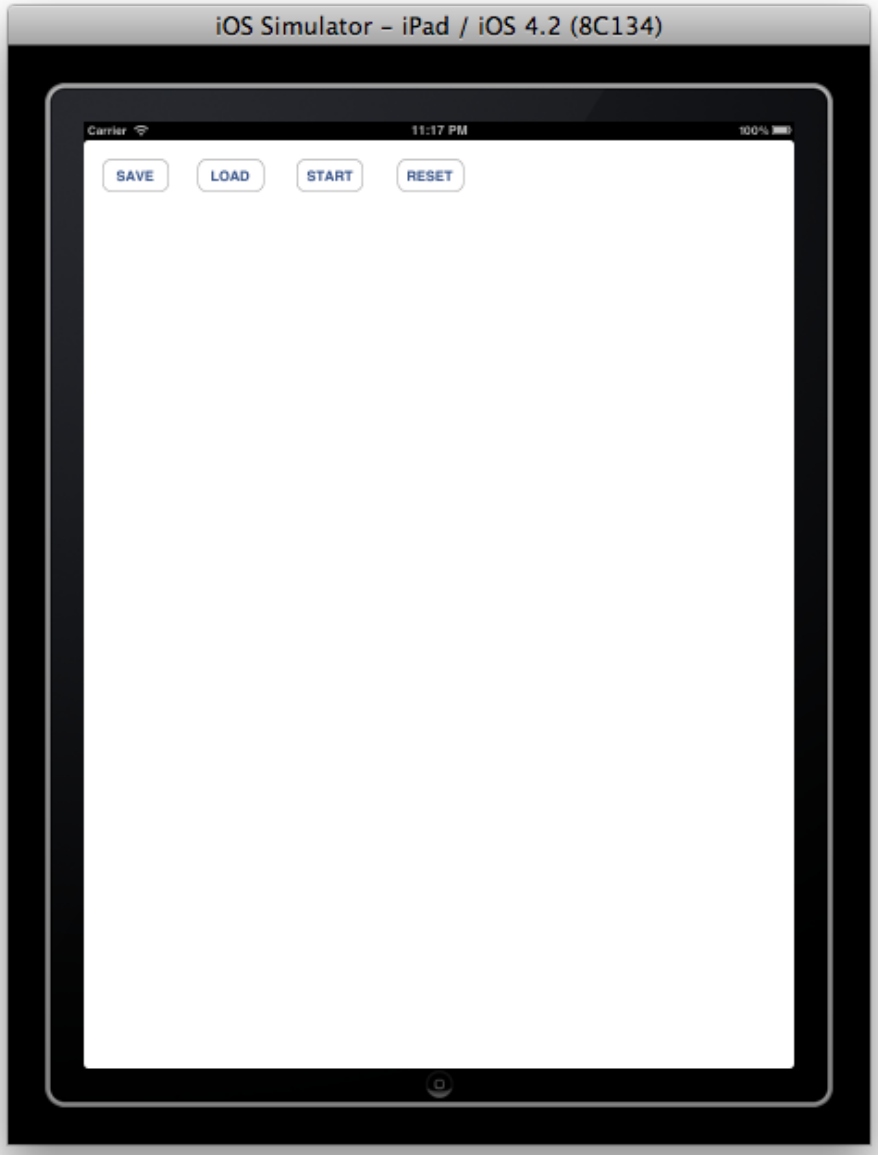

Figure 17: First simulator run.

By default, the simulator runs in portrait orientation. How do we change that? Go to the `Hardware` menu and click `Rotate Left`, as shown in Figure 18.

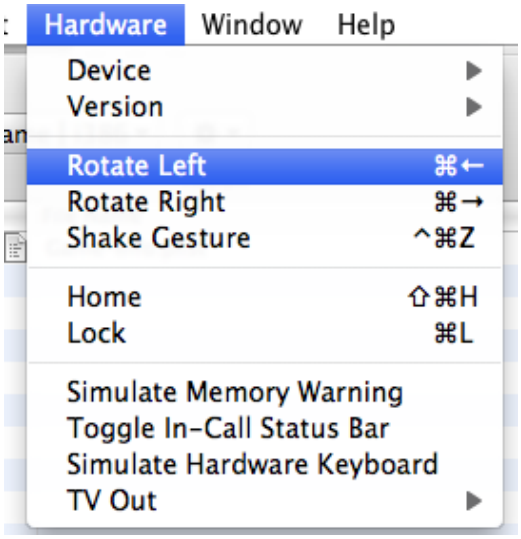

Figure 18: Rotating the simulator.

14. Currently, the button presses won't generate any responses. The reason is that we have not written any. Let's do something simple. Copy the following implementation of `buttonPressed` into `GameViewController.m`.

```
- (IBAction)buttonPressed:(UIButton *)sender
{
NSString *buttonTitle = [[sender titleLabel] text];
NSLog([buttonTitle stringByAppendingString:@" button pressed."]);
}
```

This code will print a message to the console indicating which button is pressed. Build and run the program, and test the buttons. You can open the console by pressing `shift`, `command` and `R` simultaneously.

15. The gamearea is rather dull now, isn't it? Let's place a picture in it. But before that, we need to add the picture into our project. Right click `Game` at the top of the `Groups & Files` section and add a new group called `Game Art`, as shown in Figure 19.

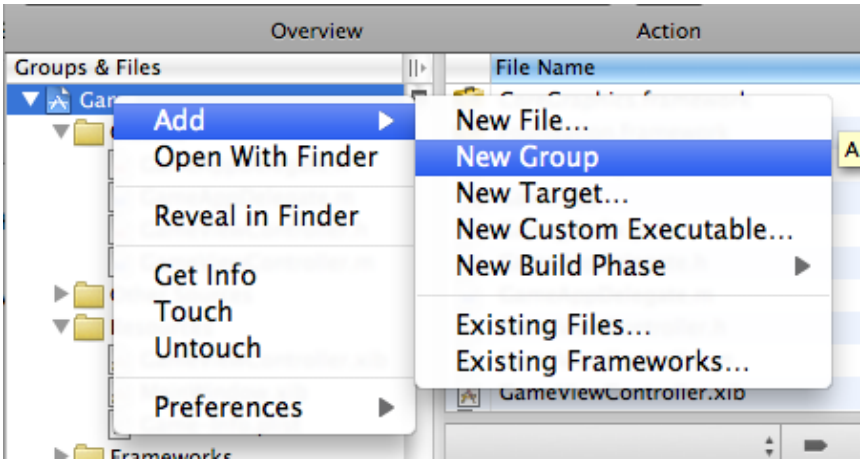

Figure 19: Add game art group.

16. Extract the images from `images.zip` to your project directory. Right click the newly added `Game Art` group and choose to add existing files, as shown in Figure 20. Browse to the image files, select them and click `add`.

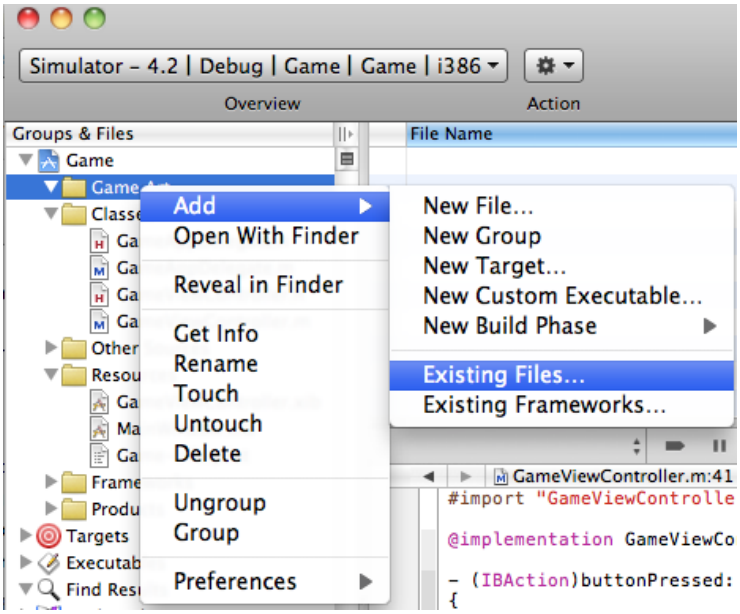

Figure 20: Add game art.

Another window is opened, as shown in Figure 21. Check the `Copy items into destination group's folder (if needed)` option, and click `add`.

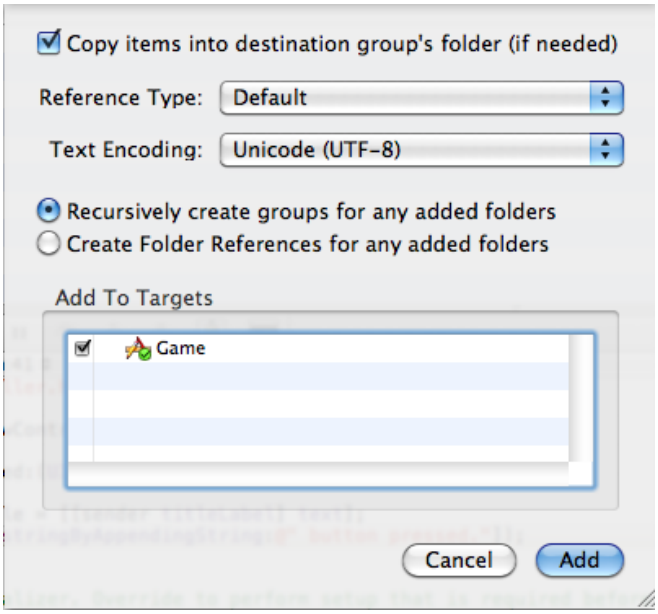

Figure 21: Add game art (part two).

17. Alright. Now let's load up images to make our gamearea look prettier. We will place one image at the bottom of the scroll view, called "ground". This represents the ground level in our game world. Then we will place another image on top of it, which serves as the background of our game world.

Open the controller's implementation. Declare these constants below at the start of it to hold the heights and widths of the two images.

```
const CGFloat backgroundWidth = 1600.0;
const CGFloat backgroundHeight = 480.0;
const CGFloat groundWidth = 1600;
const CGFloat groundHeight = 100.0;
```

Look for a method called **viewDidLoad**, which is commented out now. This method is called when the view is loaded. We can do some further customization of the view here. Uncomment it and complete it with the code below. We basically put each picture in an UIImageView, and then add the two UIImageViews as subviews of gamearea.

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

   //load the images into UIImage objects
   UIImage *bgImage = [UIImage imageNamed:@"background.png"];
   UIImage *groundImage = [UIImage imageNamed:@"ground.png"];

   //place each of them in an UIImageView
   UIImageView *background = [[UIImageView alloc] initWithImage:bgImage];
   UIImageView *ground = [[UIImageView alloc] initWithImage:groundImage];
```

```
    CGFloat groundY = gamearea.frame.size.height - groundHeight;
    CGFloat backgroundY = groundY - backgroundHeight;

    //the frame property holds the position and size of the views
    //the CGRectMake method's arguments are : x position, y position, width,
    //height
    background.frame =
          CGRectMake(0, backgroundY, backgroundWidth, backgroundHeight);
    ground.frame = CGRectMake(0, groundY, groundWidth, groundHeight);

    //add these views as subviews of the gamearea. gamearea will retain them
    [gamearea addSubview:background];
    [gamearea addSubview:ground];

    //set the content size so that gamearea is scrollable
    //otherwise it defaults to the current window size
    CGFloat gameareaHeight = backgroundHeight + groundHeight;
    CGFloat gameareaWidth = backgroundWidth;
    [gamearea setContentSize:CGSizeMake(gameareaWidth, gameareaHeight)];

    //remember to release these objects,
    //because gamearea already retains them
    [background release];
    [ground release];
}
```

Let's look at our new interface. Build and run the program. If you have done everything correctly, you should see something like the screenshot in Figure 22.

Figure 22: A better interface.

Now that you understand the basics of how to create a simple interface, feel free to modify the interface with the interface builder to improve it. Please submit a screenshot of the most creative, yet functional interface that you could can up with. Note that a screenshot of your final interface is the only deliverable part one of this problem set.

Part 2: Interpreting and Implementing Specifications

Problem 1: Polynomial Arithmetic Algorithm (15 points)

In this section, you will practice interpreting and implementing specifications, by working with three immutable ADT (Abstract Data Type) classes, `RatNum`, `RatTerm` and `RatPoly`, representing rational numbers, terms in a single-variable polynomial expression and polynomial expressions respectively. The code for these classes can be found in `RatPolyCalculator/Classes`.

First, let's discuss the algorithms for arithmetic operations on single-variate polynomials. To begin, we provide the following pseudocode for polynomial addition:

```
r = p + q:
     set r = q by making a term-by-term copy of all terms in q to r
     foreach term, t_p, in p:
         if any term, t_r, in r has the same degree as t_p,
             then replace t_r in r with the sum of t_p and t_r
             else insert t_p into r as a new term
```

You may use arithmetic operations on terms of polynomial as primitives. For the above example, the algorithm uses addition on the terms t_p and t_r. Furthermore, after defining an algorithm, you may use it to define other algorithms. For example, if helpful, you may use polynomial addition within your algorithms for subtraction, multiplication, and division.

Answer the following questions:

a. Write a pseudocode algorithm for subtraction. *(5 points)*

b. Write a pseudocode algorithm for multiplication. *(5 points)*

c. Write a pseudocode algorithm for division. The following is the definition of polynomial division as provided in the specification of RatPoly's `div` method: *(5 points)*

```
Division of polynomials is defined as follows:

Given two polynomials u and v, with v != "0",
we can divide u by v to obtain a quotient polynomial q and
a remainder polynomial r satisfying the condition u = "q * v + r",
where the degree of r is strictly less than the degree of v, the
degree of q is no greater than the degree of u, and r and q have
no negative exponents.

For the purposes of this class, the operation "u / v" returns q as
defined above.

The following are examples of div's behavior:

(x^3-2*x+3) / (3*x^2) = 1/3*x (with r = "-2*x+3").
(x^2+2*x+15) / (2*x^3) = 0 (with r = "x^2+2*x+15").
(x^3+x-1) / (x+1) = x^2-x+2 (with r = "-3").
Note that this truncating behavior is similar to the behavior of
integer division on computers.
```

Record your answers at the end of the file `RatPoly.m` as comments.

Problem 2: Rational Number Class (25 points)

The purpose of a specification is to document a program's logic, the range of input that it can safely operate on, and its expected behavior. So for ADTs, the specification describes what this ADT represents, its purpose, its representation invariant, and the behavior of each of its method. At the start of the ADT, the specification gives a summary of the ADT, mainly the abstraction function and the representation invariant. The abstraction function is a mapping from the state of an ADT instance to some mathematical object that it represents. The representation invariant is some condition that all valid instances of the ADT must satisfy, which is essentially the format of the ADT. Before each method, there are comments describing its behavior. Here we are simply describing its expected input and output. The "requires" clause state the conditions that the input must satisfy. When calling this method, we must make sure all the conditions in the "requires" clause must be met. When implementing this method, we can safely assume that the input satisfies the "requires" clause. The "returns" clause dictates the relationship between the expected output and the input.

Now let's take a look at the first ADT, `RatNum`. Unzip the file `RatPolyCalculator.zip` that comes with this assignment. Double click `RatPolyCalculator.xcodeproj` to open the project template. Read the specifications for `RatNum` in `RatNum.h`, a class representing rational numbers. Then read over the staff-provided implementation, `RatNum.m`.

Answer the following questions, and append your answers at the end of the file `RatPoly.m` as comments:

a. `add:`, `sub:`, `mul:`, and `div:` all require that "`arg != nil`". This is because all of the methods access fields of 'arg' without checking if 'arg' is null first. Why do we require `self` to be non-null? What happens when we pass a message to a nil pointer? `div` checks whether its argument is NaN (not-a-number). `add` and `mul` do not do that. Explain. *(5 points)*

b. Why is `valueOf` a class method? Is there an alternative to class methods would allow one to accomplish the same goal of generating a `RatNum` from an input String? *(5 points)*

c. Suppose the representation invariant were weakened so that we did not require that the numer and denom fields be stored in reduced form. This means that the method implementations can no longer assume this invariant held on entry to the method, but they also no longer were required to enforce the invariant on exit. The new rep invariant would then be:

   ```
   // Representation Invariant for every RatNum r:  ( r.denom ≥ 0 )
   ```

   Which method or constructor implementations would have to change? Please list them. For each changed piece of code, describe the changes informally, and indicate how much more or less complex (in terms of code clarity and/or execution efficiency) the result would be. Note that the new implementations must still adhere to the given specifications; in particular, `stringValue` needs to output fractions in reduced form. *(5 points)*

d. Calls to `checkRep` are supposed to catch violations in the classes' invariants. In general, it is recommended that one call checkRep at the beginning and end of every method. In the case of `RatNum`, why is it sufficient to call checkRep only at the end of the constructors? (Hint: could a method ever modify a `RatNum` such that it violates its representation invariant? Could a method change a `RatNum` at all? How are changes to instances of `RatNum` prevented?) *(5 points)*

It is a very good practice to write unit tests for your ADTs. They help you make sure you don't break your program when you make changes. We've built a unit test for `RatNum`. You can run it by setting

`RatNumTests` as the active target, by going to "Project" → "Set Active Target" → "RatNumTests", and then building this target, instead of "Build and Run". If the unit tests pass, you will see the lovely green ticks in your "Build Results" window, as shown in Figure 23.

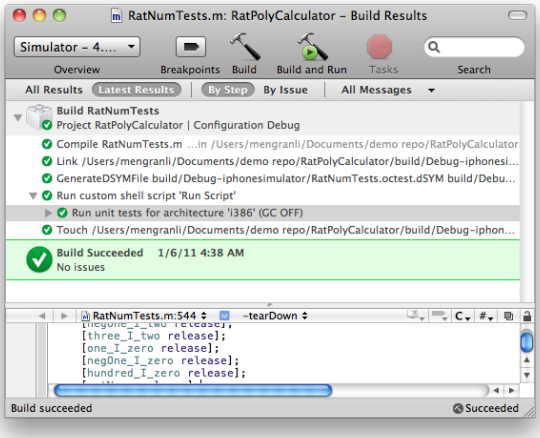

Figure 23: test success

Otherwise you will see alarming red exclamation marks. If you expand the list of unit tests, you can see exactly which test went wrong. When you click on the test case you fail, the point where you fail will be highlighted in the bottom part of the "Build Results" window, as shown in Figure 24.

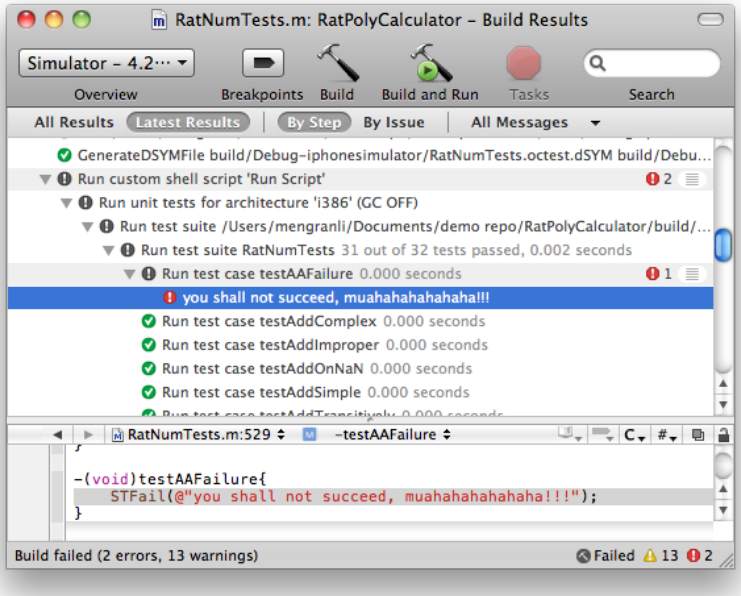

Figure 24: test failure

Problem 3: Rational Term Class (90 points)

Read over the specifications for the `RatTerm` class. Make sure that you understand the overview for `RatTerm` and the specifications for the given methods.

Fill in an implementation for the methods in the specification of `RatTerm`, according to the specifications. You may define new helper methods in `RatTerm.m` if you need them. You may not add public methods; the external interface must remain the same. *(70 points)*

We have provided a `checkRep` method in `RatTerm` that to help you test whether or not a `RatTerm` instance violates the representation invariants. We highly recommend you use `checkRep` in the code you write. Think about the issues discussed in the 2(e) when deciding where checkRep should be called.

We've build a unit test for `RatTerm` as well. Set "RatTermTests" as the active target and run the test. Make sure your implementation passes all the tests.

Answer the following questions, appending your answers at the end of the file `RatPoly.m` as comments:

a. Where did you include calls to `checkRep` (at the beginning of methods, the end of methods, the beginning of constructors, the end of constructors, some combination)? Why? *(5 points)*

b. Imagine that the representation invariant was weakened so that we did not require `RatTerm`'s with zero coefficients to have zero exponents. This means that the method implementations can no longer assume this invariant held on entry to the method, but they also no longer were required to enforce the invariant on exit. Which method or constructor implementations would have to change? Please list them. For each changed piece of code, describe the changes informally, and indicate how much more or less complex (in terms of code clarity and/or execution efficiency) the result would be. Note that the new implementations must still adhere to the given spec; in particular, `stringValue` still cannot produce a term with a zero coefficient (excluding 0). *(5 points)*

c. In the case of the zero `RatTerm`, we require all instances to have the same exponent (0). No such restriction was placed on NaN `RatTerm`'s. Imagine that such a restriction was enforced by changing the representation invariant to include the requirement:

   `[self.coeff isNaN]` $\longrightarrow$ `expt = 0`.

   This means that the method implementations can assume this invariant held on entry to the method, but they would also be required to enforce the invariant on exit. Which method or constructor implementations would have to change? Please list them. For each changed piece of code, describe the changes informally, and indicate how much more or less complex (in terms of code clarity and/or execution efficiency) the result would be. Note that the new implementations must still adhere to the given spec (except for the part where terms like `NaN*x^74` are explicitly allowed). *(5 points)*

d. Which set of `RatTerm` invariants ([self.coeff isNaN] $\longrightarrow$ expt = 0; [self.coeff isEqual:[RatNum initZERO]] $\longrightarrow$ expt = 0; both; neither) do you prefer? Why? *(5 points)*

Problem 4: Rational Polynomial Class (75 points)

Follow the same procedure given in Problem 3, and read over the specifications for the `RatPoly` class and its methods, and then fill in the implementation for `RatPoly`. The same rules apply here

(you may add private helper methods as you like). Since this problem depends on problem 3, you should not begin it until you have completed problem 3.

You should have noticed by now, if you have looked at the unit test files, that they are made exactly to test for the specifications. We ask that you to write a similar unit test for `RatPoly`. Do note that it's not a time-wasting practice! It gives you peace of mind by minimizing your bugs before submission. :) The skeleton files `RatPolyTests.h` and `RatPolyTests.m` have already been created for you. You can run unit tests for `RatPoly` by compiling the target "RatPolyTests".

Problem 5 [Bonus Question]: Reflection

Please answer the following questions:

a. How many hours did you spend on each problem of this problem set?

b. In retrospect, what could you have done better to reduce the time you spent solving this problem set?

c. What could the CS3217 teaching staff have done better to improve your learning experience in this problem set?

Submit the answer to this question in the form of a comment appended at the end of `RatPoly.m`. *(3 bonus points)*

Integration with PolyCalculator GUI

Once you finish implementing all the ADTs, you can test to see if the implementation works with the calculator. This part is not graded. Build and run your application. You should see this interface in the simulator. You can input two polynomials in the text fields, and press the + - * / buttons to perform operations on them. The result is displayed in the text field at the bottom.

Grading Scheme

The simplest way to ensure that you get a good grade on your assignment is to simply go through the list of testing items. Most of the reasons we mark an assignment down could very easily be avoided by simply taking a few minutes to go through each of the testing points before submitting. For this assignment, testing/grading will be done by running the project in the simulator. We will be looking out for the following:

- Your submission should adhere to the submission format.
- Your interface must be neat.
- You have answered the questions correctly in a concise manner.
- Your project should build without errors or warnings.
- Your project should run without crashing.
- Your unit test cases are well-designed.

Mode of Submission

Please submit a single `.zip` archive file, uploaded to the appropriate folder in the IVLE workbin. The zip file should unzip into a directory called `RatPolyCalculator` that contains all the source files, so that your TA can run your code and unit tests. `RatPolyCalculator` should also contain the screenshot of the interface that you built in part one of this problem set. The submitted zip file should adhere to the following naming convention: `CS3217-PS2-[YourName].zip`.

Clarifications and questions related to this assignment may be directed to the IVLE Forum under the header 'Problem Set 2: Objective-C & Coding to Specifications'.

Good luck and have fun!

Figure 25: Polynomial Calculator interface.

Screenshots of Sample App - Huff & Puff

Figure 26: Screenshot of Huff and Puff.