National University of Singapore
School of Computing
CS3217: Software Engineering on Modern Application Platforms
AY2011/2012, Semester 2

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

Issue date: 14 January 2012
Due date: **21 January 2012 23:59**

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:

- `ps02-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 by following the steps below:

- Start XCode and open Organizer from `Window` menu.
- Plug-in the Device.
-
  - If there is no certificate created, XCode will ask you if you want XCode to submit certificate request automatically. Say YES. Request will show online on developer portal and has to be approved by an admin user. If XCode does not prompt you to request a certificate, click `Refresh` in `Provisioning Profiles` under `Library`.
  - if this is the first time using the device for development, select your device in Organizer and press `Use for Development`.
  - if this is a new device, copy the Identifier UDID and send it to admin user to add it to the online developer portal (should not be the case since all ipads and devices are already added, unless you intend to use your own iPad or iPhone)
  - go to Provisioning Profiles in LIBRARY in Organizer and click on `Automatic Device Provisioning`. Your device should have a green color dot if everything is OK and you should be able to run apps on it.
  - if you have a yellow/orange dot, select the iOS Team Provisioning profile and hit DELETE. Then hit Refresh again. XCode will update with the new profile and should work. Try disconnecting the device and rebooting also if it's not working straight away.

Please note that:

- All students use their own certificates, issued on the Mac they connected first. If they wanna use another Mac, they must export and import their profile from XCode by themselves.
- Students may need to install the WWDC certificate authority, which can be downloaded from the online developer portal.
- Profile will expire in Jan 2013, so there won't be any expirations during projects phase :)

Once that's done, let's try to install a very simple application, in fact, just a blank application. Create a single-view iPad application, named `DeviceTest`. Make sure you fill in `nus.cs3217` (case sensitive) as your company. Then change the target platform from simulator to iPad in the scheme selector in the top part of the IDE. Run your application and see that it can be loaded into iPad to run. Now that you are able to successfully deploy an application on the iPad, you can do the same for your future applications by applying similar steps.

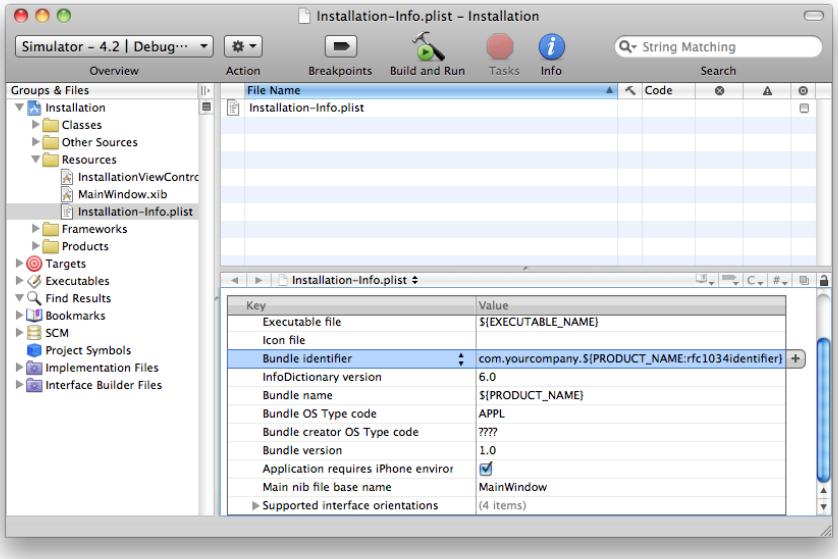

Figure 1: Bundle identifier.

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 ”Single View” application called ”Game” (or whatever else you desire). Remember to check the options `Use Storyboard` and `Use Automatic Reference Counting`. We are not going to use the full power of the `storyboard` in this tutorial, but it does not hurt getting you used to it.

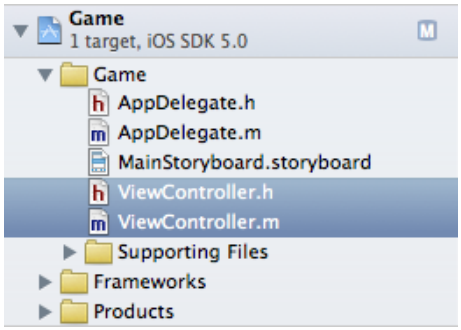

Figure 2: Controller class.

By default, the `Project Navigator` should be opened in the top left corner of the Xcode IDE. If not, you can open it by from the top menu by going `View` → `Navigators` → `Show Project Navigator`. Notice the highlighted `ViewController.h` and `ViewController.m` in the `Project Navigator`, as shown in Figure 2. These are the interface and implementation of the controller class of the view provided by the project template. Also notice the file `MainStoryboard.storyboard` in the `Project Navigator`. This is the file where the storyboard is stored. In most cases it suffices to use one storyboard for an application, even if you have quite a few different views.

2. Click on `MainStoryboard.storyboard` in the `Project Navigator`. This should open the storyboard for drawing in the centre of the IDE, as shown in Figure 3. Right now the storyboard contains only one empty view. In storyboard terminology, we call this view a `scene`. Look at the gray pane to the left of the drawing area. All the scenes are listed here. Currently there is only one scene. You can refer to Figure 4 for an enlarged view of the scene. This is where we wire the things we draw in the storyboard to our view controller class' member elements.

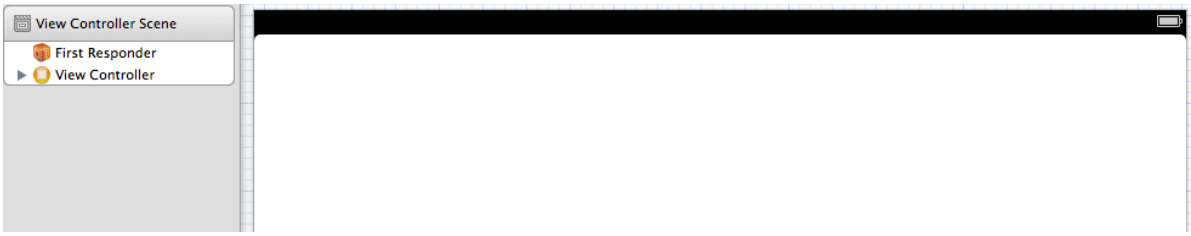

Figure 3: Empty view window.

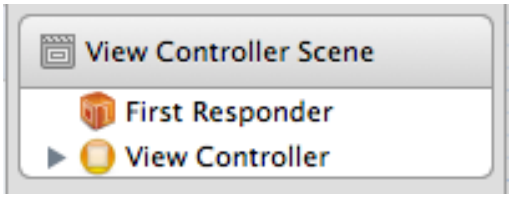

Figure 4: Scene

It is possible to create a storyboard programmatically, instead of drawing, but that is not in the scope of this problem set. You can learn how to do this on your own later.

3. Now it's the time to draw our interface. First of all we would like to change the orientation of our view. Click on `View Controller` in the scene to select it. Open the `Attributes Inspector` by going from top menu `View` → `Utilities` → `Show Attributes Inspector`. You should see it in the top right corner. Change the orientation from `portrait` to `lagdscape`, as shown in Figure 5. Notice that the view in the storyboard is automatically updated to reflect this change.

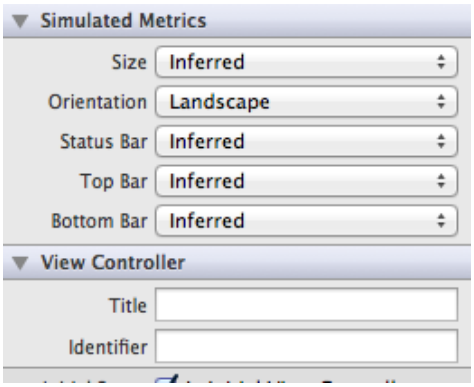

Figure 5: Attributes Inspector.

4. Open the `Object Library`. By default it should be opened in the bottom right corner. If not, open it by going from the top menu `View` → `Utilities` → `Show Object Library`, as shown in Figure 6. It contains many customizable building blocks, which we can drag and drop into our view.

Figure 6: Object Library

5. Drag and drop a `Round Rect Button` from the `Object Library` to the top left of our view. Select the button by clicking on it. In the `Attributes Inspector`, change its `Title` to **Start** and `Text Color` to `Light Gray Color`.

What we want to achieve here is that the button toggles its text color between black and light gray as we press it. In order to do that, we need an **IBAction** handler in our view controller class.

An **IBAction** handler deals with events generated by user interaction with the view. In this case, we need a handler that changes the button's text color when button press events are generated. Fortunately the IDE can help us with much of the drudgery.

6. Open the `Assistant Editor` by going from top menu `View` → `Assistant Editor` → `Show Assistant Editor`. The `Assistant Editor` should appear to the right of the main editor, with `ViewController.h` opened in it automatically, as shown in Figure 7.

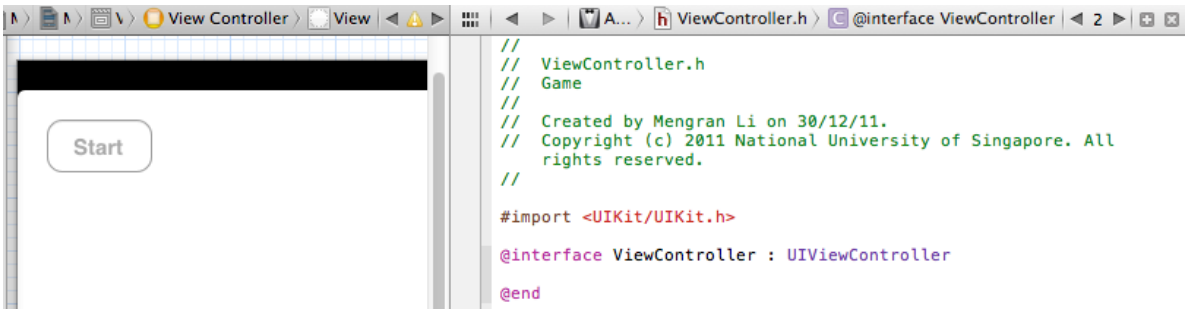

Figure 7: Assistant Editor

Click on the button to select it. Then, while holding the control key, drag the button to the interface declaration in `ViewController.h` in the `Assistant Editor`. Notice the line protruding from the button as we drag it, as shown in Figure 8.

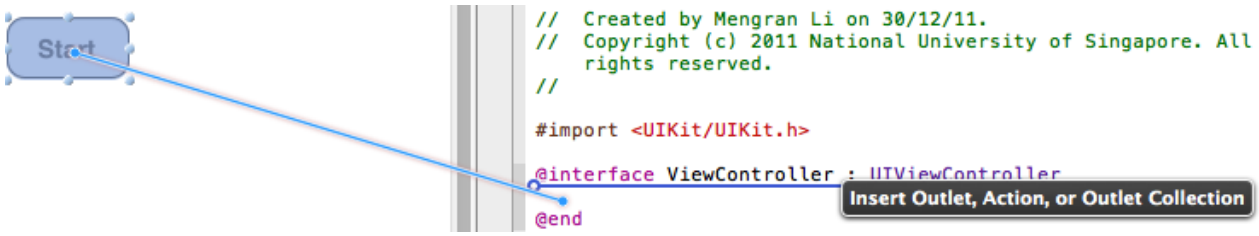

Figure 8: Dragging Button.

As you release the mouse button, a small window will popup. Change the `Connection` from `Outlet` to `Action`, then type in the name **buttonPressed**, as shown in Figure 9.

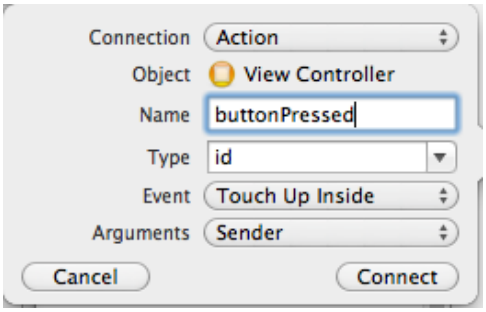

Figure 9: Wiring Button

Click `Connect`, then the following `IBAction` handler will be added to `ViewController.h`,

```
- (IBAction)buttonPressed:(id)sender;
```

Now close the `Assistant Editor` by clicking the little cross in the top right corner of it.

Open `ViewController.m` by clicking it in the `Project Navigator`.

Notice that an empty skeleton for the handler `buttonPressed` has been added here. Flesh it out like the following:

```
- (IBAction)buttonPressed:(id)sender {
    UIColor *newColor;
    UIButton *button = (UIButton*)sender;
    if ([button titleColorForState:UIControlStateNormal] ==
        [UIColor blackColor]) {
        newColor = [UIColor lightGrayColor];
    } else {
        newColor = [UIColor blackColor];
    }
    [button setTitleColor:newColor forState:UIControlStateNormal];
}
```

Now run the program, by clicking the `Run` button in the top left corner of IDE. Alternatively, you can go from top menu `Product` → `Run`. You should be able to toggle the button's text color by clicking on it.

7. If you are deploying to the simulator, you will notice that the simulator runs in portrait orientation by default, which does not reflect our design. Change that by going from the top menu of the simulator, `Hardware` → `Rotate Left`, as shown in Figure 10.

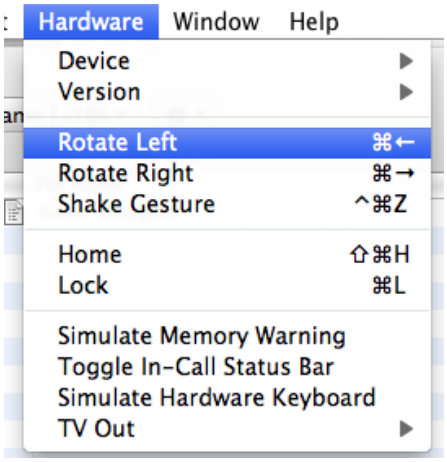

Figure 10: Rotating the simulator.

8. Copy and paste this button to create three more buttons. Note that when we copy a button, we also copy its link to the event handlers. 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.

Change the titles of the three new buttons to **Save**, **Load**, and **Reset** respectively in the `Attributes Inspector`.

9. Drag and drop a `Scroll View` to our view. Stretch it to cover the entire space below the buttons. This is where we will draw our game background later. In order to do that, we need to add an **IBOutlet** to the `ViewController` class.

An **IBOutlet** is a reference to an object through which the controller does its output. In this case, since we want a reference to the `Scroll View` that we've just added.

Open the `Assistant Editor`. While holding the control key, drag the `Scroll View` to the interface declaration in `ViewController.h`, as shown in Figure 11, the same as we did previously.

Figure 11: Wire scrollview

As you release the mouse button, a small window appears. Type in the name **game area**, as shown in Figure 12.

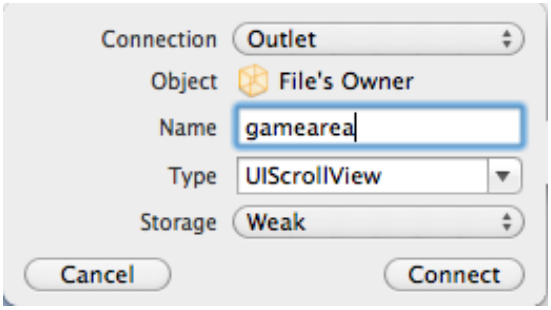

Figure 12: Wire scrollview 2

Click `Connect`. The wiring is done. You should see the following property added to `ViewController.h`:

```
@property (strong, nonatomic) IBOutlet UIScrollView *gamearea;
```

You do not have to worry about synthesizing this property in `ViewController.m`, as that is done automatically.

Close the `Assistant Editor`.

10. Click on the `Scroll View` to select it. Open up the `Attributes Inspector`. Uncheck the option `Bounces`.

11. We are done with the interface building for now. You should now have an interface like this Figure 13.

12. The game area is rather dull, isn't it. Let's place a picture in it. But before that, we need to add the picture into our project. Extract the images from `images.zip`. Right click the `Game` group in `Project Navigator` and choose `Add Files to "Game"`, as shown in Figure 14.

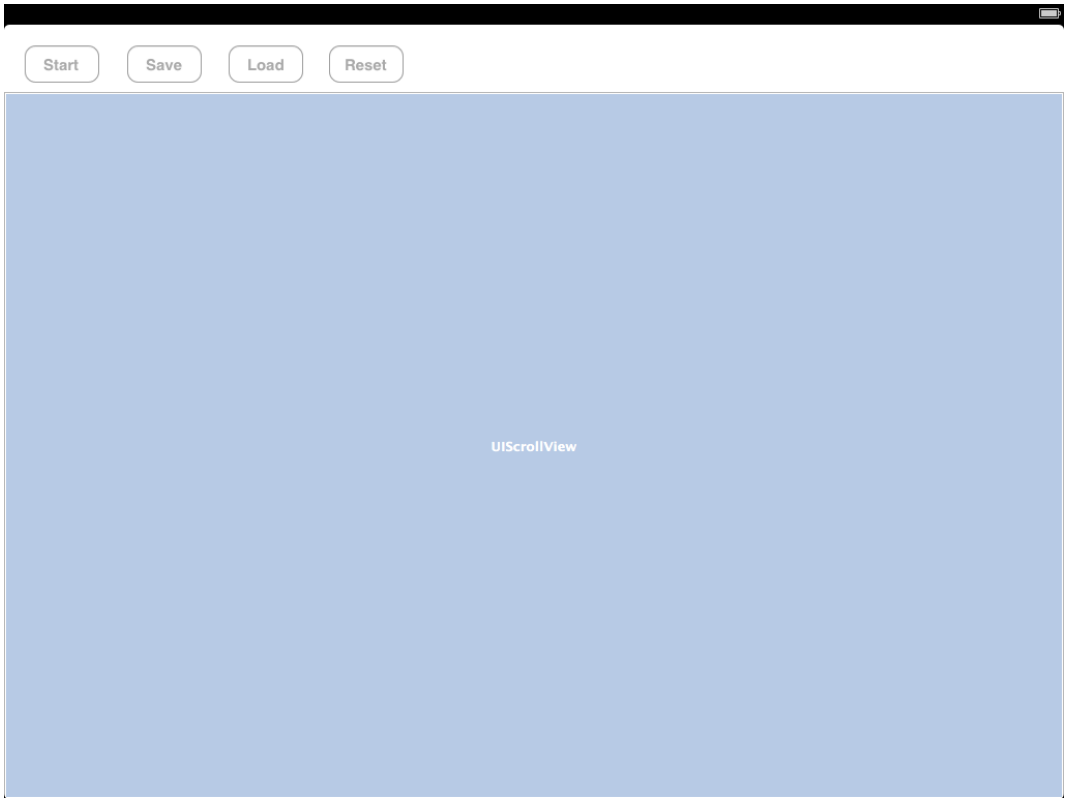

Figure 13: First interface.

Another window is opened, as shown in Figure 15. Browse to the two image files you just extracted. Select both of them, check the option `Copy items into destination group's folder (if needed)` option, and click `add`.

13. Now it's time to add codes to load up images to make our game area 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 `ViewController.m` by clicking on it in the `Project Navigator`. Look for a method called **viewDidLoad**, which is just a skeleton now. This method is called after the view is loaded from storage. We can do some further customization of the view here.

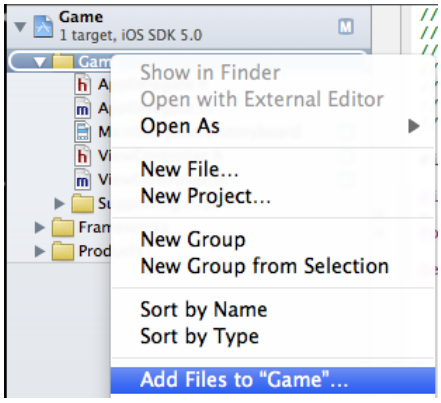

Figure 14: Add Pictures

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];
    // Do any additional setup after loading the view, typically from a nib.
    //load the images into UIImage objects
    UIImage *bgImage = [UIImage imageNamed:@"background.png"];
    UIImage *groundImage = [UIImage imageNamed:@"ground.png"];

    // Get the width and height of the two images
    CGFloat backgroundWidth = bgImage.size.width;
    CGFloat backgroundHeight = bgImage.size.height;
    CGFloat groundWidth = groundImage.size.width;
    CGFloat groundHeight = groundImage.size.height;

    //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 methods 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 addSubview:background];
    [gamearea addSubview:ground];

    //set the content size so that gamearea is scrollable
    //otherwise it defaults to the current window size
```

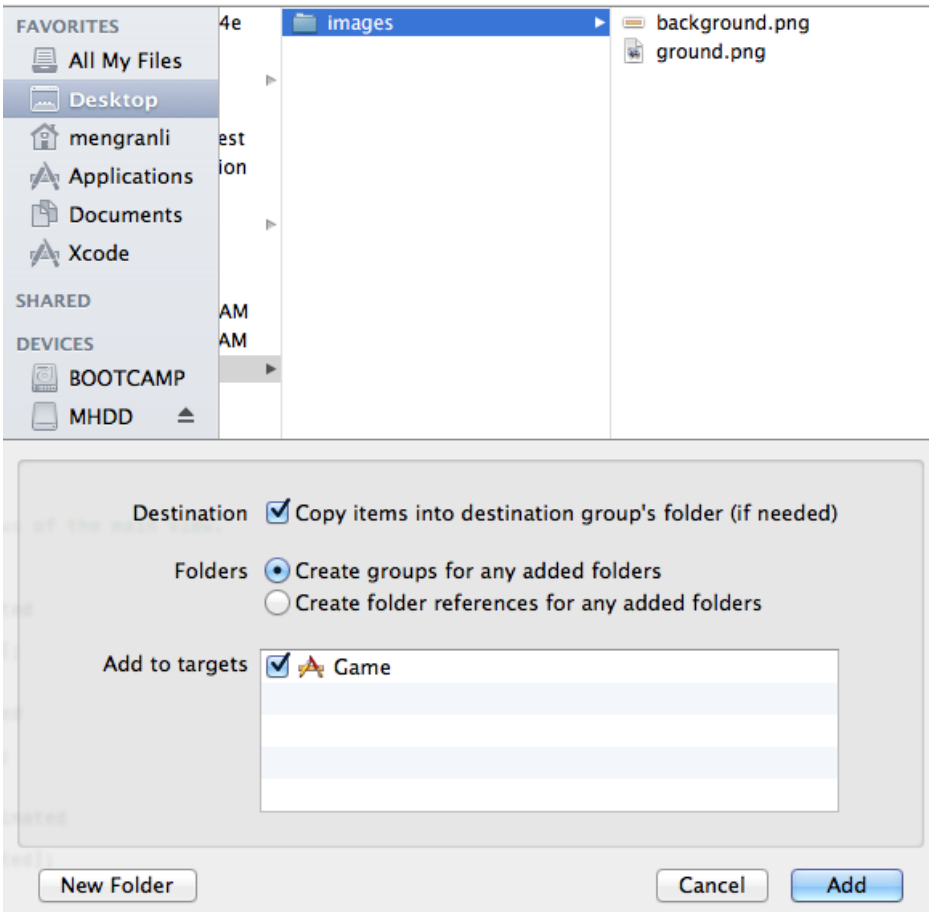

Figure 15: Add Pictures 2

```
    CGFloat gameareaHeight = backgroundHeight + groundHeight;
    CGFloat gameareaWidth = backgroundWidth;
    [gamearea setContentSize:CGSizeMake(gameareaWidth, gameareaHeight)];
}
```

Build and run the program. If you have done everything correctly, you should see something like the screenshot in Figure 16.

Figure 16: A better interface.

Now that you understand the basics of how to create a simple interface, feel free to improve it as you please. 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 for 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 (20 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`. To run it, we need to add a new scheme.

Select `Product` from the top menu and then `New Scheme`. You will be presented with a small window, as shown in Figure 17.

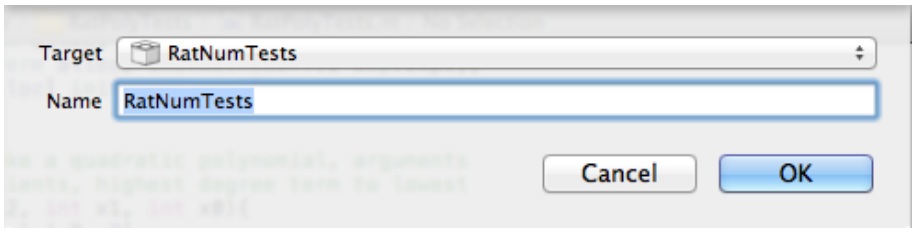

Figure 17: new scheme

Select `RatNumTests` as `Target`, and give this scheme the name `RatNumTests`, then click OK.

Similarly add a new scheme `RatTermTests` for the target `RatTermTests`, and a new scheme `RatPolyTests` for the target `RatPolyTests`. These two schemes are for the following parts.

To test our `RatNum` implementation, select `RatNumTests` as our active scheme in the scheme selector in the top left of the IDE, as shown in Figure 18.

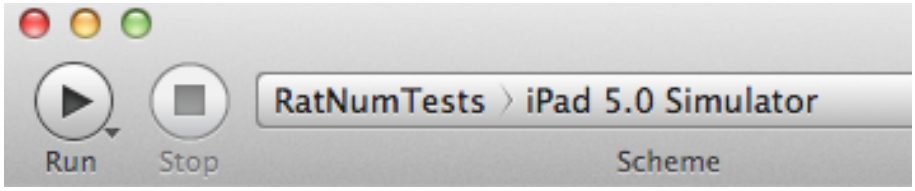

Figure 18: select scheme

Then run it by going from top menu `Product` → `Test`.

If the test succeeds, you will see a message **Test Succeeded** flash out in the screen.

Otherwise you will see the message **Test Failed** flash out in the screen, and alarming red exclamation marks in the `Issues Navigator`. If you click on one of them, you will open the corresponding file in the editor.

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(d) when deciding where checkRep should be called.

In the last section, we've added a new scheme for testing `RatTerm` as well. Set `RatTermTests` as the active scheme and run the test. Make sure your implementation passes all the test cases.

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 setting `RatPolyTests` as the active scheme and run the tests.

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

Like PS 1, the teaching staff will be grading your code directly on GitHub. Your solution for this Problem Set should be contained in a single directory called `ps02`. This directory should be inside the root directory of the private repository assigned to you. You must upload all your work to the master branch of this remote repository. You will be graded on the latest commit on the master branch before the deadline.

The directory should contain all the source files, so that your TA can run your code and unit tests. In addition, it should also contain the screenshot of the interface that you built in part one of this problem set.

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 19: Polynomial Calculator interface.

Screenshots of Sample App - Huff & Puff

Figure 20: Screenshot of Huff and Puff.