Homework #3: Trimodal Matching

Due: Tuesday, February 3 @ 12:30 PM

Submission:

Please turn in all files on Canvas before the deadline. You should compress your submission into a single file, do not submit a large number of individual files. If you know you are going to miss a deadline, contact the TA **before** the deadline. Canvas has been known to be quirky, so it is not advised to wait until 5 minutes before it is due to make your submission.

Please include a text file called "README" at the top level of your main project directory. Include the following:

- Your name
- Your email address
- How long this project took you to complete
- Any comments or notes for the grader

Overview:

This is not a group assignment. It is acceptable to consult with other class members, **but your code must be your own**.

You will expand upon homework 2 to create several views each having a variant of the card matching game in it.

Specifications:

Part 1: Model

RSCard

Take your Card class from project 2. Generalize this class into a class called "RSCard" where RS stands for "Rank and Suit."

This class should still inherit from the base class "Card".

RSCard will have a public method “validSuits”. This function should return type NSArray*. RSCard should return nil for this method. Classes that inherit from RSCard should override this method to return an NSArray containing all of the valid suit choices. (For example, @[@”♠”, ..., @”♥”]).

RSCard will have a public class method “suitColors”. This function should return type NSDictionary*. RSCard should return nil for this method. Classes that inherit from RSCard should override this method to return an NSDictionary with the mapping between suits and their appropriate color. (For example, @{@”♠”:@”black”, ..., @”♥”:@”red”}).

The setter method for the suit of this card should reject any suit that is not a valid suit.

The ranks for type RSCard are the same as the standard card ranks (Ace through King). The setter for this method should reject any rank that is not a valid rank.

Matching for these cards should follow the same rules as in assignment 2.

Reduced playing cards

Create a new type of card and deck called “ReducedPlayingCard”. This should inherit from RSCard.

This type of card may have any of the ranks allowed of playing cards (A through K), but it is only allowed to have the suit “♠”.

Create a new type of deck called “ReducedPlayingCardDeck”. A deck of reduced playing cards will have 52 cards. It will have four copies of each type of card. For example, there will be four “A♠”, four “2♠”, etc.

Standard playing cards

Create classes “PlayingCard” and “PlayingCardDeck”. Playing Card should inherit from RSCard.

These cards should have the standard four suits. The behavior of these classes should be essentially identical to their behavior in Homework 2.

Expanded playing cards

Create a new type of card called “ExpandedPlayingCard”. This class should inherit from RSCard.

The expanded playing cards have all of the suits as the standard playing cards, with the addition of four new suits.

Red suits: ♥ (heart), ♦ (diamond)
Black suits: ♠ (spade), ♣ (club)
Blue suits: ☀ (sun), ☾ (moon)
Green suits: ♆ (trident), ❧ (turnip)

If you have trouble finding any of these symbols, copy and paste them from this document.

Create a new type of deck called “ExpandedPlayingCardDeck”. There should be one of every type of card for a total of 104 cards in a deck.

Part 2: View

Add a UITabBarController to your app. This should allow you to navigate to three different views.

View 1: your matching game but with ReducedPlayingCard cards
View 2: your matching game with standard PlayingCard cards
View 3: your matching game but with ExpandedPlayingCard cards

Each variation of of the game should have a different image for the back of the card. Also, don’t forget to add an icon in the UITabBarController for each action.

Embed each matching game inside of a separate UINavigationController. There should be a button inside each game that will take the user to something called the History view. (Consider using a Bar Button Item in the Navigation Controller bar.)

In the History view there should be a scrollable text field where we will display the history of the game. The game will create a variety of log messages. These messages should be displayed as a bulleted list (or equivalent).

You may remove the following UI elements from each matching game (since they will be displayed in the history tab):

- Number of games played
- Average score
- Score value of the previous match (of the current game)

Don’t forget to properly color cards in the expanded playing card version. Any suits displayed in the history view should also be properly colored.

Part 3: Control

The behavior for the matching games should be identical to the behavior specified in Homework 2 (with the exception of the card types).

Each version of the game should have its own unique history view. That is, actions taken in the expanded version should not be visible in the reduced version.

Log the followings things in the history log:

- Card matches. Log the cards that were flipped (rank and suit) as well as the point value of the match.
- Peek. Log when the user has taken a peek action. Also log here how many cards have yet to be matched (i.e. how many cards the user is peeking at).
- Reset. Log the score of the previous game as well as the current average score. Log how many cards were matched in the previous game as well.

Notes:

You may use any code presented in class. Please type the code yourself (as opposed to copy-paste) as it is a good learning experience.

Don’t forget to add an alpha layer (i.e. transparency) to your icons down in the tab bar controller.

If you violated MVC in your solution to of Assignment 2, then this will be more difficult (that’s why you shouldn’t have violated MVC!) and you’ll probably want to go back and redo those parts.

There is no concept like “protected” in Objective-C. Unfortunately, if a subclass wants to send messages to its superclass in code (not with ctrl-drag), those methods (including properties) will have to be made public. A good object-oriented design usually keeps publication of internal implementation to a minimum!

All methods (including properties) are inherited by subclasses regardless of whether they are public or private. And if you implement a method in a subclass, you will be overriding your superclass’s implementation (if there is one) regardless of whether the method is public or private. As you can imagine, this could result in some unintentional overrides, but rarely does in practice.

If you copy and paste an entire MVC scene in your storyboard (not the components of it piece-by-piece, but the entire thing at once), then all the outlets and actions will, of course, be preserved (this can be quite convenient). Even if you then change the class of the Controller in one of the scenes, as long as the new class implements those outlets and actions (for example, by inheritance), the outlets and actions will continue to be preserved.

As you start working with multiple MVCs in a storyboard, you might get yourself into trouble by accidentally changing the name of an action or outlet or making a typo or otherwise causing your View to send messages to your Controller that your Controller does not understand. Remember from the first walkthrough of this course that you can right click on any object in your storyboard to see what it is connected to (i.e. what outlets point to it and what actions it sends) and you can also disconnect outlets and actions from there (by clicking the little X's next to the outlets and actions). If you are getting crashes that complain of messages being sent to objects that don't respond to that message (sometimes a method is referred to by the term "selector" by the way), this might be something to check.

There is no reason that the history-reporting MVC needs to be different for the two games. In fact, if you decide to use a different history-reporting MVC for each, you will want to justify your reasons for doing so in comments in your code.

Remember that every time you segue to a new MVC in a UINavigationController, it is an entirely new instance of that MVC. Also remember that that MVC is considered part of the pushing MVC's View (can only talk back to the Controller of the pushing MVC in a blind, structured way--luckily, there's no need to do that in this assignment).

The History MVC required task is mostly about creating a new MVC and how to segue to it (and only a tiny bit about using a UITextView to display text). Don't overthink the part of this which is actually displaying the attributed strings in the UITextView.