CSE 341, Spring 2004, Smalltalk Project
Code Description

Last Modified: May 20

Introduction

You are to implement a player for a card game. The rules for the game are available separately. This document discusses the details of the code your implementation will use and provide[1].

Player Code

Your player will be a class that responds to messages sent to it by the game controller. If your player raises an exception, it is expelled from the match. Exceptions can be raised by the normal sorts of bugs (for example, sending an `ifTrue:` message to something not a boolean), as well as making invalid bids (see the game rules for details).

To prevent name conflicts, all of the classes you write must be have names beginning with `P`*xx*, where *xx* is the player number that Dave will send you.

Your player should respond to the following messages. (It is probably a good idea to subclass the provided skeleton class, called `HighLowPlayer`, and override its methods as needed.) The messages your player handles are broken into categories:

- **Dealer Interactions**

  These are the basic messages you use to play.

  - `receiveCard:` passes you the card that was dealt to you.
  - `makeMove` allows you to either swap your card with the face-up card (by answering `True`) or to pass (by answering `False`). Note that `makeMove` doesn't pass in what the current face-up card is; you must track this yourself using the notifications (discussed below).
  - `makeBid` allows you to either bid, by answering your bid amount, or quit for two by answering nil.

- **Notifications**

  These messages notify you of what happens in the game. Good players base their moves on what the other players are doing.

  They are listed in roughly the order they happen in a round.

  The parameter passed in these messages after the word "by" or "from" is always a player number—an integer from 1 to 4.

  - `notifyNewRound:` tells you that a round has begun. It includes the current round number.
  - `notifyDealerCard:` tells you what the dealer's initial card is.
  - `notifyPassBy:` tells you that a player has passed (chosen to not swap their card). It includes the player number.
  - `notifySwap:by:` tells you a player has swapped their card for the dealer's. The card they swapped (the card they had, which is now the dealer's new card) is provided.
  - `notifyBid:by:` tells you of of player's bid, or `nil` to indicate they chose to quit for two.
  - `notifyCard:from:` tells you what a player's card is. This message is only sent after every player has made their bets, and only for players who have chosen not to quit this round.

- **Utility Functions**

  - `initialize` is sent when your player object is created. It can be overriden to initialize member variables, etc.

[1]If this documentation ever contradicts what the code does, believe the code.

  - `position:` informs you which player number you are.

    Within a match, your player number is constant. For each round, which player starts rotates; on round 1, it's player 1, on 2 it's 2, etc.
  - `name` should answer your player's name.

Other Code

You're welcome to browse the rest of the code, but most of it won't be too useful to your player. However:

- You can send any card object the `cardNumber` message to get its value (1–13) and the `suitNumber` message to get its suit (1–4, where lower numbers correspond to suits ranked lower by the game rules).
- You can send the message `cardValue:` to the `HighLowGame` class to get an integer from 0 to 51 indicating how the card is ranked in the game. For example, for the five of diamonds, `HighLowGame cardValue: theCard` is 13: there are thirteen cards with lower value (four each of the 2, 3, and 4, and the five of clubs).

Getting Started

1. Create a new category by clicking on the HighLow category in the category list, right-clicking, and choosing "add item…". Put all your classes in this category, and then when you're done you can simply fileOut that category to export all of your code.
2. Click on the background of the Squeak world, pick "open…", and open a workspace and a transcript. The game outputs what's happening via the Transcript and you'll want one around to test your player.
3. Create a sample game `g` in the workspace with code like this:

```
g := HighLowGame new initialize.
g addPlayer: (HighLowPlayer new initialize) at: 1.
g addPlayer: (HighLowPlayer new initialize) at: 2.
g addPlayer: (HighLowPlayer new initialize) at: 3.
g addPlayer: (HighLowPlayer new initialize) at: 4.
```

4. Now you can either run an entire match by sending `g` the `runShuffle` message, or run a single round by `g runRound: 1` (or pick another number to run another round; be sure not to run so many rounds the deck runs out of cards!).

   If you want to test this more quickly and with less output, run `g quiet: true` before running a shuffle.
5. Create your own subclass of `HighLowPlayer`, change one of the players in the above code, and start making your player better than the drones!