Daniel Sosebee
10/2/2019

Memory Card Game Design Document

For this project, I focused on writing highly extensible code, and on providing a simple and straightforward user interface and gameplay experience.

1. Usage

1.1 Starting the Game

In the base directory of this repository, there is a file titled *Memory.run*. After downloading this file, navigate to its parent directory in the console, and type:

./Memory.run competitive 13

1. This will start a game of two-player memory with 52 cards. (NOTE: if execution fails, it may be necessary to type this as well: *chmod +x Memory.run* , before trying to execute the program)

You may replace the first argument with 'solo' for single-player mode, and you may replace the second argument with any number from 1-13 inclusive to limit the number of cards in each suit. For example, a simple way to test the game would be:

./Memory.run solo 2

The above command would start a single-player game with 8 cards.

1.2 Playing the Game

Gameplay should be straightforward after starting the game. Players are prompted in turn to select pairs of cards from those available. The current player then receives feedback as to whether they selected a correct pair (A correct pair are two cards of the same face or number which are either one Hearts and one Diamonds, or one Spades and one Clubs).

'Red' cards will be surrounded by parentheses, while 'black' cards will be surrounded by curly braces, so focusing on this will likely improve your memory of the cards.

When no cards are left, players will be shown who won, or how well a single player performed, depending on the game mode.

2. Design Choices

2.1 Classes

Code was separated into simple classes, allowing for greater modularity and extensibility. See below for more information about design decisions for each class:

2.1.1 Main

The *Main* class handles the arguments passed in through the *args[]* array, and instantiates the *MemoryController* object to run the game based off of these arguments.

This class is responsible for printing specific error messages in the case that the user types incorrect arguments. These error messages guide the user towards correctly starting the program.

2.1.2 MemoryController

MemoryController is responsible for orchestrating the game of memory. This is an abstract class, meaning that it is very easy to add new child-classes of *MemoryController* for orchestrating different types of memory games.

- Players are stored in a *List<Player>*, which gives child-classes the option to choose the size of the list, and how to implement it. There is no limit to the size of this list, and while there is currently not an option for playing with more than two players, *MemoryController*'s method implementations all would hypothetically work for any reasonable number of players.
- We keep track of the turn by storing an integer index of this list.
- Cards are stored in a *CardLayout* object.

User input for choosing cards is completely validated, and players are prompted until they provide an index of a valid card.

2.1.3 SoloMemoryController

This child-class of *MemoryController* is used to set-up solo gameplay.

SoloMemoryController overrides *MemoryController*'s *endGameMessage()* method to print out the number of failed guesses when a player finished the game.

2.1.4 CompetitiveMemoryController

This child-class of *MemoryController* is used to set-up two-player gameplay.

CompetitiveMemoryController overrides *MemoryController*'s *endGameMessage()* method to print out the winner, their score in pairs won, and any ties.

2.1.5 Player

Player stores the pairs successfully collected, as well as the failed attempts of a given player as integers, and provides a *toString()* for printing this information to the user.

2.1.6 CardLayout

CardLayout uses a *List<Card>* (implemented as *ArrayList<Card>*) for storing cards, allowing for modularity in the size of the deck. This lets us build a random deck using list insertion.

The *toString()* method returns a string representing the full grid of cards, making sure that the cards are printed correctly face-up or face-down, and that all cards are the same length in characters.

2.1.7 Card

Card has an integer representing its value, and a Boolean representing whether it's face up.

The *toString()* representation of a card depends on the suit color (which is a function of the *Card*'s value): red cards get surrounded in parenthesis, while black cards get surrounded by curly braces. For example:

(AH) {10S}

This helps the users identify and remember cards by their suit, since the console interface is black and white.

2.2 Global Variables

The package *Constants* contains publicly accessible constants that are relevant to multiple classes. For example, a *public final static int PLAYERS_IN_COMPETITIVE_GAME*, which is currently set to 2, dictates the number of players who compete in competitive mode.

3. Tooling

I used Java and the Java Standard Libraries for this project. I chose Java because it has a lot of object-oriented-programming functionality that I wanted to use for my class structure in this project, and because it's a language in which I have a lot of experience and can code quickly. I considered adding Junit tests, but decided that for a project of this scope, and because of how much of the functionality is distributed across classes and across methods, it would take too much time to add these tests in any sort of comprehensive way.

In the future, for projects similar to this one, I could benefit from spending more time researching third-party libraries that may be relevant to my application.