Object Oriented Programming

Program9

Chris Graff

1) Game Description

This arcade game has been designed by Chris Graff. When you start the Arcade, you will be presented a main menu. This will allow for five options. If you select the first, you get a game of Hangman if the second, Baseball, and if the third, TicTacToe. If you happen to select the fourth, you will be presented with a ScoreCard and the fifth option will quit the game.

a) In Baseball, you will have the option of playing a game of baseball which you will have to guess the correct number key based on your strikes and balls given. You will guess by entering a three digit number, and pressing Enter. You will be told how many strikes and balls you got, and be told to make a new guess. A strike is a correct number in a correct spot, and ball being a correct number, not in the correct spot. When you guess it, you will be assigned a score which is calculated by taking the turns it took you to guess the correct number, minus three, and then multiplied by 100. The score will then be recorded, and you will be returned to the main menu.
b) In Hangman, you will be given a word from a FiniteDictionary, and a will have to guess it. When you guess a letter, the letter either appears on the line, or does not if it does not exist. Regardless, the letter is printed under used letters, and the letter is taken away from the letter bank by replacing the used letter with a '.' character. When you have guessed the correct work, you will be returned to the arcade to play another game, and the score recorded. The score is calculated by taking the number of guesses, subtracting the number of letters in the word, and multiplying that by 100. The Source code for this game was received from Kenton Standard, who in turn I believe received it from Micah Gustin.
c) In Tic-Tac-toe, the game will allow the player to place his pieces (O's), and the computer will place a random piece(X) in an empty slot. When the board is full, or the game is won, then you will be assigned a score, and returned to the main menu. If you win you receive 100 points, of you lose, you receive zero, and a draw earns you 50.
d) In Scores you will be able to see all of your scores, and what game they were for. As well as the number of times playing each game and the average score for the game.
e) Pressing option 5 will exit the game, losing any arcade game data acquired.

As far as the behind the scenes implementation goes, your selection is based upon a switch-case statement to determine input validation, as well as changing the choice, and performing the option. All I/O operations take place in either the main function, on in the Overridden playGame helper functions which handle I/O for each game. Those will be found under HangmanGame, BaseballGame, and TicTacToeGame, respectively. All game logic, and internal workings take place in the other classes, with little other than calculating the score and generating output and handling input being done here. Each time a game is played, the

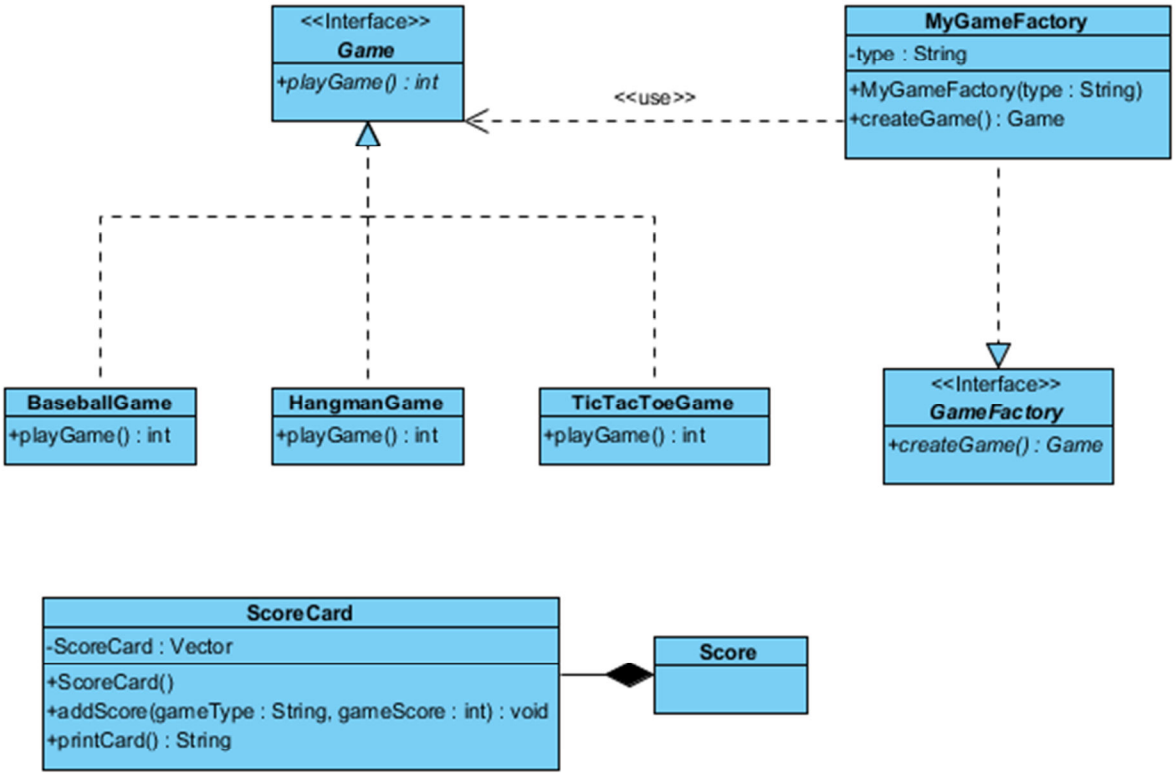
GameFactory using MyGameFactory to implement the createGame function creates a new Game with PlayGame being overridden in each [Type]Game class for individuality. Once a game of the given type is created, it will use its [Type]Game class to then call the [Type] class which houses the internal game logic. Some of those call even deeper functions in order to perpetuate the game in its entirety. The ScoreCard is a Vector of Scores. I implemented Vectors in order to use the Iterator Design Pattern.

2) Responsibilities of Objects

Object	States	Behaviors
MyGameFactory	• Type	• createGame
Game		• playGame
GameFactory		• createGame
Score	• gameType • Score	• getScore • getSCoreNum • getType
ScoreCard	• ScoreCard<Score>	• addScore • printCard
Key	• KeyCode • First • Second • Third	• getKey • checkKey
BaseballGame	• Guesses • MyKey • Check • Result	• playGame
HangmanGame	• myWords • game1 • guessletter • alreadyguessed	• playGame
Hangman	• Alphabet • Wordlist	• chooseLetter • modifyAlphabet • wordListToString • serCurrentWord • evaluateGuess • getcount • getBlankedOut • getCurrentWord • getAlphabet • gettitle
FiniteDictionary	• Dictionary • lastIndexUsed	• getRandomWord • toString

		• populate • getMAX_CAPACITY
TicTacToeGame	• Turn • TTT	• playGame • makeXmove
TicTacToe	• boardPosition • Board	• getBoard • checkMove • addMove • boardUpdate • getFull • getWin

3) UML class Diagram



.

4) Design patters
   a) Iterator Pattern
      i) One could also prove that this is a Template Pattern given that Iterator requires that it be given a type, much like Vector.
      ii) Iterators used to access data within vector without actually directly accessing data.

iii) I would look deeper into the actual Iterator class, but cannot actually find the underlying code. Everything is hidden and can't be accessed (Iterator and Aggregate would be located here, but cannot be named).
iv) As discussed in lecture, Iterators are found in certain types of array lists, including vectors, and ScoreCard() initializes the Vector ScoreCard, which is a Vector of Scores.
v) Uses {Vector}.add() inside of the method addScore() in order to use built-in iteration technique to Automatically Iterate to the end of the Vector, and then store the data at the end.
vi) In printCard(), uses Iterator.next() (Concrete Aggregate) function in order to retrieve data, and adjust iteration. Also uses Iterator.hasNext() (Concrete Iterator) function to determine whether another Iteration is possible.

b) Factory Pattern
   i) Factory pattern starts in the Game(Product) class which creates an abstract of the playGame() method.
   ii) The playGame() functions are Overridden at the BaseballGame, HangmanGame, and TicTacToeGame (Concrete Product) classes in order to evolve functionality. They handle the specific I/O functions for each game, as helpers for the main, and call the game logic, which is located deeper in the files.
   iii) GameFactory (Creator) creates an abstract of the createGame() method.
   iv) This is then Overridden in MyGameFactory (Concrete Creator) by gathering which type is necessary via a string passed to the constructor, or the createGame() method.

c) Unknown Pattern
   i) I could not find another pattern amongst the 15 given that I could implement without rewriting massive amounts of my code. It should be noted however, that I sent my code to a friend who works at Amazon.com, and the first thing that he noticed was that I was using a Factory pattern, and he was extremely impressed. He said that he wished that he had known how to do design patterns at the end of his college career, because learning them in the workplace was much more difficult.