Assignment 2 Report

Linked Lists

Linked list was used for both the tile bag and the box lid, for the tile bag it is used as the tiles would be added in random order and can be taken out in that order while also removing those tiles from the list which is the best fit for this as it allows individual tiles pulled out and added into factories which are held in arrays. The varying storage size also helps in this case.

For the box lid the varying storage size works and being able to pull tiles from the back and the front also allows for a function to assist in randomising the tiles as they can be pulled from both the back and the front of the box lid in a random order.

Vectors

A 2D vector was used for the player factory lines as it can hold vectors of varying sizes unlike 2D arrays which can only have the same size for every array stored. It also has functions for identifying the size of the vector which can help for checking how many more tiles can be inserted into the players lines.

A vector was also used in one of the load functions as the rotate function is needed to compare the load data with what it should be in the player wall so the mosaic pattern is correct with what it should be.

Arrays

The factory wall has a 2d array as the wall is a 5x5 set of tiles, with the numbering of 2d arrays it also has an intuitive way to compare adjacent tiles by comparing index numbers of where the values are stored in the array.

A 1D array was also used for the broken floor tiles for each player as it has a fixed size for how many tiles can be inserted and can also be easily iterated over to check for open slots to insert tiles.

Factories also have 1D arrays which for factory 0 can only have so many tiles that are present on the game board and the rest of the factories can only hold 4 tiles, similar to broken floor tiles 1D arrays are easy to iterate over and add less complications as varying storage sizes aren’t necessary.

Main

The main class includes minimal functions and only includes the GameEngine object as to call the start of the game as well as the load game functions. Outside of this it only includes functionality for the menu including showing the credits. This also includes the functionality to pass through launch parameters which is used to modify the seed used for the random functions to determine what the results will be of that random function.

LinkedList

LinkedList was created for the linked list object to create dynamically sized lists of objects, this was created as its necessary to create multiple linked list objects, it has the ability to add/remove objects from the back and front of the list as well as retrieve the value of an object stored at a certain index in the list. We feel it was well designed as it has all the functionality necessary and it keeps it simple which makes the code less prone to errors as well as easier to read/understand.

GameEngine

GameEngine was designed to hold all the objects associated with the game being the GameBoard and Players which also includes the PlayerBoard objects by proxy as they are carried in the Player object. This class is designed to carry the functions for the players to manipulate the game board including processing the input for that and carries the main gameplay functions, this class also calls functions for the other classes to manipulate them for gameplay purposes and this class also has the functions for the saving and loading the game.

We decided it would be good design to have all the other class objects stored in the same class where they can call all the functions to run the game so we decided to create the GameEngine class. As we have stored all the other objects in this class it allows all functions to be called and allows the appropriate data to be passed through via parameters to other classes as necessary.

We also decided it would be best to have the save and load functions included in this class as well as it allows the function to access all other classes to get the data necessary to save the functions to a text file as well as modify these values to load the game from a saved state. The load function also ensures that the data is valid before saving it to the class variables to ensure that the game cannot get to an invalid game state after being loaded. In GameEngine we decided it would be best to track the current active player with a boolean as opposed to a name as it would allow us to track the players independently of name as they could have the same first name, this also applies to the save file as it shows a boolean rather than a name which would have situations in which it cannot be accurately interpreted.

Player

Player was created to store player information such as name and score and has the PlayerBoard object which is created with each player as each player needs their own board to play the game. This class is in charge of its PlayerBoard objects and also storing and clearing the points of the player board after each turn after adding the points into the total Player score.

GameBoard

GameBoard was created to store fix sized factory arrays, box lid and tile bag link lists, and a string to store the order of tiles. We decided to have the factories as fix sized arrays because a factory's size is constant and therefore no array resizing is needed. The box lid and tile bag on the other hand are not fix sized, thus we decided to use linked lists on them because it is far more efficient than using the regular array. We have the order of tiles declared as string because it is by far the most accurate way of storing a character order.

We defined the five fix sized factory arrays with max size of 4 each (defined on Types.h), and a centre factory array with max size of 16. We also decided to have the centre factory's size 16 because considering both player take a single tile from each of the 5 factories the number of left over tiles put into the centre factory would be 16 (in addition to the first player marker). Since our factories are made of separate different named arrays, we have the retrieveFactory method to return a specific factory given an int, thus it's easier for us to iterate through factories. For ease of use in GameEngine, we have a method to directly take tiles from centre factory given tile colour and take the first player marker from centre factory. We also have an array to check whether or not the factories are empty. There's also a printFactory method to print out the factories in a nice format.

For the random order of tiles, we have the generateTileOrder (parameter: Random seed) method that is going to shuffle the tiles (R, L, Y, B, U) using a random engine. We have the random engine produce number 1 to 5 in a random order and assign each tile to the given number.

The fillTileBag (parameter: Random seed) method produces random tiles and puts each of them into the tile bag Linked List. There will be a maximum number of 20 for each different tile. We are only going to call the fillTileBag method once per game. The fillTileBagFromBoxLid on the other hand fills up tile bag from the box lid Linked List and will be called in the end of each round.

We decided to let the takeTile method to handle the taking tiles from a factory and store the left-over tiles into the centre factory, because it is more efficient to let the object automatically handle the tile movement rather than having to call different methods separately. It has return value of type int to return the number of same coloured tiles exist in the factory. First we are going to iterate through the given factory tiles and checks if a tile matches the given tile, then we set the matching tile back to its former state (a dot char), put different coloured tiles into a temporary array and increment the counter. When it's done iterating, we check whether or not the given tile exists in the factory (that is if we did increment the counter). If the method did find the given tiles, it is first going to set the leftover tiles value to a dot (remove them) and then store the temporary array containing the left-over tiles into the centre factory. Finally, the return will help telling GameEngine the number of same coloured tiles in the factory to store into player's mosaic.

In addition to above methods we also have a method to return box lid for use in GameEngine , methods to return the stored arrays and lists for saving the current GameBoard state and load methods for the load game functionality.

PlayerBoard

PlayerBoard contains the state of the player board which has 5 rows of storage lines (2D vector) and a player mosaic (2D array) which is a grid of completed tiles. It also contains a broken tile line (1D array) which handles extra tiles that cannot be filled into the player tile storage lines and player wall.

We decided to let the PlayerBoard handle everything related to calculating the score and the inserting of tiles from the factory, because it is more efficient to let the object handle it rather than passing every state into a higher level object and using those passed values to calculate the score, or insert tiles.

There's an insertIntoLine function to check if the tile and the line it is being inserted into is valid with regards to the current state of the PlayerBoard. If not valid it does not insert the tile and returns a false letting the GameEngine deal with what happens after that. There is a check for the First Player Marker tile and if this tile is inserted it is automatically sent to the broken line. The next check is to confirm if the tile can be entered into a player line (if line is empty or contains similar tiles) or if that corresponding row in the player mosaic has not been filled. If either one of those conditions are not met, it returns a false, and it doesn't add the tile to that specific line. As tiles can be added in multiple sets if that player line is full each extra tile is then added to the broken line. If the specified line at the start is the broken line, then it automatically adds all tiles into the broken line. If the player lines and broken lines are full, all tiles are then sent to the box lid. After a successful insertion the value true is then returned and once again the GameEngine deals with what happens after that.

The next method is the insertToWall function. This checks if a single player storage line is full. If it is, it moves a single tile of that line into the player Wall and sends the rest of it to the box lid. It is during this moment that, the calculateScore method is used to calculate the points depending on the number of tiles and their locations on the player mosaic.

There is an extra endScoring method which is called at the end of the game to score additional points depending on the state of the player wall at the end.

The scoring is done in the PlayerBoard as it is best to calculate everything as soon as the tiles are inserted into the player mosaic. You would only enter tiles into the player mosaic (if possible) at the end of a round. The Player object in charge of the PlayerBoard will then take these points and store it in their Player score variable and then use a PlayerBoard function to clear the points stored in a player board object making it ready to calculate the points for the next round. This way we have the total Player score and an instance of the score from a player board for each round.

The PlayerBoard contains many more helper methods such as checkLine or checkWall, all included to help the inserting Tile method and it also includes a print mosaic method which can be used to print the current status of the Player mosaic in a nice format.

Group Feedback

As a group we felt we worked well together, there was a lot of communication and we were able to agree on how to delegate the work as well as on how to resolve any design conflicts we came across. Group members were quick to respond on issues if anything came up or if someone needed to make quick adjustments on their code. Everyone was also able to quickly complete the parts of their code so we had a good amount of time to test our code and ensure bug free implementation.

Everyone was also always present in tutorials to show off work and we were able to easily co-ordinate what we needed to show and who would show it and there was never a point where anyone was lacking in communication overall.

Overall it was a very positive experience and everyone learnt a lot in regards to working as a team in regards to handling delegated work, working alongside others in a programming environment and handling any issues that arose in regards to conflicts between ideas and code in these situations.