**ISAIAH**

Linked List

The Linked List implemented in our assignment includes only the methods necessary to its functionality within the qwirkle application. For example, our Linked List contains a method to allow nodes to be added to the back of the list, however no methods to add to the front or middle of the list, as these functions would be useless in qwirkle.

Further, out implementation contains multiple enhancements to allow our Linked List to be more efficient. These include the additional of a tail field (which acts similarly to the head), a counter for tracking the number of nodes (numNodes) & a toString method.

The numNodes counter makes adding, removing & iterating through the Linked List much easier, whereas the toString method allows easy printing of the contents of the Linked List for gameplay purposes.

Design of Implementation

The program was built with defensive programming in mind. The program heavily relies on the user input to run & as a result we needed to ensure that the program would only accept certain inputs from the user and act accordingly. To validate the user input, a special Validator class was created which is responsible for checking the structure & value of the user’s input & throwing exceptions if it is invalid. This implementation also made it easy to present the user with helpful error messages & continue asking for their input until it was valid, as thrown exceptions were efficiently used as “break” statements to prevent code being executed if the input was invalid.

Test Case Effectiveness

Tests cases allows quicker & easier testing of certain methods & allowed us to more efficiently make changes to our program based on its behaviour. Instead of running out long & time-consuming commands over and over, test cases let us see errors, make changes & see the result of the change in a fraction of the time. This let us have more time to implement additional enhancements in the program and test for as many bugs as possible.

Group Co-Ordination & Project Management

A screenshot of a social media post

Description automatically generatedThe majority of our group communication was done through “messenger” and also face-to-face. These allowed us to keep all members updated on the status of the program and any alerts regarding the project. We used GitHub to manage the project & ensure everyone has the most recent build of the program. This also made concurrent working & integration of work easier through the effective us of branches & GitHub merge functionality.

Example of Project Management.

Screenshot of GitHub page.

**SHREY**

**File I/O**

**Design**

The functions of saving and loading a game are carried out through a separate class called FileIO. Instead of using the main qwirkle.cpp to carry out file input and output, another class is created to maintain cohesion. To save and load, there are multiple functions that needed to be created and used. Keeping them in the main, would have led to unnecessary additions that do not have much to do with gameplay. Instead the main just creates an instance of the FileIO class which calls upon the different functions.

In order to save and load the game, an instance is created on the stack, because it is only needed in one function of the main class. The save function is a void method because it does not need to set any of the objects or variables. On the other hand, in order to load in a game, a current player is returned. This is because when loading, objects need to be set and recreated for gameplay to take place.

The save file is segregated in 4 sections; players, board, bag and the current player. Each section is separated by an empty end line. The empty line provides as a terminating condition for a loop which reads in a particular section. Having said that, each section is read in using a separate function which loops over the specified portion of the save file. Moreover, because the instances of the FileIO are created on the stack, memory links would not be caused due to the instances.

**Functionality**

In order to save the gameplay, FileIO takes in the vector of players, a vector of vector strings also known as the board, the bag and finally the current player. Each one of these are printed in the same order as mentioned into a file and saved as a ‘.save’. To load in from a file, FileIO takes in the same but they are updated. Additionally, a vector of board positions is passed in as well. A file is only loaded in if it exists and is immediately closed after the functions are carried out. If a file does not exist, then an exception is thrown stating that the file does not exist. Using an exception allows the program to continue running even after the attempt to open a non-existent file. Loading in players requires new players to be created and added to the vector. Next, the player has his/her points updated and new game tiles added to their hand.

Loading in the board calls upon a read row function in the board class. The read row method takes in one row at a time and sets the board for visualization. For the gameplay to actually go forward, the vector of board positions needs to be updated. For every tile in the row a new board position is created and added to the vector. With this the gameplay can resume from the previous state and will be able to recognize valid moves.

Bag load in requires each tile to be recreated and added back to the linked list of the pre-created bag. Finally, the current player or the player who goes first is loaded in. In order to do so, first the player is grabbed from the save file. Then the player is searched for within the vector of players and only if the player exists, the current player is set and returned.

**Testing**

The FileIO class is essential in testing. If not creating a new game in the testing, we use a .save file to load in a current state of the game. Not only, does the testing become more efficient but it allows the load function to be tested repeatedly. Efficiency is improved because there are no large number of repeated operations being carried out. Hence the time taken to test is decreased for each test. Hence the test cases for purely unit testing, saving and loading contributes to every other test as well. Without the unit tests passing all other unit tests that use automated game states would fail as well.