Battle Boggle

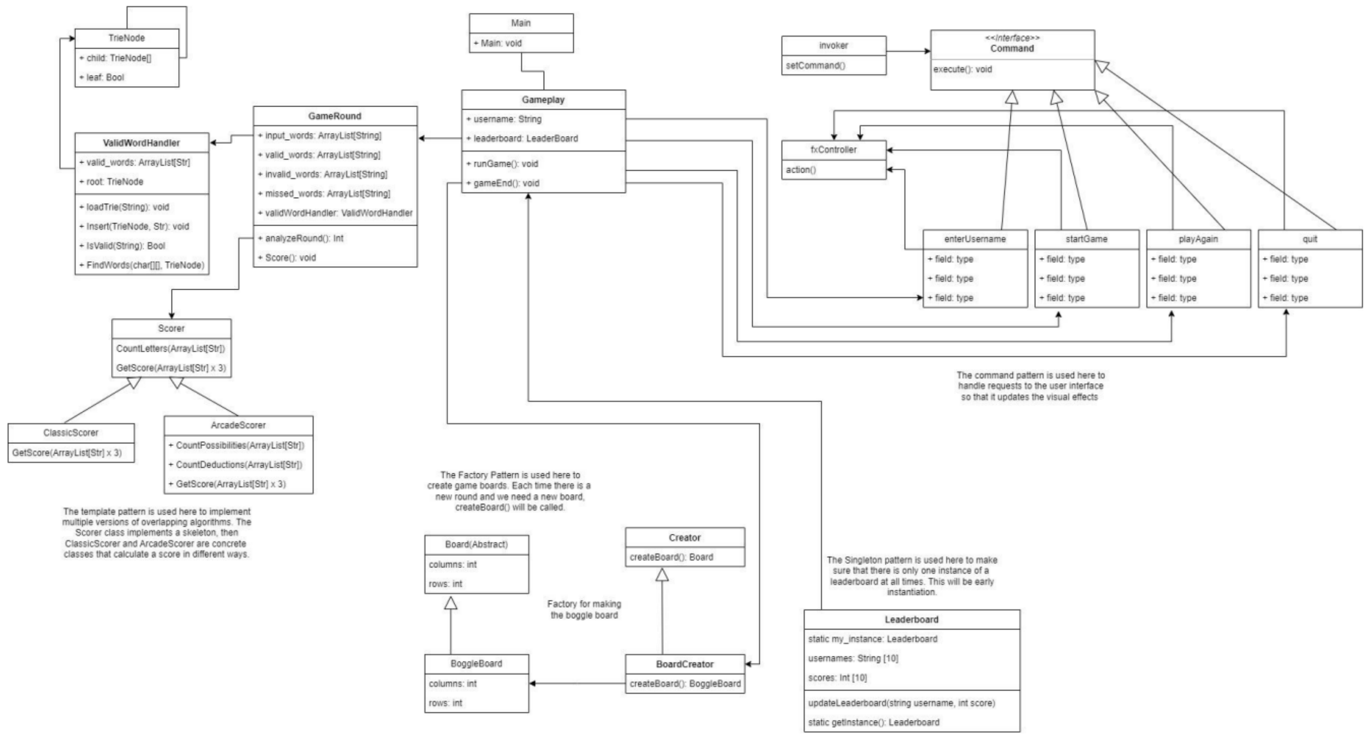
Blake Hamilton, Matthew McDermott

Final State

Most of the features that we originally planned in project 5 and 6 were successfully implemented. The piece that was not fully implemented was the user interface: we had lots of trouble getting the JavaFX library to work how we wanted. Although the user interface was not completely implemented, all functionality (business logic, back end) was successfully completed. The piece that was not implemented in the UI was the leaderboard page, but as we are using a .csv file to store the leaderboard data, it is easy enough to show the leaderboard without the actual leaderboard page. The rest of the user interface (welcome screen, game board, user input) was implemented. There were a couple twists in the implementation (mostly switching from an Observer pattern to an MVC pattern in the first sprint), but the final state of the app is what we were aiming for. We implemented a modified twist on the classic Boggle game that presents itself as a one-player arcade game. You are scored based on the correctness of your words, all possible words, and a classic Boggle score. The most important backend goal was to create an algorithm to find all possible words in a Boggle board in an efficient enough manner so that the user did not have significant wait times, and we did this successfully. We also implemented two different types of board (different dice possibilities): standard and original (Boggle switched its dice in 2008). As another little feature, we show the user what their standard Boggle score is after each round, not just the arcade score. The only change between project 6 and 7 was adding the original board instead of a more difficult board. We thought this was a more enjoyable second option, as coming up with a more difficult board proved very challenging (it was a letter frequency statistics problem that was non-trivial).

Final Class Diagram and Comparison Statement

Original Class diagram:


TrieNode
+ child: TrieNode[]
+ leaf: Bool
ValidWordHandler
+ valid_words: ArrayList[Str]
+ root: TrieNode
+ loadTrie(String): void
+ Insert(TrieNode, Str): void
+ IsValid(String): Bool
+ FindWords(char[][], TrieNode)
GameRound
+ input_words: ArrayList[String]
+ valid_words: ArrayList[String]
+ invalid_words: ArrayList[String]
+ missed_words: ArrayList[String]
+ validWordHandler: ValidWordHandler
+ analyzeRound(): Int
+ Score(): void
Main
+ Main: void
Gameplay
+ username: String
+ leaderboard: LeaderBoard
+ runGame(): void
+ gameEnd(): void
invoker
setCommand()
<<interface>>
Command
execute(): void
fxController
action()
enterUsername
startGame
playAgain
quit
+ field: type
Scorer
CountLetters(ArrayList[Str])
GetScore(ArrayList[Str] x 3)
ClassicScorer
GetScore(ArrayList[Str] x 3)
ArcadeScorer
+ CountPossibilities(ArrayList[Str])
+ CountDeductions(ArrayList[Str])
+ GetScore(ArrayList[Str] x 3)
Board(Abstract)
columns: int
rows: int
BoggleBoard
columns: int
rows: int
Creator
createBoard(): Board
BoardCreator
createBoard(): BoggleBoard
Factory for making the boggle board
Leaderboard
static my_instance: Leaderboard
usernames: String [10]
scores: Int [10]
updateLeaderboard(string username, int score)
static getInstance(): Leaderboard

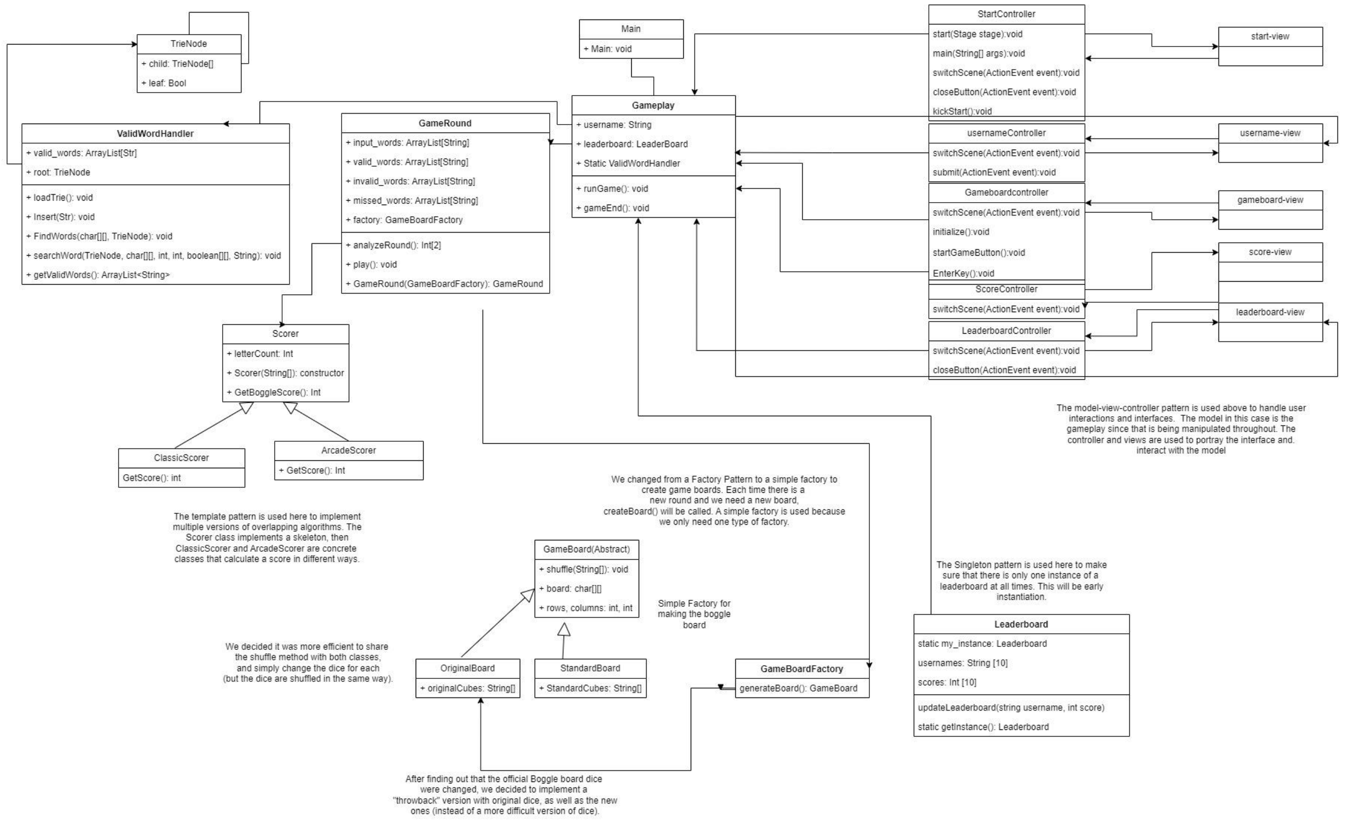
The command pattern is used here to handle requests to the user interface so that it updates the visual effects

The Factory Pattern is used here to create game boards. Each time there is a new round and we need a new board, createBoard() will be called.

The template pattern is used here to implement multiple versions of overlapping algorithms. The Scorer class implements a skeleton, then ClassicScorer and ArcadeScorer are concrete classes that calculate a score in different ways.

The Singleton pattern is used here to make sure that there is only one instance of a leaderboard at all times. This will be early instantiation.

Final class diagram:
https://drive.google.com/file/d/1sKRM1dBjTLxuZqJTSFB7T8mkIMNQ833p/view?usp=sharing


TrieNode
+ child: TrieNode[]
+ leaf: Bool
ValidWordHandler
+ valid_words: ArrayList[Str]
+ root: TrieNode
+ loadTrie(): void
+ Insert(Str): void
+ FindWords(char[][], TrieNode): void
+ searchWord(TrieNode, char[][], int, int, boolean[][], String): void
+ getValidWords(): ArrayList<String>
GameRound
+ input_words: ArrayList[String]
+ valid_words: ArrayList[String]
+ invalid_words: ArrayList[String]
+ missed_words: ArrayList[String]
+ factory: GameBoardFactory
+ analyzeRound(): Int[2]
+ play(): void
+ GameRound(GameBoardFactory): GameRound
Main
+ Main: void
Gameplay
+ username: String
+ leaderboard: LeaderBoard
+ Static ValidWordHandler
+ runGame(): void
+ gameEnd(): void
StartController
start(Stage stage):void
main(String[] args):void
switchScene(ActionEvent event):void
closeButton(ActionEvent event):void
kickStart():void
usernameController
switchScene(ActionEvent event):void
submit(ActionEvent event):void
Gameboardcontroller
switchScene(ActionEvent event):void
initialize():void
startGameButton():void
EnterKey():void
ScoreController
switchScene(ActionEvent event):void
LeaderboardController
switchScene(ActionEvent event):void
closeButton(ActionEvent event):void
start-view
username-view
gameboard-view
score-view
leaderboard-view
Scorer
+ letterCount: Int
+ Scorer(String[]): constructor
+ GetBoggleScore(): Int
ClassicScorer
GetScore(): int
ArcadeScorer
+ GetScore(): Int
GameBoard(Abstract)
+ shuffle(String[]): void
+ board: char[][]
+ rows, columns: int, int
OriginalBoard
+ originalCubes: String[]
StandardBoard
+ StandardCubes: String[]
GameBoardFactory
generateBoard(): GameBoard
Simple Factory for making the boggle board
Leaderboard
static my_instance: Leaderboard
usernames: String [10]
scores: Int [10]
updateLeaderboard(string username, int score)
static getInstance(): Leaderboard

The model-view-controller pattern is used above to handle user interactions and interfaces. The model in this case is the gameplay since that is being manipulated throughout. The controller and views are used to portray the interface and interact with the model

We changed from a Factory Pattern to a simple factory to create game boards. Each time there is a new round and we need a new board, createBoard() will be called. A simple factory is used because we only need one type of factory.

The template pattern is used here to implement multiple versions of overlapping algorithms. The Scorer class implements a skeleton, then ClassicScorer and ArcadeScorer are concrete classes that calculate a score in different ways.

The Singleton pattern is used here to make sure that there is only one instance of a leaderboard at all times. This will be early instantiation.

We decided it was more efficient to share the shuffle method with both classes, and simply change the dice for each (but the dice are shuffled in the same way).

After finding out that the official Boggle board dice were changed, we decided to implement a "throwback" version with original dice, as well as the new ones (instead of a more difficult version of dice).

Patterns used: Template, MVC, Singleton, Simple Factory (more details in diagram).

As seen in the diagrams, the big change between our original class diagram and the final model was the use of an MVC pattern. We originally thought that using JavaFX would fit well into an Observer pattern, but we quickly realized that the JavaFX library is made to be a Model-View-Controller pattern. Although we did not get the user interface working well, the structure we were attempting to implement is possible and we believe the best way we could have gone about it. A smaller change we made was changing the variations of the Simple Factory (more detail in "Final State" section).

Third-Party Code vs Original

The biggest place where third-party code is used is with JavaFX and SceneBuilder, as these were used to create the GUI. A Trie algorithm for searching all possible words in a Boggle board was used from GeeksForGeeks. The algorithm pieces were cut up and used in different functions and classes, and any place they were used is cited in the code. This was very important to the project, as the runtime without using the trie structure would have been extremely slow. Any other third party code use was very minimal (i.e. Stackoverflow), not more than a couple lines, and cited in our code.

- https://openjfx.io/
- https://gluonhq.com/products/scene-builder/
- https://www.geeksforgeeks.org/boggle-using-trie/

Overall Process Statement

1. The biggest issue in our project was the use of JavaFX and SceneBuilder. We had a very hard time getting SceneBuilder to actually build components that worked in our code. We wouldn't be opposed to trying to use it again, but development with it was very slow and clunky, and we were not able to get the full functionality that we were hoping for in the user interface.
2. Using a CSV file to persist data worked very well. We are not experts with SQL or any other database, but it was very easy to write simple data (our leaderboard) to the CSV and then get that data the next time the app starts up. We would certainly use this technique in the future, as it served the purpose we needed without using databases.
3. Our general approach was to develop the front end and the back end independent of each other, then mix them together in the second sprint. This did not work well, even without considering the problems we had with the JavaFX. It would have been much smoother to grow the project together from the beginning, instead of making two separate pieces and mixing them together at the end of the project. It might have created more merge conflicts and more problems with version control flow, but this would have been more manageable than the problems we ran into when trying to mix the front end and back end.