Blake Hamilton, Matthew McDermott
Boggle Battle

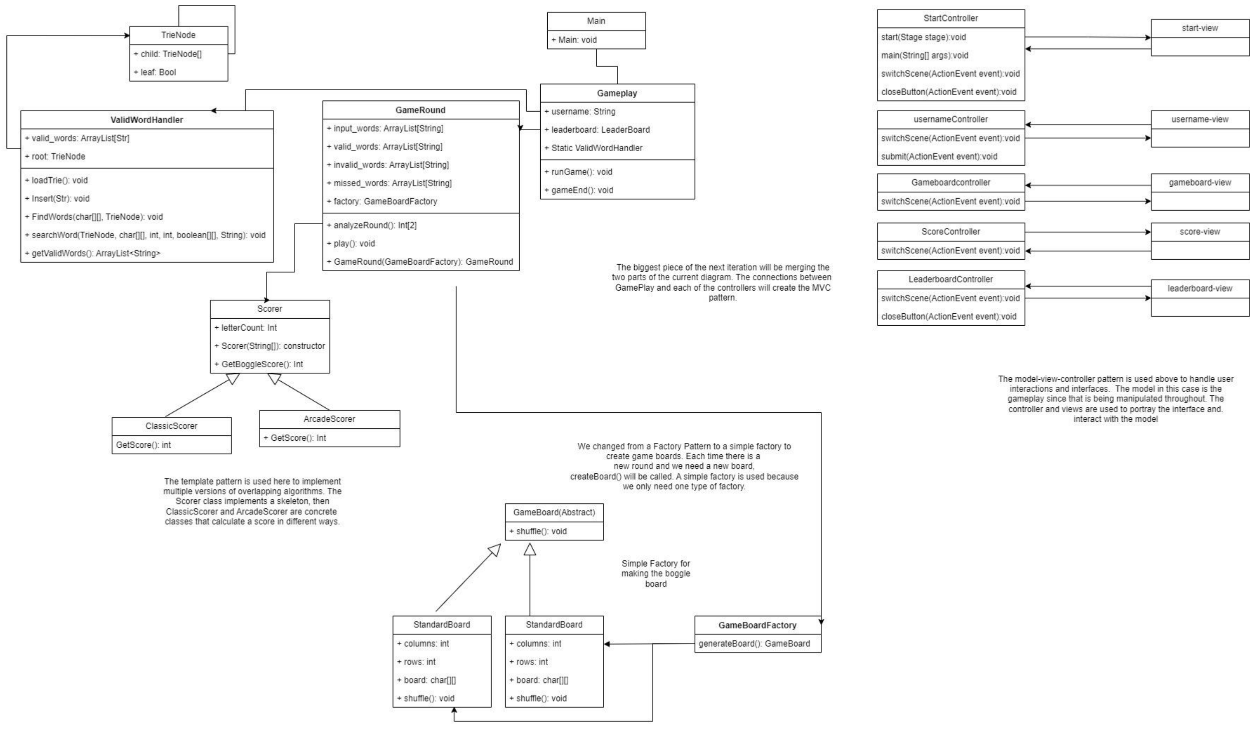
Status Summary

- **Work done:**
  - Blake - Game play has been set up to create a new round object each new round. This object creates a Trie object with all words in dictionary that will persist while app is open, a very efficient implementation for word validation.
  - Blake - Algorithm/classes to calculate every word possibility in a Boggle board using a dictionary with 50,000 words. The result of this is to have a list of valid user words, invalid user words, and missed words. These three lists will be used to create a normal score and an arcade score for each round.
  - Blake - Create scoring classes to create standard Boggle score and Arcade score
  - Blake - Implement a factory to create boards (both standard and difficult). The dice shuffling algorithm has not been implemented yet, but everything else has.
  - Matthew - helped restructure the files so that existing files would be inside a javafx project instead of being outside of it
  - Matthew - Started implementing the MVC pattern using controllers, views, and a model using javafx.
  - Matthew - Got all the basics of each view to work so that user can see each the interface.
- **Changes/Issues encountered:**
  - Some issues we encountered was figuring out how to organize the existing backend files with the javafx since the backend files were originally in a java project and not a javafx project
  - Another issue was setting up javafx and scenebuilder since my computer was not liking me edit the fxml files initially but after messing around with it for a little bit it was fixed.
  - A change we decided to make was to use the mvc pattern instead of the command pattern for dealing with the user interface.
- **Patterns:**
  - We implemented the factory pattern to create game boards. The creation of new objects is removed and abstraction is used to create two different types of boards

- Implemented template factory to efficiently have two different methods to generate two different types of scores (standard and arcade) which reuse algorithm code through the template pattern so there is no duplication.
- We implemented mvc for handling the user interface(views) and how it interacts(controller) with the backend parts like the score and words entered(model).

Class Diagram

[Link to high resolution diagram]()


TrieNode
+ child: TrieNode[]
+ leaf: Bool
Main
+ Main: void
Gameplay
+ username: String
+ leaderboard: LeaderBoard
+ Static ValidWordHandler
+ runGame(): void
+ gameEnd(): void
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
The biggest piece of the next iteration will be merging the two parts of the current diagram. The connections between GamePlay and each of the controllers will create the MVC pattern.
Scorer
+ letterCount: Int
+ Scorer(String[]): constructor
+ GetBoggleScore(): Int
ClassicScorer
GetScore(): int
ArcadeScorer
+ GetScore(): Int
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.
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.
GameBoard(Abstract)
+ shuffle(): void
Simple Factory for making the boggle board
StandardBoard
+ columns: int
+ rows: int
+ board: char[][]
+ shuffle(): void
StandardBoard
+ columns: int
+ rows: int
+ board: char[][]
+ shuffle(): void
GameBoardFactory
generateBoard(): GameBoard
StartController
start(Stage stage):void
main(String[] args):void
switchScene(ActionEvent event):void
closeButton(ActionEvent event):void
start-view
usernameController
switchScene(ActionEvent event):void
submit(ActionEvent event):void
username-view
Gameboardcontroller
switchScene(ActionEvent event):void
gameboard-view
ScoreController
switchScene(ActionEvent event):void
score-view
LeaderboardController
switchScene(ActionEvent event):void
closeButton(ActionEvent event):void
leaderboard-view
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

Plan for next iteration

The major thing we need to do is merge the UI of the app with the back end. We have the JavaFX primarily set up, and the backend business logic is also complete. We need to implement the Model View Controller, and merging these two pieces of the project will form the complete MVC design pattern. This could take a significant amount of time since neither of us has used JavaFX or created a GUI before. We also need to implement the leaderboard object and write the leaderboard to a file (singleton pattern), and this should be fairly easy. For the user interface, the only big thing we need to do is implement a timer for the user round time. Although we changed a couple things in the first iteration, we don't plan to change anything from our original plan for the next iteration. Our final deliverable will be a working GUI Boggle game by the end of the next iteration. Our goal is to make the app cross platform, but it may only be Mac compatible.