Quick Start
To create a new game, users must first import the engine.
An example of a basic setup:

```
from engine import *

class MyGame(engine.Game):

        def __init__(self, rows=10, columns=10):
                self.board = engine.Game.generateBoard(rows, columns)
                self.player = engine.Player("", "", 0)
                super().__init__(list([self.player]), self.board, None)
                self.timer = Timer()
                # … game specific initialization
```

Classes and Properties

Class Board

Board.grid
- Holds the grid the 2x2 grid object for the game

Board.getBoardState()
- Method that returns what the current grid looks like

Board.displayBoard()
- Method that prints what the current grid looks like

Class Game

Game.players
- A list of players who are participating in the current game

Game.gameBoard
- Stores the instance of Board in this variable

Game.timer
- Stores the instance of timer in this variable

Game.events
- A FIFO queue that stores the game events

Game.generateBoard(rows: int, cols: int)
- Creates a new instance of Board and stores it inside the gameBoard variable

Game.addPlayer(newPlayer: Player)
- Adds an instance of the player to the playerList

Game.handleEvents()
- Pops an event from the events queue and handles that event

Class GameEvent

GameEvent.event_action
- Stores the lambda function that gets ran when we handle that event

GameEvent.execute()
- Executes the function stored in event_action. This is used by handle events inside Game.

Class GameRuleEvent(GameEvent)

GameRuleEvent.rule
- Stores the name of the game rule

Class PlayerInputEvent(GameEvent)

PlayerInputEvent.player_input
- Stores the name of the player input used

Class Player

Player.username
- Stores the username string for the player

Player.password
- Stores the password string for the player

Player.score
- Stores the score for that player

Player.get_username()
- Returns the username for that player

Player.update_score(new_score: int)
- Updates the score with a new replaced score.

Class Tile

Tile.row
- Stores the current row the tile is on

Tile.col
- Stores the current column the tile is on

Tile.point_value
- Stores the amount of score that tile will yield the player.

Class Timer

Timer.tick
- Stores the current tick the timer is on

Timer.tick_rate
- Stores the tick rate of the timer

Timer.is_running
- Boolean value indicating whether the timer is running or not

Timer.start()
- Starts the timer

Timer.stop()

- Stops the timer

Timer.reset()

- Resets the timer

Timer.update()

- Increments the tick counter by the tick_rate of the game

Timer.set_TickRate(rate: int)

- Changes the tick rate to a new tick rate

High Points

- We were able to work together to quickly create the UML diagram and begin work on coding the engine and our individual games.
- We all knew how to use github so we were able to efficiently create branches, pull requests, and merges.
- We were able to work well as a team to work together to create the engine and then sub teams to develop the different games.

Low Points

- We got stuck trying to figure out how to implement the game engine into actual games, since none of us had experience with developing games or using a game engine.
- We had issues with debugging our games, due to the player having the controls they were able to do a bunch of different actions that could create a bug that breaks the game.

Major Challenges

- At the beginning of the project we were unorganized and were unable to evenly distribute tasks. So we had to start using Jira to help keep track of completed tasks and assigning tasks.
- We struggled with debugging because at first we were manually playing the game to test for bugs but then we decided to write small test cases in our code to test the code whenever we run it to help quickly see and fix issues.

For our final design, we removed the Engine class, the function updateDisplay() from the Game class, and made Game Events an implementable interface. We also changed how the Tile class implemented its value. With the engine class, its removal was a result of a newfound understanding of the project, where the entire system itself was the engine and not a separate class that contained the different classes, which was why we removed the Engine class. Within the Game class, the function updateDisplay() was removed due to a similar reason, as we believed the display would be game-dependent, and thus each game would have a specific way of displaying, resulting in the removal of the updateDisplay() function. As for the Game Events, we wanted to ensure that any events that can happen within the game were registered and classified. Because of this change, Game Events was transformed from a class to an interface and implemented into two separate classes, PlayerInputEvents and GameRuleEvents.