TGME Final Project Retrospective

INF 122 Team 9

# Team Members

Ace Jack Lowder

Vivian Nguyen

Martin P. Murata

Haoming Zhou

Anh Ngoc Lan Nguyen

Huy Luong

Nha Truc Vuong

Jonathan Tran

# Step-by-step Instructions

The Tile-Matching Game Environment, or TGME, comes packaged as a set of six classes:

1. Game Logic
   1. Grid
   2. Tile
2. Custom Structures
   1. Coordinate
   2. Direction
3. User Interface
   1. Game
   2. UserProfile

This package has been optimized to be a foundation for tile-matching games that require a fixed grid to act as the playing board, tiles that can match with other tiles and disappear, and tiles that can move and shift due to user control.

To implement a custom tile game on top of our interface, the user must first build the game logic of the grid and tile piece.

1. Grid
   1. The grid can be specified to have a fixed width and height.

E.g: 2048 has a 1:1 grid ratio that is defaulted to a 4x4 grid, and Same-Game typically has a 5x5 grid.

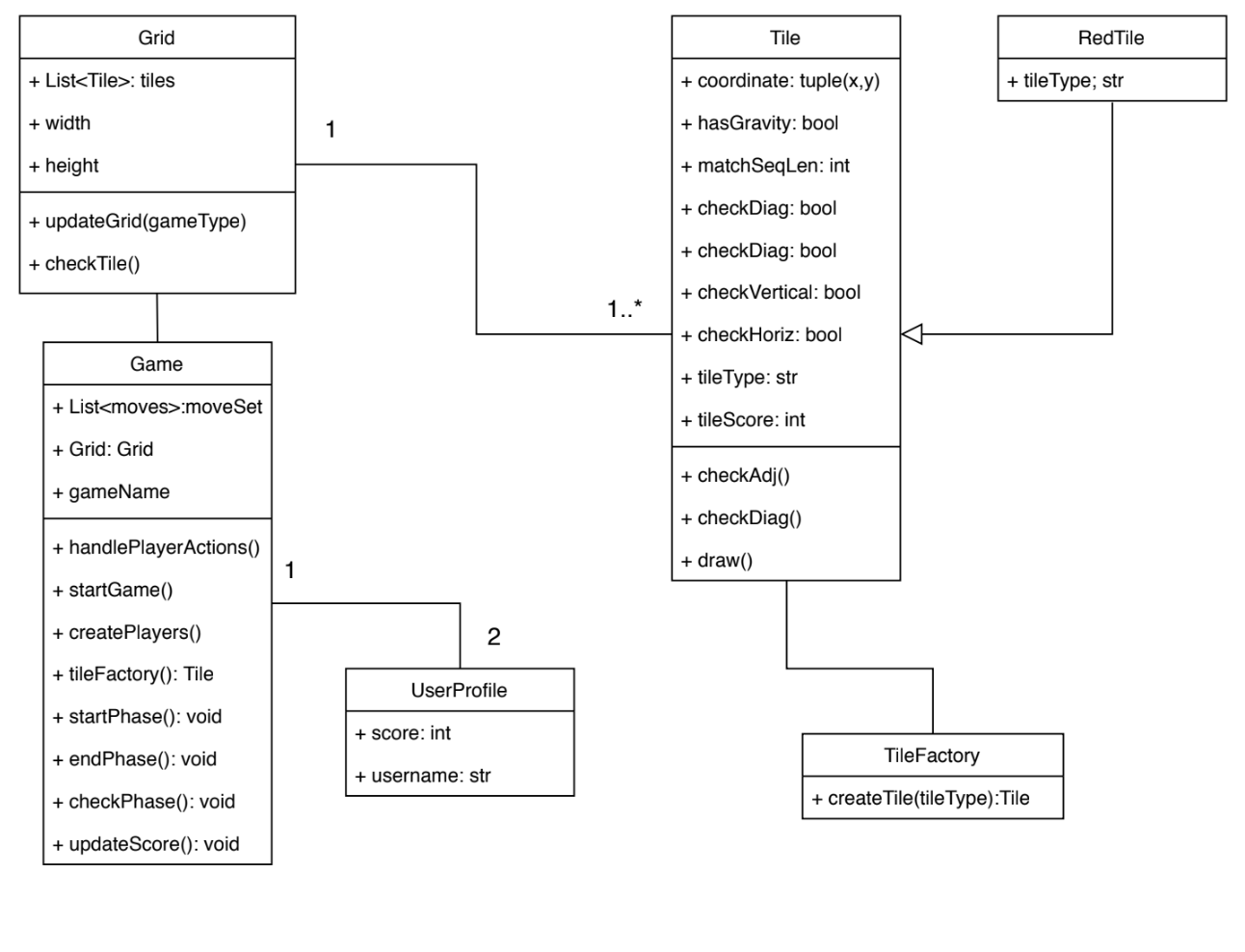
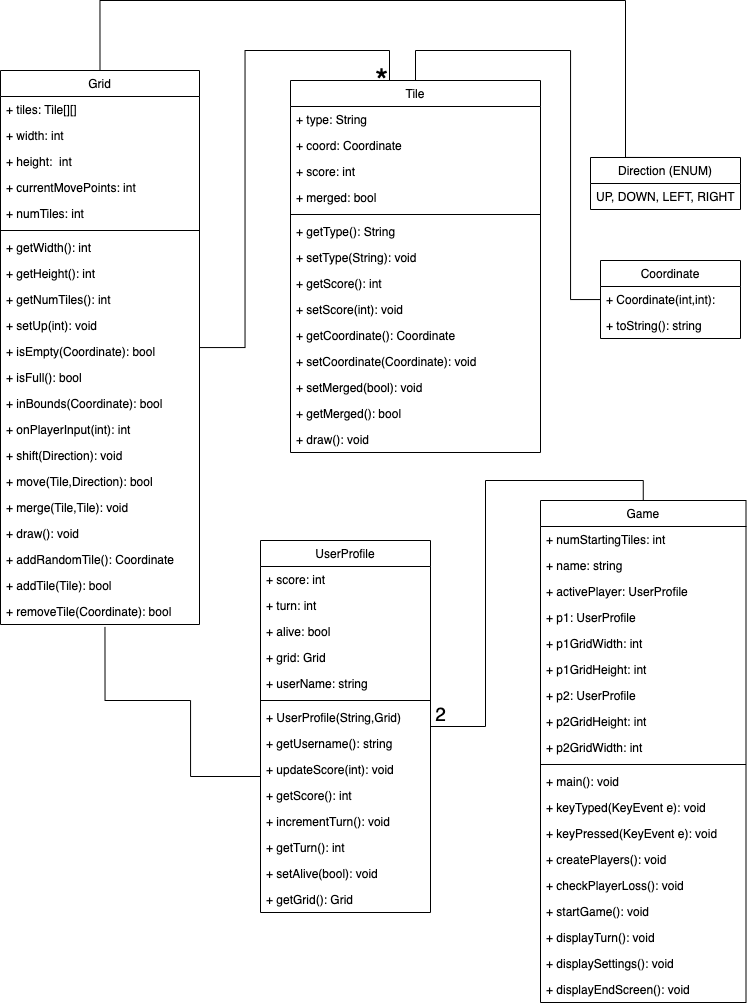
* 1. Each grid keeps track of whether the grid is full/has reached max. capacity. This is useful for many games that require the board filling up in order to determine win/lose logic.
  2. Each grid has an isEmpty and inBounds method that can take a set of coordinates to determine if the space on the grid is either empty or within the bounds of the board. This is useful in determining win/lose logic as well as tile spawning mechanics. E.g Running out of room for Dr. Mario and Tetris tiles.
  3. Each grid also handles Tile movement via the onPlayerInput, shift, and move methods. onPlayerInput handles four directions via arrow keys at the very minimum, but can be overridden to implement other controls such as rotations. Shift determines how each tile can move; in 2048, controls are locked to four directions, and each input can have cascading effects (all tiles move over).
  4. Each grid can clear matching Tiles via the delete function, so in the Same Game, players can clear matching Tiles with the same character in every turn until the whole grid is clear.
  5. Each grid can also handle Tile drop-down after matching Tiles are cleared, which is necessary for some games, such as the Same Game or Tetris.

1. Tile
   1. Each tile has a coordinate and score/value attribute. Tiles that occupy multiple grid spaces can be implemented on top of the interface by creating a collection of tiles with different coordinates.
2. Coordinate/Direction
   1. Coordinate is a simple structure that we created to hold a pair of Integers, signifying the row and column that a tile occupies within a board. Each tile can be assigned a default coordinate or a random coordinate to signify a random spawn.
   2. Direction is just an enum of constants that hold possible movements that a tile can perform. Additional movements such as rotations can be added on an as-need basis.
3. Game
   1. Game acts as the “main menu” in which a user can specify which game they want to play and whether it’ll be single-player or multiplayer. Win/loss actions that are specific to each game can be overridden and implemented in this class.
   2. In this TMGE, there are two games which are 2048 and the Same Game, so players can choose one game before entering the username to be able to play the game. For 2048, users use the arrows key to move and add up tiles. With Same-Game, users use the arrows to move between tiles and the “space” key to kill titles of the same character.
4. UserProfile
   1. UserProfile keeps track of each user currently playing a tile game. Every user has a username to act as a “login,” as well as game attributes such as score, whether it is their turn or not, whether they are alive, and which game they are currently playing.
   2. The alive attribute is meant to be altered depending on the desired tile game. In some games, alive would mean that the user has not placed a tile out of bounds e.g. Tetris and Puzzle Bobble while other games would require the user to not run out of moves e.g. Candy Crush. Regardless that functionality is meant to be extended from this class.

Although these class methods have default logic implemented into them, many are meant to be overridden to accommodate each specific game.

# Design Evolution

# 

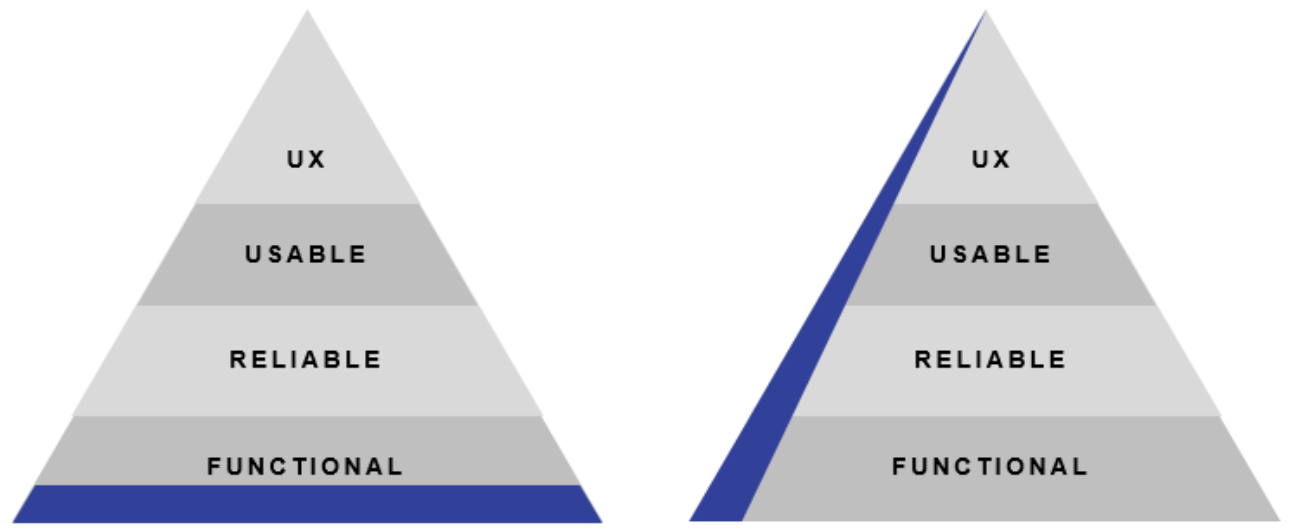
1. Initial UML
   1. 
   2. Shortcomings:
      1. Needed a custom structure to hold our Tile coordinates.
      2. Matching/merging logic in the Tile needed information about other tiles on the board.
      3. Game needed to support two-player games and multiple other games.
      4. Win/Loser mechanics were not present anywhere.
      5. Factory Design Method for Tiles was not efficient because most tiles share very similar attributes, but we could not generalize it all into one category.
      6. Creating a tileType for every type of tile was not necessary because most tile matching games have a grid of homogenous tiles.
      7. Player Movement is not accounted for.
2. Revised UML
   1. 
   2. Revisions:
      1. The grid should be handling match/merge logic because it holds the information of all tiles currently on the grid.
      2. Created a custom structure for our Tile coordinates and possible Tile movements.
      3. UserProfile is elaborated to hold login information and attributes needed to determine win/loss functionality such as alive.
      4. Grid is completely redesigned to focus functionality on what tiles exist on the grid and what that type of information provides. Grid also handles Tile movement and matching/merging functionality.
      5. Game provides the functionality to play different games, multiplayer games, and handles interface movement.
   3. As with all designs, this UML will perpetually continue to be altered as more tile-games are produced and more revisions/optimizations can be made.

# High Points

# 

1. Discovering patterns and similarities
   1. Building a TMGE forced us to think more generally/abstractly and to get away from implementations and ideas that were too ad hoc. Deciding on a better baseline TMGE architecture with versatility in mind, allowed the implementation process to only build on, and not conflict with the existing design.
2. Building the game
   1. Thinking more abstractly led to a smoother implementation. Building the game went fairly quickly. It was interesting to see the end result working in a short amount of time, especially since this would not be how some of us would normally implement a game.

# Low Points/Major Challenges

1. Design Challenges
   1. Generalization vs. Specialization
      1. Our largest design challenge was deciding what all tile-matching games share in common, and whether those attributes can be generalized into an interface for the TGME.
      2. Designing a grid was relatively easy because most tile games simply had a fixed height and width, but match sequences and game mechanics for each grid varied heavily and we were unsure of what approach to take. Games like Tetris and Dr. Mario begins with an empty board and has tiles constantly spawning whereas games like Candy Crush and Bust-A-Move begin with the board full/semi-full and have different spawning mechanisms.
      3. We decided the best approach was to provide the TGME with only the most general information in order to not limit the type of games that could be made, even if that meant that the game implementations would have to be more in depth.
   2. Building the Minimum Viable Product (MVP) the wrong way
      1. A large lesson we learned during development was that developing an MVP with both functionality and the interface simultaneously was much more productive than trying to build the complete project in stages.
      2. We began with designing only the back-end, and focusing on elaborating on the game mechanics and logic before moving forward with the interface and etc. This caused us to not only have stage dependencies, but also we lost sight of other requirements that would have required changes to our backend anyways. We were building the functionality blindly by not accounting for the whole system.
      3. We then switched to a new approach: we would build the games first and then extract what was shared amongst them. This way, we learned to account for user input and how we should design our classes to handle requirements such as multiplayer games. Designing user input and display functionality also helped us improve iterative testing so that our less abstracted logic was more accurate.
2. Project Challenges
   1. Group work on a small subset of code
      1. With a large group of people, it was difficult to work on designing the TGME and the games, especially during the early stages of our project.
      2. Our UML and TGME were not finalized and we continuously made revisions which slowed down how fast and accurately we could work. We found that we had to constantly refer back to the TGME and its updates because there were optimizations that could be made. Ultimately this was a plus because our TGME can accommodate for much more and our games have the same guideline/structure that we made the TGME to specify.