Final Project Design Document

Luis E. Morales

Cristy A. Gonzales

Kyle K. Lam

Michael Dinh

Nico Platt

Overview

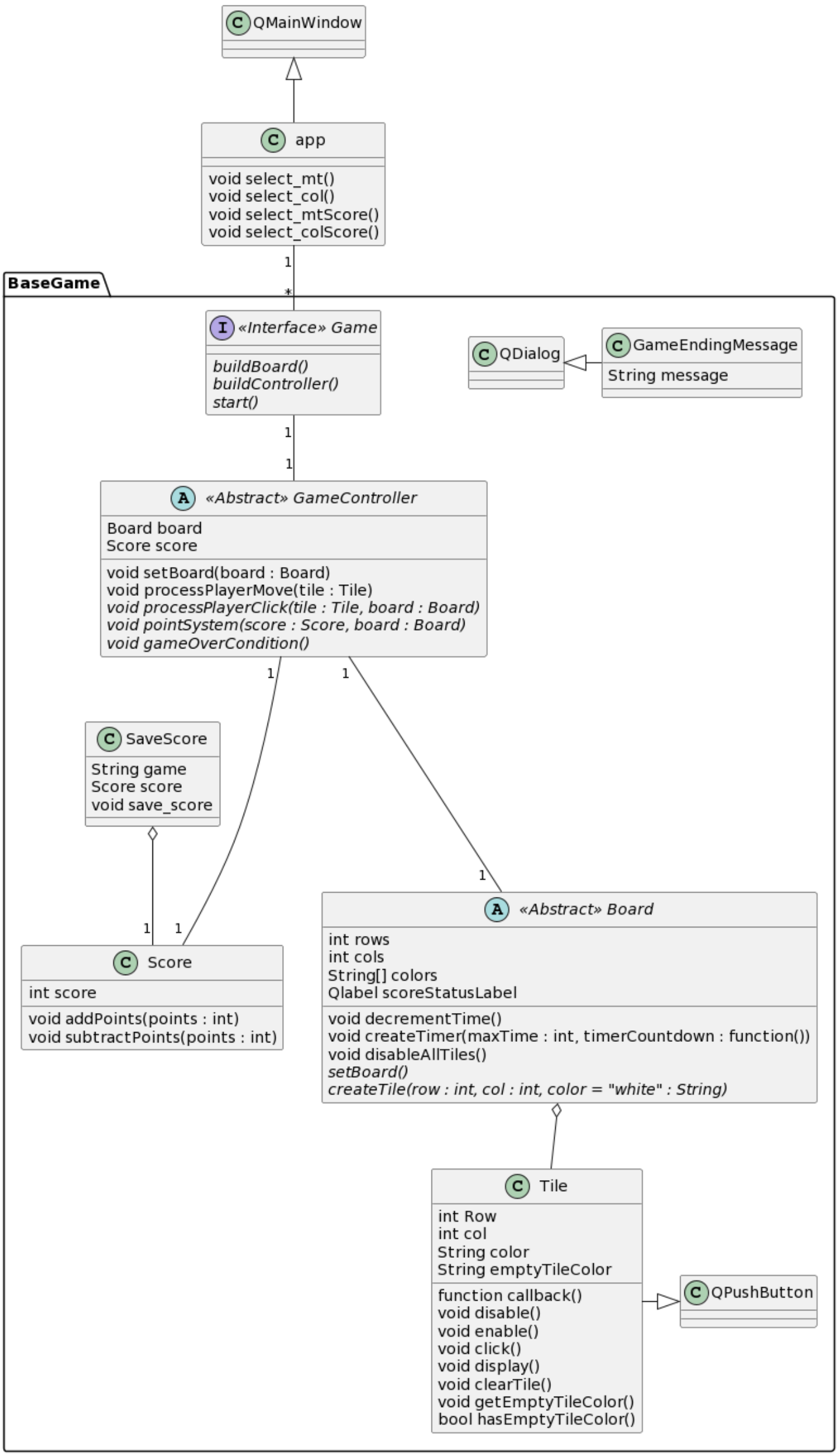
Our Tile Matching Game Environment (TMGE) is a collection of interfaces and files that can be used to create tile-matching games in the PyQt framework. By following our design, a user should be able to create a tile game, implement game conditions, optionally incorporate timer and scoring systems and define end states easily. Such an environment is accomplished by having a collection of components that are documented, interfaced, and comprehensive enough to support multiple Tile Matching Games. In particular, we have used our environment to implement a modified version of Columns and a memory-matching tile game. Given sufficient coding experience, a user could use our environment to create a Tile Matching Game such as Bejeweled. Like memory game, Bejeweled would start the game with all tiles populated, and it would allow for the selection of two tiles to compare, with the additional functionality of switching the two selected tiles. It would then incorporate tile falling logic from columns, which will occur after matches are removed, falling both present tiles, along with newly generated tiles at the top of the board.

Our implementation of Columns abides by slightly different rules than the original. Players are able to click on any tile on the board, and once clicked, the column of 3 tiles from the top right indicator will be dropped into the corresponding column the clicked tile is located in. Unlike the original Columns game, players are unable to shift the order of the queued column. To gain points, players must match 3 or more tiles in a row together. Vertical, horizontal, and diagonal lines are considered. Once a line of 3 or more tiles is registered, the tiles in the line disappear, and the tiles above it will drop. If another line is formed, the appropriate points are added, and the new line is also removed. These steps are repeated until there are no newly generated matched lines. The game ends when the player runs out of time, or an entire column is filled past the top of the board.

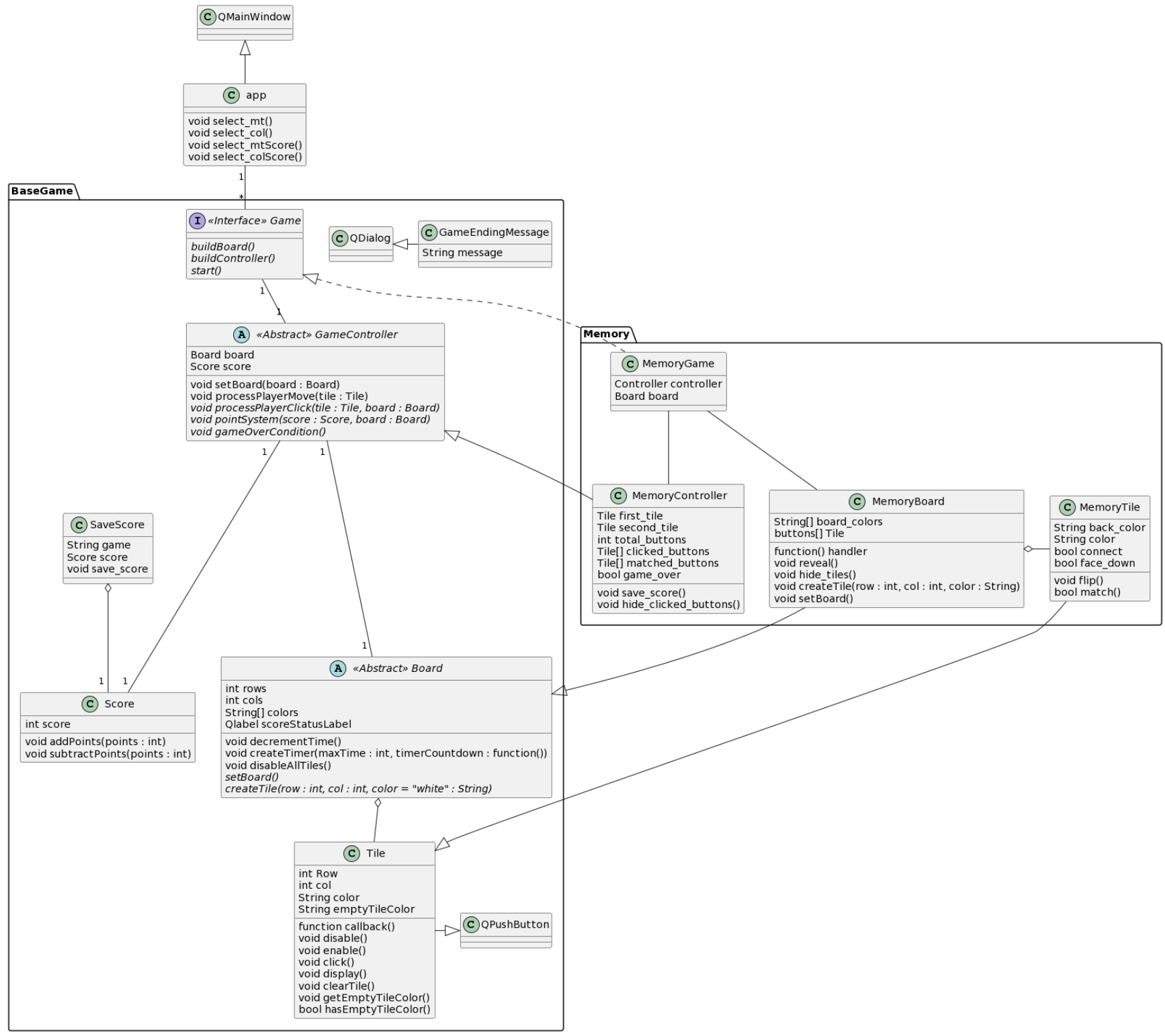
Our implementation of Memory Game functions like a matching game. At the start of the game, all tiles are revealed for a second, and are hidden afterwards. Players must choose 2 tiles, and if they match, the tiles are cleared and appropriate points are added. The game ends when the player runs out of time, or all tiles on the board are matched and cleared. If a player clears the board with time to spare, the remaining number of seconds is added to their score, incentivizing faster gameplay.

UML

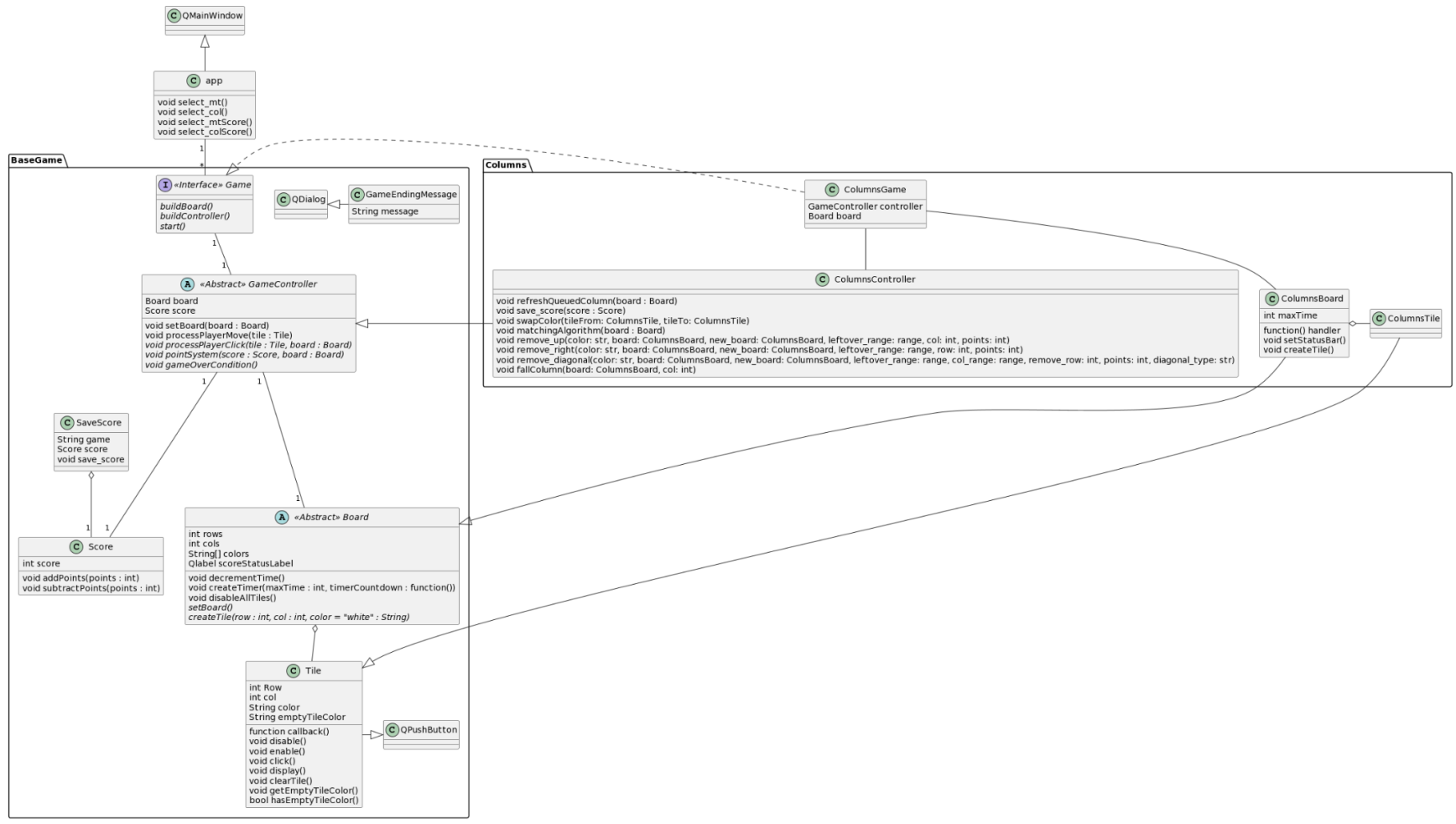
BaseGame Environment



BaseGame Environment and MemoryGame



BaseGame Environment and ColumnsGame



UML Description

Our environment utilizes the PyQt5 framework, which allows for basic GUI functionality. We've adapted core pieces of the PyQt5 framework to represent game components, such as tiles or a board. To that end, much of our environment extends PyQt5 components. In our UML, we've included stubs (for example, QPushButton) to signify that these classes are not part of our own environment but are extended in our environment. This reduces the need for clutter in our UML - we do not use all parts of a given PyQt5 component and are only interested in how it is used in our game.

Our environment is represented by the BaseGame module. This module contains the classes that need to be extended and/or implemented as well as some optional components that we created to help implement our selected games. In the following table, the term "variant" will be used to describe a component or functionality that can vary between games.

Python File	Description
App	The App class is the entry point into the entire environment. It is a main menu that features buttons to trigger the various games and game scoreboards created using our environment.
Board	The Board is an extension of the PyQt5 QGridLayout class. It is the visual representation of the game, and each prospective game in our environment uses a Board implementation. A board is fixed to always be a collection of tiles, but can have additional functionality in a given game's implementation of it.
GameController	The GameConditions class handles game logic events, such as a scoring system or the disappearing of tiles. A new game needs to: 1. Define the changes of the board that occur once a tile is clicked 2. Define how points are calculated 3. Determines whether or not the game is over These three steps are executed in the above order once a tile is clicked. These actions can be implemented in any appropriate manner, but they exist in all games using our TMGE.
Score	The Score class is a small class used to keep track of points. It is optional and is considered variant.
Tile	The Tile class is a key component. Instances of Tiles are placed on the board. Each tile has two possible states: cleared and populated. When a tile is cleared, it is colored white, and is considered an empty section of the board. When a tile is populated, this means that the specific section of the board that the tile is located in has a game tile in it that can be interacted with. Games which call for additionality functionality would need to extend this class.
GameEndingMessage	The GameEndingMessage just shows a message to the user when the game is over. This is an optional component and thus is variant.
SaveScore	The SaveScore class inherits from QDialog which provides a dialog window for the user to save their score and include their initials. The user may choose to save their score or cancel. The save_score method writes the score and initials to a .txt file. The SaveScore is an optional component of our environment. Some games may choose not to implement a score, and as such this is variant.
Game	The Game class puts everything together by creating the GUI window and initializing the game's board and controller. This is not variant.

MemoryGame is a tile-matching game that challenges the player to recall as much information as possible. The user views a "face-up" grid of tiles for one second, after which they are challenged to try and find as many matching color tile pairs as quickly as possible. MemoryGame extends the core four components of the TMGE, adding additional functionality. The MemoryBoard adds functionality to "flip" the board. MemoryTile has functions to check if a given tile matches another, as well as methods and attributes to track its state. MemoryController handles all necessary game logic, including tracking the tiles a player has selected and flipping them as necessary.

ColumnsGame is a tile-matching game that challenges the player to match as many tiles by color as possible by "dropping" a 3-tile column of tiles repeatedly onto the game board space. Like MemoryGame, ColumnsGame extends the four basic TMGE components, adding necessary functionality. ColumnsController handles the game logic for "dropping" a tile group and checking for resulting matches. ColumnsBoard contains logic for the tile group preview window.

Game Creation Instructions

Our deliverable contains 4 modules: BaseGame, Columns, Memory, and Scoreboard. Columns and Memory extend the BaseGame core module, which is itself an extension of PyQt5. To create a new game, a user would adapt our core BaseGame files and extend with custom functionality as they see fit. Each class must contain at least one implementation/extension of the classes in the core BaseGame library. Scoreboard is also an extension of PyQt5. It is composed of three files and is primarily used by the driver file, app.py for scorekeeping purposes.

Game Creation Steps

1. Create tile class; inherits from BaseGame.Tile
    a. Additional tile functionality/helper functions can be added
2. Create board class; inherits from BaseGame.Board
    a. Must implement setBoard and createTile functions
        i. createTile creates a single game tile
        ii. setBoard calls createTile and places them on the board's 2D list representation and adds tiles widget to GUI window
    b. If a timer is desired, call createTimer in the constructor
    c. Additional board functionality/helper functions can be added
3. Create controller class; inherits from BaseGame.GameController
    a. Must implement processPlayerClick, pointSystem, gameOverCondition functions
        i. processPlayerClick defines the changes of the board that occur once a tile is clicked
        ii. pointSystem defines how points are calculated
        iii. gameOverCondition determine whether or not the game is over
4. Create game class; inherits from BaseGame.Game
    a. Must implement start, buildBoard, buildController functions
        i. start sets up the appearance of the GUI window
        ii. buildBoard creates the Board object and connects it to the Game object
        iii. buildController creates the Controller object and connects it to the Game object
5. Add buttons to access the game and scoreboard to app.py.

Design Evolution

After deciding our games of choice, we initially began by creating the components from the ground up. While this seemed like the most logical approach, we quickly realized that the games were quite different in their logic. Comparatively, the Memory-Matching game would be much easier to implement, while Columns would require a greater amount of implementation work in order to maintain modularity and SOLID principles. As such, we decided to first work on the Columns game while keeping in mind that we would need to create an infrastructure flexible enough to also incorporate the Memory-Matching implementation.

When the Columns game was being developed, we had to think about features that are shared not only across our two games being implemented, but all tile matching games in general. We determined that these types of games needed a board, tiles, and some sort of logic controller as a baseline. Because of the difference in complexity between the two games, columns became more of the foundation in planning, since we agreed that it would be easier to work on the more complex variant first, followed by adapting the simpler variant.

As development progressed, we found ourselves adding unnecessary classes and functions that detracted from the design, and resulted in a design that was confusing and needlessly complex. Certain functions and classes were being added from columns to the base TMGE, but were not necessarily used in most/all TMGE games. Classes were also depending too much on one another, which contributed to the confusion. To remedy this, we refactored out unused classes, and merged classes that were closely related in order to increase cohesion and reduce coupling.

Our final TMGE module contains 7 classes: Board, Game, GameController, GameEndingMessage, SaveScore, Score, and Tile. The Board and GameController classes are both abstract, with Game and GameController following the template design strategy. GameController delegates the processPlayerClick, pointSystem, and gameOverCondition implementations to the child classes, and the processPlayerMove function is the template function, calling the three functions above. Game delegates the GUI window setup (start function), board building/assignment (buildBoard function), and controller building/assignment (buildController function) to the child classes, and the Game class constructor functions as the template function, calling the above three functions. The Tile and Board classes follow the Open-Closed principle, as the child classes are able to define additional functionality if needed.

High Points

As with any project, there is an inherent high point when seeing combined efforts culminate in a functioning product. For us, this came about when creating a functioning and playable prototype. Though prototypes need more work to become finished products, seeing the game working was a definite high point and milestone in our project.

Low Points

When identifying the features of games that should be included in the TGME, it was difficult to determine whether or not certain parts should be created in the TGME or the individual game itself. Since there was such a drastic difference in complexity between the tile matching game and columns, certain parts of columns' functionality crept into the design of the TGME, when in the grand scheme of things, does not make sense to be included in the tile matching game. Identifying these situations and reeling back the changes made took a while, and even though it made creating columns a bit harder, it resulted in a TGME that made more sense.

Major Challenges

One major challenge we have is adapting to the intricacies of group work. While our project has largely gone smoothly, there have been instances when there have been misunderstandings or miscommunication between members. So far, there has not been a problem that could not be solved with a robust communication system and a willingness to compromise on both sides of the conversation. However, we understand that this is and will continue to be an ongoing risk that the group will have to be proactive in addressing.

Another major challenge came from our choice of coding language. Without additional libraries, Python does not support abstract classes or interfaces, so we struggled to properly implement some of the design patterns we learned in class. For example, we attempted to apply the strategy pattern, using the FallStartegy interface, and the FallColumn class that implemented it. When we set a FallColumn instance variable, and called its fall method, the program would crash without calling the method -- even if the method had no internal logic to cause a crash (ie: only pass statement). After many hours of fruitless debugging, we were forced to rethink our design around these limitations, sacrificing that design pattern in favor of working software. Ultimately, this proved to be a complex challenge, however, with proper teamwork and communication, we were able to procure a working design.

Game Rules and Scoring Scheme

Both games included with our TMGE use a point-based scoring system.

MemoryGame presents a 4-by-4 grid of tiles. When the game is started, each tile is initially face-up, revealing their color. After one second, the tiles are all flipped down, shown as black tiles. The user must select any tile and then its twin. If the user selected correctly such that both tiles match in color, then both the initial tile selected and its twin are rendered white to show as "disappeared." If the user selected incorrectly, then the tiles are returned to the "face-down" position. The user's goal is to match tiles to "disappear" all of the tiles on the board before time runs out.

MemoryGame generates a score upon completion of the game. A score is comprised of the base amount of points for completing the memory game (16 tiles * 5 points = 80 points) with additional points being awarded based on the amount of time remaining. More time remaining after the user completes the round results in a higher score. If the user fails to complete the game, they are awarded a score of 0.

ColumnsGame creates a large board, full of "empty" tiles. At the right side of the screen, a column of 3 tiles is shown. This is the next "incoming" tile group to be placed. The user then selects any tile on the grid, choosing the column of the chosen tile. Once the user has selected a tile on the grid, the "incoming" tile group is placed onto the board as if it was dropped from the top of the board, falling down and landing on top of the topmost existing non-white tile in the column the user selected. The game then checks to see if the placement of the "incoming" tile group resulted in the existence of any three blocks connected vertically, horizontally, or diagonally. This is counted as a "match," and the tiles comprising the "match" are then removed. The game then repeats checking for "matches" until no "match" is found. The user's goal is to generate as many "matches" as possible, until time runs out.

ColumnsGame calculates a score based on the amount of "matches" completed. Once time runs out and the user attempts to select a column, the game will notify the user that the game is over and prompt the user to enter in a name to record their score.