Team 7 Final Project

Tile-Matching Game Environment

Luke Ren
Brianna Huynh-Tong
Cong Khang Nguyen le
Nathanael John Scudder
Sean Garrett Loveland

Table of Contents

Introduction

For the final project, we designed and implemented our Tile-Matching Game Engine (TMGE) using Java as the programming language and JavaFX as the GUI library. We then used our game engine to implement the games Bejeweled and MemoryMatch, both of which are turn-based tile-matching games that pit two players against each other to see who scores higher.

In the game Bejeweled, players are able to repeatedly swap two tiles until jewels line up three or more in a column or row. If swapping two tiles matches three or more jewels, they are cleared from the board and new tiles fall in to fill their place. This process repeats automatically if the falling tiles happen to form matching sets of jewels. The score is increased proportionally to the number of tiles cleared. This repeats for a limited number of swaps for both players, after which the score of the two players is compared and the player with the higher score wins.

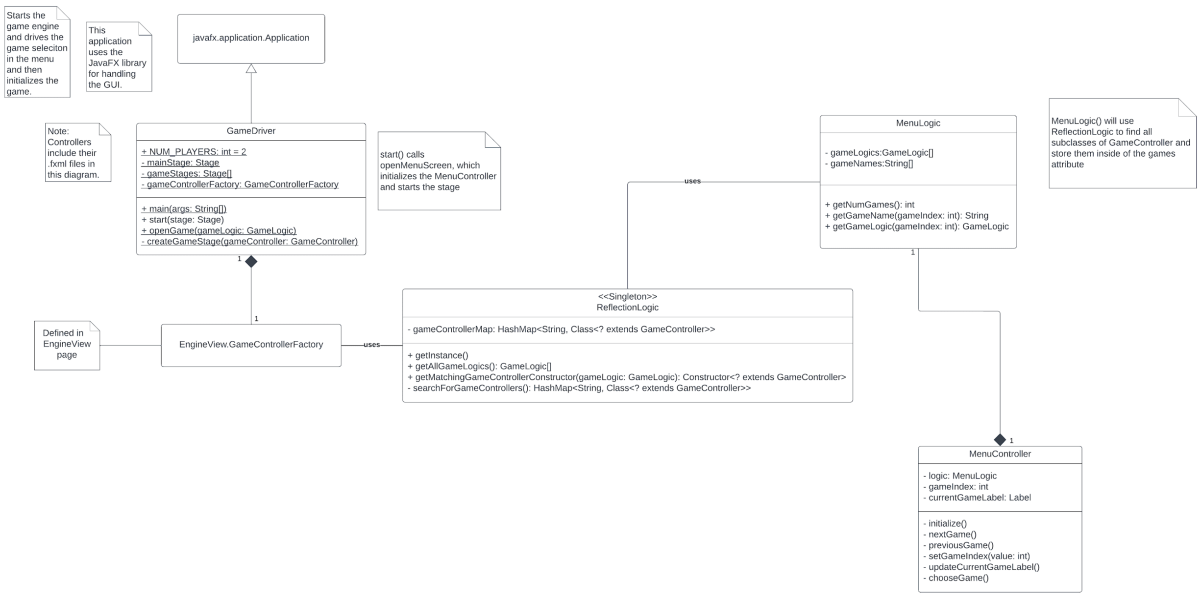
As for the game MemoryMatch, players flip face-up two cards at a time, matching identical cards until no more cards are left on the board. If two cards that do not match are flipped over, they will be flipped back face-down after a few seconds, and the player will lose points. If two cards are flipped face-up that match, they will be cleared from the board and the player will gain points.

Instructions to run the TMGE for various operating systems are included in the README.md file located in the root directory of the project.

UML Design

Below is the final version of the UML diagram for our TMGE, which contains the designs for the engine itself as well as the two individual games, Bejeweled and MemoryMatch. The design is split into five packages (or parts): the GameDriver package, the EngineView package, the EngineModel package, the Bejeweled package, and the MemoryMatch package.

GameDriver



GameDriver

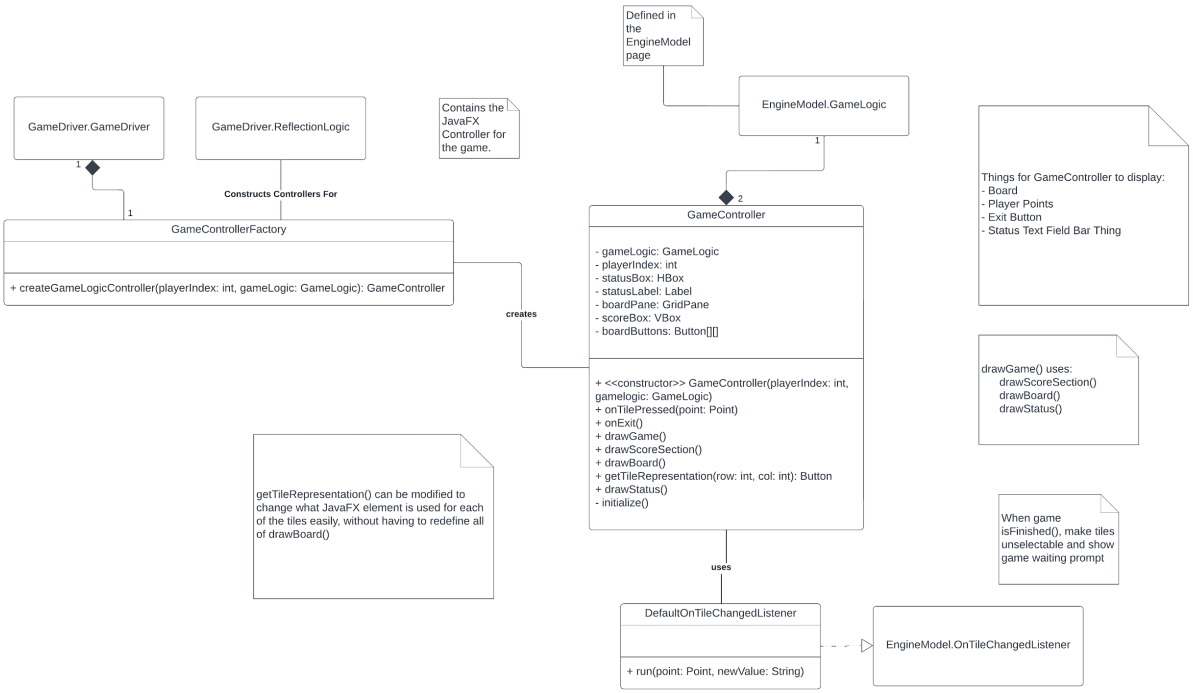
GameDriver is the main Application class that begins the TMGE. It is in charge of initiating the Menu view as well as opening the game screens once a game is selected. The Game Driver uses the GameControllerFactory to create a screen for each player based on the particular GameLogic that is selected in the Menu screen.

ReflectionLogic

ReflectionLogic uses the reflections library to scan the classes automatically for the different GameLogic and GameController classes in the project, which then get stored inside ReflectionLogic to be used by other classes. ReflectionLogic will determine which subclasses of GameController are associated with each GameLogic based on whether or not they are a part of the same package.

EngineView

The EngineView (view) package consists of three classes: GameController and GameControllerFactory



GameController, DefaultOnTileChangedListener

GameController contains the JavaFX controller for the game. It is in charge of visualizing the screen for a single run of a game, and draws three main sections: the board, the score section, and the status section.

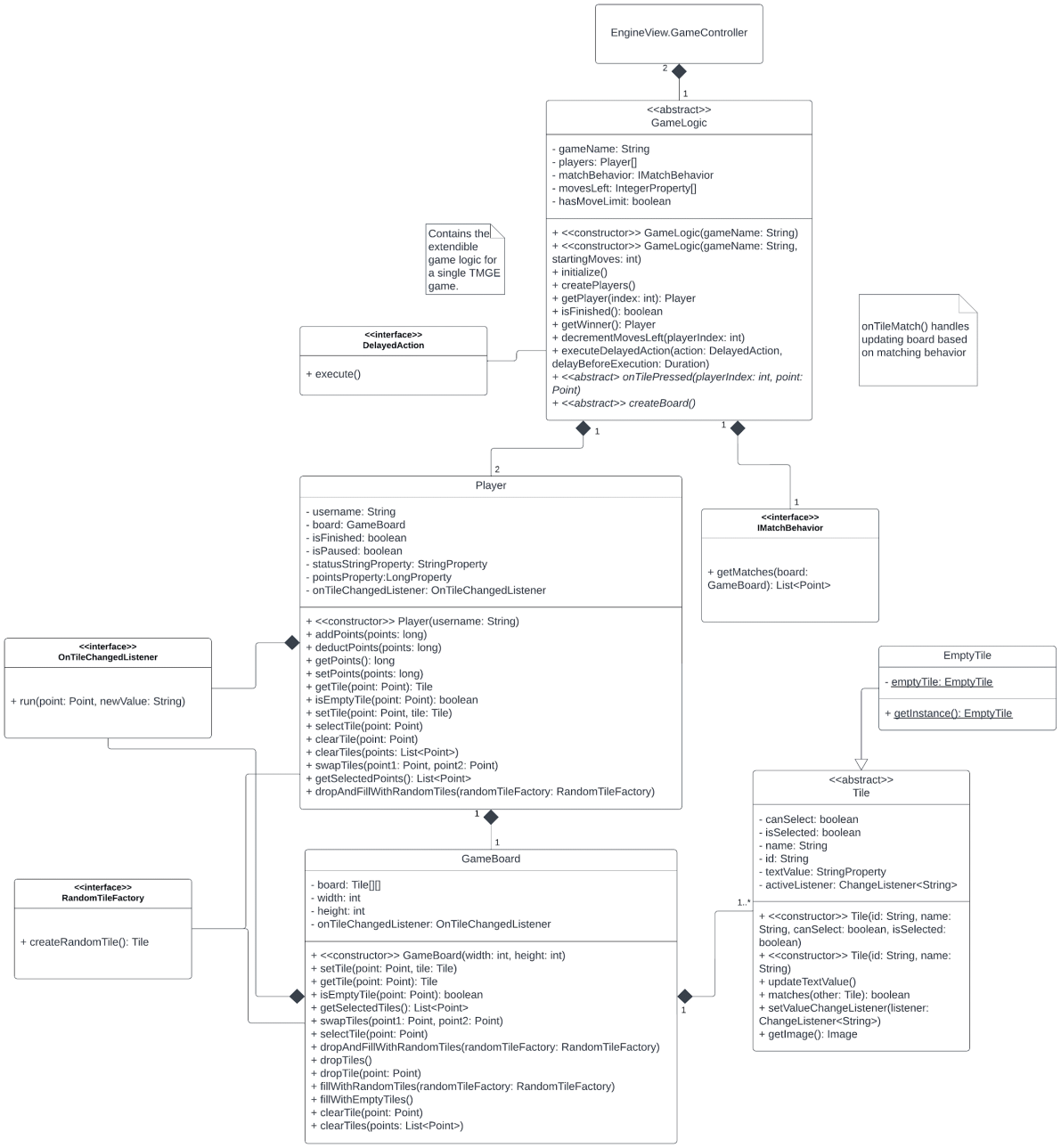
Within GameController is the private class, DefaultOnTileChangedListener, which implements the OnTileChangedListener interface in EngineModel. This listener interface is in charge of updating individual tile representations in the view as the data in the tiles get updated.

GameControllerFactory

GameControllerFactory is in charge of constructing a GameController based on a particular GameLogic. The GameControllerFactory will determine which subclasses of GameController should be associated with which GameLogic based on the scan done by ReflectionLogic, and will default to the GameController class itself if there are none.

EngineModel

The EngineModel package consists of three classes, Player, GameBoard, and TIle; two abstract classes, GameLogic and Tile; and four interfaces, DelayedAction, IMatchBehavior, OnTileChangedListener, and RandomTileFactory.



GameLogic

GameLogic is an abstract class that contains the extendible game logic for tile-matching games. GameLogic comes with many methods for retrieving and modifying player and game board data. The two abstract methods that each individual game must implement are the createBoard and onTilePressed methods.

The createBoard method is in charge of initializing a GameBoard for a particular game, including determining the width, height, and tiles for the board.

The onTilePressed method is in charge of determining the logic of when a particular player clicks on a particular tile, and will be unique for each game.

GameBoard

GameBoard is a class that is in charge of storing and modifying the various tiles that are in a board, which is represented as a matrix of Tile objects.

Player

Player is a class that stores and modifies the data for a particular user, including their username, points, and game board.

Tile, EmptyTile

Tile is an abstract class that represents an individual tile on a GameBoard, and contains the data for an individual tile. Tiles can be selected, and when they are, will be represented differently visually than non-selected tiles. Tiles have textual representations that, when updated, will change the equivalent visual representation of the tile in the GUI.

EmptyTile extends the Tile class. It is simply a Null Object (Tile) that is not selectable and does nothing.

IMatchBehavior, DelayedAction, OnTileChangedListener, RandomTileFactory

IMatchBehavior is an interface that may be commonly used for tile matching games, and contains an abstract method for finding which locations on the board represent matching tiles for a particular board and game.

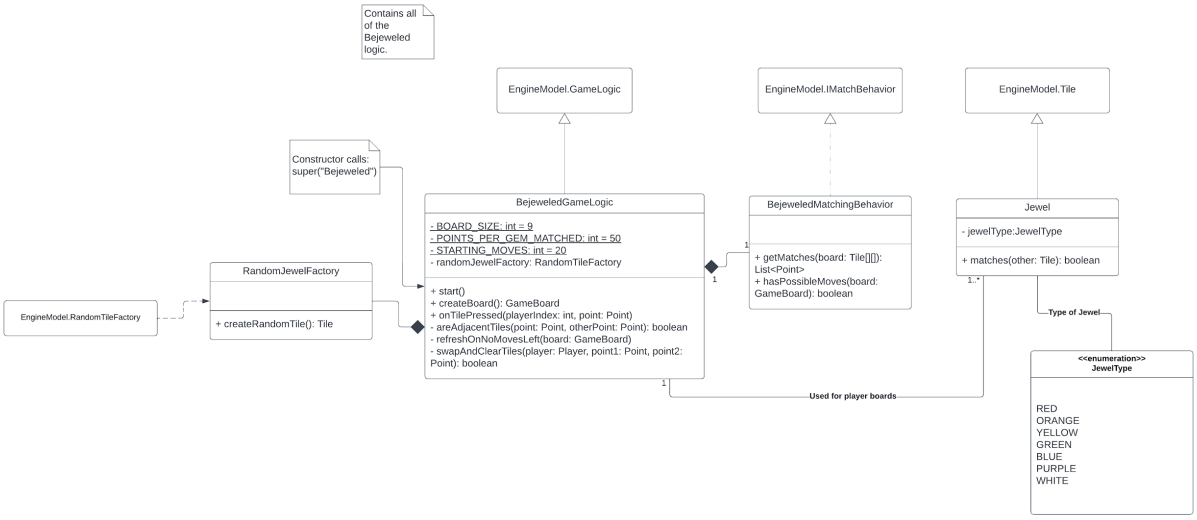
DelayedAction is an interface that represents an action that must be performed after a particular amount of time rather than a turn-based event.

OnTileChangedListener is an interface that represents an action that must be executed when the value of a particular tile changes, such as updating the corresponding UI.

RandomTileFactory is an interface that provides a method for creating random tiles, such that when GameBoard must generate a random tile, it has an abstract method to do so.

Bejeweled

The diagram below contains all of the logic for Bejeweled. It consists of four classes and one enum: BejeweledLogic, BejeweledMatchingBehavior, RandomJewelFactory, Jewel, and JewelType.



BejeweledLogic

BejeweledLogic extends GameLogic from EngineModell and writes its own logic for the abstract methods createBoard and onTilePressed, and contains some utility methods to define the behavior of when a tile is clicked, such as areAdjacentTiles, refreshOnNoMoves, and swapAndClearTiles. It has additional private member variables for the random jewel factory and various game constants.

BejeweledMatchingBehavior

BejeweledMatchingBehavior implements the IMatchBehavior interface and contains the logic for finding jewels on the board that are 3+ in a row either horizontally or vertically.

Jewel, JewelType, RandomJewelFactory

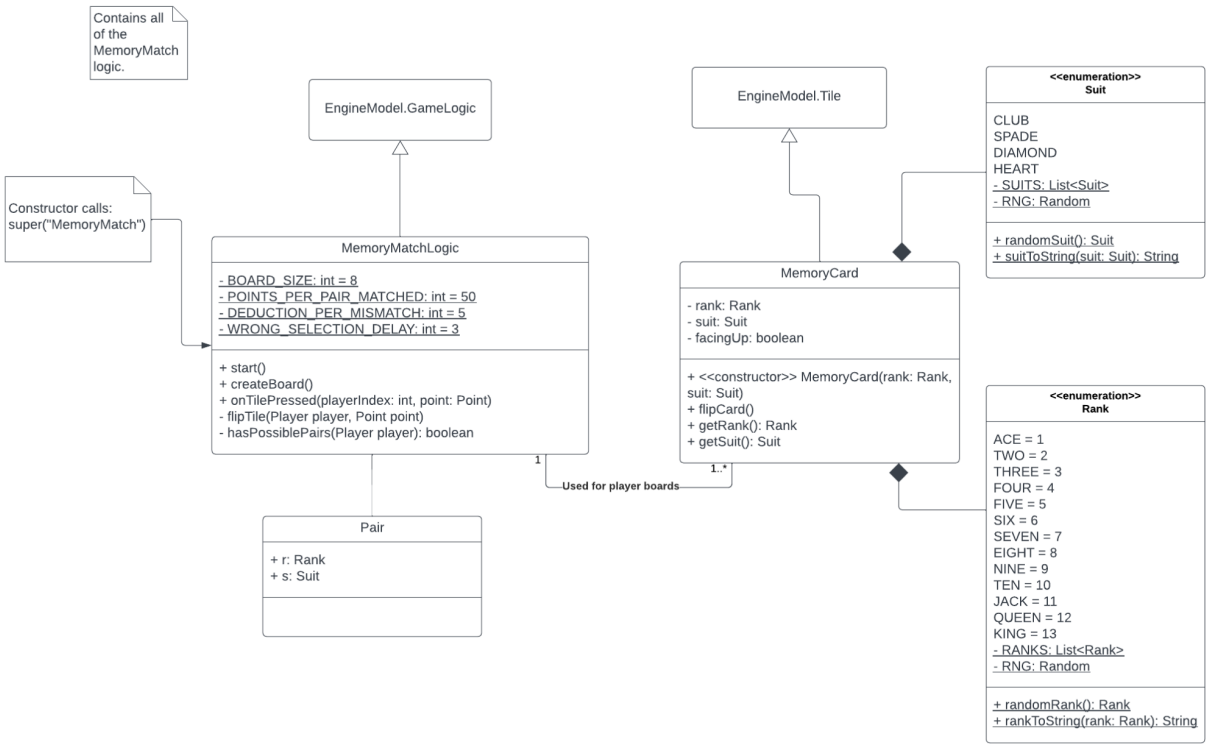
Jewels are the Bejeweled implementation of tiles, and contain a JewelType that defines how it matches with other Jewels.

JewelType is an enum with 7 different values, RED, ORANGE, YELLOW, GREEN, BLUE, PURPLE, and WHITE.

RandomJewelFactory implements the RandomTileFactory interface and creates a random Jewel with a uniform distribution.

MemoryMatch

The diagram below contains all of the logic for MemoryMatch, which consists of two classes, MemoryMatchGameLogic and MemoryCard, and two enums, Rank and Suit.



MemoryMatchLogic, Pair

MemoryMatchLogic extends GameLogic from EngineModel and writes its own logic for the abstract methods createBoard and onTilePressed. It has additional private member variables for the board size, points per match or mismatch, and delay.

Within the MemoryMatchLogic class is the Pair class, which is used to create a Pair of rank and suit for cards/Tiles.

To ensure each MemoryCard will have at least one other card to match with, two MemoryCards of each Pair of Rank and Suit are created, added to a list, and then shuffled. The createBoard method creates a new GameBoard and fills it using the list of shuffled MemoryCards.

The MemoryMatch game does not implement the IMatchBehavior interface as the matching behavior of the game is quite simple and only requires checking if two clicked Tiles have the same Rank and Suit, which is handled in the onTilePressed method.

MemoryCard, Rank, Suit

MemoryCard extends Tile from EngineModel and these cards are representative of real-life playing cards with a Rank and a Suit, which are designed as enums. Each has their own method that will randomly generate a Rank or Suit.

Implemented Design Patterns

Strategy

Our TMGE utilized the strategy pattern heavily within the Engine Model package. The different classes in the engine's model utilize interfaces, rather than concrete classes for many of its operations so that the exact implementation remains more flexible to the needs of each specific game. The interfaces used in this way include DelayedAction, IMatchBehavior, RandomTileFactory, and OnTileChangedListener.

Factory

We also utilized the Factory design pattern to encapsulate the construction for some of the game engine's more complex objects. This is exemplified in the GameControllerFactory and the RandomTileFactory classes, which produce their respective object instances without requiring their utilizers to know the exact details of how each of the objects is being constructed.

Template

Additionally, we implemented the template design pattern within the GameController class through its drawGame method. The drawGame method itself cannot be overridden, but only contains calls to drawScoreSection, drawBoard, and drawStatus. This allows developers to redefine those methods individually without changing the whole structure of how the user interface is drawn.

Null Object

Our design also implements the Null Object pattern with the use of the EmptyTile class, which implements the necessary methods of a Tile object but does nothing. This was useful in empty space on the game boards since we always knew the kind and behavior of the object taking those spots in the two-dimensional array of tiles, without having to place non-null assertions everywhere in the code.

Singleton

Lastly, our design incorporates the Singleton design pattern for the ReflectionLogic class. ReflectionLogic allows us to hold a map that holds the package name of each game controller to its respective controller class. Applying the Singleton design pattern ensures that only one ReflectionLogic instance is ever created, and whenever it is used or called in the app, it simply gets an instance of the ReflectionLogic that has been created.

Software Design Principles

Our system was designed to be in line with many of the SOLID principles we discussed in class.

Single Responsibility

The Single Responsibility principle states that each class should have only one reason to change. In other words, each class should be “in charge” of handling the operations specific to that class. We used this principle extensively throughout our design. The GameController class acts as our view and controller for our GUI, essentially drawing the initial state of our GUI and updating it based on changes made as games progress. The GameLogic class handles all game related logic such as creating and populating the game boards with relevant tiles, handling the logic for tile matchings, keeping track of the status of games and winners, and handling tile pressing events. The Player class only maintains player related information and the GameBoard class only maintains information and functionality related to the actual game board and its tiles.

Open-Closed

The Open-Closed principle states that classes should be open to extensibility while being closed to modification. One example of this is how we implemented the GameLogic class. We made the GameLogic class abstract and implemented many base functionalities that could potentially be used by many different games (e.g. determining the winner via point values, maintaining a limited number of moves, delaying programmatic actions, etc.). This class is closed to modifications, and if developers need additional functionality, they must extend the class (which they already need to do since GameLogic is abstract) in order to add changes specific to that game. Further, we separated out the matching logic for games into a separate interface called IMatchBehavior, so you won't have to directly change the GameLogic class in order to create game-specific logic, you would simply extend IMatchBehavior and use it in GameLogic.

Liskov Substitution

Add Liskov Substitution section if you can find an example of it (dont think we have… might just remove)

Interface Segregation

The Interface Segregation Principle states that having many different fine grained interfaces is preferable than having less interfaces that are more general. Essentially, we don't want to force developers to implement functions that they do not need. We use this in the GameLogic class as we have two interfaces: DelayedAction and IMatchBehavior. Each of these interfaces only has one function, an execute function for executing delays, and a matching function for identifying matches on a board, respectively. However, since we separated these interfaces, we do not force developers that use our TMGE to implement an execute function for delays or a matching function if they don't use that functionality in their games. For example, in MemoryMatch, the DelayedAction interface is implemented and used, but not the IMatchBehavior interfaces. In Bejeweled, we implement IMatchBehavior, but we do not implement DelayedAction.

Dependency-Inversion

The Dependency-Inversion principle states that developers should depend on abstractions rather than on concretions. We did this at several points in our design. Firstly, GameControllers depend on GameLogic, which is an abstract class. Another example is that the GameBoard

class relies on the Tile class which again is an abstract class. Furthermore, many of our classes depend on different interfaces (GameLogic depends on DelayedAction and IMatchBehavior, Player depends on onTileChangedListener, etc.). By depending on abstractions, we can easily extend those classes and modify them according to the desired functionality we would like. Also, following the Dependency Inversion principle helps ensure that we are following the Open-Closed principle in our system's design.

How to Make a New Game Using TMGE

To make a new game using the TMGE, such as Candy Crush, developers are provided with many tools to ease the overall process. Firstly, to make a new game, a developer must extend the GameLogic class corresponding to their game (e.g. CandyCrushLogic extends GameLogic). Extending GameLogic will require the implementation of the onTilePressed and createBoard functions, which control the action taken when pressing a tile and the game-specific generation of the board respectively. GameLogic also provides default implementations for many general features of tile-matching games, such as determining the winner (by point value), maintaining a limited number of moves, delaying programmatic actions on the GUI, and so on.

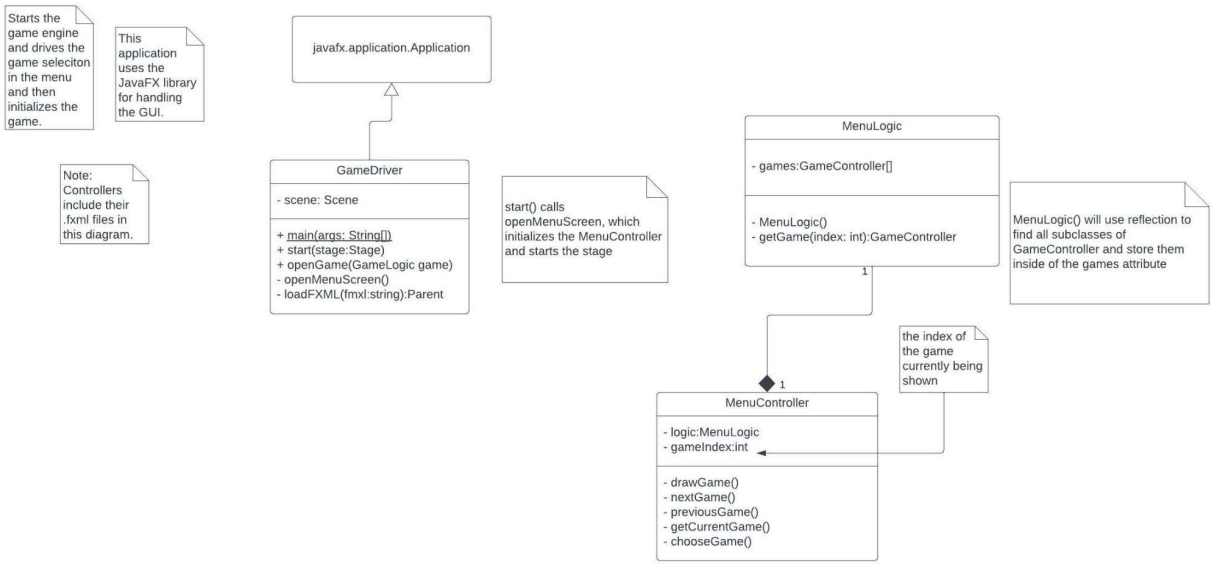
Developers are also required to define tiles for their game by extending the Tile class (e.g. CandyTile extends Tile). These Tiles should be defined to know how they match other Tile objects, depending on the game's needs. Accordingly, developers must define the result of matching tiles, given the state of the game board such as through the IMatchBehavior class (e.g. CandyCrushMatchingBehavior extends IMatchBehavior). For example, in Candy Crush, matching three or more tiles in a row will remove those tiles and replace them. Matching four or more tiles will instead create new special tiles at the position that can clear entire sections of the board on a subsequent match. All of this can be accomplished by defining the matching behavior IMatchBehavior and then determining how it transforms the board when implementing GameLogic's onTilePressed method.

In terms of the user interface, developers are provided with a standard GameController, which simply draws each of the tiles in the game board as buttons on a grid, along with live scoring sections for each player, and a status window for displaying prompts and responses from the game. Optionally, developers can override the drawScoreSection, drawBoard, getTileRepresentation, and drawStatus methods to gain as much or as little control over the user interface as desired.

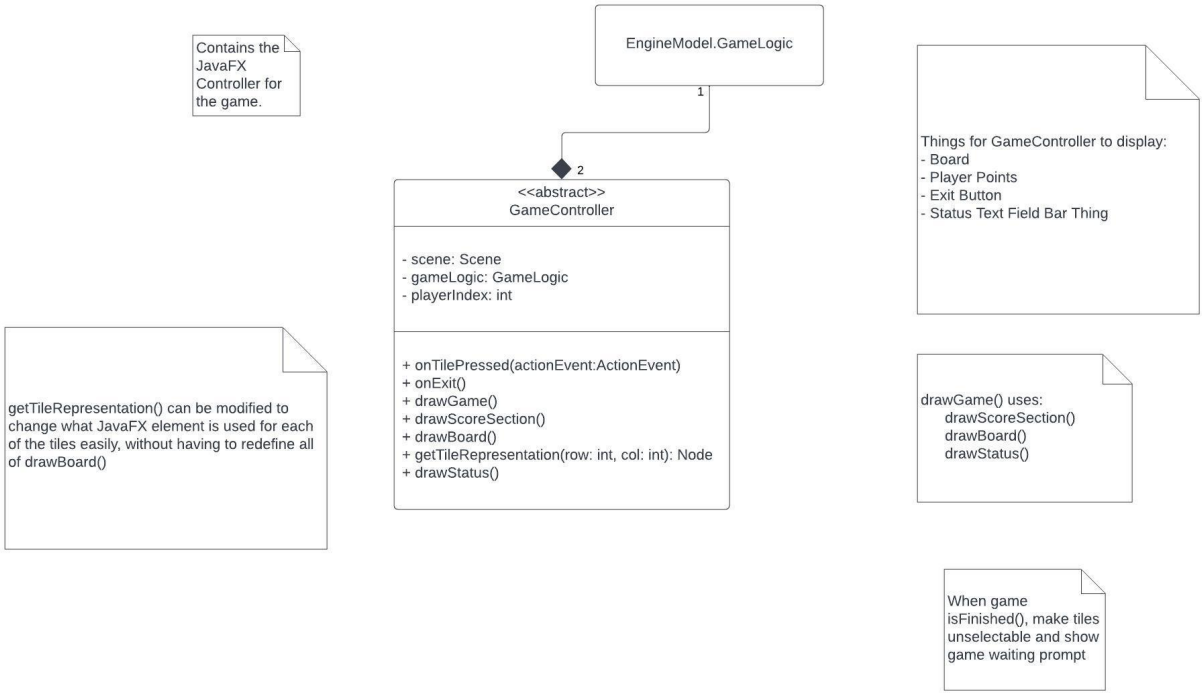
Design Evolution

This design originally started out as the UML diagram below:

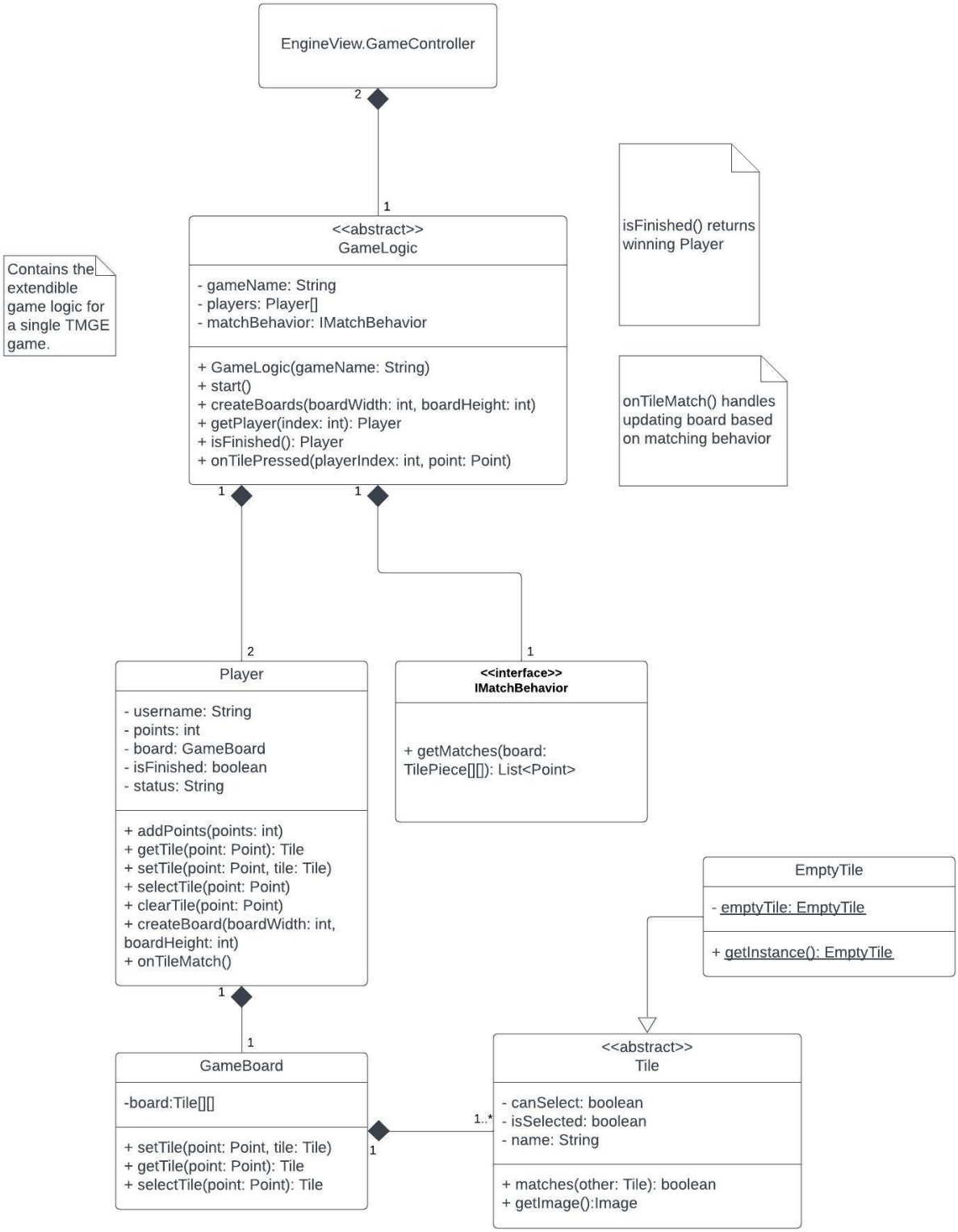
GameDriver v1



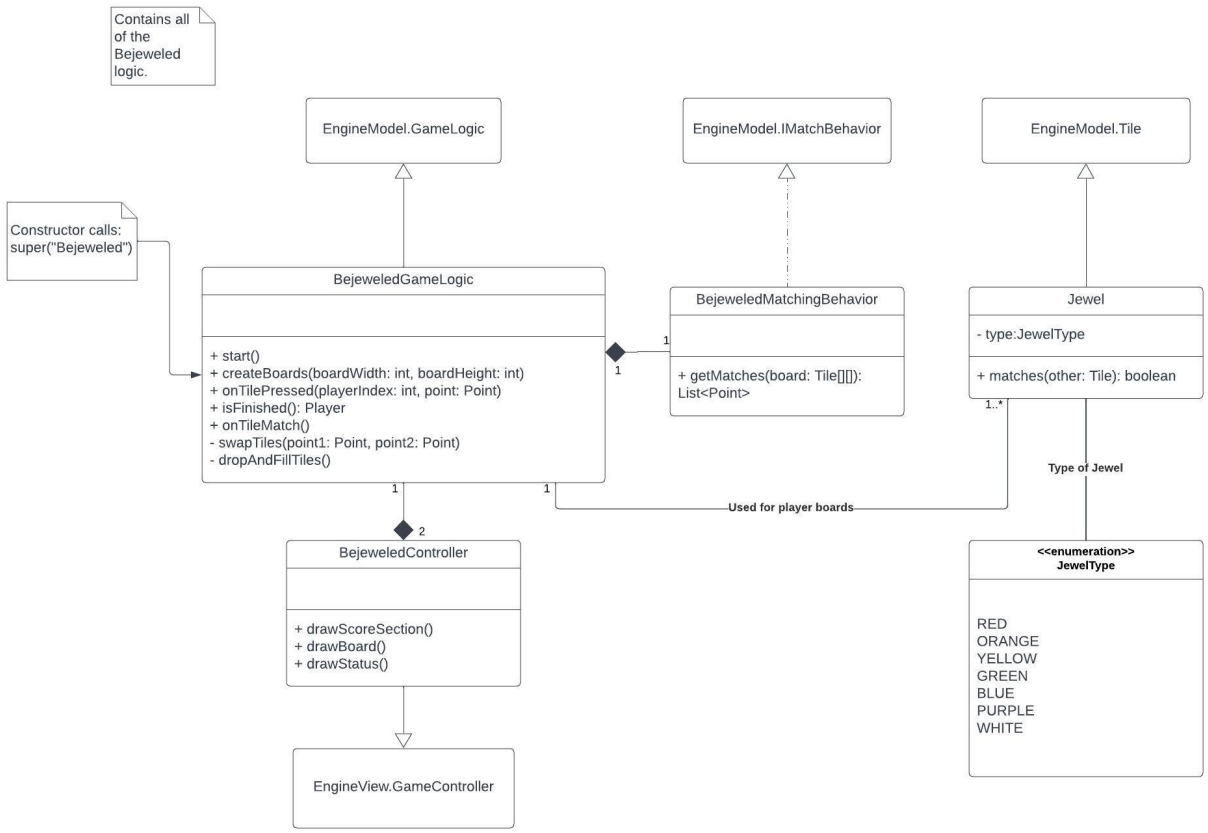
EngineView v1



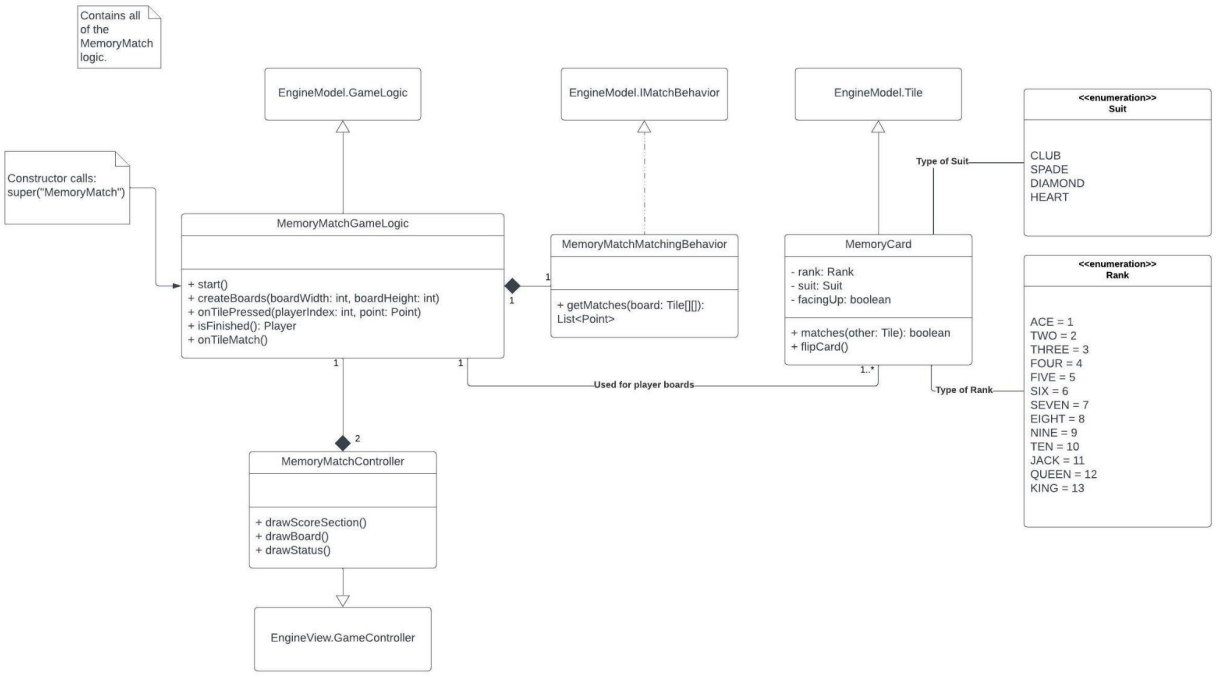
EngineModel v1



Bejeweled v1



MemoryMatch v1



One of the first changes that was made to the original version of the UML was the inclusion of the ReflectionLogic class. Given that much of the logic with reflection seemed irrelevant to the MenuLogic itself, the methods related to reflection were separated into its own ReflectionLogic class. This change also opened up the potential for other parts of the code to use reflection as well, such as the creation of tiles or game boards.

Another significant change in the design was the inclusion of many GameBoard methods inside of the Player class. By allowing the Player class to serve as a medium between the GameLogic and the GameBoard, there could be more decoupling between those two classes, allowing for changes in how the GameBoard works internally to be abstracted by the Player class.

A smaller but also significant change that was made to the design was making the GameController class not abstract, which allowed games to use the GameController class as the default controller for each game's view. By making this change, games that desired to use the default engine's view did not have to reimplement the GameController class unnecessarily.

Some of the high points for this process were that making changes to an already well thought out design required much less change to the code itself since many of the parts were already decoupled and modularized. Design changes only influenced one or a few other classes, which made recovering those designs in the actual code itself much simpler than recovering a highly coupled design.

One of the low points while working on this design was just for the group to collectively learn how to work with JavaFX. Since none of us had prior experience with JavaFX, it made working on the design a bit difficult. We had started working on a design and developed a baseline UML at first, we just had a bit of difficulty learning how we would integrate JavaFX into the design for the UI. After we had done our research though, we were able to continue working on and complete the design.

Some challenges we faced during this phase were figuring out what the GameDriver, EngineView, and EngineModel should contain. Only one of us in the group had experience with game design and even then we were having trouble figuring out where what classes should be and what each class should do. But after a couple meetings we were able to flesh out our design and delegate the correct classes and what methods they should have to their respective packages.

Lessons Learned

One of the high points of this project was that using design patterns that we learned in class and how they can improve the quality of the code was quite rewarding. Being able to see the design patterns in action and how they can minimize the amount of change required from modifying or adding features to the code made the project a lot more fun and easy. Because we had thought out the design before hopping into the code, writing the actual code and making changes was significantly easier and allowed for more asynchronous development.

One of the low points or challenges we faced was not being super familiar with game development or JavaFX. Figuring out what kinds of abstractions or classes we would need in advance before actually seeing the code was difficult because it required foresight and experience that many of us did not have. It was not always clear what the best approach to

design a particular part of the system would be, such as how the GUI would be updated to represent the internal data, so we had to make a lot of guesses. Since we could not try all of the possibilities because we were limited on time, it was not totally clear by the end of it what changes would necessarily improve the design.