Monster Trading Card Game by David Zelenay

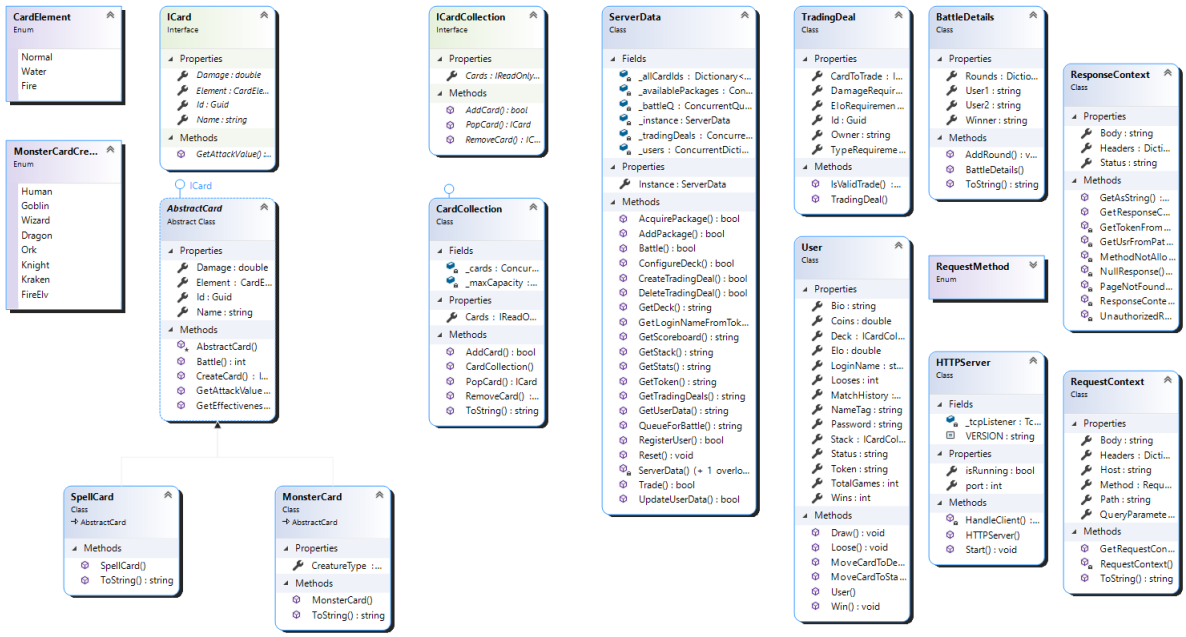
Time Tracking:

Date	Time in h	Comment
15-oct-2020	3	Setup Github repo, initial OOP class design (TDD)
17-oct-2020	0.5	Setup Github Actions
18-oct-2020	1.5	Further developed CardDeck Class
HTTP Webservice	-	-
20-oct-2020	2.75	Started developing HTTP Server
21-oct-2020	3	Added MessageCollection + Tests
11-nov-2020	3	Further developed MsgColl + Implemented GET /messages and POST /messages
12-nov-2020	4	Implemented other request Methods and msg in Path (GET /msgs/1)
15-nov-2020	2	Recorded Video + minor tweaks
mtcg	-	
02-jan-2021	1.5	Made HTTP Server Multithreaded
02-jan-2021	3.5	Read requirements thoroughly and almost completely redesigned class diagram
03-jan-2021	7	Developed Card Logic (GetAttackValue() + GetEffectivenessMultiplier() + CardCollection)
04-jan-2021	5	Developed ServerData + Logic
05-jan-2021	8	Developed ServerData + Logic
06-jan-2021	8	Developed ServerData + Logic + developed http server
07-jan-2021	9	Developed ServerData + Logic + started developing http server
08-jan-2021	7.25	Refactoring + Implemented Trading + wrote documentation

Approximate time spent on mtcg: 54.25h

Approximate time spent total: 69h

Class Diagram



Documentation

Design Thoughts

Every response from the Server is a Json string by default

Cards can either be Spell- or Monster-Cards. An abstract Class that implements the ICard interface handles the common tasks between all CardTypes. It has a static method to create either Monster- or Spell-Cards, assign cards the correct element and if applicable a MonsterCreatureType based on the Name. The battle logic is also implemented in the AbstractCard class (effectiveness calculation, actual damage calculation, ...).

A custom class called CardCollection is being used to Store Multiple Cards. The CardCollection has basic Add and Remove methods that check that the upper card limit of the collection is not exceeded.

The TradingDeal class stores trading deals and is able to check if a trading deal is valid.

The BattleDetails class stores basic information about a Battle (Winner, players, rounds).

The User class stores information about a user (player). Besides some general properties a user has a MatchHistory (Dictionary<DateTime, BattleDetails>), a Stack (all Cards ICardCollection) and a Deck (ICardCollection with max of 4 Cards) The user also hase some methods to easily change the stats (elo, matches won/lost,..) after a battle and to move cards between the deck and the stack.

The central element of my solution is the Singleton class "ServerData". The class contains most of the business logic. The ResponseContext then calls each Method based on the Request. E.g. GET to "/cards" -> Request Context calls The Method serverData.GetStack(...). The ServerData Class also stores and manages all data. It has a Collection of currently available Packages (ConcurrentBag<ICardCollection), a Dictionary<string, User> for all users where the key is the id of the user, a collection of all cards that have been created to guarantee the uniqueness of the card ids, if this wouldn't be there all users' decks and stacks would need to be checked before creating a card -> looking it up in a dictionary is the faster approach, a Dictionary<string, TradingDeals> to

store the currently registered trading deals and allow a fast search for the trade id (key of the dictionary) and a Queue of Players currently waiting for a battle.

Almost all public methods of ServerData, User, Card are tested with unit tests. Methods like GetScoreboard() are not tested because they do not contain any logic and just parse instances/objects to a JSON string. The Method Battle is not tested as well, because I was not sure on how to execute a test with 2 Threads, however the major logic of a battle e.g. GetAttackValue is covered by tests.
Furthermore the class ResponseContext is not tested, as it does not contain any "business logic" and just calls the correct methods of the ServerData class. (Total Unit Tests 36)

Additional Features

- Winrate (win loose ratio) in user stats
- Trade Cards vs Elo

Pitfalls

The way the two player threads receive the battle result/log is just a dirty fix/poc and not implemented cleanly. Currently the second player that enters the queue calls the battle method, the battle method dequeues both players "plays" the game and stores the result of the Match in each users' match history. The first user which entered the queue does not receive any value from the battle method directly. The way the first user knows if a battle has been finished is the number of total matches is saved before entering the queue, then the thread sleeps in a loop (max timeout 20s) and checks every 100 ms if the current number of matches has increases, if so then the latest BattleDetails of the MatchHistory is returned. Before that I had an even worse approach, I used a queue in the ServerData where the match results were stored.

Furthermore I wanted to prevent the situation where I have just a single class that contains all the logic. I wanted to have a more or less clean object oriented architecture and then implement the database connection. So I waited to long to start implementing a database connection up to the point where I am finished with the project except the database connection. My initial fear (why I did not implement a DB connection from the beginning) was that my program could just end up as a "interface" between the client and the database, which (I thought) could result in me neglecting the object oriented architecture. Since the non-functional-requirements make up 10 point and consist of 4 subsections my thought was, to implement everything else properly and hope, that the persistence (DB) indeed makes up only 2.5 points.

How I would have handled the database connection: My plan was to have every class handle the persistence itself. Furthermore every Class that needs some information to be stored in the DB, has a boolean private variable that says whether changes to the properties should be stored to the DB (for testing purposes). Each property has a custom "set" method where the changes are stored to the DB if the private variable says so. The ServerData Class would have had a function that initializes the fields (users, available packages, ...) with values from the DB. If the the database does not exist it would create the tables required to store all the information. This method would only be called once in the private constructor of the singleton ServerData class.

Curl script needed to be updated (order of acquiring packages is random to some degree)

Github Link