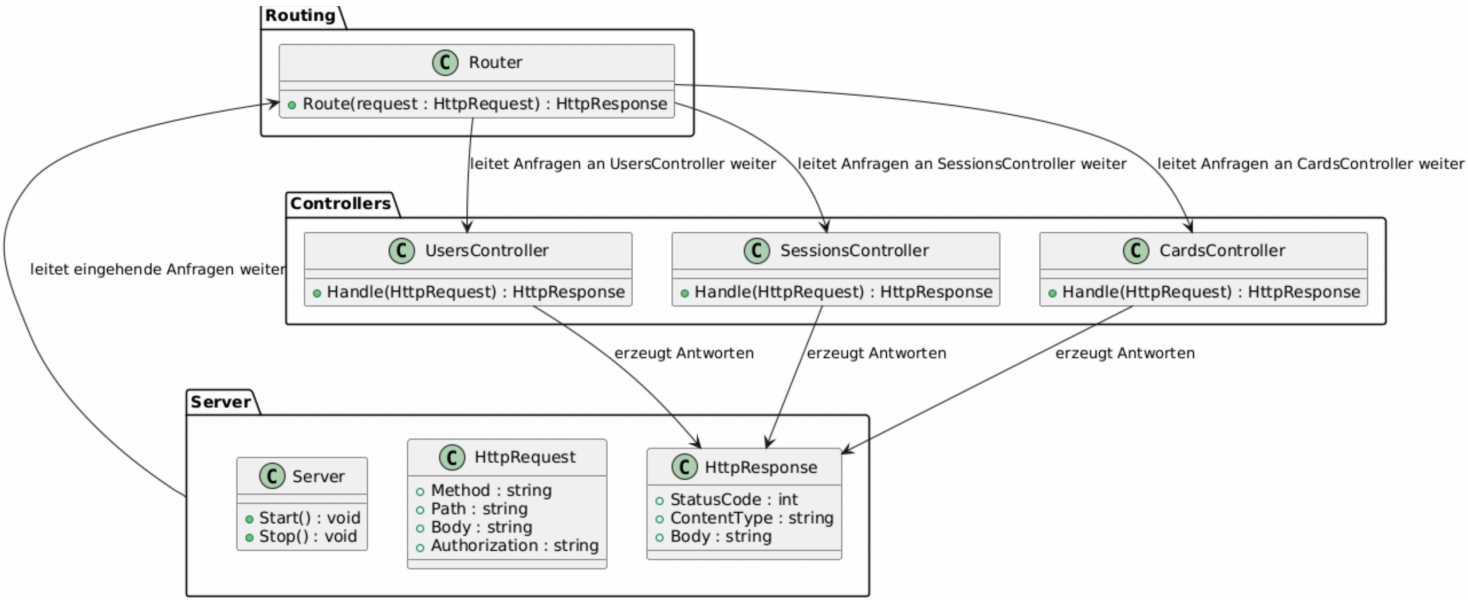
Monster Card Trading Game

Dokumentation

Petra Scheuer - 12. Jänner 2025

1.) Server Funktionalität

Dieses Diagramm zeigt, wie der Server Anfragen verarbeitet, sie an die entsprechenden Controller weiterleitet und die Antworten zurückgibt.



Server:

- Startet den Server und stoppt den Server.

HttpRequest:

- Eingehende HTTP-Anfrage mit Attributen wie Method, Path, Body und Authorization.

HttpResponse:

- Repräsentiert eine HTTP-Antwort mit StatusCode, ContentType und Body.

Router:

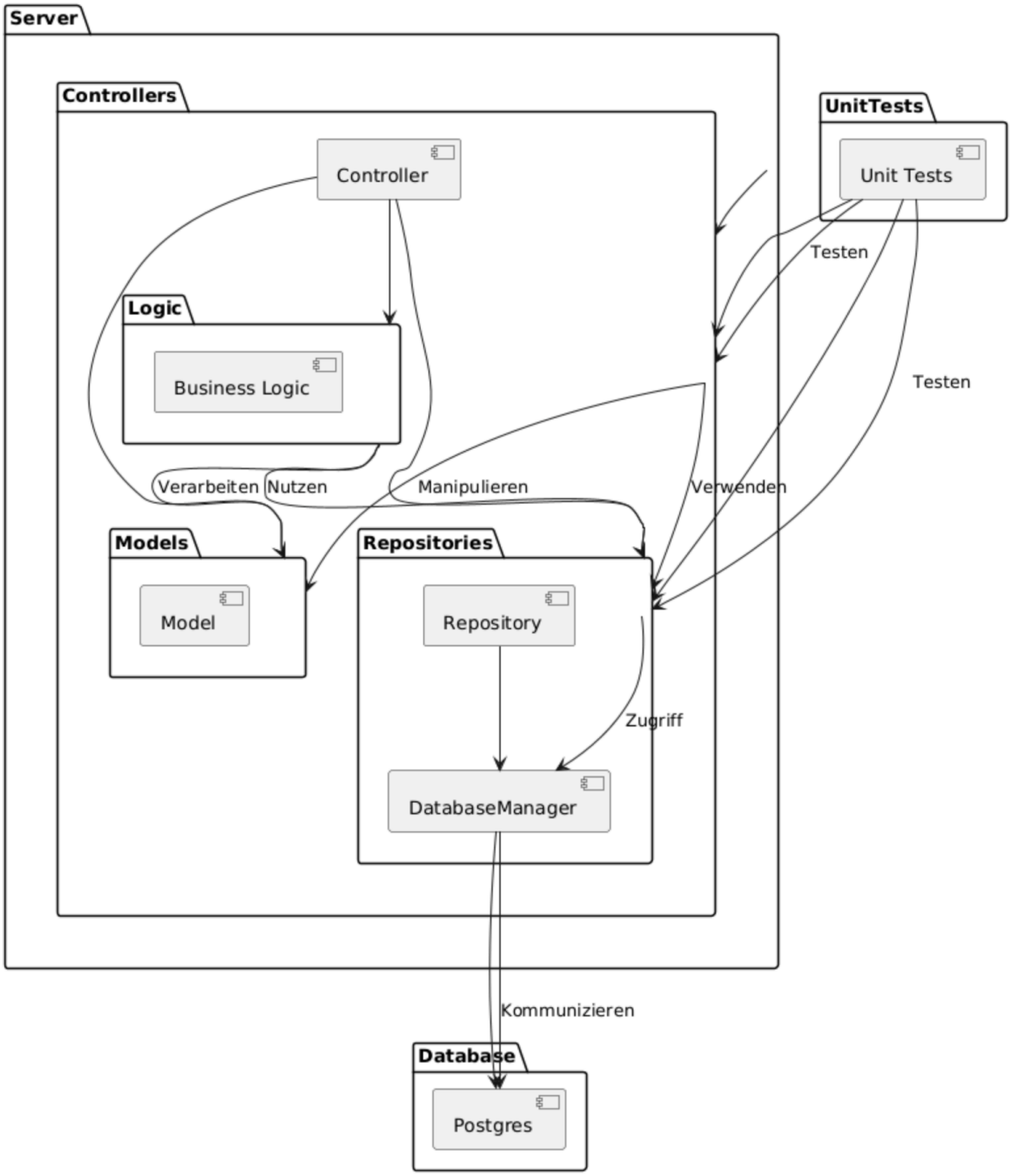
- Nimmt eine HttpRequest entgegen, bestimmt den passenden Controller basierend auf dem Path und leitet die Anfrage weiter.

Controllers:

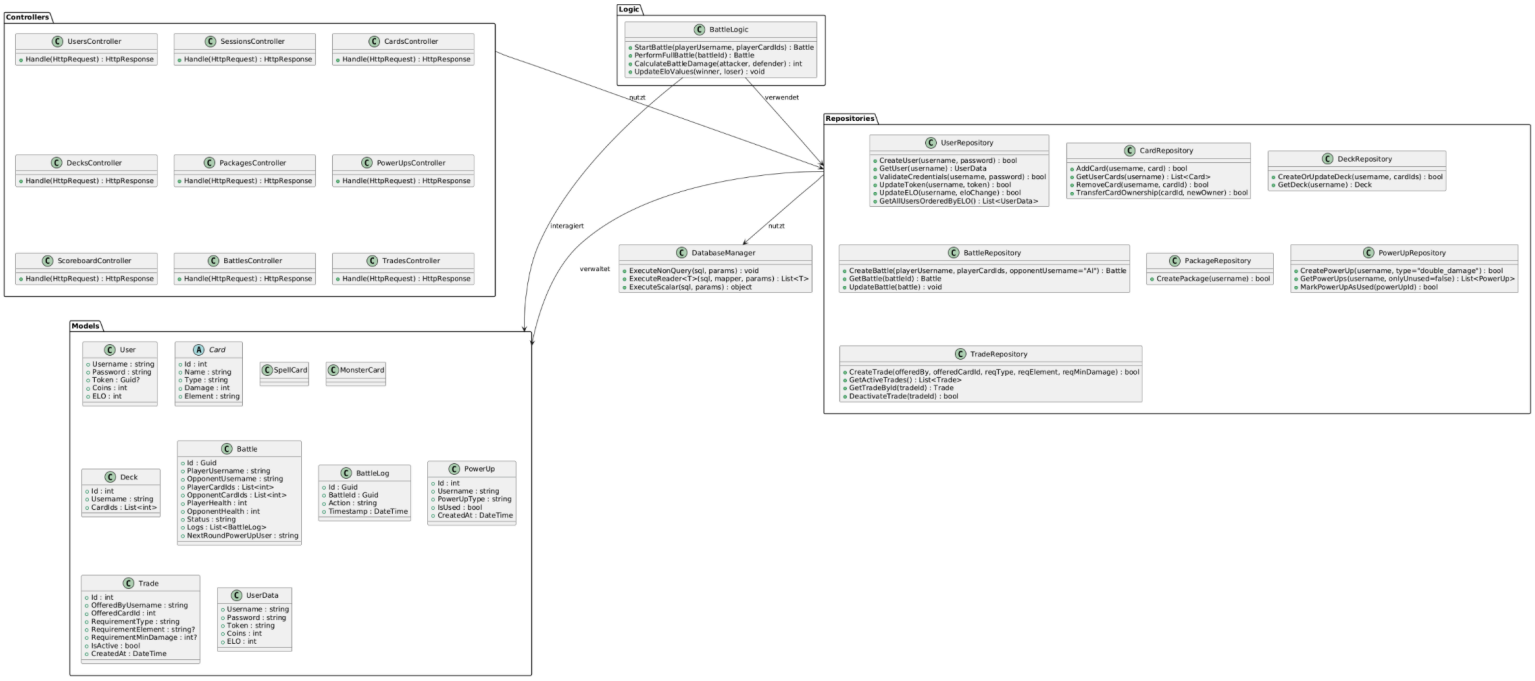
- Jeder Controller (z.B. UsersController, SessionsController, CardsController) verarbeitet die Anfragen und erzeugt entsprechende HttpResponse-Objekte.

2.) APP Funktionalität

Dieses Diagramm ist ein vereinfachtes UML-Klassendiagramm, das die Hauptkomponenten und ihre Beziehungen in meinem Projekt zeigt:



Dieses Diagramm zeigt die Hauptkomponenten meiner spezifischen Anwendung, ihre Beziehungen und Verantwortlichkeiten, einschließlich Controllers, Repositories, Models und Logik.



- Controllers:

  Verarbeiten eingehende HttpRequest-Objekte und erzeugen HttpResponse-Objekte.

  Jeder Controller (z.B. UsersController, CardsController) ist für einen spezifischen Funktionsbereich Verantwortlichkeit.

- Repositories:

  Verantwortlich für den Datenzugriff und die Interaktion mit der PostgreSQL-Datenbank über den DatabaseManager.

  Methoden wie CreateUser, AddCard ermöglichen CRUD-Operationen.

- Models:

  Repräsentieren die Datenstrukturen der Anwendung, wie User, Card, Deck, Battle, etc.

  Card ist eine abstrakte Klasse mit Unterklassen SpellCard und MonsterCard.

- Logic:

  BattleLogic: Enthält die Geschäftslogik für Kämpfe, einschließlich Methoden zur Kampfanalyse und ELO-Updates.

Beziehungen:

Controllers nutzen Repositories zur Datenverarbeitung.

Repositories verwalten Models und nutzen den DatabaseManager für Datenbankoperationen.

BattleLogic verwendet sowohl Repositories als auch Models zur Durchführung von Kämpfen.

3.) Unique Feature: PowerUps (Double Damage)

Mein einzigartiges Feature besteht darin, dass Benutzer innerhalb des Spiels spezielle **PowerUps** beanspruchen und einsetzen können. Ein PowerUp „Double Damage“ ermöglicht es dem Spieler, in der **nächsten Battle-Runde** den verursachten Schaden zu verdoppeln und so einen entscheidenden Vorteil gegenüber dem Gegner zu erhalten.

Funktionsweise:

1. PowerUp beanspruchen (Claim):
    - Über den Endpunkt `/powerups/claim` kann ein Benutzer, sofern er noch keinen ungenutzten PowerUp besitzt, ein neues „Double Damage“-PowerUp anfordern.
2. PowerUp einsetzen (Use):
    - Beim **Battle** (z.B. `/battles/{id}/usepowerup`) kann dieses PowerUp aktiviert werden, um einmalig den Schaden in der kommenden Runde zu verdoppeln.
3. Limitierte Nutzung:
    - PowerUps sind nur einmalig verwendbar. Nach ihrem Einsatz wird das PowerUp als „verbraucht“ markiert und steht nicht länger zur Verfügung.

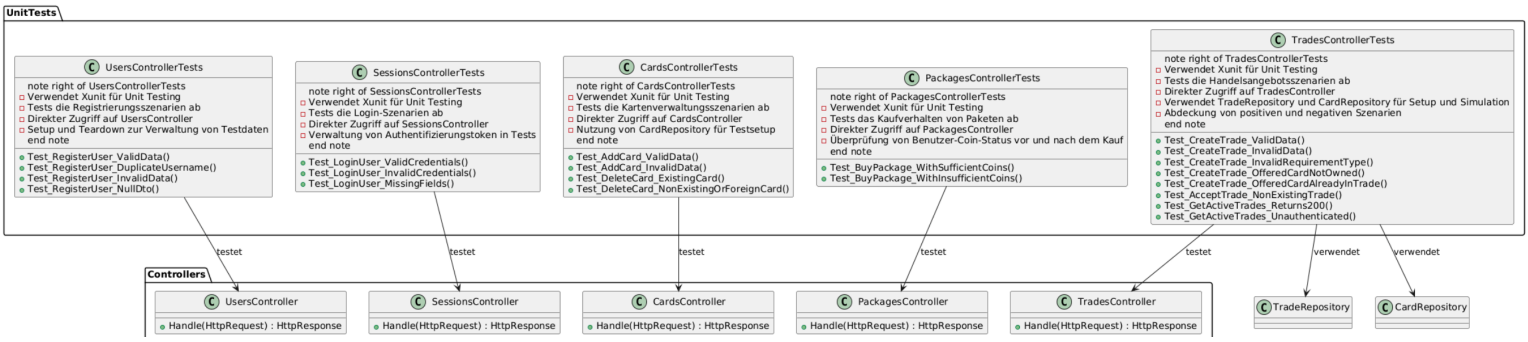
Das „Double Damage“-Feature belohnt strategische Entscheidungen beim Kämpfen und erzeugt zusätzliche Spannung in den Battle-Runden. Spieler können den richtigen Zeitpunkt abwarten, um das PowerUp möglichst effektiv einzusetzen.

Unterschied zum Basisspiel:

- In der Standard-Spezifikation des MTCG sind solche temporären Buffs nicht vorgesehen. Durch die Einführung der PowerUps wird das Kampfsystem erweitert: Neben Kartenauswahl und Element-Boni kommt nun ein zusätzlicher taktischer Aspekt hinzu, der die Spieltiefe erhöht.

3.) Unit Tests

Dieses Diagramm zeigt die Unit-Test-Klassen und welche Controller sie testen. Es veranschaulicht die Testabdeckung und die Beziehungen zwischen den Tests und den zu testenden Komponenten.



Controllers:

- Die zu testenden Controller-Klassen (UsersController, SessionsController, etc.) sind aufgelistet.

UnitTests:

- Jede Testklasse (UsersControllerTests, SessionsControllerTests, etc.) enthält mehrere Testmethoden, die spezifische Funktionen der Controller überprüfen.
- Die Beziehungen zeigen, welche Controller von welchen Testklassen getestet werden.

Beziehungen:

- Die Unit-Test-Klassen sind direkt mit den Controllers verbunden, die sie testen.
- Einige Tests (z.B. TradesControllerTests) nutzen zusätzlich Repositories, um bestimmte Szenarien zu simulieren oder Daten vorzubereiten.

Erklärung der Unit Testing Decisions

Verwendetes Testframework:

- Xunit wurde gewählt wegen seiner Flexibilität und einfachen Integration in .NET-Projekte. Es unterstützt parallele Testausführung und bietet erweiterte Features wie Data-Driven-Tests.

Testisolierung:

- Tests sind so gestaltet, dass sie unabhängig voneinander laufen. Jeder Test initialisiert seine eigenen Testdaten und bereinigt diese nach der Ausführung, um Seiteneffekte zu vermeiden.

Direkter Zugriff auf Controller:

- Die Tests interagieren direkt mit den Controller-Klassen, um die Endpunkte und deren Logik zu überprüfen. Dies ermöglicht eine klare und direkte Überprüfung der Controller-Funktionalität.

Datenmanagement in Tests:

- Setup-Methoden: Jede Testklasse initialisiert notwendige Daten (z.B. Testbenutzer, Testkarten) im Konstruktor.
- Teardown-Methoden: Implementiert durch die Dispose()-Methode, um nach den Tests aufgeräumt zu werden, indem Testdaten aus der Datenbank entfernt werden.

Testabdeckung:

- Positive Tests: Überprüfen, ob die Controller erwartungsgemäß auf gültige Eingaben reagieren (z.B. erfolgreiche Registrierung, erfolgreicher Kartenkauf).
- Negative Tests: Überprüfen, ob die Controller korrekt auf ungültige Eingaben oder fehlerhafte Szenarien reagieren (z.B. Registrierung mit doppeltem Benutzernamen, Kauf eines Pakets mit unzureichenden Münzen).

Assertions und Validierung:

- Tests verwenden spezifische Assertions, um die Genauigkeit der Ergebnisse zu überprüfen (z.B. Assert.Equal, Assert.NotNull, Assert.True).

4.) Lessons learned

❖ Frühzeitiges Integrieren von Unit-Tests ist entscheidend

Zu Beginn des Projekts lag mein Fokus hauptsächlich auf der Implementierung der Hauptfunktionen und der Architektur. Unit-Tests habe ich erst am Ende hinzugefügt, nachdem die meisten Features bereits entwickelt waren.
-> Das späte Hinzufügen von Unit-Tests stellte sich als problematisch heraus. Es war schwieriger, Tests für bereits bestehenden Code zu schreiben, da die Logik nicht immer testfreundlich gestaltet war. Fehler wurden erst spät im Entwicklungsprozess erkannt, was zu zusätzlichem Aufwand bei der Fehlerbehebung führte. Zusätzlich habe ich sehr mit den Unit tests gestrudelt da meine IDE die Tests nicht als solche erkannt hat - hier hatte ich ein Riesen learning bzgl Properities in .csproj dateien.

Daraus habe ich gelernt:
Unit-Tests sollten von Anfang an in den Entwicklungsprozess integriert werden. Durch das kontinuierliche Schreiben von Tests während der Entwicklung hätte ich Fehler frühzeitig erkannt und beheben können - dies fördert auch ein besseres Design, da testbarer Code oft klarer und modularer ist.

❖ Später Projektstart erfordert selbstständiges Lernen

Ich habe das Projekt später als geplant begonnen, was dazu führte, dass ich viele Technologien und Konzepte eigenständig erlernen musstest, ohne auf Feedback in der Klasse zurückgreifen zu können.
-> Ohne die Möglichkeit, Feedback zu bekommen, musste ich viele Probleme alleine lösen. Dies führte zu längeren Entwicklungszeiten und manchmal zu suboptimalen Lösungen, da der Austausch von Ideen und Peer-Reviews fehlte.

Daraus habe ich gelernt:
Eine frühzeitige Projektplanung und der Aufbau einer unterstützenden Gemeinschaft oder die Zusammenarbeit mit Kollegen können helfen, Herausforderungen effizienter zu meistern.

❖ Die Bedeutung von Asynchroner Programmierung (`async`)

Während der Entwicklung habe ich begonnen, asynchrone Programmierung in deinem Projekt zu implementieren, um die Performance und Skalierbarkeit zu verbessern. Allerdings gab es anfänglich Schwierigkeiten beim Verständnis und der korrekten Anwendung von `async` und `await`.
-> **A**synchrone Programmierung ist komplex und erfordert ein tiefes Verständnis der zugrunde liegenden Konzepte wie Threads, Tasks und Synchronisation. Fehlerhafte Implementierungen können zu Deadlocks, unerwartetem Verhalten und schwer zu diagnostizierenden Bugs führen.

Daraus habe ich gelernt:

- Befolgen bewährter Methoden, wie das Vermeiden von `async void`-Methoden (außer bei Event-Handlern) und das korrekte Verwalten von Ausnahmebehandlungen in asynchronen Methoden.
- Asynchronität schrittweise in Projekte einführen . Beginnen mit I/O-gebundenen Operationen wie Datenbankzugriffen oder HTTP-Anfragen, bevor man komplexere Szenarien angeht.

5.) Protokol

Datum	Dauer in h	Task
25.12.2024	3h	Requirements durchlesen und verstehen. Vorbereitung mittels Moodle Folien
27.12.2024	3h	Projektinitialisierung, Einrichten der Entwicklungsumgebung, Git-Repository erstellt, NPGSQL-Bibliothek hinzugefügt.
27.12.2024	5h	Minimales TCP Server eingebaut. Erstes HTTP-Parsing implementiert. HTTP Antworten formatiert. Erste tests mit curl
28.12.2024	6h	Aufsetzen und Integration von PostgreSQL, Verbindungs aufbaue und Implementierung der Benutzerverwaltung.
28.12.2024	5h	Benutzerauthentifizierung implementiert, Tests mit Curl, Bugfixing
29.12.2024	1h	Ordnerstruktur refaktutiert, namespaces überall hinzugefügt.
29.12.2024	5h	Analyse der Requirements bzgl. DB Model, Erstellung des relationales DB Model (Fremdschlüssel, Constraints).
30.12.2024	4h	Implementierung der ersten REST-Endpunkte (PW ändern funktioniert noch immer nicht, versuch von Bugfix)
30.12.2024	1h	Manuelle Test mit Curl. Überprüfen von HTTP Statuscodes
31.12.2024	6h	Erstellung der Basisklassen für Monster- und Zauberkarten, Definition von Attributen. Implementierung erster Methoden . Testen der Kartenlogik.
01.01.2025	5h	Token Basierte Authentifizierung einbauen. Speicherung und Validierung von Tokens in der Datenbank. Neue Features eingebaut - Vorbereitung für Trading Funktionen
02.01.2025	4h	Problem mit Login gelöst.
03.01.2025	7h	Entwicklung der Handelslogik lt. Anforderungen und Integrationstests mit Curl.
04.01.2025	4h	Arbeiten an der Bettle Logik. Testen. Mit Curl.
07.01.2025	7h	Arbeiten an der Battle Logik. Erfolgreiches Anlegen eines ersten Decks und Starten mit Curl. Erweitern der Battle Logik mit Elementlogik und Spezialregeln
08.01.2025	3h	Weitere Spiel Logik Implementiert - Special Feature. Testen mittels Curl

Datum	Dauer in h	Task
09.01.2025	6h	Immer und immer wieder versucht ein Unit Test Projekt hinzuzufügen - große Probleme da ich compiler Probleme hatte (Mal war es doppelte doppelte Attribute, mal errors das es mehrere Einstiegspunkte im Projekt gibt, Versionskonflikte im Zuge des ausprobieren etc.). Immer wieder andere Konfiguration (unterschiedliche Testing Frameworks probiert, verschiedene C# und .Net Versionen) und nichts hat funktioniert, danach mit git reset zurück zum start und das wiederholt bis ich keine lust mehr hatte.
10.01.2025	2h	Unit Tests endlich erfolgreich hinzugefügt!!! Problem lag in den .csproj dateien (Generate Target Framework Attribute auf False setzen)
10.01.2025	4h	Arbeiten an den Unit Tests
11.01.2024	1h	Dokumentation schreiben, Diagramme erstellen.
12.01.2024	3h	Code nochmal Reaktoren - ungenützte Funktionen rausschmeissen. Code kommentieren
15.01.2024	1h	Dokumentation schreiben, letzte kleine Überprüfungen mit der Checkliste ob alle Kriterien erfüllt sind
16.01.2024	1h	Ertsellen des Curl Scripts
16.01.2024	3h	Dokumentation schreiben, Unit tests nochmal ausgetestet - habe wieder nicht funktioniert. Bugfix

6.) Github Link

https://github.com/petra-scheuer/Monster_Trading_Cards.git