SWEN1 - Final Protokoll MCTG

Samuel Hammerschmidt
`GitHub: github.com/shammerschmidt1999/SWEN1`

Januar 2025

1 Einführung

1.1 Projektstruktur

Das Projekt MCTG ist in mehrere Klassen aufgeteilt. Durch Vererbung und Polymorphismus wird eine einfache Erweiterung des Spiels ermöglicht. Die Klassen sind in verschiedene Dateien und Namespaces aufgeteilt, um die Struktur des Projekts übersichtlich zu halten. Weiters wurden Interfaces implementiert, um die Klassen flexibler und austauschbarer zu machen. Neben ihren Properties besitzen Klassen auch private Fields, dies dient der Steuerung des Zugriffs auf die Eigenschaften. Die Verwendung von Enumerationen verbessert die Lesbarkeit des Codes und ermöglicht eine einfache Erweiterung des Spiels. Die meisten Methoden, die Zugriff auf die Datenbank benötigen oder HTTP-Anfragen verarbeiten, sind asynchron, um die Performance des Spiels zu verbessern.

2 Klassenbeschreibung

2.1 User

Die `User`-Klasse repräsentiert einen Benutzer des Spiels. Um die Daten persistent zu halten, werden User in der Datenbank gespeichert. Die `User`-Klasse ist eine der wichtigsten und am häufigsten verwendeten Klassen im Projekt, da die meisten Operationen von dem `User` abhängen. Änderungen an den User-Objekten werden in der Datenbank gespeichert, um die Daten konsistent zu halten, während Informationen auch direkt von der `User`-Tabelle abgerufen werden.

2.2 Card

Die `Card`-Klasse repräsentiert eine Karte im Spiel. Es gibt verschiedene Arten von Karten, die von der `Card`-Klasse abgeleitet sind, `MonsterCard` und `SpellCard`. Die Karten werden in der Datenbank gespeichert und können von `Usern` gekauft und ins Deck gelegt werden. Diese Operationen werden von der `CardRepository`-Klasse durchgeführt. Da `User`, wie in einem Trading Card Game üblich, mehrere Karten derselben Art besitzen können, wurde die Zwischentabelle `user_cards` erstellt, die die Beziehung zwischen `Usern` und `Cards` repräsentiert.

2.3 Battle

Die `Battle`-Klasse repräsentiert ein `Battle` zwischen zwei `Usern`. Die Karten der `User` werden in einem `Stack` gespeichert und die Karten werden in einer Runde gespielt. Jede Runde wird im `BattleLog` gespeichert und am Ende des Kampfes an beide `User` ausgegeben. Außerdem übernimmt der Gewinner der Runde die Karte des Verlierers. Weiters regelt die `Battle`-Klasse die Berechnung des Schadens der Karten nach den definierten Regeln und prüft den Sieger des Kampfes.

2.4 Package

Ein Objekt der `Package`-Klasse repräsentiert eine Sammlung von Karten, die vom `User` mit seinen `Coins` gekauft werden können. Ein Package speichert seinen `PackageType`, die direkten Einfluss auf den Preis, Anzahl der Karten und Anzahl der Möglichkeiten für den User definiert.

2.5 Stack

Ein Objekt der `Stack`-Klasse repräsentiert eine Sammlung von Karten, die einen `User` gehören. Verwendet werden `Stack`-Entitäten vor allem bei Operationen mit den gesamten `Cards` eines `Users` und den Karten, die der `User` in seinem Deck hat. Unterschieden wird hier durch den `bool inDeck`, der bei einem `Card`-Objekt hinterlegt ist.

2.6 CoinPurse & Coin

`Coins` dienen als Währung im Spiel und haben je nach `CoinType` einen anderen Wert. Die `CoinPurse`-Klasse speichert die Anzahl der `Coins` eines `Users` und ermöglicht dadurch den Kauf von `Packages`.

3 Design- und technische Entscheidungen

3.1 GlobalEnums

Die `GlobalEnums`-Klasse ist eine statische Klasse, die alle Enumerationen enthält, die im Spiel verwendet werden. Sie enthält die folgenden Enumerationen:

- `ElementType`: Die Elementtypen der Karten (Fire, Water, Normal).
- `MonsterType`: Die Monstertypen der Monsterkarten (Dragon, Goblin, Wizard, Knight, Ork, FireElve, Kraken).
- `CoinType`: Die Münztypen (Diamond, Platinum, Gold, Silver, Bronze).
- `RoundResult`: Die Ergebnisse einer Runde (Victory, Defeat, Draw).
- `PackageType`: Die Typen von `Packages` (Basic, Premium, Legendary).

Die Verwendung einer globalen Klasse für die Enumerationen ermöglicht eine einfache Anwendung der Enumerationen in verschiedenen Bereichen des Spiels. Weiters führt die Verwendung von Enumerationen zu einer verbesserten Lesbarkeit des Codes und die

globale Sammlung der Enumerationen ermöglicht eine einfache Erweiterung und eine übersichtliche Struktur.

3.2 Unit Tests

Für die Klassen `User, Card` und `Battle` sowie für das Repository `UserRepository` wurden Unit Tests implementiert. Die Tests der Klassen achten vor allem auf eine richtige Initialisierung der Objekte und das korrekte Verhalten bei Änderungen der Objekteigenschaften. Die Tests des Repositories überprüfen die korrekte Speicherung und den korrekten Zugriff auf die Daten in der Datenbank.

3.3 Unique Feature

User haben die Möglichkeit, `Packages` von verschiedner Qualität zu kaufen. Im `PackageType`-Enum sind die jeweiligen Typen aufgelistet. Ein `Package` kostet bestimmt viele `Coins`, je nach Qualität. Um die `Coins` des `Users` zu speichern, wurde die `CoinPurse`-Klasse implementiert. Das `CoinPurseRepository` und die `PackageService`-Klasse übernehmen die nötigen Schritte für den Kauf von `Packages` in der Datenbank. `Coins` haben außerdem auch einen verschiedenen Wert, je nach `CoinType`. Dies wird im `CoinType`-Enum festgelegt.

3.4 Trading

Die Trading Funktionalität wurde nicht implementiert.

3.5 Datenbank

Die Datenbank wurde in `PostgreSQL` erstellt und beinhaltet die folgenden Tabellen:

- `users`: Speichert die User-Daten.
- `cards`: Speichert die Karten-Daten.
- `user_cards`: Speichert die Beziehung zwischen Usern und Karten.
- `coin_purses`: Speichert die Coin-Daten der User.
- `tokens`: Speichert die Tokens der User.

Die dafür notwendigen Operationen werden von dem zugehörigen Repository durchgeführt. Diese erben alle von der Base-Klasse `RepositoryT`. Der `Connection-String` zur Datenbank wird in einer JSON-Datei `appsettings.json` gespeichert und von der `AppSettings`-Klasse geladen.

3.6 HTTP-Server

Der HTTP-Server wurde implementiert, um die Anfragen der User zu ermöglichen. Der eigentlichen Operation werden in den Handlern, die von der `Handler` Basisklasse erben, durchgeführt. Folgende Handler wurden implementiert:

- `UserHandler`: Verarbeitet die Erstellung, Anmeldung, Änderung und Informationsabfrage von `Usern`.
- `SessionHandler`: Ermöglicht die Anmeldung von `Usern`.
- `BattleHandler`: Ermöglicht das asynchrone Spielen von `Battles` zwischen zwei `Usern`.
- `PackagesHandler`: Dient zum Kauf von `Packages`.
- `ScoreboardHandler`: Ermöglicht das Anzeigen des Scoreboards.
- `CardsHandler`: Ermöglicht das Anzeigen von Karten.
- `DeckHandler`: Ermöglicht das Anzeigen und Bearbeiten von Decks.
- `StatsHandler`: Ermöglicht das Anzeigen der User-Stats.

Der HTTP-Server funktioniert folgendermaßen:

1. In der `Program`-Klasse wird ein `HttpSvr`-Objekt erstellt und mittels der Methode `Run()`gestartet. Der Server lauscht auf eingehende HTTP-Anfragen.
2. Das Incoming-Event des Servers wird mit der asynchronen Methode `Svr_Incoming()` verknüpft, die aufgerufen wird, wenn eine Anfrage eingeht.
3. Die Methode `Svr_Incoming()` empfängt die Anfrage und gibt die HTTP-Methode, den Pfad und die Header der Anfrage auf der Konsole aus.
4. Basierend auf dem Pfad der Anfrage wird entschieden, welcher `Handler` die Anfrage weiterverarbeiten soll. Dies geschieht durch die `HandleEventAsync()` Methode der `Handler`-Klasse.
5. Sollte kein passender Handler gefunden werden, wird eine 404-Fehlermeldung zurückgegeben.
6. Die `HttpSvrEventArgs`-Klasse enthält Informationen über die eingehende HTTP-Anfrage, einschließlich Methode, Pfad, Header und Payload.
7. Die Methode `Reply` der `HttpSvrEventArgs`-Klasse wird verwendet, um eine HTTP-Antwort an den Client zu senden, einschließlich Statuscode und optionalem Antworttext.

3.7 Tokens

Beim Einloggen eines `Users` wird ein `Token` generiert und an den Client zurückgegeben. Dieser `Token` wird im weiteren Verlauf des Projektes bei jeder Anfrage des Clients an den Server mitgeschickt und dient zur Authentifizierung des Benutzers. Der `Token` wird in der Datenbank gespeichert und mit dem `User` verknüpft. Der `Token` wird wie folgt generiert:

1. Nach dem der entsprechende Endpunkt (`/sessions`) vom Client erhalten wurde, wird die `_CreateSessionAsync()`-Methode des `SessionHandler` aufgerufen.

2. In der Anfrage sind ein username und password string enthalten, diese werden mit den Daten in der `users` Datenbank verglichen.

3. Bei einem Match wird die `_CreateTokenForAsync()`-Methode der `Token`-Klasse aufgerufen, die einen zufälligen Token erstellt und in der Datenbank gespeichert. Der generierte Token wird in der Reply mitgeschickt. Ansonsten wird eine 401-Statuscode Fehlermeldung als Reply returniert.

3.8 Interfaces

Für die meisten Klassen wurden Interfaces implementiert. Dies hat den Grund, dass die Klassen dadurch flexibler und austauschbarer werden. So kann z.B. ein `MonsterCard`-Objekt als `Card`-Objekt behandelt werden, wenn es das `ICard`-Interface implementiert. Dadurch können Methoden, die ein `Card`-Objekt erwarten, auch ein `MonsterCard`-Objekt akzeptieren. Das führt zu einer verbesserten Lesbarkeit und Wartbarkeit des Codes und ermöglicht eine einfachere Erweiterung des Spiels.

4 Lessons Learned

4.1 Datenbank

Die Verwendung von `PostgreSQL` als Datenbank hat sich als gute Wahl erwiesen. Die Datenbank ist einfach zu bedienen und bietet eine gute Performance. Weiters bietet das grafische Interface von pgAdmin 4 als Verwaltungstool eine einfache Möglichkeit, die Datenbank zu verwalten und Abfragen auszuführen. Weiters half mir pgAdmin 4 speziell am Anfang der Datenbank-Entwicklung, da das Tool simpel zu bedienen ist und eine gute Übersicht über die Datenbankstruktur bietet.

4.2 Unit Tests

Die Unit Tests wurden mit dem `MSTest`-Framework implementiert. Vor allem die dadurch entstandene Trennung der Tests und des eigentlichen Codes, ermöglicht eine gute Übersicht des Repositories. Weiters hat das kontinuierliche Entwickeln von Tests während der Funktionsimplementierung dazu geführt, dass die Klassen und Methoden besser strukturiert und getestet wurden.

4.3 Klassenstruktur

Die Erstellung von eigenen Klassen für spezielle Operationen, wie z.B. `AppSettings` und `PasswordHelper`, hat sich als sinnvoll erwiesen, um den Code übersichtlicher zu halten und Verwirrung zu vermeiden.

4.4 C# und Projektarbeit

Zwar habe ich im Laufe meiner ersten Betriebspraxisphase bereits Erfahrung mit C# gesammelt, jedoch war die Entwicklung eines größeren Projektes ohne ein Team eine neue Herausforderung für mich. Durch die Entwicklung des MCTG-Projektes konnte ich meine

Kenntnisse in C# vertiefen und habe mich auch darin verbessert, ein Projekt alleine zu verwalten und auf eigenständige, für mein Projekt passende Lösungen zu finden.

5 Timetable

Arbeitspaket	**Zeit (Stunden)**
Erste Projektstruktur	2
Recherchen	3
Erste Erstellung der Klassen	5
Implementierung des HTTP-Servers	10
Erstellung Intermediate Abgabe	5
Datenbankstruktur	7
Datenbankanbindung	2
Erstellung der Repositories	15
Implementierung Unique Feature	7
Anpassung des Codes von In-Memory zu DB	12
(Unit) Testing	6
Battle & Asynchronität	10
Final Refactoring	8
Erstellung der finalen Abgabe	4
Summe	**96**

Tabelle 1: Timetable - MCTG Projekt

6 UML-Diagramme

6.1 Class Pattern

Hinweis: Hier wird nur die `Stack`-Class gezeigt, um das Pattern der Classes zu darzustellen. Der Aufbau ist für sämtliche Classes ähnlich.

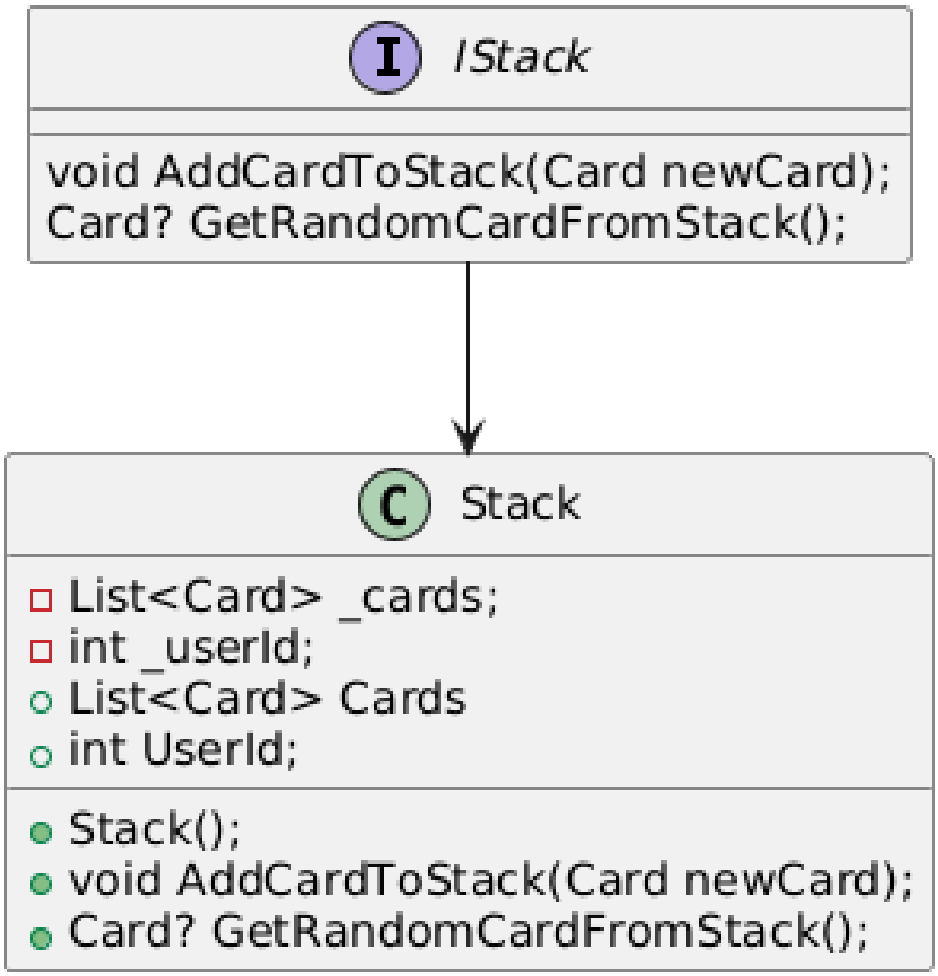

Abbildung 1: UML-Diagramm des Class Patterns

6.2 Repository Pattern

Hinweis: Hier wird nur das `UserRepository` gezeigt, um das Pattern der Repositories zu darzustellen. Der Aufbau ist für sämtliche Repositories ähnlich.

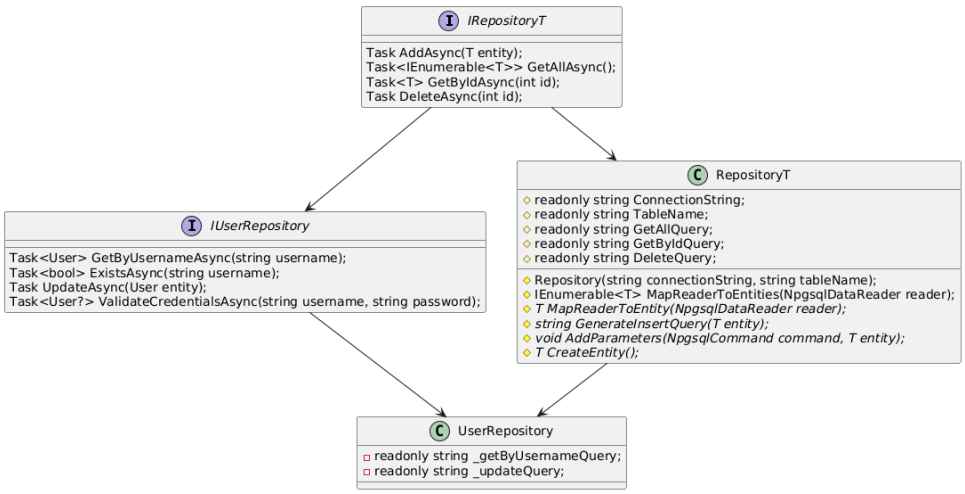

Abbildung 2: UML-Diagramm des Repository Patterns

6.3 Handler Pattern

Hinweis: Hier wird nur der `UserHandler` gezeigt, um das Pattern der Handler zu darzustellen. Der Aufbau ist für sämtliche Handler ähnlich.

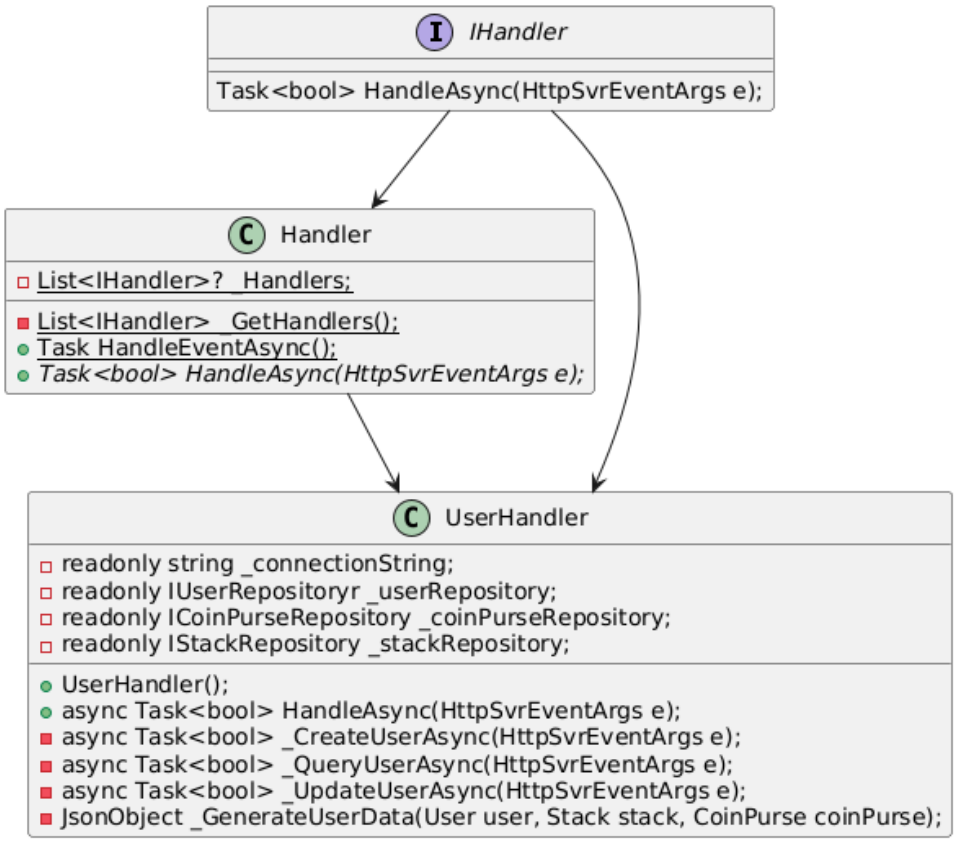

Abbildung 3: UML-Diagramm des Handler Patterns