SWEN1 - Intermediate Protokoll MCTG

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

November 2024

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. Der integrierte HTTP-Server ermöglicht die Benutzerregistrierung und -anmeldung. Benutzer werden derzeit noch nicht in einer Datenbank gespeichert, sondern nur im Arbeitsspeicher. Die Implementierung der Datenbankanbindung ist für das Final Hand-In geplant.

1.2 Anforderungen für Intermediate Hand-In

Für die Zwischenabgabe stehen vor allem die Klassen `User` und `Card` sowie die für den HTTP-Server benötigten Klassen im Fokus. Diese werden im Laufe des Protokolls genauer beschrieben. Weitere technische Entscheidungen werden ebenfalls erklärt.

2 Klassenbeschreibung

2.1 User

Die `User`-Klasse repräsentiert einen Benutzer des Spiels. Sie enthält die folgenden Attribute:

- `static Dictionary<string, User> _Users`: Eine Liste von Benutzern, die im Spiel registriert sind.
- `string Username`: Der Benutzername des Spielers.
- `string Password`: Das Passwort des Spielers.
- `int Elo`: Die Elo-Zahl des Spielers.
- `Stack UserCards`: Eine Liste von Karten, die der Spieler besitzt.

- `Stack UserDeck`: Eine Liste von Karten, die der Spieler in seinem Deck hat (Untermenge von `UserCards`).
- `Stack UserHand`: Eine Liste von Karten, die der Spieler in der Hand hält (Untermenge von `UserDeck`).
- `Stack UserDiscard`: Eine Liste von Karten, die der Spieler abgeworfen hat (Untermenge von `UserDeck`).

Die `User`-Klasse enthält die folgenden Methoden:

- `void PrintUser()`: Gibt die Informationen des Spielers auf der Konsole aus.
- `void PrintStack(Stack stack)`: Gibt die Informationen einer Kartenliste auf der Konsole aus.
- `static void Create(string username, string password)`: Erstellt einen neuen Benutzer und fügt ihn zur Liste der registrierten Benutzer hinzu.
- `static (bool Success, string Token) Logon(string username, string password)`: Loggt den Benutzer ein und gibt ein Token zurück, falls die Anmeldedaten korrekt sind.
- `void Save(string token)`: Ändert die Informationen des Benutzers und speichert diese.
- `static User? Get(string userName)`: Gibt den Benutzer mit dem gegebenen Benutzernamen zurück.
- `static bool Exists(string userName)`: Überprüft, ob ein Benutzer mit dem gegebenen Benutzernamen existiert.

Ein User wird über den integrierten HTTP-Server erstellt und eingeloggt.

2.2 Card

Die `Card`-Klasse repräsentiert eine Karte im Spiel. Sie enthält die folgenden Attribute:

- `string Name`: Der Name der Karte.
- `double Damage`: Dier Schaden, den die Karte verursacht.
- `ElementType ElementType`: Der Elementtyp der Karte.

Die `Card`-Klasse enthält die folgenden Methoden:

- `abstract void PrintInformation()`: Abstrakte Methode, die die Informationen der Karte auf der Konsole ausgibt.

Die `Card`-Klasse ist eine abstrakte Klasse, von der die Klassen `MonsterCard` und `SpellCard` erben.

2.3 MonsterCard

Die `MonsterCard`-Klasse repräsentiert eine Monsterkarte im Spiel. Sie erbt von der `Card`-Klasse und enthält folgende zusätzliche Attribute:

- `MonsterType MonsterType`: Der Monstertyp der Karte.

Die `PrintInformation()`-Methode wird überschrieben, um die Informationen der Monsterkarte auf der Konsole auszugeben.

2.4 SpellCard

Die `SpellCard`-Klasse repräsentiert eine Zauberkarte im Spiel. Sie erbt von der `Card`-Klasse und enthält keine zusätzlichen Attribute. Die `PrintInformation()`-Methode wird überschrieben, um die Informationen der Zauberkarte auf der Konsole auszugeben.

2.5 Weitere Klassen

Es wurden bereits zahlreiche weitere Klassen implementiert, (z.B. Battle, Stack, CoinPurse, etc.) die jedoch nicht im Detail beschrieben werden, da diese für das Intermediate Hand-In nicht relevant sind.

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).

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` wurden bereits Unit Tests implementiert. Diese überprüfen vor allem die korrekte Konstruktion der Klassenobjekte. Die `Battle`-Klasse Unit Tests achten auf die korrekte Berechnung des Schadens und des Gewinners der Runde. Allerdings ist die `Battle`-Klasse und die Tests dafür nicht für diese Abgabe relevant. Weitere Unit Tests werden für das Final Hand-In implementiert.

3.3 HTTP-Server

Der HTTP-Server wurde implementiert, um die Benutzerregistrierung und -anmeldung zu ermöglichen. Der Server enthält die folgenden Endpunkte:

- `POST /user`: Erstellt einen neuen Benutzer.
- `POST /sessions`: Loggt einen Benutzer ein.

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 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 `HandleEvent()` Methode der `Handler`-Klasse.
5. Wenn der Pfad `/users/` enthält, wird die Anfrage an den `UserHandler` weitergeleitet. Wenn der Pfad `/sessions` enthält, wird die Anfrage an den `SessionHandler` weitergeleitet. Andernfalls wird eine 404-Fehlermeldung zurückgegeben.
6. Jeder spezifische `Handler` erbt von der `Handler`-Klasse und kann die `Handle`-Methode überschreiben, um spezifische Logik für die Verarbeitung der Anfrage zu implementieren.
7. Die `HttpSvrEventArgs`-Klasse enthält Informationen über die eingehende HTTP-Anfrage, einschließlich Methode, Pfad, Header und Payload.
8. Die Methode `Reply` der `HttpSvrEventArgs`-Klasse wird verwendet, um eine HTTP-Antwort an den Client zu senden, einschließlich Statuscode und optionalem Antworttext.

3.4 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 derzeit im Arbeitsspeicher gespeichert. Der Token wird wie folgt generiert:

1. Nach dem der entsprechende Endpunkt (`/sessions`) vom Client erhalten wurde, wird die `_CreateSession()`-Methode des `SessionHandler` aufgerufen.
2. In der Anfrage sind ein username und password string enthalten, diese werden mit den Daten in der `_Users`-Liste verglichen.

3. Bei einem Match wird die `_CreateTokenFor()`-Methode der `Token`-Klasse aufgerufen, die einen zufälligen Token erstellt und im `internal static Dictionary<string, User> _Tokens` gespeichert. Der generierte Token wird in der Reply mitgeschickt. Ansonsten wird eine 401-Statuscode Fehlermeldung als Reply returniert.

3.5 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.

3.6 Fields und Properties

Für jede `public` Property wurde ein `private` Field erstellt, um den Zugriff auf die Eigenschaften zu steuern.

4 UML-Diagramme

4.1 Klassendiagramm User und Card

Abbildung 1: Klassendiagramm der Klassen User und Card

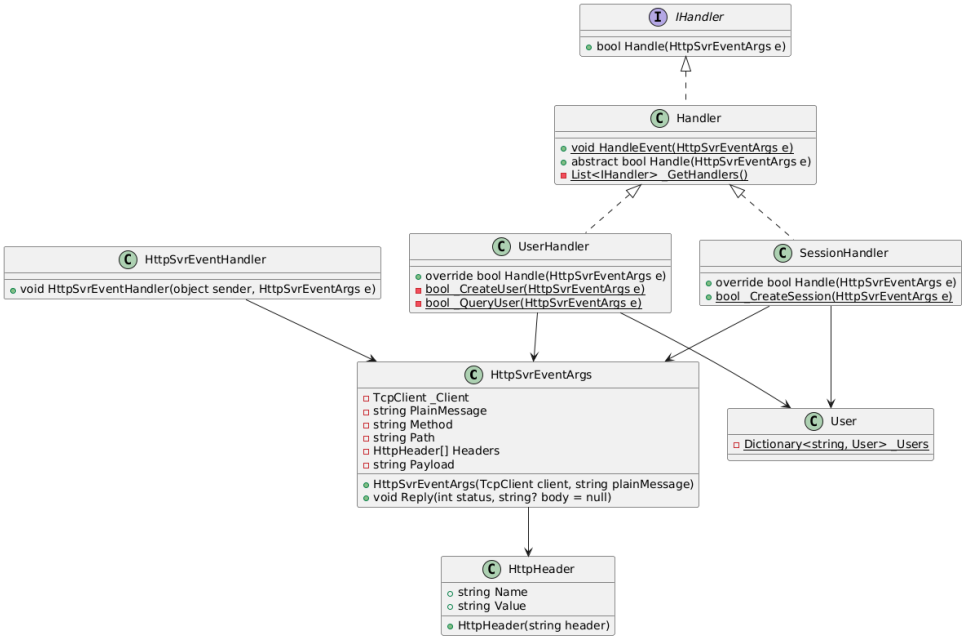

Abbildung 2: Klassendiagramm der Klassen HTTP-Server und Handler Klassen