Beschreibungen und Anmerkungen zu MCTG

Inhalt

[GitHub Link 1](#_Toc187091471)

[Klassendiagramm 2](#_Toc187091472)

[Layer 3](#_Toc187091473)

[Erklärung der Klassen 4](#_Toc187091474)

[DatabaseConnector 4](#_Toc187091475)

[Main 4](#_Toc187091476)

[HttpServer 4](#_Toc187091477)

[HttpRequestLine und HttpHeaders 4](#_Toc187091478)

[RequestHandler 4](#_Toc187091479)

[PostRequestHandler 4](#_Toc187091480)

[GetRequestHandler 5](#_Toc187091481)

[PutRequestHandler 5](#_Toc187091482)

[Card 5](#_Toc187091483)

[CardFactory 5](#_Toc187091484)

[Package 5](#_Toc187091485)

[PackageService 5](#_Toc187091486)

[Stack 5](#_Toc187091487)

[User 6](#_Toc187091488)

[UserService 6](#_Toc187091489)

[Deck 6](#_Toc187091490)

[AuthorisationService 6](#_Toc187091491)

[Battle 7](#_Toc187091492)

[Unique Feature 7](#_Toc187091493)

[Unit Tests 7](#_Toc187091494)

[Anmerkungen: 7](#_Toc187091495)

[Zeitinvestment 8](#_Toc187091496)

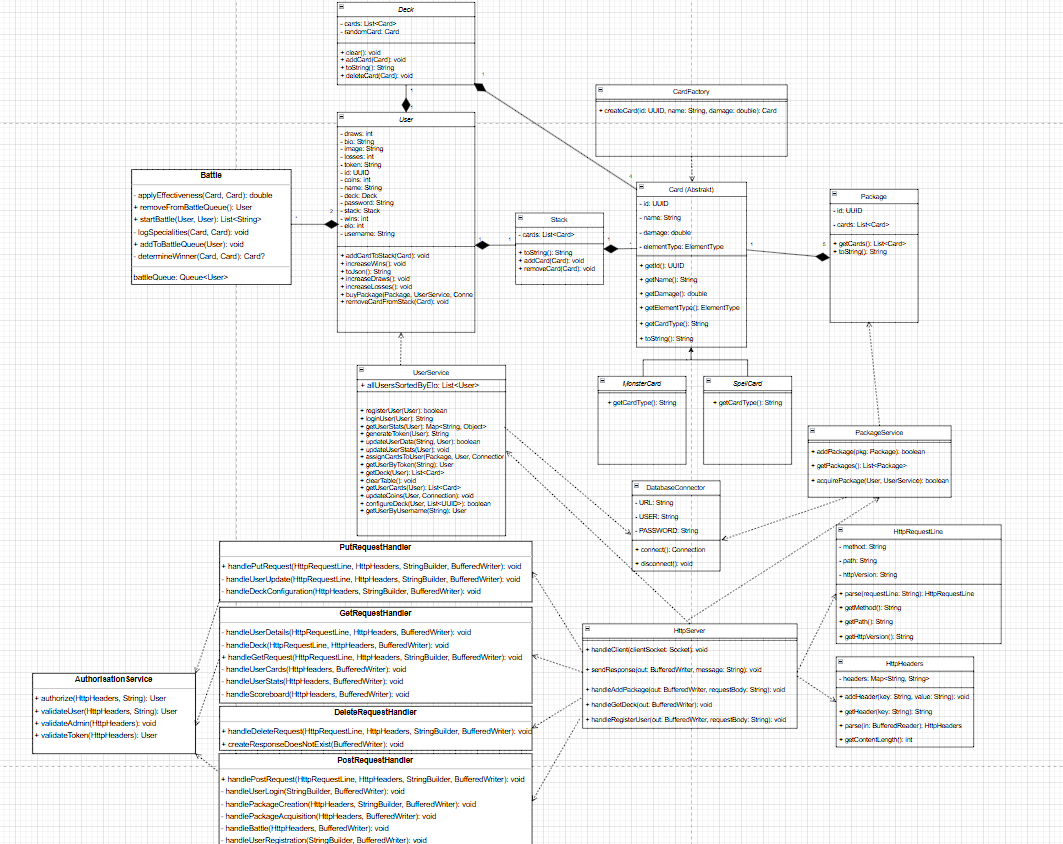
[Learned Lessons 9](#_Toc187091497)

# GitHub Link

https://github.com/Marcel19985/MCTG\_Goessl

# Klassendiagramm

Eine detaillierte Struktur kann dem Klassendiagramm entnommen werden. Dieses wurde mit draw.io erstellt und kann daher mit dieser Anwendung geöffnet werden. Ich habe mich für draw.io entschieden, weil wir dafür eine kurze Einführung im letzten Semester erhalten haben und ich seitdem damit arbeite. Eine .io Datei befindet sich in der Abgabe. Überblick:



# Layer

services: kümmern sich um Methoden, bei denen mit der Datenbank interagiert wird.

server: kümmert sich um Client Requests und Responses

models: Modellklassen

handlers: teilen die http Methoden auf, um die CRUD Anfragen voneinander zu trennen

database: stellt Verbindung zur Datenbank her

# Erklärung der Klassen

## DatabaseConnector

Stellt eine Verbindung zur Datenbank mit den jeweiligen Eigenschaften (URL, Username und Passwort) her.

## Main

void main(…): Startet einen ServerSocket auf Port 10001 (Portnummer ist im CURL Script vorgegeben). Sobald sich ein Client verbindet, wird ein neuer Thread mithilfe eines Thread- Pools (der Größe 10) gestartet und der clientSocket wird in der Klasse HttpServer weiterverarbeitet. Der Threadpool wurde mit 10 Threads gewählt, da Moderne Rechner etwa so viele Threads unterstützen. Zum Beispiel bietet ein Einsteiger AMD Ryzen 3 Prozessor 8 Threads, wärend ein Highend AMD Ryzen 9 Prozessor 24 Threads unterstützt.

## HttpServer

void handleClient(…): Liest aus dem clientSocket die HTTP Request Line und die headers aus und speichert diese jeweils als Objekt. Der Request Body wird als String gespeichert und weiterverarbeitet. Abhängig von der HTTP Methode wird eine der folgenden Methoden aufgerufen, die die jeweilige CRUD Operation ausführt:

* handlePostRequest
* handleGetRequest
* handlePutRequest
* handleDeleteRequest

Für jede dieser Methode wurde eine eigene Klasse erstellt, die sich im Package „handlers“ befinden. Diese Struktur habe ich erstellt, nachdem der Code sehr unübersichtlich in der handleClient Methode wurde (zunächst wurden alle Methoden in if-else Blöcken abgearbeitet, wodurch seit dem Hinzufügen aller API Endpoints sehr vielen Zeilen entstanden sind).

## HttpRequestLine und HttpHeaders

Durch diese Klassen lassen sich die Request Line und Headers als Objekte abspeichern. Die Klassen enthalten nützliche Methoden, um ein Request übersichtlicher bearbeiten zu können.

## RequestHandler

Die nachfolgenden vier RequestHandler ähneln sich in ihrer Struktur. Für jeden Pfad gibt es eigene Methoden, in denen die Methoden anderer Klassen aufgerufen werden. Diese geben ein Objekt zurück, welches in ein JSON umgewandelt und ausgegeben wird.

### PostRequestHandler

handlePostRequest(…): Basierend auf dem Pfad („/users“ für die Registrierung, „/sessions“ für die Anmeldung oder „/packages“ für das Erstellen neuer Packages, "/transactions/packages" für das kaufen von Packages und "/battles" um ein Battle zu starten) wird die jeweilige Anfrage verarbeitet. Der Ablauf ist bei den ersten beiden ähnlich: Aus den JSON-Daten wird ein Objekt erstellt und die Registrierung/Anmeldung durchgeführt. Abhängig vom Erfolg der Durchführung, wird die jeweilige Response erstellt und an den Client zurück gesendet.

Bei der Erstellung neuer Packages wird zunächst geprüft, ob ein Admin-Token übermittelt wurde. Danach wird das übermittelte JSON in einer Liste gespeichert, in der jedes Element eine Map (Key-Value Pair) ist. Es ist noch nicht möglich, Card Objekte zu erstellen, da noch nicht klar ist um welches Objekt es sich handelt (MonsterCard vs. SpellCard). Hier habe ich mich nämlich für Vererbung entschieden, weil die Karten für die Battles unterschiedliche Implementierungen bekommen werden. Mithilfe einer CardFactory- Klasse werden aus jedem Elemente der Liste (jede Karte) zu den jeweiligen Card- Objekten. Diese werden in der Datenbank in den Tabelle Cards und Packages abgebildet.

### GetRequestHandler

Hier werden alle Get-Requests weiterverarbeitet. Darunter befinden sich die Pfade "/users/" um die Userdaten auszugeben, "/cards" um die gekauften Karten anzuzeigen, "/deck"um ein konfiguriertes Deck auszugeben, "/stats" um die Spielerstatistik des anfragenden Spieler anzuzeigen und "/scoreboard" um eine Liste an ELO- Werten sortiert auszugeben. Jeder Pfad wird von einer eigenen Methode abgearbeitet, die jeweils Methoden anderer Klassen aufruft und das zurückgegebene Objekt als JSON ausgibt.

### PutRequestHandler

Der PutRequestHandler kümmert sich um die Put-Anfrage der Pfade "/deck" (um ein Deck mit Karten zu befüllen) und "/users/" (um Benutzerdaten zu aktualisieren).

## Card

Diese Klasse enthält alle Attribute für eine Karte laut Angabe. Der ElementType (Fire, Water oder Normal) wurde als ENUM definiert. Konstruktor und Getter sind implementiert. Es handelt sich um eine abstrakte Klasse, da die Methode getCardType() erst in den Kindklassen SpellCard und MonsterCard implementiert werden.

In der Datenbank besteht eine 1:n Beziehung zwischen cards und packages, wodurch cards einen Fremschlüssel auf packages enthalten.

## CardFactory

Card createCard(…): Erstellt eine Karte der Kindklassen (SpellCard oder MonsterCard) basierend auf dem Namen, der im CURL command übergeben wird.

## Package

Enthält eine Liste für 5 Karten und dazugehörige getter und einen Konstruktor. Ich habe mich für eine Liste als Container entschieden, um bei Bedarf die Größe der Packages verändern zu können. Um Leistungs- und Speichereffizienz zu erhöhen könnte stattdessen auch ein Array verwendet werden.

## PackageService

boolean addPackage(…): Fügt ein Package und die zugehörigen Karten in die Datenbank ein.  
Dabei wird darauf geachtet, dass entweder beide Datenbank query’s erfolgreich durchgeführt werden oder keines davon. Damit wird verhindert, dass ein leeres Package (also ohne Karten) hinzugefügt wird.

acquirePackage(User user, UserService userService): Wählt ein Paket aus der Tabelle Packages aus und überträgt die Karten dieses Packages dem übergebenen User. Das Package wird als der Tabelle gelöscht, sodass es kein weiterer User erwerben kann.

deletePackageById(UUID packageId, Connection conn): löscht ein Package mit der übergebenen Package-ID. (wird von acquirePackage als Hilfmethode verwendet).

## Stack

Enthält eine Liste für alle Karten, die ein Benutzer besitzt und zugehörige Methoden: hinzufügen und entfernen von Karten um Tradings in der Zukunft implementieren zu können.

## User

Enthält alle Attribute für den User. Derzeit werden noch nicht alle verwendet, da diese erst bei späteren CURL Befehlen eine Rolle spielen. Da die Attribut- Namen mit kleinem Anfangsbuchstaben beginnen sollen, werden diese mithilfe von @JsonProperty zugewiesen. In den CURL Befehlen werden nämlich großgeschriebene Keys übergeben. Ich habe einige Methoden hinzugefügt (z.B. für den Kauf von Packages, um Karten vom Stack zu entfernen oder hinzuzufügen, getter und setter. Viele davon werden derzeit noch nicht verwendet.

## UserService

Kümmert sich um die Datenbank inserts der user:

boolean registerUser(User user): Stellt eine Datenbankverbindung her und fügt das übergebene Objekt in die Tabelle users ein. Falls der Benutzer bereits existiert, wird eine SQLException geworfen, da der username unique sein muss. Das braucht weniger Rechenaufwand als separat (extra SQL Abfrage) zu überprüfen, ob ein Benutzer bereits existiert.

generateToken(User user): Generiert einen Token basierend auf dem übergebenen user- Objekt. An den Benutzernamen wird „-mtcgToken“ angehängt.

loginUser(User user): Meldet einen Benutzer an wenn ein zusammengehöriges Paar an Benutzernamen und Passwort übergeben wurde. Der Token wird nach der Anmeldung an den Client zurückgegeben.

updateCoins(User user, Connection conn): Updated die Anzahl an Coins eines Users.

assignCardsToUser(Package pkg, User user, Connection conn): Weist dem übergebenen User das übergabene Package (also die darin befindlichen Karten) zu.

getUserCards(User user): gibt eine Liste an Karten zurück, die zum übergebenen User gehören.

getUserByToken(String token): Ruft die notwendigen Daten eines Benutzers ab, erstellt dessen Deck und gibt das User Objekt zurück.

getDeck(User user): Ruft das Deck eines Users ab und gibt dieses zurück.

getUserByUsername(String username): Ruft einen Benutzer anhand seines Username ab, konstruiert ein User Objekt und gibt dieses zurück.

configureDeck(User user, List<UUID> cardIds): Weist einem Benutzer ein Deck zu (4 Karten).

updateUserData(String username, User updatedData): Aktualisiert die Daten eines Benutzers.

updateUserStats(User user): Aktualisiert die Statistik eines Benutzers nach einem Battle.

getUserStats(User user): gibt die Statistik eines übergebenen Users zurück.

getAllUsersSortedByElo(): ruft alle Benutzer, deren statistik und ELO ab und gibt eine Liste an User zurück, die nach dem ELO- Wert sortiert sind.

## Deck

Besteht aus einer Liste an 4 Karten und die dazugehörigen Methoden (getter setter usw.) um diese zu bearbeiten.

## AuthorisationService

Stellt Methoden zur Verfügung, die häufig für die Autorisierung von Benutzern gebraucht werden.

validateAdmin(HttpHeaders headers): überprüft, ob sich in der Anfrage der Admin Token befindet.

validateUser(HttpHeaders headers, String username): gibt den User basierend auf dem Username zurück. Dabei wird überprüft, ob der Username zum übergebenen Token passt.

validateToken(HttpHeaders headers): Ruft den Benutzer basierend auf dem übergebenen Token ab.

authorize(HttpHeaders headers, String username): überprüft, ob der username mit dem übergebenen Token übereinstimmt.

## Battle

Besteht aus einer Warteschlange für Spieler. Es wird erst gestartet, sobald zwei Spieler von unterschiedlichen Threads spielen möchten. Im Battle wird die Spiellogik rundenbasiert abgearbeitet. In jeder Runde treten zwei Karten gegeneinander an. Gewinner ist der Spieler, dessen Gegner zuerst alle Karten verliert.

Die Ereignisse werden in eine Liste geschrieben, die am Ende des Battles als Log ausgegeben wird.

### Unique Feature

Wenn eine Runde unentschieden ausgeht (beide Karten gleich stark), werden verlieren beide Spieler ihre Karten dieser Runde.

Es gibt eine rundenbasierte Siegesserie, die beim 3. Aufeinanderfolgenden Rundensieg zum Einsatz kommt. Die Damage der derzeitigen Karte wird dann per Zufall verändert (33% Chance für keine Veränderung, 33% Chance für verdoppeln, 33% Chance für verdreifachen).

Beide eingebauten Änderungen sollen dazu führen, dass die meisten Battles mit einem Sieger und einem Verlierer enden. Zuvor ist es nämlich häufig zu einem Unentschieden gekommen, da meistens jeder Spieler zumindest eine sehr starke Karte hatte, durch die eine Niederlage verhindert wurde.

# Unit Tests

Meine Vorgehensweise war es, nach der erfolgreichen Implementierung eines Features, Unit Tests dafür zu generieren. Dadurch wollte ich sicherstellen, dass die Funktionalität nicht durch spätere Änderungen beeinträchtigt wird. Ich habe versucht, Methoden bei Möglichkeit einzeln und unabhängig voneinander zu testen. Dies war nicht immer einfach bzw. möglich, da viele Methoden weitere Methoden aufrufen.

Dabei habe ich JUnit und Mockito eingesetzt. Dadurch konnte ich einige Operationen Mocken, um kleinere Codeteile eigenständig zu testen.

Es gibt für nahezu jede Klasse eine eigene Testklasse, um diese zu Testen. Insgesamt gibt es 56 Tests.

# Anmerkungen:

Bei allen Tabellen der Datenbank wird eine UUID als Primärschlüssel verwendet. Diese ist sicherer gegenüber einem fortlaufenden integer und die ID’s sind global eindeutig.

Bei allen Methoden, die Datenbank query’s ausführen, werden SQL exceptions geworden und gecatched, um etwaige Fehlermeldungen zu sehen.

Für die Umwandlung von JSON in Java Objekte habe ich mich für Jackson entschieden. Dazu habe ich die meisten Informationen im Internet gefunden.

Die PostgreSQL Datenbank verwende ich über einen Docker, um Speicherplatz am Rechner zu schonen. Zu Docker gibt es einige Materialien im Internet, wodurch das Aufsetzen rasch möglich war.

Bis auf Trading’s wurden alle Anforderungen der Excel-Liste Implementiert und haben nach mehrmaligen Testen des CURL-Scripts ihre Funktionalität gezeigt.

# Zeitinvestment

Die nachfolgende Einschätzung basiert auf einer Schätzung, da ich meine Arbeitszeit nicht exakt getrackt habe:

Bis zur Zwischenabgabe habe ich gut 30 Stunden investiert. Das setzt sich aus Programmieren und Internet Recherchen zusammen. Die größte Herausforderung waren:

* Die Projektanforderungen zu verstehen und gedanklich zu strukturieren. Zu Beginn ist es mir schwer gefallen, das Spiel zu verstehen und hatte auch keine Idee, wie eine Implementierung aussehen könnte, da ich noch kein Wissen über HTTP+REST und Java hatte.
* Implementierung von Teilen, die noch nicht Teil des Unterrichts waren  
  (z.B. ist der Grundstein für dieses Projekt HTTP+REST, was erst Thema von Class 8 ist -> daher habe ich beim Beginn des Projektes einiges durch Internetrecherchen erarbeitet)
* Wissen rund um Datenbanken auffrischen, da das erste Semester schon ein bisserl her ist
* Hinzufügen von Packages inklusive Cards war zeitaufwendig, da die Zuteilung der Package-Kindklassen basierend auf dem Karten- Namen erfolgt. Herausforderungen waren das Parsing des Requests und das handling der Klassen.

Ab der Zwischenabgabe habe ich etwa 60 Stunden investiert. Die meiste Zeit hat das Programmieren von acquire Packages, configure Deck und Battles benötigt. Die Herausforderungen waren:

* Mehr Fokus auf Unit Tests: Sinnvolle Tests erstellen hat sich als schwieriger als geplant erwiesen. Das liegt unter anderem Daran, dass einige Methoden mit der Datenbank interagieren bzw. viele Methoden weitere Methoden aufrufen, was unabhängiges Testen erschwert. Das Einbauen von Mocking mithilfe von mockito hat Probleme bereitet. Häufig wurden die Dependencies im .pom File als Fehler markiert bzw. musste Maven mehrmals aktulisiert werden.
* Datenbankoperationen und Aufbau der Datenbank: Vorallem acquire packages und configure decks hat einige debug Arbeitsstunden benötigt, da bei diesen Vorgängen mehrere Tabellen mit Fremdschlüssel angepasst werden müssen.
* Aquire Packages: Da hier alle Karten eines Packages einem User zugeordnet werden müssen, benötigt diese Methode Datenabankabfragen der Tabellen Packages und Cards. Damit die Datenbank bei verschiedenen Anfragen in einem korrekten Zustand gehalten wird, musste ich einiges testen und Debuggen.
* Configure Deck: Auch hier sind komplexe Datenbankabfragen notwendig, bei denen die Tabellen Cards und Decks verändert werden müssen.
* Die Implementierung der Battles inklusive Stats, Scoreboard und unique feature waren hingegen einfacher als geplant. Da die Funktionen rund um Battles einen großen Teil der erreichbaren Punkte ausmachen, hatte ich dafür mehr Zeit eingepalnt. Die Logik der Battles (welche Karte gewinnt gegen andere etc.) konnte ich relativ schnell implementieren. Die Herausforderung war allerdings das Starten eines Battles laut Curl Befehl. Hier habe ich nach Recherchen eine Warteschlange implementiert, damit sich zwei Spieler für ein Battle finden. Das Log des Battles wird in einer Liste gespeichert und nach dem Ablauf ausgegeben. Dadurch muss nicht ständig zwischen dem PostRequestHandler und der Battle Klasse kommuniziert werden.

# Learned Lessons

Bei diesem Programm handelt es sich um die vermutlich umfangreichste Applikation, die wir bis jetzt im Studium programmiert haben. Dadurch gab es auch einige learned lessons:

* IntelliJ bietet viele praktische Funktionen:
  + Klassen können nicht nur schnell umbenannt werden, sondern auch in allen Codeteilen automatisch umbenannt werden
  + Das Erstellen von Gettern und Settern funktioniert für die grundlegenden Funktionen automatisch
  + Das erstellen von Testklassen funktioniert automatisch
  + Klassen können per Drag und Drop leicht verschoben und refactored werden
  + Autocomplete ist praktisch, um Tipparbeit zu sparen
* Sollte es Probleme mit Dependencies geben, hilft das refreshen der Maven Dependencies (ab und zu muss das öfter gemacht bzw. etwas gewartet werden)
* Die Vorteile der objektorientierten Programmierung wurden spätestens bei der Battle Implementierung klar: Programme können sehr übersichtlich gestaltet werden und Methoden können Objekte anstatt unzähliger Attribute übergeben werden. Methoden können logisch abstrahiert, also vereinfacht betrachtet werden, indem Codeteile in andere Klassen ausgelagert werden. Bei einer sinnvollen Benennung von, muss nicht ständig nachgeschaut werden, was sie macht.
* Es ist sinnvoller im Vorhinein genau zu überlegen, wie man ein Programm strukturiert als es im Nachhinein zu ändern. Das ist natürlich eine Herausforderung, da wir im Studium bis jetzt selten mit so umfangreichen Programmen gearbeitet haben.
* Durch die ersten zwei Semester, hatte ich den Eindruck, dass Programmier- Syntax auswendig gekonnt werden muss. Bei diesem Projekt war es jedoch sinnvoller, sich auf Programmstruktur zu fokussieren und Syntax bei Bedarf nachzuschlagen.
* Es ist zeitsparender Unit Tests zu kreieren als nach allen Änderungen das gesamte Curl Script durchlaufen zu lassen.
* Datenbankoperationen sind meist ähnlich aufgebaut und bieten sehr viele SQL Funktionen. Es ist meist nicht notwendig, Programmlogik neu zu erfinden. Z.B. Nutzen von Constraints und SQL Exceptions