Protocol Stefan Werner MTCG

Inhalt

GitHub Repo:
https://github.com/RxndyOG/SWEN_MTCG_StefanWerner.git

Die Abgabe MTCG von Stefan Werner geht bis Punkt 20 im CURL script. Es werden Request geparsed und durchgeführt. Nach Abschluss einer Request wird dem Client eine Response geschickt mit Codes je nachdem ob es funktioniert hat oder ob Fehler passierten.

Das Programm besteht aus sieben Klassen. Diese teilen sich in „Unterklassen“ und „Hauptklassen“ auf. Das Word „Unterklasse“ wird hier benutzt, um Klassen zu beschreiben die benutzt werden um bestimmte Konzepte wie Karten oder Packages dazustellen.

Klassenbeschreibung:

TCPServer Klasse:

Die TCPServer Klasse ist der Anfang des Programms. Mit der Funktion „Start()“ wird der Server gestartet. In dieser Funktion wird in einer while schleife jedes Mal, wenn ein User sich auf der IP und dem Port befindet ein Client angelegt, der auf einem eigenen Thread arbeitet. Die jeweiligen Routen, die dem User zur Verfügung stehen werden, erstellt. Die Routen werden in einer Klasse Routes erstellt und initialisiert. Die Routes werden in einem Directory gespeichert das als Key die Methode (POST, GET, PUT) und die Url annimmt. Hierauf wird im Abschnitt Routes Klasse nochmal genauer eingegangen.

Die „HandleRequest()“ Methode wird vom Routes Objekt gecalled und verarbeitet die Eingabe des Users. Hier wird herausgefunden welche Funktion des TCPServers ausgeführt werden soll. Die TCPServer Klasse dient als HUB in dem alles startet und endet. Responses an den User werden mit der „SendResponse()“ geschickt die unterschiedliche Status haben kann. Der Status wird hierbei mit globalen Variablen gesetzt (OK, Created, NotFound).

In der TCPServer Klasse gibt es zu viele Methoden, um alle einzeln durchzugehen jedoch sind viele gleich aufgebaut. Zuerst wird der User authentisiert. Hierfür wird der angegebene Token des Users mit Token in einer Datenbank verglichen und getestet, ob der User existiert. Je nach Methode oder Wunsch des Clients werden zugriffe auf die Datenbank durchgeführt. Hierzu wird als Synchronisation Locks benutzt damit keine Kollision stattfindet.

Routes Klasse:

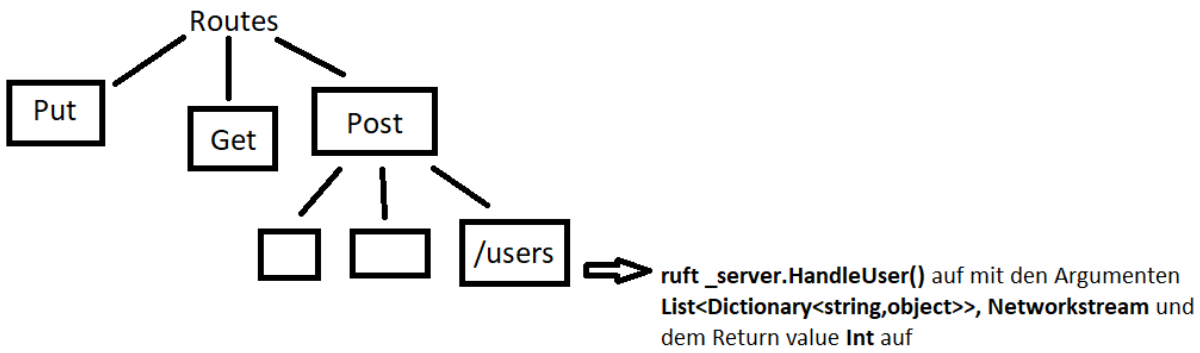
Die Routes Klasse ist im Namespace der TCPServer Klasse. Sie besteht aus Methoden die die zur Verfügung gestellten Routen initialisiert und berechnet. Die „HandleRouting()“ Methode die vom TCPServer aufgerufen wird erstellt ein Dictionary das ein Dictionary enthält, mit einem Delegate das ein Func Klasse darstellt um zurück zum TCPServer zu linken.

Hier ist es einfacher es darzustellen:

```
public void HandleRouting()
{
    routes = new Dictionary<string, Dictionary<string, Delegate>>
    {
        { _methodeGet, new Dictionary<string, Delegate>() },
        { _methodePost, new Dictionary<string, Delegate>() },
        { _methodePut, new Dictionary<string, Delegate>() }
    };

    InitRoutes(routes);
}

private void InitRoutes(Dictionary<string, Dictionary<string, Delegate>> routes)
{

    routes[_methodePost]["/users"] = new Func<List<Dictionary<string, object>>, NetworkStream, int>(_server.HandleUser);
    routes[_methodePost]["/sessions"] = new Func<List<Dictionary<string, object>>, NetworkStream, int>(_server.HandleSession);
    routes[_methodePost]["/packages"] = new Func<List<Dictionary<string, object>>, string, NetworkStream, int>(_server.HandlePackages);
    routes[_methodePost]["/transactions/packages"] = new Func<List<Dictionary<string, object>>, string, NetworkStream, int>(_server.HandleTransPackages);
    routes[_methodePost]["/battles"] = new Func<string, NetworkStream, int>(_server.HandleBattleList);
    routes[_methodePost]["/tradings"] = new Func<List<Dictionary<string, object>>, string, NetworkStream, int>(_server.HandleTradingPost);

    routes[_methodeGet]["/cards"] = new Func<string, NetworkStream, int>(_server.HandleStack);
    routes[_methodeGet]["/deck"] = new Func<string, NetworkStream, int>(_server.HandleDeck);
    routes[_methodeGet]["/deck?format=plain"] = new Func<string, NetworkStream, int>(_server.HandleDeckOther);
    routes[_methodeGet]["/usersControll"] = new Func<string, NetworkStream, int>(_server.HandleUserControllGET);
    routes[_methodeGet]["/stats"] = new Func<string, NetworkStream, int>(_server.HandleStats);
    routes[_methodeGet]["/scoreboard"] = new Func<NetworkStream, int>(_server.HandleLeaderboard);
    routes[_methodeGet]["/tradings"] = new Func<string, NetworkStream, int>(_server.HandleTrading);

    routes[_methodePut]["/deck"] = new Func<List<Dictionary<string, object>>, string, NetworkStream, int>(_server.HandleDeckInput);
    routes[_methodePut]["/usersControll"] = new Func<List<Dictionary<string, object>>, string, NetworkStream, int>(_server.HandleUserControllPUT);
}
```

Die Methode gibt hier den Key für das erste Dictionary an und die Url den Key für das Zweite Dictionary. Das zweite Dictionary enthält hier das Delegate. Dies wird benutzt, um mit Invoke eine Function aufzurufen. So werden keine If Statements oder Switch Statements benutzt.

Bei Eingabe „POST /users“:



Die Methode „HandleRequest()“ wird dann aufgerufen um die richtige Route zu bekommen.

Authent Klasse:

Die Authent Klasse wird benutzt um den User sowie Routen zu Authentifizieren. „IsAuthentic()“ wird benutzt um den Token des Users zu testen. Die anderen Authent Methoden werden benutzt

um sicher zu gehen das die ausgewählte Route auch wirklich die richtige ist und Invoken sie dann.

User Klasse:

Eine User Klasse kann Karten Klassen bekommen, indem es Packages aufmacht. Ein Package ist eine Klasse die eine Liste enthält „cardsInPack“ und eine Methode „createPackage()“. In der Liste werden 4 Karten Objekte gespeichert, die vom Admin erstellt wurden. „createPackage()“ ist eine Methode die dem Admin erlaubt Packages zu erstellen. Die User Klasse wird benutzt um die variablen die im User Objekt sind zu Printen oder Werte zu verändern. Ebenfalls werden Karten im Deck gespeichert und der User erstellt sowie eingeloggt.

Battle Klasse:

Die Battle Klasse wird benutzt, um Clients in eine Queue zu geben und diese gegeneinander kämpfen zu lassen. Ebenfalls wird es benutzt um das Leaderboard zu Printen. Wenn ein Battle zwischen 2 Clients startet, wird berechnet welchen Typ die Karten der Clients haben und je nachdem 2 Methoden aufgerufen. „CalcMonsterBattle()“ berechnet den Kampf zwischen 2 Monster und „CalcNormalBattle()“ zwischen Spells und Monstern. Der Grund hierzu ist da Monster unterschiedliche Eigenschaften haben je nachdem gegen welches Monster sie Kämpfen und es sonst zu viele Zeilen in einer Methode geworden wären. Hier wird ebenfalls auch das Unique Event durchgeführt.

In der Angabe wird nach einem Unique Feature oder einer Special Ability gefragt. Wenn eine Karte mehrfach hintereinander gewonnen hat, bekommt sie einen Strength Boost. Für 2 Runden wird sie um 20 Punkte Stärker und kann dadurch öfter gewinnen jedoch, wenn sie zu oft gewinnt, wird sie Müde und verliert ihre Stärke. Es ist quasi ein temporärer Power boost.

Database Klasse:

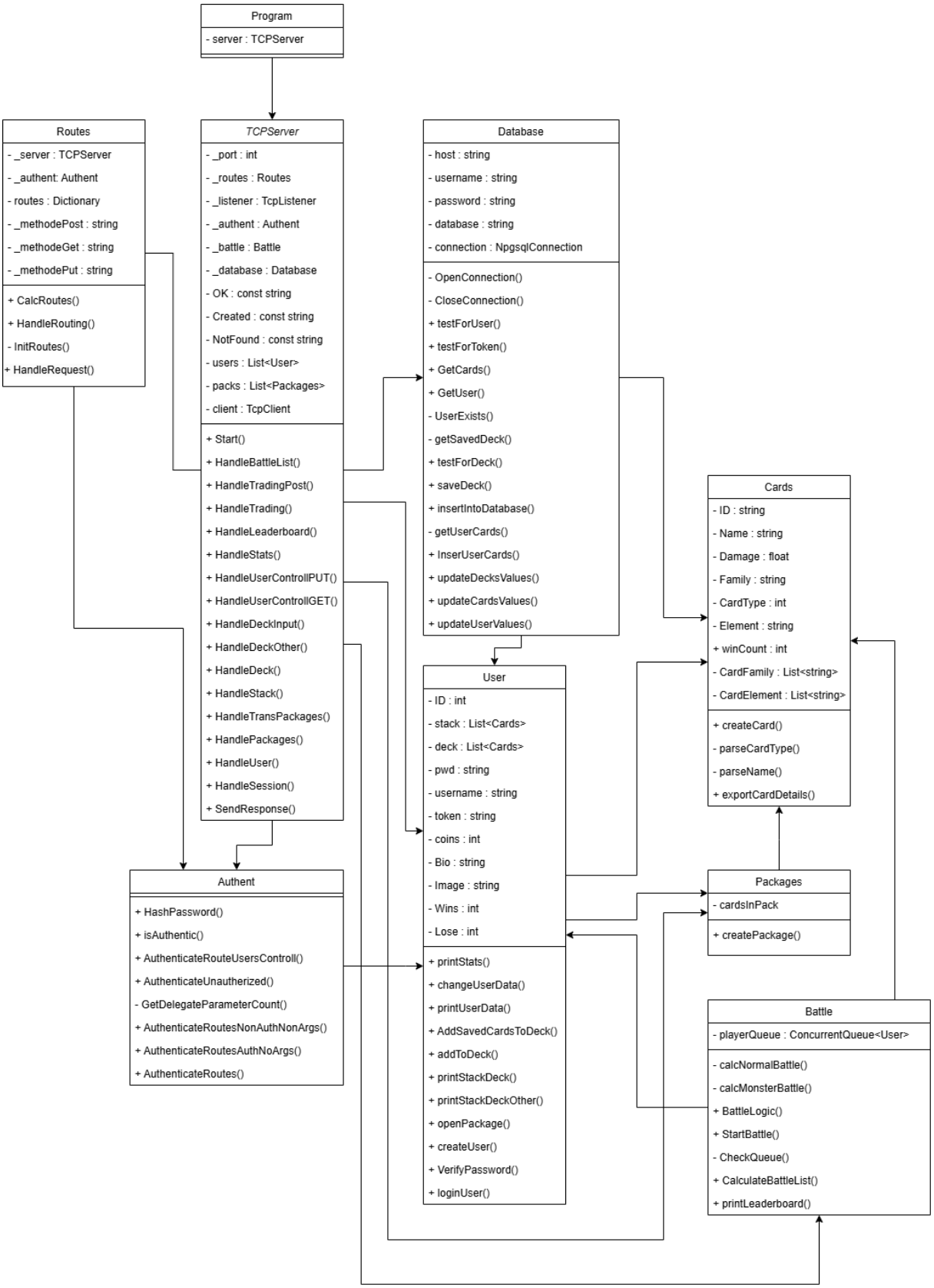
Die Database Klasse wird benutzt, um auf die Datenbank zuzugreifen. Hier wird die Datenbank mit „OpenConnection()“ geöffnet und mit „CloseConnection()“ geschlossen. Die unterschiedlichen Methoden sind wie beim TCPServer zu viel, um auf sie Speziell einzugehen. Die Methoden Selecten, Inserten oder Updaten bestimmte Values in der Datenbank.

Die Datenbank besteht aus 3 Tabellen, „users“, „deck“ und „cards“. Die „users“ Tabelle speichert alle werte der User ab. Die „cards“ Tabelle speichert alle möglichen Karten ab und wem sie gehören. Dies wird mit dem Username durchgeführt. (Sollte der User seinen Namen ändern wird es hier automatisch auch geändert). In der „deck“ Tabelle werden die Decks der User gespeichert. Hierzu wird der Username gespeichert und die ID der Karten. (Username wird wieder automatisch aktualisiert)

Klassendiagramm:

Das Klassendiagramm ist ebenfalls im GitHub Repository.

https://github.com/RxndyOG/SWEN_MTCG_StefanWerner.git


Program
- server : TCPServer
Routes
- _server : TCPServer
- _authent: Authent
- routes : Dictionary
- _methodePost : string
- _methodeGet : string
- _methodePut : string
+ CalcRoutes()
+ HandleRouting()
- InitRoutes()
+ HandleRequest()
TCPServer
- _port : int
- _routes : Routes
- _listener : TcpListener
- _authent : Authent
- _battle : Battle
- _database : Database
- OK : const string
- Created : const string
- NotFound : const string
- users : List<User>
- packs : List<Packages>
- client : TcpClient
+ Start()
+ HandleBattleList()
+ HandleTradingPost()
+ HandleTrading()
+ HandleLeaderboard()
+ HandleStats()
+ HandleUserControllPUT()
+ HandleUserControllGET()
+ HandleDeckInput()
+ HandleDeckOther()
+ HandleDeck()
+ HandleStack()
+ HandleTransPackages()
+ HandlePackages()
+ HandleUser()
+ HandleSession()
+ SendResponse()
Database
- host : string
- username : string
- password : string
- database : string
- connection : NpgsqlConnection
- OpenConnection()
- CloseConnection()
+ testForUser()
+ testForToken()
+ GetCards()
+ GetUser()
- UserExists()
- getSavedDeck()
+ testForDeck()
+ saveDeck()
+ insertIntoDatabase()
- getUserCards()
+ InserUserCards()
+ updateDecksValues()
+ updateCardsValues()
+ updateUserValues()
Cards
- ID : string
- Name : string
- Damage : float
- Family : string
- CardType : int
- Element : string
+ winCount : int
- CardFamily : List<string>
- CardElement : List<string>
+ createCard()
- parseCardType()
- parseName()
+ exportCardDetails()
User
- ID : int
- stack : List<Cards>
- deck : List<Cards>
- pwd : string
- username : string
- token : string
- coins : int
- Bio : string
- Image : string
- Wins : int
- Lose : int
+ printStats()
+ changeUserData()
+ printUserData()
+ AddSavedCardsToDeck()
+ addToDeck()
+ printStackDeck()
+ printStackDeckOther()
+ openPackage()
+ createUser()
+ VerifyPassword()
+ loginUser()
Authent
+ HashPassword()
+ isAuthentic()
+ AuthenticateRouteUsersControll()
+ AuthenticateUnautherized()
- GetDelegateParameterCount()
+ AuthenticateRoutesNonAuthNonArgs()
+ AuthenticateRoutesAuthNoArgs()
+ AuthenticateRoutes()
Packages
- cardsInPack
+ createPackage()
Battle
- playerQueue : ConcurrentQueue<User>
- calcNormalBattle()
- calcMonsterBattle()
+ BattleLogic()
+ StartBattle()
- CheckQueue()
+ CalculateBattleList()
+ printLeaderboard()

Unit Testing:

Fürs Unit Testing wurde NUnit mit Nsubstitute für mocking ausgewählt. Es wurde für die Folgenden Klassen Unit Tests geschrieben, „User“, „Cards“, „Packages“ und „Authent“. Diese Klassen wurden ausgewählt da sie die Hauptstruktur des Codes bilden. Die anderen Oberklassen wir TCPServer benutzen die Methoden dieser Klassen für ihre Methoden und haben sonst nur wenig Code, der nicht auf andere Methoden zugreift. Die Methoden der getesteten Klassen werden getestet, ob der Richtige Wert ausgegeben wird und dann nach Fehlerhandeling getestet. Es wird getestet ob wenn ein Fehler passiert, dieser auch richtig verarbeitet wird. Insgesamt gibt es 25 Unit Tests. Es können noch mehr hinzugefügt werden jedoch hatte ich keine zeit dazu.

Tracked Time:

#	Name	Startdatum	Enddatum	Aufwand (Stunden)	Beschreibung
1	GesamtProjekt	19.09.2024	06.01.2025	319	Das Gesamtprojekt wurde insgesamt fünfmal neu gestartet, jedes Mal mit einem anderen Ansatz. Zunächst wurde es funktional programmiert. Nachdem ich darauf hingewiesen wurde, dass es objektorientiert sein muss, wurde von Grund auf neu begonnen. Außerdem wurden verschiedene Ansätze ausprobiert, die entweder in das Hauptprojekt integriert oder weiter getestet wurden. Die bisher erfassten Stunden für das Gesamtprojekt beziehen sich ausschließlich auf das aktuelle Projekt und berücksichtigen die anderen Versuche nicht.
2	TCPListener	19.09.2024	01.01.2025	78	Der TCPListener musste zweimal neu gestartet werden, da ich ein HTTP-Framework verwendet hatte, ohne es zu wissen. Außerdem konnte die Arbeit nur abschnittsweise durchgeführt werden, da uns zwar am 19.09 die Aufgabe gegeben wurde, aber erst am 10.10 erklärt wurde, was ein TCPServer ist und wie man ihn aufbaut. Der Listener wurde zudem mehrfach neu begonnen oder mit neuen Teilen ergänzt. Die Einführung von Multithreading verursachte erhebliche Probleme, da nahezu alle Methoden neu geschrieben werden mussten, insbesondere diejenigen, die mit der Datenbank in Verbindung standen.
3	User Klasse	20.09.2024	28.12.2024	15	Die User-Klasse existierte von Anfang an, musste jedoch häufig verändert oder angepasst werden.
4	Cards Klasse	20.09.2024	20.11.2024	10	Die Cards-Klasse wurde ebenfalls früh erstellt und blieb bis zur Neugestaltung der Datenbank bestehen. Anfangs wurden Werte in einer Liste gespeichert. Als diese jedoch in eine Datenbank übertragen werden mussten, musste die Cards-Klasse entsprechend angepasst werden.
5	Routing Klasse	24.09.2024	20.12.2024	33	Die Routing-Klasse wurde insgesamt dreimal neu geschrieben. Zunächst war sie in der TCP-Klasse integriert, wurde jedoch später in eine externe Klasse ausgelagert. Ursprünglich wurde sie mit Switch- und If-Statements implementiert, die sich jedoch als nicht erweiterbar erwiesen. Daher musste die Klasse erneut überarbeitet werden, bis sie schließlich mit einem Dictionary funktionierte.
6	Package Klasse	27.09.2024	28.09.2024	7	Die Packages-Klasse musste nicht oft verändert werden, und wenn, dann wurden nur ein paar Code-Snippets angepasst.
7	Authent Klasse	03.10.2024	10.12.2024	24	Die Authent-Klasse war in diesem Fall ein Problem, da nicht genau klar war, was eigentlich authentifiziert werden muss. Zuerst wurde nur der User authentifiziert, jedoch wurde die Klasse mit jeder Änderung etwas größer, da die Routing-Klasse auf die Authent-Klasse zugreifen musste. Die letzte Veränderung wurde in der Routing-Klasse durchgeführt. Jetzt werden auch die unterschiedlichen Routen authentifiziert.
8	Database Klasse	24.10.2024	01.01.2025	56	Die Database-Klasse wurde häufig neu gestartet, insbesondere in Kombination mit dem TCPServer. Es half auch nicht, dass die Datenbank mit Docker eingerichtet werden musste und das dazugehörige Tutorial in Moodle Fehler aufwies. Sobald die Datenbank jedoch einmal eingerichtet war, funktionierte sie einwandfrei. Trotzdem gab es Probleme mit PostgreSQL. Die Methoden der Database-Klasse mussten oft überarbeitet werden, vor allem in Bezug auf Multithreading. Das Multithreading selbst machte es notwendig, die Art und Weise, wie auf die Datenbank zugegriffen wird, komplett neu zu überdenken.
9	Battle Klasse	07.12.2024	29.12.2024	28	Die Battle-Klasse wurde relativ spät begonnen, da in der Zeit zwischen der Database-Klasse und dem Start der Battle-Klasse viele Prüfungen anstanden. Daher wurde die Battle-Klasse etwas hastig entwickelt und erreicht nicht die Qualität der anderen Klassen. Sie hatte große Probleme mit der Warteschlange und vor allem mit dem Multithreading, was zu zwei neuen Versuchen führte. Die funktionierende Version der Klasse verwendet nun eine Concurrent Queue, um das Problem zu lösen.
10	Testing	30.12.2024	01.01.2025	18	Das Testing wurde schnell durchgeführt was dazu führte das es nicht so tiefgehend ist wie es eigentlich sein sollte.
11	Protokolle / Klassendiagramme	01.01.2025	01.01.2025	5	Zu den Protokollen gibt es nichts weiter zu sagen.
12	Error Handeling / Bug Fixing	24.09.2024	01.01.2025	45	Es mussten viele Bugs und Fehler behoben werden. Die Datenbank und der TCPServer allein haben zwei Drittel der Bugs verursacht.

Insgesamt hat das Projekt 319 Stunden gebraucht, wenn die neu angefangenen Projekte sowie Testprojekte nicht mitgezählt werden.

Lessons Learned:

Wie in Tracked Time schon geschrieben wurde das Projekt oder Teile des Projekts sehr oft neu gestartet. Was ich herausgefunden habe, ist das die Angaben in Moodle nicht viel bringen

speziell was Mocking und Unit Testing angeht. Ich wollte das Beispiel in der Unit Testing Präsenz durchgehen jedoch existiert der GitHub Link nicht mehr was dazu führte das ich, ohne es mir selbst beizubringen keine Unit Tests durchführen konnte. Deshalb sind die Unit Tests auch so wie sie sind. Es gibt zu vielen Präsenzen keine wirkliche Erklärung. Die Datenbank war genau so ein Problem da das, was im Moodle steht nicht hilfreich ist. Dies führte zu falschen Ansätzen die dazu führten das das Projekt neu gestartet werden musste.