# Netzwerk Protokoll Dokumentation

By Joe's Buddler Corp.

May 10, 2019

# Inhalt

# 1 Übersicht und Hierarchie

## 1.1 Protokolltyp

Unser Netzwerkprotokoll basiert auf dem Übertragungssteuerungsprotokoll (engl. TCP). Wir stellen für jeden Client in einem eigenen Thread eine Verbindung mittels der Java Bibliothek *java.net.Socket* her. Die übertragenen Daten sind einfach, von Menschen lesbare Strings. Jede übertragene Nachricht besteht aus einem 5-stelligen Buchstabencode, gefolgt von einem Leerzeichen, gefolgt von einem String, welcher alle Daten enthält welche, bei Bedarf, mit unserem Protokoll-Trennzeichen || verbunden resp. aufgeteilt werden können.

Die Steuerung des Netzwerkprotokolls ist auf 5 Kernklassen verteilt, welche im folgenden beschrieben werden.

## 1.2 Klassenhierarchie

### 1.2.1 Client Seite

**GameGUI** Wird aufgerufen beim Spielstart und fungiert als Interface mit dem Client. Nimmt Port und IP entgegen und erstellt damit eine Instanz der *ClientLogic*. Das GUI bietet ein Konsolen- oder ein Grafisches Interface. Befehle über das Interface werden in protokolleigenen Paketen verarbeitet und dann via die Klasse *ClientLogic* an den Server gsesendet. Das starten eines clients wird über das jar erledigt und erfolgt mit den Konsolenparamentern "client <hostadress>:<port> [<username>]". Der Username kann auch weggelassen werden. Dann erhält der Spieler entweder den im vorherigen Spiel ausgewählten Namen oder bei einem ersten Spiel den Namen "Joe Buddler".

**ClientLogic** Auf die Klasse ClientLogic wird weitgehend statisch zugegriffen. Sie wird vom *GameGUI* einmalig mit IP und Port über den Konstruktor initialisiert und verbindet so zum Server via *java.net.Socket*. Dann setzt sie ihre statischen Variablen für den In- und Output zum Server. Im Konstruktor wird ausserdem ein Thread gestartet, in welchem die Klasse den Socket von Server stetig ausliest und vor-verarbeitet.

Die ClientLogic verwaltet den In- und Output Stream zum Server und stellt Methoden zum Senden von protokolleigenen Paketen an den Server zur Verfügung.

### 1.2.2 Server Seite

**ServerGUI** Nimmt einen Port entgegen und initialisiert mit diesem Port die *ServerLogic* Klasse. Falls ein serverseitiges Konsoleninterface gebraucht wird, dann würde dies in dieser Klasse entstehen. Die Main Klasse befindet sich im *net* Packet und heisst *StartServer*. Man kann den Server aber auch über das jar starten mit den Konsolenparametern "server <port>" gestartet werden.

**ServerLogic** Auf die Klasse ServerLogic wird weitgehend statisch zugegriffen. Sie wird vom *ServerGUI* einmalig mit dem Port über den Konstruktor initialisiert und erstellt via *java.net.ServerSocket* einen Socket auf welchen die Clients verbinden können. Die ServerLogic verwaltet ausserdem zwei Listen: Eine Liste mit allen Lobbies, welche auf dem Server aktiv sind und eine Liste aller Clients mit ihrem Thread welche zum Server verbunden sind.

In einer Schleife wartet die ServerLogic auf neue Verbindungen von Clients. Verbindet sich ein neuer Client, kriegt dieser Client eine einzigartige clientId, es wird ein neuer *ClientThread* erstellt und schliesslich wird beides in der entsprechenden Liste gespeichert. Nachrichten an einen Client gehen durch die ServerLogic, welche die Nachricht an den ensprechenden ClientThread weiterleitet.

**ClientThread** Der ClientThread wird für jeden verbundenen Client initialisiert, hat eine eindeutige ID und ist für die Verbindung zu diesem Client verantwortlich. Im ClientThread sind die via *java.net.Socket* erstellten In- und Output Streams zum entsprechenden Client gespeichert. Die Klasse ClientThread befindet sich im Packet *net.playerhandling*.

Jeder ClientThread läuft in einem eigenen Thread und liest in einer Schlaufe auf dem input Socket die vom Client gesendeten Nachrichten. Die Nachrichten werden anhand des Headers an das verantwortliche protokolleigene Paket weitergeleitet. Der ClientThread enthält ebenfalls eine Methode um ein protokolleigenes Paket an den Client zu übermitteln.

# 2 Struktur der protokolleigenen Pakete

Unser Protokoll funktioniert mit vordefinierten Netzwerkpaketen. Jedes Paket übernimmt genau eine Funktion. Die Pakete sind selber verantwortlich für die Validierung, Verarbeitung und Fehlerbehandlung der eigenen Daten.

## 2.1 ENUM für PaketTypen

Das ENUM für die PaketTypen befindet sich in der Abstrakten Klasse für alle Pakete und definiert alle Paket-Typen welche das Protokoll unterstützt. Ein PaketTyp ist definiert durch den 5-stelligen Buchstabencode, welcher auch als Header für die Nachrichten dient. Das ENUM weist ausserdem jedem PaketCode eine beschreibende Variable zu, damit der Code deutlich an Leserlichkeit gewinnt. Das Paket mit dem Code "STNMS" zum Beispiel wird so im Code mit PaketTypes.SET_NAME_STATUS referenziert. Ein ENUM bietet uns ausserdem viel Flexibilität und ermöglicht eine einfache Erweiterung um zusätzliche Pakete.

## 2.2 Abstrake Klasse für alle Pakete

Protokolleigene Pakete haben einen Typ und enthalten einen String mit allen Daten. Falls das Paket von einem Client versendet wurde, wird dies in clientId gespeichert. Ausserdem hat jedes Paket eine Liste mit Fehlern im String (klartext) Format. Ist diese Liste nicht leer, dann können die Fehler mit den entsprechenden Methoden ausgelesen und behandelt werden.

```
private PacketTypes packetType;
private String data;
private int clientId;
private List<String> errors = new ArrayList<>();
```

### 2.2.1 Wichtige Methoden der abstrakten Klasse

Die abstrakte Klasse enthält Methoden welche für alle Pakete identisch sind, wie:

- Getter und Setter für data und clientId
- Bool'sche Methode zum überprüfen ob ein Paket Fehler enthält
- Getter und Adder für Fehler sowie eine Methode welche alle Fehler zu einem String verbindet und zurück gibt
- toString: Transformiert Daten zu einem lesbaren String welcher als Nachricht über das Protokoll geschickt wird
- Funktionen zum Senden des Pakets an Client, Lobby oder Server
- Allgemeine bool'sche Validierungsfunktionen, welche von mehreren Paketen genutzt werden

### 2.2.2 Abstrakte Methoden

Ausserdem schreibt die abstrakte Klasse vor, dass jedes Paket die folgenden zwei Methoden implementieren muss:

- validate(): Validiert alle Klassenvariablen und erstellt gegebenenfalls Fehler mit aussagekräftigen Fehlermeldungen. Fehler werden im Paket gespeichert wie oben beschrieben. Diese Funktion wird am Ende des Konstruktors aufgerufen und hat **keinen Zugriff** auf Informationen ausserhalb des Pakets! Sprich, die Funktion muss auf Server- und Clientseite gleichermassen funktionieren.

- processData(): Wird nach dem empfangen eines Pakets ausgeführt und enthält einen Grossteil der Logik des Pakets. Diese Funktion kann ebenfalls Fehler zum Paket hinzufügen und hat Zugriff auf Klassen und Informationen auf der Server- und/oder Client Seite. Hier werden oft Antwortpakete erzeugt und versendet beziehungsweise spezifische Befehle auf Client- beziehungsweise Serverseite ausgeführt.

## 2.3 Implementierung der Pakete

Pakete werden von der abstrakten Klasse abgeleitet und definieren individuell weitere Klassenvariablen zum speichern und validieren der Daten für welche sie zuständig sind. Jeder PaketTyp ist genau für eine Funktion zuständig wie etwa *Login*, *Login Status* oder *Disconnect*.

Pakete müssen wenn möglich clientId und data setzten, sowie validate() und processData() implementieren, wie in 2.2 beschrieben.

Pakete sind in Gruppen geordnet, welche jeweils eine Funktionskategorie umfassen. Im nächsten Kapitel wird jede dieser Funktionskategorien mit ihren Paketen erläutert.

## 2.4 Kontrollfluss der Pakete

Ein Paket geht typischerweise durch die folgenden Schritte:

1. Der Konstruktor wird aufgerufen mit Spieldaten oder Userinput
    - Die übergebenen Variablen werden den entsprechenden Klassenvariablen zugeteilt
    - Die *data* Variable wird generiert: Eine Aneinanderreihung der Klassenvariablen, getrennt durch das Protokoll-Trennzeichen
    - Die Klassenvariablen werden mittels der validate() methode validiert. Allfällige Fehler werden dem Paket angehängt
2. Falls das Paket Fehler enthält, können diese vor dem verschicken behandelt werden, indem zum Beispiel die entsprechende Send-Methode überschrieben wird
3. Das Paket wird versendet mittels einer der Send-Methoden welche den Output der toString() Methode verschicken
4. Der String wird auf der anderen Seite empfangen und anhand des Paket-Typ-Codes einem Paket zugewiesen
5. Der Konstruktor dieses Pakets wird aufgerufen mit dem *data-String* und falls vorhanden, der clientId
    - Der *data-String* wird in der *data* Variable gespeichert und dann Mittels dem Protokoll-Trenzeichen in die Klassenvariablen aufgesplittet
    - Die Klassenvariablen werden mittels der validate() methode validiert. Allfällige Fehler werden dem Paket angehängt
6. Die Methode processData() kann aufgerufen werden. Hier Findet die Fehlerbehandlung und die ganze Logik des Pakets Platz. Falls notwendig werden hier Antwortpakete erstellt

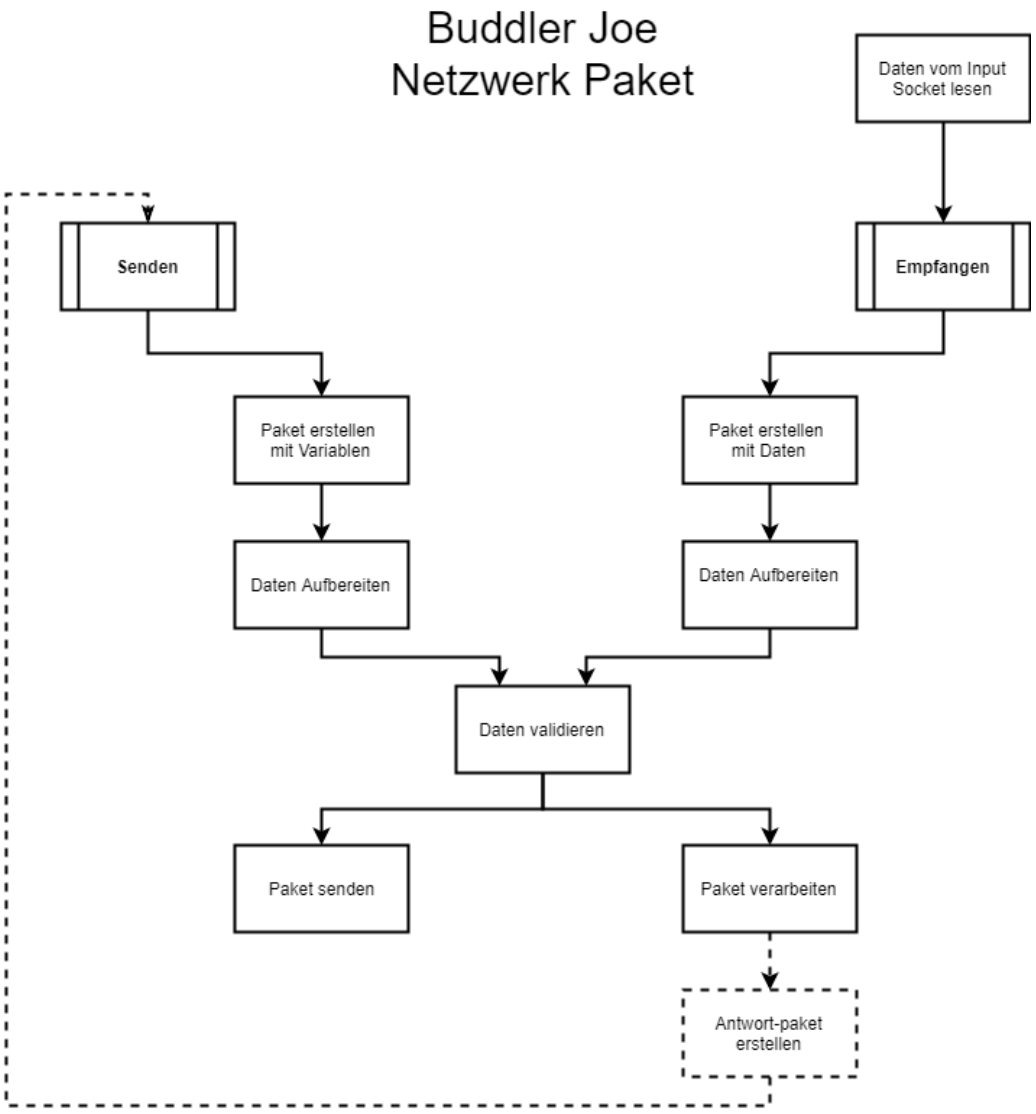


Figure 1: Grafische Darstellung des Paket-Kontrollfluss

## 2.5 Nachrichten Formatierung

Unser Netzwerkprotokoll basiert auf dem Versenden von Strings über die I/O-Reader des Servers und der Clients. Die Formatierung dieser Strings ist spezifisch und wird im Allgemeinen von allen Packeten gleich eingehalten. Nachfolgend erklären wir, wie diese Strings formatiert und aufgebaut werden. Später im Dokument wird dann die spezifische Verarbeiteung der Nachrichten behandelt.

Ein String, welcher entweder vom Server an den Client, oder umgekehrt, gesendet wird hat immer den folgenden Aufbau. Zwischen jedem aufgezählten Punkt wird unser einheitliches Trennzeichen (||) automatisch von den Packeten eingefügt:

- Der Code des Packetes, welcher in der ENUM der abstrakten Packet Klasse definiert ist. Im weiteren Verlauf des Dokumentes kann man diesen Code von jedem Packet im Teil Allgemein unter Code finden
- Die zu versendenden Daten. Diese werden entweder vom Client oder vom Server aneinander gehängt und dann mit dem Trennzeichen || wird unterschieden, welche Daten an welcher Stelle sind. Im weiteren Verlauf des Dokumentes kann man die unterschiedlichen Informationen, welche in einem jeden String enthalten sind, unter der Rubrik Klassenvariablen finden. Diese werden in der aufgeführten Reihenfolge an den String angehängt. Falls gewisse Klassenvariablen nicht in jedem Fall versendet werden, so wird dies erwähnt.

Bei der Verarbeitung werden die Strings dann an den Trennzeichen (||) aufgesplittet und weiter verarbeitet. Die Verarbeitung kann im weiteren Verlauf des Dokumentes unter der Rubrik Verarbeitung eingesehen werden.

Die Beispiele dienen nur der Illustration eines Kommunikation-Ablaufes zwischen dem Client und dem Server oder umgekehrt.

# 3 Funktionskategorien der Pakete

## 3.1 Login und Logout

**Java-Package:** net.packets.login_logout

Der Server empfängt den Login eines Clients mit Username. Der Username wird validiert und der Server prüft, ob der Spieler ohne Konflikte erstellt werden kann. Ist dies möglich wird der Spieler erstellt in die Liste der eingeloggten Spieler eingetragen. In jedem Fall wird der Server eine Antwort an den Client senden mit dem Resultat des Logins.

Das Disconnect Packet kann entweder vom User gesendet werden bei einem ordnungsgemässen Logout, oder es wird vom Server generiert, falls die Verbindung zum Client nicht ordnungsgemäss abbricht. Das Disconnect Packet löscht den Spieler aus allen relvanten Listen und sendet die notwendigen Pakete an andere Clients um diese vom Logout zu informieren.

### 3.1.1 Packet: LOGIN

**Allgemein**

| | |
|---|---|
| Code | PLOGI |
| Klasse | PacketLogin.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**username**

| | |
|---|---|
| Beschreibung | Gewünschter Benutzername im Spiel |
| Validierung | Ist vorhanden. Mit Hilfe der checkUsername Methode der Abstrakten Packet Klasse wird überprüft, ob der username nicht zu kurz/zu lang ist, ob er im extended Ascii ist und ob er überhaupt vorhanden ist. Siehe 2.2 für mehr Informationen. |

**Verarbeitung**. Zuerst wird überprüft, ob das Packet Fehler enthielt. Wenn ja, wird dem Client ein String mit "ERRORS: [Fehler im Packet]" zurückgeschickt. Falls nicht, werden die Daten weiter verarbeitet. Nun wird der neue Client mit seinem username in die Spielerliste auf der Serverlogic eingefügt. Falls der Benutzername bereits existiert auf dem Server, wird eine aufsteigende Zahl an den Namen angehängt, bis der Name einzigartig ist. Danach wird versucht den Spieler zu erstellen sofern dieser noch nicht eingeloggt ist und den Spieler zur Liste der eingeloggten Spieler hinzuzufügen.

In jedem Fall wird ein Login-Status-Packet erstellt und dem Client zurück gesendet.

**Beispiel**. "PLOGI Joe_Buddler" von Client zu Server. Dieser fügt den Spieler "Joe_Buddler" der Spielerliste auf dem Server hinzu und sendet ein Login Status Packet (*net.packets.loginlogout.PacketLoginStatus*) mit dem Inhalt "OK||Peter Gryffin" zurück.

### 3.1.2 Packet: LOGIN_STATUS

**Allgemein**

| | |
|---|---|
| Code | PLOGS |
| Klasse | PacketLoginStatus.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**status**

| | |
|---|---|
| Beschreibung | Resultat des Login-Versuchs |
| Validierung | Nur extended ASCII und nicht leer |

**username**

| | |
|---|---|
| Beschreibung | Username, welcher der Spieler erhalten hat |
| Validierung | Extended ASCII und nicht leer. |

**Verarbeitung**. Falls der Login erfolgreich war, wird status = "OK|||[username]" an den Client gesendet. Ansonsten wird "ERRORS: [Liste der Fehler]", welche der Client individuell anzeigt oder verarbeitet. Falls der Server den Username geändert hat, weil es den username schon auf dem Server gab, erhält der Spieler einen String "CHANGE|||[neuer username]". Der username wird dann beim Spieler der username im Game gesetzt.

**Beispiel**. "PLOGS OK||Peter Gryffin" von Server zu Client signalisiert einen erfolgreichen Login und setzt beim Client den Namen auf "Peter Gryffin" und zeigt den aktuellen Namen im GUI an.

### 3.1.3 Packet: DISCONNECT

**Allgemein**

| | |
|---|---|
| Code | DISCP |
| Klasse | PacketDisconnect.java |
| Absender | Client oder Server |
| Empfänger | Server |

**Klassenvariablen**

keine

**Verarbeitung**. Der Server ruft die Methode removePlayer in der Classe ServerLogic auf mit der SpielerID als Parameter. Diese Methode prüft ob der dazugehörender Spieler in der Spielerliste vorhanden ist. Anschliessend wird überprüft, ob sich der Spieler in einer Lobby befindet. Wenn ja, wird der Spieler aus der Lobby geworfen. Der Spieler Thread wird geschlossen und der Spieler wird von der Spielerliste genommen. Dann wird eine Nachricht an die Spieler in der Lobby geschickt, dass der entsprechende Spieler das Spiel verlassen hat. Zuletzt wird der Lobbystatus an alle in der Lobby gesendet, damit sie den Überblick über die Lobby haben.

**Beispiel**. Ein Spieler, welcher in einer Lobby ist, schliesst das Spiel und sendet damit automatisch ein DISC Packet an den Server. Der Spieler wird dann aus der Lobby und von der Spielerliste genommen. Die restlichen Spieler in der Lobby bekommen dann eine Nachricht vom Server das der Spieler die Lobby verlassen hat.

### 3.1.4 Packet: UPDATE_CLIENT_ID

**Allgemein**

| | |
|---|---|
| Code | UPCID |
| Klasse | PacketUpdateClientId.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**clientIdString**

| | |
|---|---|
| Beschreibung | Aktuelle clientId des Client als String |
| Validierung | Ist vorhanden. Es wird überprüft ob eine clientId vorhanden ist oder nicht. |

**clientId**

| | |
|---|---|
| Beschreibung | Aktuelle clientId des Client als Integer |
| Validierung | Ist vorhanden. Es wird überprüft ob eine clientId vorhanden ist und ob die clientIdasString wirklich ein Integer und damit eine clientId ist. |

**Verarbeitung**. Falls es keine Fehler bei der Validierung gab wird das Packet wiefolgt verarbeitet. Nachdem in der Validierung der String in einen Integer umgewandelt wurde, wird diese client Id nun dem Spieler zugewiesen. Dadurch kennt der Spieler seine eigene clientId und kann bei anderen Operationen (wie zum Beispiel bei den Items) einen Clientvvergleich von sich selber und anderen clientId's machen.

Das Packet wird vom Server aus automatisch an den Client geschickt und der Client antwortet nicht auf das Packet.

**Beispiel**. Der Server erstellt ein PacketUpdateCliendId, sendet den String "UPCID 1" an den Client mit der ClientId 1, dieser vermerkt sich die ClientId 1 in seiner eigenen Player Klasse, welche im Game erstellt wurde.

## 3.2 Name

**Java-Package:** net.packets.name

Dieses Package enthält alle Paket-Klassen die mit dem Name setting zu tun haben. Dazu gehört zum Beispiel eine setName Klasse, welche dem Spieler ermöglicht, den Namen zu ändern.

### 3.2.1 Packet: SET_NAME

**Allgemein**

| | |
|---|---|
| Code | SETNM |
| Klasse | PacketSetName.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**username**

| | |
|---|---|
| Beschreibung | Ein String, der den username enthält, welcher der player gerne setzen möchte. |
| Validierung | Ist vorhanden. Mit Hilfe der checkUsername Methode wird überprüft, ob der username nicht zu kurz/zu lang ist, ob er im extended Ascii ist und ob er überhaupt vorhanden ist. |

**Verarbeitung**. Falls in der validierungsmethode ein Fehler gefunden wird, wird eine Fehler Nachricht erstellt und als status an ein PacketSetNameStatus mit dem String "ERROR|||[Errors im Packet]" weitergegeben und an den client zurückgeschickt. Falls der Name schon in der serverPlayerList vorhanden ist, wird mit Hilfe eines Zählers so lange der username variiert, bis der Name einmalig ist und dann geändert. Sobald der Name einmalig ist, wird eine Statusmeldung erstellt und auch einer PacketSetNameStatus Klasse übergeben und dem Client geschickt. Dieser String hat dann die Formatierung "CHANGED|||[neuer username]". Falls der Name schon von Anfang an einmalig ist, wird einfach dieser neue Name gesetzt und ein Status mit einer PacketSetNameStatus Instanz mit dem String "OK|||[username]" an den Client zurückgeschickt.

**Beispiel**. "SETNM Peter Gryffin" von Client zu Server. Angenommen der username Peter Gryffin ist schon vorhanden, so wird so lange mit dem Counter hochgezählt, bis Peter Gryffin_ + Counter einmalig ist. Nehmen wir an, dies sei bei Peter Gryffin_1 der Fall, so würde der username vom Client auf Peter Gryffin_1 geändert und ein Status "CHANGED||Peter Gryffin_1" mit einem PacketSetNameStatus an den Client zurückgeschickt.

### 3.2.2 Packet: SET_NAME_STATUS

**Allgemein**

| | |
|---|---|
| Code | STNMS |
| Klasse | PacketSetNameStatus.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**status**

| | |
|---|---|
| Beschreibung | Ein String Array, der den Status und den aktuellen username des Name settings enthält |
| Validierung | Ist vorhanden. Wird geprüft, ob der Status vorhanden ist und ob der Status extended Ascii ist. |

**Verarbeitung**. Falls in der Validierungsmethode ein Fehler gefunden wird, wird eine Fehler Nachricht erstellt und dem Client ausgegeben. Falls der Status Array an Position 0 mit "OK" beginnt, war das name setting erfolgreich, falls der Status Array an Position 0 mit "CHANGE" beginnt, wurde der Name noch abgeändert und es wird der neue Username angezeigt.

**Beispiel**. "CHANGED||Peter Gryffin_1" vom Server zum Client. In diesem Fall war das Name setting erfolgreich aber mit einer abgeänderten Version des Namens und der neue username des Clients ist Peter Gryffin_1. Dieser Name wird dem Spieler dann im GUI angezeigt.

## 3.3 Lobby

**Java-Package:** net.packets.lobby

Dieses Packet enthält alle Packet-Klassen die mit dem Lobbysystem etwas zu tun haben. Dazu gehören Beispielsweise Pakete, die dazu verwendet werden um dem Benutzer eine Übersicht über verfügbare Lobbys zu geben, Pakete für das Erstellen neuer Lobbys oder auch Pakete, die es dem Benutzer erlauben einer Lobby beizutreten, beziehungsweise sie wieder zu verlassen.

### 3.3.1 Packet: GET_LOBBIES

**Allgemein**

| | |
|---|---|
| Code | LOBGE |
| Klasse | PacketGetLobbies.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

keine

**Verarbeitung**. Falls der Benutzer, der das Packet an den Server gesendet hat, noch nicht angemeldet ist wird eine Fehlermeldung zurück gegeben. Ansonsten wird eine Auflistung von maximal zehn Lobbys erstellt, welche nicht voll sind (Maximale Anzahl an Lobbymitgliedern ist noch nicht definitiv festgelegt). Die Auflistung wird in einem String gespeichert, mit welchem dann ein LOBBY_OVERVIEW Paket erstellt wird. Falls Fehler auftraten enthält das neu erstellte Packet die Fehlermeldung. Ansonsten beginnt der String mit "OK".

**Beispiel**. "LOBGE" von Client zu Server. Angenommen es gäbe eine Lobby namens myLobby, mit der lobbyId 1 und darin befänden sich 3 Spieler. Es wird eine Auflistung der verfügbaren Lobbys erstellt und ein LOBBY_OVERVIEW Paket an den Benutzer geschickt, der "LOBGE" gesendet hat.

### 3.3.2 Packet: LOBBY_OVERVIEW

**Allgemein**

| | |
|---|---|
| Code | LOBOV |
| Klasse | PacketLobbyOverview.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**in[]**

| | |
|---|---|
| Beschreibung | Das Array enthält an jedem Index einen String. Falls es zu keinen Fehlern kam, ist das Array dann wie folgt besetzt: ("OK", "numberOfLobbies", "lobbyname", "numberOfPlayers", "mapSize", "lobbyname", "numberOfPlayers", "mapSize"...). Wobei numberOfLobbies eine Zahl ist, die angieb wie viele Lobbys enthalten sind.NumberOfPlayers ist eine Zahl, die angibt wieviele Spieler in einer Lobby sind. MapSize entspricht dem Buchstaben "s" ,"m" oder "l" Falls Fehler auftraten ist der erste Eintrag nicht "OK" sondern eine Fehlermeldung. |
| Validierung | Ist vorhanden. Nur extended ASCII wird akzeptiert. Es wird geprüft ob das Array "in" an denn benötigten stellen ein Integer enthält und ob zu jeder Lobby eine SpielerZahl und eine legale Kartengrösse mitgegeben wurde. |

**Verarbeitung**. Es wird überprüft ob die empfangenen Daten mit "OK" beginnen. Ist dies der Fall und es ist auch zu keinem Validierungsfehler gekommen, wird aus denn Strings ein neuer "Lobbykatalog" erstellt welcher im Menu ChooseLobby angezeigt werden kann.

**Beispiel**. Der Server erhielt ein "LOBGE" von einem Benutzer. Es gibt eine Lobby auf dem Server welche 5 Spieler enthält. Es wird also ein LOBBY_OVERVIEW-Paket, als String "LOBOV OK||1||lobbyname||5||m", an den Client geschickt von dem "LOBGE" kam. Im GUI des clients wird darauf hin die Tabelle des ChooseLobby-Menus aktualisiert.

### 3.3.3 Packet: CREATE_LOBBY

**Allgemein**

| | |
|---|---|
| Code | LOBCR |
| Klasse | PacketCreateLobby.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**info[]**

| | |
|---|---|
| Beschreibung | Das Array enthält an jedem Index einen String. Falls es zu keinen Fehlern kam, ist das Array dann wie folgt besetzt: ("lobbyName", "mapSize"). |
| Validierung | Hat zwei Einträge. Erster Eintarag mindestens 4 und höchstens 16 Zeichen lang sein. Zweiter Eintrag ist gleich "s" , "m" oder "l". Nur extended ASCII wird akzeptiert. |

**Verarbeitung**. Es wird geprüft, ob der Benutzer, der das Packet gesendet hat, eingeloggt ist und ob er sich bereits in einer Lobby befindet. (Nicht eingeloggt oder bereits in einer Lobby wären hier ein Fehler). Wenn keine Fehler existieren, wird eine neue Lobby mit dem gewünschten Lobbynamen und eingestellter Kartengrösse erstellt und der Liste, die der Server über die Lobbys führt, hinzugefügt. Falls das erstellen erfolgreich war, sendet der Server nun an alle Benutzer die noch nicht in einer Lobby sind ein LOBBY_OVERVIEW-Paket um sie über die neu Lobby zu informieren. Auf jeden Fall wird aber ein CREATE_LOBBY_STATUS-Packet an den Benutzer geschickt der den Befehl zum neu erstellen einer Lobby gesendet hat. Dieses enthält beim Erstellen einen String, welcher allfällige Fehlermeldungen oder die Information über denn Erfolg des Lobbyerstellen enthält.

**Beispiel**. Der Client sendet ein CREATE_LOBBY-Paket, als String "LOBCR myLobby||s", an den Server. Dieser erstellt eine Lobby mit dem Namen myLobby und Kartengrösse s und schickt uns ein CREATE_LOBBY_STATUS-Packet mit dem Inhalt "OK" zurück. An alle Benutzer die nicht in einer Lobby sind (auch wir selber) wird ausserdem ein LOBBY_OVERVIEW-Paket gesendet.

### 3.3.4 Packet: CREATE_LOBBY_STATUS

**Allgemein**

| | |
|---|---|
| Code | LOBCS |
| Klasse | PacketCreateLobbyStatus.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**status**

| | |
|---|---|
| Beschreibung | Ein String der Informationen darüber enthält ob das Erstellen der Lobby funktioniert hat. Wenn nicht Ist eine Fehlerbeschreibung enthalten, sonst "OK". |
| Validierung | Ist vorhanden. Nur extended ASCII wird akzeptiert. |

**Verarbeitung**. Es wird überprüft ob die empfangenen Daten mit "OK" beginnen. Ist dies der Fall wird "Lobby-Creation Successful" ausgegeben, andernfalls die Fehlermeldung die in status enthalten ist oder eine Auflistung der Fehler, die bei der Validierung von status auftraten.

**Beispiel**. Der Server erhielt ein CREATE_LOBBY-Paket, als String "LOBCR myLobby||s" und hat erfolgreich eine neue Lobby namens myLobby und Kartengrösse s erstellt. Nun erstellt er ein CREATE_-LOBBY_STATUS-Packet, als String "LOBCS OK" ("OK" wurde von ServerLogic.getLobbyList().addLobby(lobby) zurückgegeben als die neue Lobby der Lobbyliste des Servers hinzugefügt wurde). Dieses Packet wird nun an den Client gesendet, welcher daraufhin das Menu zur Lobbyerstellung schliesst.

### 3.3.5 Packet: JOIN_LOBBY

**Allgemein**

| | |
|---|---|
| Code | LOBJO |
| Klasse | PacketJoinLobby.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**lobbyname**

| | |
|---|---|
| Beschreibung | Name der Lobby der beigetreten werden soll. |
| Validierung | Nur extended ASCII. |

**Verarbeitung**.Es wird überprüft ob eine Lobby existiert, die nach dem angegebenen Lobbynamen benannt ist, ob der Benutzer, der einer Lobby beitreten will auf dem Server angemeldet ist und ob der Benutzer bereits in einer Lobby ist. (Fehler wären: Es existiert keine entsprechende Lobby, der Benutzer ist noch nicht auf dem Server angemeldet oder er ist bereits in einer Lobby.). Sind keine Fehler festgestellt worden, wird der Benutzer der Lobby hinzugefügt. Danach wird an alle Benutzer die sich in der Lobby befinden der beigetreten wurde ein CUR_-LOBBY_INFO-Paket geschickt. Damit werden sie über denn neuen Nutzer informiert. Zudem wird an alle Benutzer die sich aktuell nicht in einer Lobby befinden ein LOBBY_OVERVIEW-Paket geschickt. Dadurch werden sie darüber informiert dass sich die Spielerzahl in der behandelten Lobby geändert hat. In jedem Fall wird aber ein JOIN_-LOBBY_STATUS-Paket an den Benutzer zurück geschickt der einer Lobby beitreten wollte. Dieses enthält dann entweder "OK", im Falle eines Erfolges und andernfalls eine entsprechende Fehlermeldung als String.

**Beispiel**. Wir haben uns auf dem Server eingeloggt und wollen nun der Lobby myLobby beitreten, die bereits existiert. Es wird also ein JOIN_-LOBBY-Paket, als String "LOBJO myLobby", an den Server gesendet. Da wir noch nicht in einer Lobby sind, fügt der Server uns der gewählten Lobby hinzu und schickt ein JOIN_LOBBY_STATUS-Paket an uns zurück. Alle anderen Benutzer werden zusätzlich auch über die Änderung informiert.

### 3.3.6 Packet: JOIN_LOBBY_STATUS

**Allgemein**

| | |
|---|---|
| Code | LOBJS |
| Klasse | PacketJoinLobbyStatus.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**status**

| | |
|---|---|
| Beschreibung | Ein String der Informationen darüber enthält, ob das Beitreten in die gewünschte Lobby funktioniert hat. Wenn nicht, Ist eine Fehlerbeschreibung enthalten. |
| Validierung | Ist vorhanden. Nur extended ASCII. |

**Verarbeitung**. Es wird überprüft, ob die empfangenen Daten mit "OK" beginnen und es zu keinen Fehlern kam. Ist beides gegeben, wechselt das GUI in das InLobby-Menu.

**Beispiel**. Wir schicken ein JOIN_LOBBY-Paket, als String "LOBJO myLobby", an den Server, dieser fügt uns erfolgreich der Lobby myLobby hinzugefügt und schickt uns ein JOIN_LOBBY_STATUS-Paket, als String "LOBJS OK", zurück.

### 3.3.7 Packet: CUR_LOBBY_INFO

**Allgemein**

| | |
|---|---|
| Code | LOBCI |
| Klasse | PacketCurrLobbyInfo.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**info**

| | |
|---|---|
| Beschreibung | Ein String der Informationen über die Lobby enthält in der sich der Empfänger gerade befindet. Dies sind der Namen der Lobby, die Namen und Ids aller Benutzer sowie deren "ready-Status". Er hätte beispielsweise die Struktur „OK\|\|lobbyName\|\|PlayerId1\|\|playerName1\|\|readyStatus1. Falls es zuvor zu Fehlern kam, beispielsweise bei einem „LOBGI“ enthält der String entsprechende Fehlermeldungen. |
| Validierung | Ist Extendet ASCII. Ist vorhanden ist. |

**infoArray**

| | |
|---|---|
| Beschreibung | Ein String-Array, welches im Konstruktor des Paketes mit den Substrings, die entstehen wenn info an den Stellen „\|\|“ gespalten wird, gefüllt wird. Falls info eine Fehlermeldung enthält, wird diese in infoArray[0] gespeichert. |
| Validierung | Es wird überprüft ob infoArray zu jeder Lobby drei Einträge hat, also für jede Lobby genügend Informationen vorhanden sind. |

**Verarbeitung**. Falls es in der Validierung zu Fehlern kam, wird eine entsprechende Fehlermeldung ausgegeben. Sonst wird geprüft ob infoArray[0] „OK“ entspricht, ist dies der Fall werden die restlichen Elemente von infoArray ausgegeben,bzw ein neuer "InLobbyKatalog" erstellt, dessen Elemente in der Tabelle des InLobby-Menus dargestellt werden können. Falls infoArray [0] ungleich „OK“ ist, wird nur infoArray [0], welches eine Fehlermeldung enthält, ausgegeben.

**Beispiel**. Wir haben uns als Benutzer auf dem Server eingeloggt und sind bereits einer Lobby beigetreten. Nun schicken wir ein GET_LOBBY_INFO-Paket, als String „LOBGI“ an den Server. Dieser antwortet uns darauf mit einem CUR_LOBBY_INFO-Paket, als String "OK||myLobby||42||Joe||false". Das GUI wird aktualisiert.

### 3.3.8 Packet: LEAVE_LOBBY

**Allgemein**

| | |
|---|---|
| Code | LOBLE |
| Klasse | PacketLeaveLobby.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

keine

**Verarbeitung**.Es wird überprüft, ob der Benutzer der das Packet geschickt hat, eingeloggt ist und sich in einer Lobby befindet. Falls eines der beiden nicht gegeben ist, wird ein Fehler vermerkt. Sind keine Fehler vorhanden, entfernt der Server den Benutzer, der das Packet geschickt hat, aus seiner aktuellen Lobby. Danach wird an alle Benutzer, die sich in der Lobby befinden, aus der der Benutzer entfernt wurde, ein CUR_LOBBY_INFO-Paket geschickt. Damit werden sie über das Verlassen des Nutzers informiert. Zudem wird an alle Benutzer, die sich aktuell nicht in einer Lobby befinden (dazu zählt auch der Benutzer der jetzt gerade eine Lobby verlassen hat), ein LOBBY_OVERVIEW-Paket geschickt. Dadurch werden sie darüber informiert, dass sich die Spielerzahl in der behandelten Lobby geändert hat. In jedem Fall wird ein LEAVE_LOBBY_STATUS-Paket an den Benutzer zurück geschickt der die Lobby verlassen wollte. Dieses enthält dann entweder "OK", im Falle eines Erfolges und andernfalls eine entsprechende Fehlermeldung als String.

**Beispiel**. Wir sind als Benutzer in einer Lobby und wollen sie verlassen. Dazu senden wir ein LEAVE_LOBBY-Paket, als String „LOBLE“ an den Server. Da dieser keine Fehler feststellt, entfernt er uns aus der Lobby und informiert uns mit einem LEAVE_LOBBY_STATUS-Paket darüber, dass wir die Lobby verlassen konnten. Zudem erhalten wir auch gleich ein LOBBY_OVERVIEW-Paket, um wieder eine Übersicht über die verfügbaren Lobbys zu haben. Alle anderen Benutzer werden zusätzlich auch über die Änderung in der Lobby informiert.

### 3.3.9 Packet: LEAVE_LOBBY_STATUS

**Allgemein**

| | |
|---|---|
| Code | LOBLS |
| Klasse | PacketLeaveLobbyStatus.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**status**

| | |
|---|---|
| Beschreibung | Ein String der Informationen darüber enthält ob das Verlassen der Lobby funktioniert hat. Ist dies der Fall ist der String „OK“. Sonst enthält er eine entsprechende Fehlerbeschreibung. |
| Validierung | Ist vorhanden. Nur extended ASCII. |

**Verarbeitung**.Es wird überprüft ob die empfangenen Daten mit "OK" beginnen. Ist dies der Fall, wird das GUI aktualisiert, dass heisst wir sind nun im Hauptmenu oder im ChooseLobby-Menu.

**Beispiel**. Wir befinden uns in einer Lobby im InLobby-Menu und schicken ein „LEAVE_LOBBY-Paket an den Server, dieser entfernt uns erfolgreich aus der Lobby und schickt uns ein LEAVE_LOBBY_STATUS-Paket, als String "LOBLS OK", zurück. Da dieses den Status „OK“ enthält, wird uns nun wieder das ChooseLobby-Menu angezeigt.

## 3.4 Ping und Pong

**Java-Package:** net.packets.pingpong

Dieses Package enthält die Ping und die Pong Klasse, welche eine wichtige Rolle für den Erreichbarkeitstest von Server und Clients haben. Ein Ping Paket wird mit seiner Erstellungszeit von der Server wie auch von der Client Seite verschickt. Als Antwort darauf sollte ein Pong Paket von der Gegenseite erstellt und mit dem selben Inhalt zurückgeschickt werden, worauf der Absender das Pong Paket erhält und sich die Zeit merkt. Nun wird die Zeitdifferenz berechnet. An der Zeitdifferenz kann abgelesen werden, wie lange der Transport von Packeten hin zum Adressaten und zurück zum Absender braucht. Die Pingpakete werden im Hintergrund automatisiert verschickt und können vom Client selbst ausgelöst werden.

### 3.4.1 Packet: PING

**Allgemein**

| | |
|---|---|
| Code | UPING |
| Klasse | PacketPing.java |
| Absender | Client und Server |
| Empfänger | Server und Client |

**Klassenvariablen**

**keine**

**Verarbeitung**. Falls sich keine Fehler in der im Pingpaket übergebenen Uhrzeit verstecken, wird ein Antwort- bzw. ein Pongpaket erstellt, welchem abhängig von der Art des eingetroffenen Pingpakets zum Server oder zum Client verschickt wird. Enthielt das Pingpaket eine Nummer, welche den Client identifiziert, so wird das Pongpaket zum Client verschickt, anderenfalls zum Server.

**Beispiel**. Möchte ein Client die Erreichbarkeit testen, kann er ein Pingpaket mit dem Befehl "ping" erstellen und dies dem Server schicken. Der Server verarbeitet anschliessend das erhaltene Paket.

### 3.4.2 Packet: PONG

**Allgemein**

| | |
|---|---|
| Code | PONGU |
| Klasse | PacketPong.java |
| Absender | Server und Client |
| Empfänger | Client und Server |

**Klassenvariablen**

**keine**

**Verarbeitung**. Falls sich keine Fehler in der vom Pingpaket übergebenen Uhrzeit verstecken, so erfolgt die Berechnung der Zeitdifferenz. Hier wird von der aktuellen Uhrzeit die übergebene Zeit abgezogen und somit die Zeitdifferenz berechnet.

**Beispiel**. Man könnte in der Konsole den Befehl "PONGU" eingeben, jedoch würde er keine relevante Aufgabe erfüllen.

## 3.5 Chat

**Java-Package:** net.packets.chat

Dieses Package enthält alle Paket-Klassen die mit dem Nachrichtenaustausch unterhalb den Spielern zu tun haben. Dazu gehört ein Paket, welches dem Spieler ermöglicht Nachrichten an den Server zu versenden, ein Paket welches der Server dem Spieler zurückschickt als Bestätigung das seine Nachricht angekommen ist und ein Paket welches der Server benutzen kann um Nachrichten an die Spieler zu versenden.

### 3.5.1 Packet: CHAT_MESSAGE_TO_SERVER

**Allgemein**

| | |
|---|---|
| Code | CHATS |
| Klasse | PacketChatMessageToServer.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**chatmsg**

| | |
|---|---|
| Beschreibung | Ein String der die Nachtricht des Spieler enthält welche gesendet werden soll. |
| Validierung | Ist vorhanden. Die Nachricht darf nicht leer sein und darf höchstens 100 Zeichen enthalte. |

**timestamp**

| | |
|---|---|
| Beschreibung | Ein String, der die Absendezeit der Chat Nachricht enthält. |
| Validierung | Keine Validierung wird vorgenommen. |

**Verarbeitung**. Zuerst wird geprüft ob sich der Spieler in der Spielerliste vorhanden ist. Dann wird geprüft ob der Spieler sich in einer Lobby befindet. Ist alles erfüllt nimmt der Server das Paket mit der Nachricht und speichert die fertige Nachricht in der Stringvariablen fullmessage. Beginnt die Nachricht mit einem "@", dann prüft der Server anschliessend ob ein Spieler mit dem darauf folgendem Name existiert. Wenn ja wird der Empfänger gespeichert und das "@" und der Empfänger wird aus der Nachricht entfernt, wenn nicht wird ein CHAT_MESSAGE_TO_CLIENT Paket erstellt und es wird dem Spieler mitgeteilt, das der angegebene Empfänger nicht existiert. Ist alles geprüft wird ein neues CHAT_MESSAGE_TO_CLIENT Paket erstellt welches dann mittels LobbyID vom spieler oder an einen Empfänger verschickt wird.

**Beispiel**. Möchte ein Spieler eine Nachricht an andere Spieler senden zum Beispiel "hallo" dann kann er dies entweder in der Lobby oder im Spiel tun. Der Chat fird aufgerufen und die Nachricht wird eingegeben und abgeschickt. Das Paket mit der Nachricht wird an den Server geschickt. Der Server nimmt die Informationen vonm Spielername und der Nachricht welche anschliesend in soch einen String verpackt wird <['Spielername'-'Zeit'] hallo> , der mittels CHAT_MESSAGE_TO_-CLIENT Paket weiterverschickt wird an die Spieler.

### 3.5.2 Packet: CHAT_MESSAGE_TO_CLIENT

**Allgemein**

| | |
|---|---|
| Code | CHATC |
| Klasse | PacketChatMessageToClient.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**chatmsg**

| | |
|---|---|
| Beschreibung | Ein String der die endgültige Nachtricht enthält, welcher vom Server Verschickt wird und beim Spieler nur noch auf dem Chat Gui ausgegeben werden muss |
| Validierung | Ist vorhanden. Die Nachricht darf nicht leer sein und darf höchstens 130 Zeichen enthalte. 100 Zeichen für die Nachricht selbst und 30 Zeichen für den Benutzernamen und die Zeit. |

**Verarbeitung**. Der Server erstellt das Paket mit dem endgültigen String. Dieses Paket wird an den Spieler geschickt und wird dann im Chat Gui ausgegeben.

**Beispiel**. Der Server erhält ein CHAT_MESSAGE_TO_SERVER Paket mit der Nachricht "hallo", welche keine Fehler aufweist. Daraufhin wird ein CHAT_MESSAGE_TO_CLIENT Paket erstellt mir dem String <['Spielername'-'Zeit'] hallo> welcher dann an die jeweiligen Spieler geschickt wird. Das Paket muss den String entnehmen und nochmals prüfen ob der ganze String nicht zulange ist. Dannach wird der String im Chat Gui des Spieles ausgegeben.

## 3.6 Block

**Java-Package:** net.packets.block Dieses Package enthält alle Paket-Klassen,

welche sich mit relevanten Informationen betreffend den Blocks befassen. Dazu gehört eigentlich nur das Packet PacketBlockDamage, welches dem Server mitteilt, dass ein Block beschädigt wurde. Dies dient dazu, um diesen Schaden erstens mit den anderen Spielern zu teilen und zweitens um den Status des Blockes auf dem Server zu updaten.

### 3.6.1 Packet: BLOCK_DAMAGE

**Allgemein**

| | |
|---|---|
| Code | BLDMG |
| Klasse | PacketBlockDamage.java |
| Absender | Server und Client |
| Empfänger | Client und Server |

**Klassenvariablen**

**blockDestroyerClient**

| | |
|---|---|
| Beschreibung | Ein Integer, welcher die ClientId des Clients enthält, welcher dem Block Schaden zufügt. |
| Validierung | Ist vorhanden. Es wird überprüft ob es sich um einen Integer handelt. |

**blockX**

| | |
|---|---|
| Beschreibung | Ein Integer, welche die x-Koordinate des Blockes darstellt. |
| Validierung | Ist vorhanden. Es wird überprüft ob es sich um einen Integer handelt. |

**blockY**

| | |
|---|---|
| Beschreibung | Ein Integer, welche die y-Koordinate des Blockes darstellt. |
| Validierung | Ist vorhanden. Es wird überprüft ob es sich um einen Integer handelt. |

**damage**

| | |
|---|---|
| Beschreibung | Ein Float, welcher den Schaden darstellt, welcher der Client dem Block zugefügt hat. |
| Validierung | Ist vorhanden. Es wird überprüft ob es sich um einen Float handelt. |

**dataArray**

| | |
|---|---|
| Beschreibung | Ein String array, welcher mit den Stringinformationen vom client/server gefüllt wird. |
| Validierung | Ist vorhanden. es wird überprüft ob der Array genau vier Elemente enthält |

**Verarbeitung**. In der Verarbeitung wird zwischen dem Server und dem Client unterschieden. Falls der Server vom Client ein solches Paket erhalten hat, holt sich der Server zuerst die aktuelle Karte der Lobby, dann wird überprüft ob die Koordinaten des Blockes auf der Karte sind. Danach wird dem Block auf der Server Karte schaden zugefügt, damit der Status des Blockes aktualisiert werden kann. Danach wird in der Servermap Klasse allen Spielern der Lobby mit einem neuen PacketBlockDamage mitgeteilt, dass der Block Schaden erfahren hat.

Falls wir uns auf der Client Seite befinden, wird die aktuelle ClientMap mit dem Schaden des Blockes aktualisiert und so wird der Status des Blockes auf der Client Seite aktualisiert.

**Beispiel**. Ein Spieler hat einen Block beschädigt, der Client erstellt ein PacketBlockDamage mit den nötigen Informationen, diese werden als String "BLDMG 150||200||0.5f" (blockX, blockY, damage) an den Server geschickt. Dieser entpackt diese Informationen, aktualisiert die ServerMap, und sendet allen Clients in der Lobby ein Packet mit dem String "BLDMG 5||150||200||0.5f" (clientId, blockX, blockY, damage). Diese entpacken dann diese Information und aktualisieren, sofern sie nicht der Client sind, der den Schaden verursacht hat, ihre ClientMaps.

## 3.7 Game Status

**Java-Package:** net.packets.gamestatus Dieses Package enthält alle Paket-

Klassen, welche sich mit dem Game Status befassen. Dazu gehören zum Beispiel das PacketHistory, welches dem Spieler eine History aller vergangenen Spiele ausgibt, das PacketReady welches der Lobby mitteilt ob die Spieler bereit für ein Spiel sind oder das PacketGameEnd, welches den Spielern sagt, ob ein Spiel vorbei ist.

### 3.7.1 Packet: GET_HISTORY

**Allgemein**

| | |
|---|---|
| Code | HISGE |
| Klasse | PacketGetHistory.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**Keine Klassenvariablen**

**Verarbeitung**. Der Server erhält eine Anfrage für das Packet von einem Client, erstellt einen String aus der Geschichte der Spiele und sendet die Informationen an den Client durch ein PacketHistory und dem String aus der History.

**Beispiel**. Ein Client fordert eine History der Spiele an, dazu sendet er ein GET_HISTORY-Paket, als String "HISGE" an den Server. Dieser erstellt eine History aller Spiele und sendet diese mit in einem HISTORY-Paket zurück an den Client.

### 3.7.2 Packet: HISTORY

**Allgemein**

| | |
|---|---|
| Code | HISTO |
| Klasse | PacketHistory.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**in[]**

| | |
|---|---|
| Beschreibung | Ein String Array, welches an der nullten stelle "OK" enthält oder eine Fehlermeldung. Falls "OK" sind die restlichen einträge jeweil Zeilen der aktuellen History. Der Aufbau wäre also folgendermassen: ("OK", "line0", "line1", "line2"...) |
| Validierung | Ist vorhanden. ist extended ASCII. |

**Verarbeitung**. Falls es zu keinen Fehlern kam und der erste Eintrag von in[] "OK" ist wird der restliche Inhalt Zeile für Zeile ausgegeben.

**Beispiel**. Ein Client schickt ein GET_HISTORY-Paket an den Server dieser antwortet daruf in dem er eine History erstellt und ein HISTORY-Paket, als String "HISTO OK||Open Lobbies:||myLobby1||Lobbies Of Running Games:||none||Old Games:||myLobby2||" zurück schickt. (In diesem Fall gäbe es also gerade keine Lobby, die ein laufendes Spiel hat) Die History wird dem Client nun im GUI angezeigt.

### 3.7.3 Packet: READY

**Allgemein**

| | |
|---|---|
| Code | READY |
| Klasse | PacketReady.java |
| Absender | Client |
| Empfänger | Server |

**Keine Klassenvariablen**

**Verarbeitung**. Es wird überprüft ob der Benutzer der das Paket geschickt hat, eingeloggt ist und sich in einer Lobby befindet. Ist dies der Fall, wird der Ready-Status des Absenders auf bereit(true) gesetz und die restlichen Lobbymitglieder darüber informiert(mit einem CUR_-LOBBY_INFO-Paket). Es wird geprüft ob nun alle Lobbymitglieder bereit sind. Sind alle bereit wird die Spielrunde gestartet und die Lobbymitglieder mit einem START-Paket darüber informiert.

**Beispiel**. Wir sind in einer Lobby mit einem anderen Mitspieler und wollen die Spielrunde Starten. Wir schicken also ein READY-Paket, als String "READY", an denn Server welcher darauf unseren Ready-Status auf bereit setzt. Der Server informiert darüber mit einem CUR_LOBBY_-INFO-Paket. Im GUI werden wir nun auch als ready angezeigt. Da unser Mitspieler aber noch nicht bereit ist wird die Runde noch nicht gestartet.

### 3.7.4 Packet: START

**Allgemein**

| | |
|---|---|
| Code | START |
| Klasse | PacktetStartRound.java |
| Absender | Server |
| Empfänger | Client |

**Keine Klassenvariablen**

**Verarbeitung**. Die Aktive Stage INLOBBY wird durch die Stage PLAYING ersetzt, wodurch die Anzeige des Benutzers vom InLobby-Menu zum eigentlichen Spiel gewechselt. Das Spiel beginnt.

**Beispiel**. Wir befinden uns in einer Lobby mit 5 anderen Mitspielern, wobei der 5-te noch nicht bereit ist. Als dieser nun ein READY-Paket an den Server sendet, Startet der Server das Spiel und schickt ein START-Paket, als String "START" an alle Lobbymitglieder, wodurch bei diesen gleichzeitig das eigentliche Spiel gestartet wird.

### 3.7.5 Packet: GAME_END

**Allgemein**

| | |
|---|---|
| Code | STOPG |
| Klasse | PacktetGameEnd.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**winner**

| | |
|---|---|
| Beschreibung | Ein String welcher dem Nutzernamen des Rundengewinners entspricht. |
| Validierung | Ist vorhanden. |

**time**

| | |
|---|---|
| Beschreibung | Eine Zahl welche der Dauer der Spielrunde entspricht(in Millisekunden). |
| Validierung | Ist vorhanden. Ist Long. |

**Verarbeitung**. Falls keine Fehler im Paket festgestellt wurden. Wird die Anzeige des Benutzers zum GameOver Menu gewechselt. Darauf wird der Name des Gewinners und die Spieldauer angezeigt.

**Beispiel**. Wir sind im Spiel und einer unserer Mitspieler erreicht die benötigte Anzahl Gold, um die Runde zu gewinnen. Der Server erstellt ein GAME_OVER-Paket mit dem Namen des Gewinners und der Spielzeit, als String "STOPG Joe||1349333576093" und sendet es an alle Lobbymitglieder. Dadurch wird bei diesen die Spielrunde beendet und das GameOver Menu gezeigt.

## 3.8 Items

**Java-Package:** net.packets.items Dieses Package enthält alle Packete, welche

sich mit der Verarbeitung der Items befasst. So kann mit dem PacketItemUsed dem Server mitteilen, dass ein Packet verbraucht wurde, oder mit dem PacketSpawn Item ein Item bei den Clients spawnen.

### 3.8.1 Packet: ITEM_USED

**Allgemein**

| | |
|---|---|
| Code | ITMUS |
| Klasse | PacketItemUsed.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**itemId**

| | |
|---|---|
| Beschreibung | Ein Integer, welcher die ItemId des zerstörten Items enthält. |
| Validierung | Ist vorhanden. Es wird überprüft ob es sich bei den Daten wirklich um einen Integer hält. |

**Verarbeitung**. Wenn der Server ein solches Packet erhält, löscht er das Item aus der ServerItemState Liste.

**Beispiel**. Ein Client verbraucht ein Dynamit mit der ItemId 5, sendet dem Server ein Packet "ITMUS 5". Der Server entpackt die Daten und löscht das Item aus der Liste.

### 3.8.2 Packet: SPAWN_ITEM

**Allgemein**

| | |
|---|---|
| Code | ITMSP |
| Klasse | PacketSpawnItem.java |
| Absender | Client und Server |
| Empfänger | Server und Client |

**Klassenvariablen**

**owner**

| | |
|---|---|
| Beschreibung | Ein Integer, welcher die ItemId des zerstörten Items enthält. |
| Validierung | Ist vorhanden. Es wird überprüft, ob es sich bei den Daten wirklich um einen Integer hält. |

**position**

| | |
|---|---|
| Beschreibung | Ein Integer, welcher die Position des erstellten Items enthält. |
| Validierung | Ist vorhanden. Es wird überprüfrt ob es sich bei den Daten wirklich um einen Vector3f hält. |

**type**

| | |
|---|---|
| Beschreibung | Ein String, der den Itemtyp enthält. |
| Validierung | Braucht keine Validierung |

**dataArray**

| | |
|---|---|
| Beschreibung | Ein String Array, welcher alle Daten enthält.. |
| Validierung | Wird dann in die einzelnen Daten umgewandelt. |

**Verarbeitung**. Auf der Server Seite wird überprüft, ob alle Daten vorhanden sind, danach wird überprüft ob der Spieler in einer Lobby ist und dann wird die Information an die Lobby versendet. Auf der Client Seite Wird überprüft, um welches Item es sich handelt und dann wird ein neues Item des korrekten Typen an der korrekten Position erzeugt. Weiter wird beim Item überprüft, ob das Item dem aktuellen Client gehört oder nicht. Zusätzlich wird geprüft, ob das Tutorial für dieses Item schon einmal ausgeführt wurde. Falls nicht, wird dieses noch angezeigt.

**Beispiel**. Ein Client zerstört einen Fragezeichenblock, der Server erstellt ein PacketSpawnItem Packet und schickt es an die Lobby. Der Client, welcher den Fragezeichenblock zerstört hat wird als Besitzer festgelegt und das Item wird bei allen Spielern in der Lobby gespawnt.

## 3.9 Life

**Java-Package:** net.packets.life Dieses Package enthält alle Packete, welche mit der Lebensinformation und deren Verarbeitung zu tun haben.

### 3.9.1 Packet: LIFE_STATUS

**Allgemein**

| | |
|---|---|
| Code | LSTAT |
| Klasse | PacketLifeStatus.java |
| Absender | Client und Server |
| Empfänger | Server und Client |

**Klassenvariablen**

**newLives**

| | |
|---|---|
| Beschreibung | Ein Integer, welcher den Lebensstand enthält. Wäre der Server der Absender, so würde newLives den definitiven Lebensstand darstellen. Wäre der Client der Absender, würde newLives die in seiner Auffassung stattgefundene Änderung des Lebensstands des betroffenen Clients darstellen. |
| Validierung | Ist vorhanden. Es wird überprüft, ob es sich um einen Integer handelt und ob sich die Zahl im Interval [-1,3] befindet. |

**effectedPlayerId**

| | |
|---|---|
| Beschreibung | Ein Integer, welcher die ClientId des betroffenen Spielers enthält. |
| Validierung | Ist vorhanden. Es wird überprüft, ob es sich um einen Integer handelt und ob sich der betroffene Spieler in der gleichen Lobby wie der Spieler, der das Geschehen meldet, befindet. |

**Verarbeitung**. Geschieht die Verarbeitung auf der Server Seite, so übergibt er das Paket an einen Referee. Falls das Paket auf der Client Seite verarbeitet wird, wird der Lebensstand des betroffenen Spielers entsprechend neu gesetzt.

**Beispiel**. Ein Client verliert wegen einem Steinblock oder wegen eines Dynamits ein Leben. Jeder Client, der das Geschehen realisiert hat, schickt dem Server einen String "LSTAT 1||3". Die "1" steht für den neuen möglichen Lebensstand und die "3" für die ClientId des betroffenen Spielers. Der Server erhält den String, wandelt ihn in ein PacketLifeStatus um und übergibt die Meinung des Senders an einen Referee. Falls die Refereeinstanz genug Meinungen über das Geschehen gesammelt hat, so wird ein Lifestatus Paket mit dem entgültigen Lebensstand und mit der ClientId des betroffenen Spielers auf Serverseite durch diesen Referee erstellt und an die Lobby verschickt.

## 3.10 Lists

**Java-Package:** net.packets.lists Dieses Package enthält alle Packete, mit denen ein Spieler Listen anfordern kann. Dazu gehört zum Beispiel das PAcketGamesOverview, welches dem Spieler einen Overview über alle Spiele gibt, das PacketPlayerList, welches dem Spieler eine komplette Liste aller Spieler auf dem Server schickt und noch mehr.

### 3.10.1 Packet: HIGHSCORE

**Allgemein**

| | |
|---|---|
| Code | HIGHS |
| Klasse | PacketHighscore.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**highscore**

| | |
|---|---|
| Beschreibung | Ein String Array, welcher den aktuellen Highscore auf dem Server enthält. |
| Validierung | Ist vorhanden. Nur extended ASCII und ob Daten vorhanden sind. |

**Verarbeitung**. Auf der Server Seite erhält der Server eine Highscore Anfrage von einem Client. Der Server erstellt dann einen String, welcher aus einem Indikator, ob die Daten korrekt sind, und dem aktuellen Highscore. Auf der Spielerseite wird der String in den String Array eingefüllt und danach dem Client ausgegeben.

**Beispiel**. Ein Client fragt den Highscore an, der Server erhält die Anfrage und erstellt einen String als Antwort an den Client. Dieser String "HIGHS OK(den aktuellen Highscore)" wird dann dem Client zurückgeschickt. Der Client entpackt die Daten und gibt den Highscore aus.

### 3.10.2 Packet: PLAYERLIST

**Allgemein**

| | |
|---|---|
| Code | PLALS |
| Klasse | PacketPlayerList.java |
| Absender | Client und Server |
| Empfänger | Server und Client |

**Klassenvariablen**

**dataArray**

| | |
|---|---|
| Beschreibung | Ein String Array, welches die nötigen Daten enthält. |
| Validierung | Ist vorhanden. Es wird überprüft ob die Daten vorhanden sind und extended ASCII sind. |

**Verarbeitung**. Die Verarbeitung auf Server Seite erstellt der Server eine Liste aus allen Spielern und sendet diese an den korrekten Client. Auf der Clientseite werden die Daten, welche vom Server gekommen sind ausgegeben.

**Beispiel**. Der Client ertellt ein PacketPlayerLsit Packet ohne Inputparameter um die daten beim Server anzufragen, dieser stellt diese zusammen und schickt sie zurück an den Client. Dieser verarbeitet dann die Informationen und gibt sie dem Spieler aus.

## 3.11 Map

**Java-Package:** net.packets.map Dieses Package enthält alle Packete, mit denen die Map generierung und aktualisierung durchgeführt werden. So wird zum Beispiel mit PacketBroadcastMap die aktuelle Karte mit den Spielern geteilt.

### 3.11.1 Packet: FULL_MAP_BROADCAST

**Allgemein**

| | |
|---|---|
| Code | MAPBC |
| Klasse | PacketBroadcastMap.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**mapString**

| | |
|---|---|
| Beschreibung | Ein String welcher die gesamte Map beinhaltet. |
| Validierung | Nicht nötig. |

**mapArray**

| | |
|---|---|
| Beschreibung | Ein String Array, welcher die Mapenthält und dann in die Karte umgewandelt wird. |
| Validierung | Vorhanden, es wird überprüft ob die Karte die korrekte Länge grösse hat und ob die Blocktypen korrekt umgewandelt wurden. |

**Verarbeitung**. Der Client erstellt entweder eine neue Karte oder aktualisiert die bestehende. Falls Fehler vorhanden sind, wird dies auch ausgegeben.

**Beispiel**. Der Server erstellt ein PacketBrodcastMap mit dem String, welcher die ganze Karte beihnaltet und schickt dies dem Client. Dieser entpackt den String "MAPBC (Karteninformationen" und aktualisiert die Karte.

## 3.12 Player Properties

**Java-Package:** net.packets.playerprop Dieses Package enthält alle Packete,

mit denen die Spielerpositionen an die verschiedenen Spieler geschickt werden.

### 3.12.1 Packet: POSITION_UPDATE

**Allgemein**

| | |
|---|---|
| Code | POSXY |
| Klasse | PacketPos.java |
| Absender | Server und Client |
| Empfänger | Client |

**Klassenvariablen**

**playerId**

| | |
|---|---|
| Beschreibung | Ein Integer, welche die playerId des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Integer vorhanden ist und ein Integer ist. |

**posX**

| | |
|---|---|
| Beschreibung | Ein Float, welche die Position auf der X-Achse des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**posY**

| | |
|---|---|
| Beschreibung | Ein Float, welche die Position auf der Y-Achse des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**rotY**

| | |
|---|---|
| Beschreibung | Ein Float, welche die Rotation des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**Verarbeitung**. Auf der Serverseite wird der Spieler im Gamestate aktualisiert und die neue Position des Spielers wird an die anderen Spieler der Lobby verschickt. Auf der Client Seite wird die Position des sich veränderten Spielers aktualisiert.

**Beispiel**. Spieler A bewegt sich nach Links, er verschickt ein PacketPos an den Server, dieser überprüft die Daten und verschickt allen anderen Spielern in der Lobby die aktualisierte Position. Diese erhalten die Informationen und verarbeiten diese entsprechend und aktualisieren die Spielerposition von Spieler A.

### 3.12.2 Packet: PLAYER_DEFEATED

**Allgemein**

| | |
|---|---|
| Code | PDEAD |
| Klasse | PacketDefeated.java |
| Absender | Server |
| Empfänger | Client |

**Klassenvariablen**

**defeatedClientId**

| | |
|---|---|
| Beschreibung | Ein Integer, welche die clientId des ausgeschiedenen Spielers enthält. |
| Validierung | Es wird überprüft ob der Integer vorhanden ist und ein Integer ist. |

**Verarbeitung**. Auf der Client Seite wird die "defeated" Flag des Spieler auf true gesetzt, was dessen Model durch einen Grabstein ersetzt und beim betroffenen Spieler alle Controls ausschaltet und die Kamera frei bewegbar macht.

**Beispiel**. Spieler 3 verliert sein zweites Leben. Der Server schickt ein Player Defeated Packet via "PDEAD 3" an die Lobby. Spieler 3 geht in den Zuschauer Modus und die anderen Spieler aktualisieren ihre Modelle.

### 3.12.3 Packet: VELOCITY_UPDATE

**Allgemein**

| | |
|---|---|
| Code | VELXY |
| Klasse | PacketVelocity.java |
| Absender | Client |
| Empfänger | Server |

**Klassenvariablen**

**playerId**

| | |
|---|---|
| Beschreibung | Ein Integer, welche die playerId des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Integer vorhanden ist und ein Integer ist. |

**curvX**

| | |
|---|---|
| Beschreibung | Ein Float, welcher die Geschwindigkeit auf der X Achse beinhaltet. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**curvY**

| | |
|---|---|
| Beschreibung | Ein Float, welcher die Geschwindigkeit auf der Y Achse beinhaltet.. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**tarvX**

| | |
|---|---|
| Beschreibung | Ein Float, welcher die Zielgeschwindigkeit auf der X-Achse des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**tarvY**

| | |
|---|---|
| Beschreibung | Ein Float, welcher die Zielgeschwindigkeit auf der Y-Achse des sich bewegenden Spielers enthält. |
| Validierung | Es wird überprüft ob der Float vorhanden ist und ein Float ist. |

**Verarbeitung**. Auf der Serverseite wird der Spieler im Gamestate aktualisiert und die neue Geschwindigkeit des Spielers wird an die anderen Spieler der Lobby verschickt. Auf der Client Seite wird die Geschwindigkeit des sich veränderten Spielers aktualisiert.

**Beispiel**. Spieler A bewegt sich nach Links, er verschickt ein PacketVelocity an den Server, dieser überprüft die Daten und verschickt allen anderen Spielern in der Lobby die aktualisierte Geschwindigkeit. Diese erhalten die Informationen und verarbeiten diese entsprechend und aktualisieren die Geschwindigkeit von Spieler A.

# List of Figures