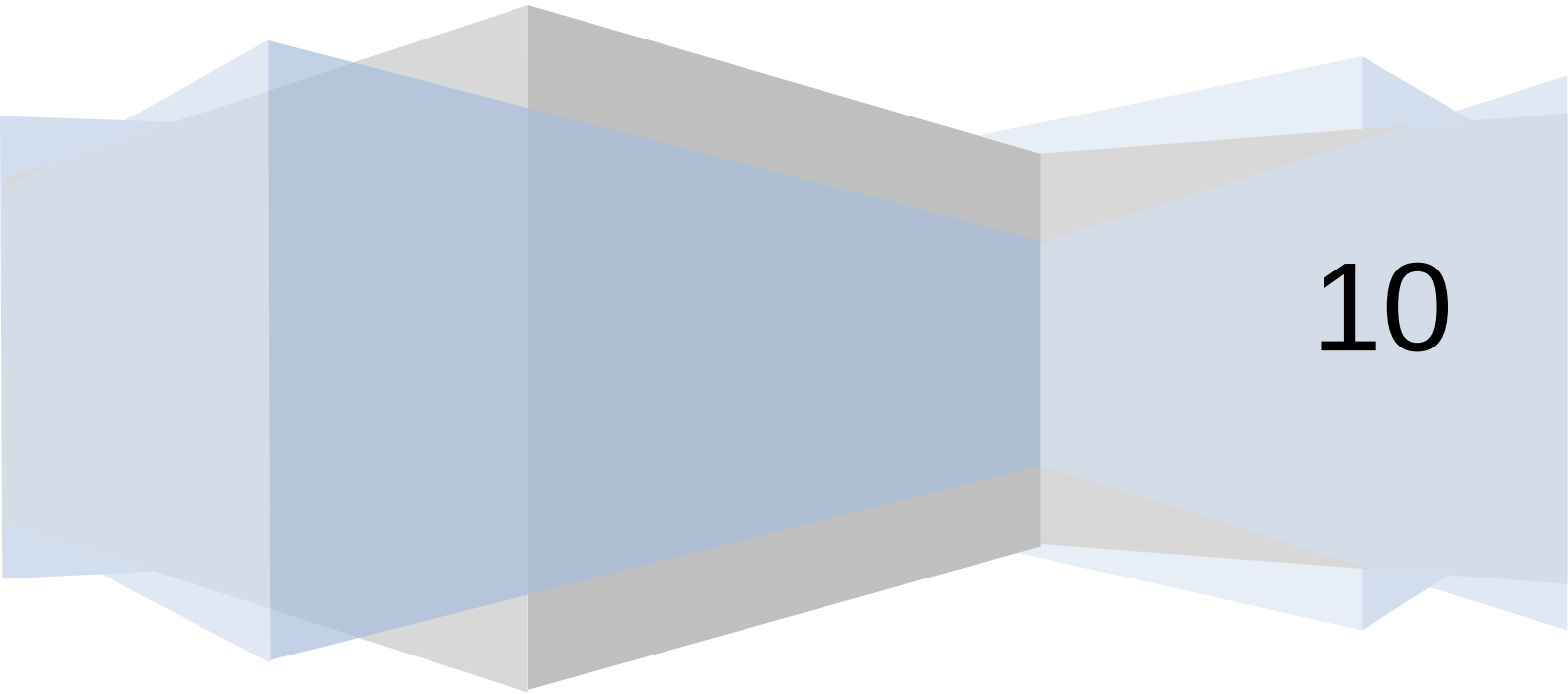
Berner Fachhochschule <-> Informations-Technologie

7301p-Projekt 1

Multiplayer-Pacman-Game

Thomas Häni (hanit1) <-> Andreas Emch (emcha1)



10

Inhalt

1 Ziel Projekt-Auftrag

Dieses Projekt wird mit dem Ziel ausgeführt, uns mit der Praxis der Projektführung vertraut zu machen. Zu diesem Zweck realisieren Thomas Häni und Andreas Emch das Projekt „Multiplayer Pacman“. An regelmässigen Sitzungen wird mit der betreuenden Person (Herr Jürgen Eckerle) der aktuelle/weitere Verlauf besprochen sowie die weitere Planung erarbeitet. Weiter soll versucht werden, auf den bisherigen Grundlagen aufzubauen sowie durch Selbststudium weiteres Wissen zu erlangen, welches sich spezifisch auf unser Projekt bezieht.

Unser Ziel ist es, ein PacMan Game zu realisieren, welches multyplayer fähig ist. Die besondere Herausforderung wird darin bestehen, das das laufende aktuelle Spiel auf jedem Rechner abgeglichen wird. Um dies zu realisieren, werden wir uns mit der Synchronisation in Java beschäftigen. Es sollen maximal vier Spieler an verschiedenen Standorten gleichzeitig miteinander spielen können. Jeder Spieler muss sich zuvor registrieren und kann sich danach am Spiel anmelden. So sieht man immer gegen wenn man spielt und die Verbindung zur Datenbank und dem Internet sind sichergestellt. Das Internet selber wird nur für die Registrierung und Spieleröffnung gebraucht, sobald ein Spiel im Netzwerk eröffnet wird sollen die Computer direkt miteinander kommunizieren.

2 Einleitung

Im Rahmen des Modules „Projekt 1 7301“ geht es darum, ein gewähltes Projekt in Gruppen selbstständig von A bis Z zu erarbeiten. Dabei standen uns diverse Projekte zur Auswahl. Wir haben uns somit für das Projekt **abgeleitetes Spiel der PacMan-Version (multiplayer fähig)** entschieden. Die Gründe für unsere Wahl sind, weil es einen sehr vielseitigen Aspekt hat. Wir sind dabei sehr interessiert daran, diese für uns neuen Techniken der Programmierwelt zu erforschen sowie neues Wissen darüber zu erlangen. Zu diesen Aspekten zählen wir unteranderem die komplette Netzwerkprogrammierung. Wie können Spielzustände synchronisiert oder abgeglichen werden, sodass alle Spieler im Netzwerk den gleichen Stand auf dem Bildschirm haben. Oder mit welchen Netzwerk-Protokollen wie **UDP** oder **TCP** man am besten arbeitet, um das bestmöglichste Ergebnis zu erhalten und wie die ganze Client-Server Geschichte wie zum Beispiel Sockets funktioniert. Dies sind unteranderem die Aspekte, warum wir uns für dieses Projekt entschieden haben. Es ist eine neue Herausforderung für uns Studenten und wir werden dabei versuchen, das bestmöglichste Ergebnis herauszuholen.

3 Planung

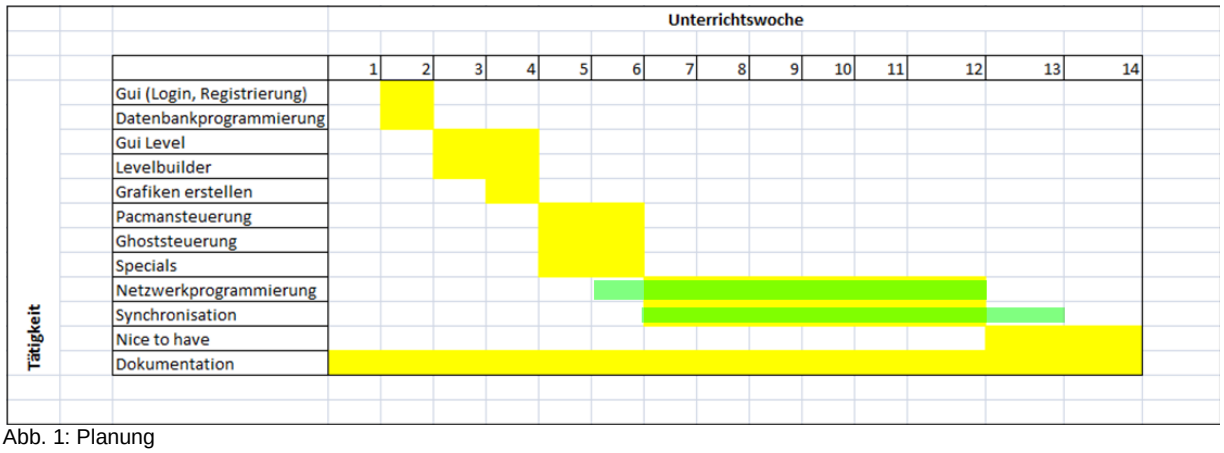


Abb. 1: Planung

3.1 Ablauf

Zwei Diagramme -> Ist/Soll Zustand

- Grundsätzlich konnten wir unsere Planungsvorgaben einhalten. Einziger Punkt, bei dem wir mehr Zeit brauchten als geplant, war die Netzwerkprogrammierung/Synchronisation (grüne Markierung). Wir mussten leider relativ viel im vorhandenen Programcode ändern um das ganze Spiel überhaupt anständig Netzwerkfähig zu halten. Zwar haben wir einige Aspekte schon im Voraus beachtet, jedoch nicht alle. So verloren wir unter anderem Zeit für bereits bestehende Funktionen anzupassen und netzwerkfähig zu machen.
- Ansonsten hatten wir keine anderen Probleme betreffend der Planung/Einhaltung.

4 Analyse Pacman

Hier wird eine grobe Übersicht über das Spiel gegeben (mit UML). Diese Ansicht soll auf der obersten Schicht stattfinden.

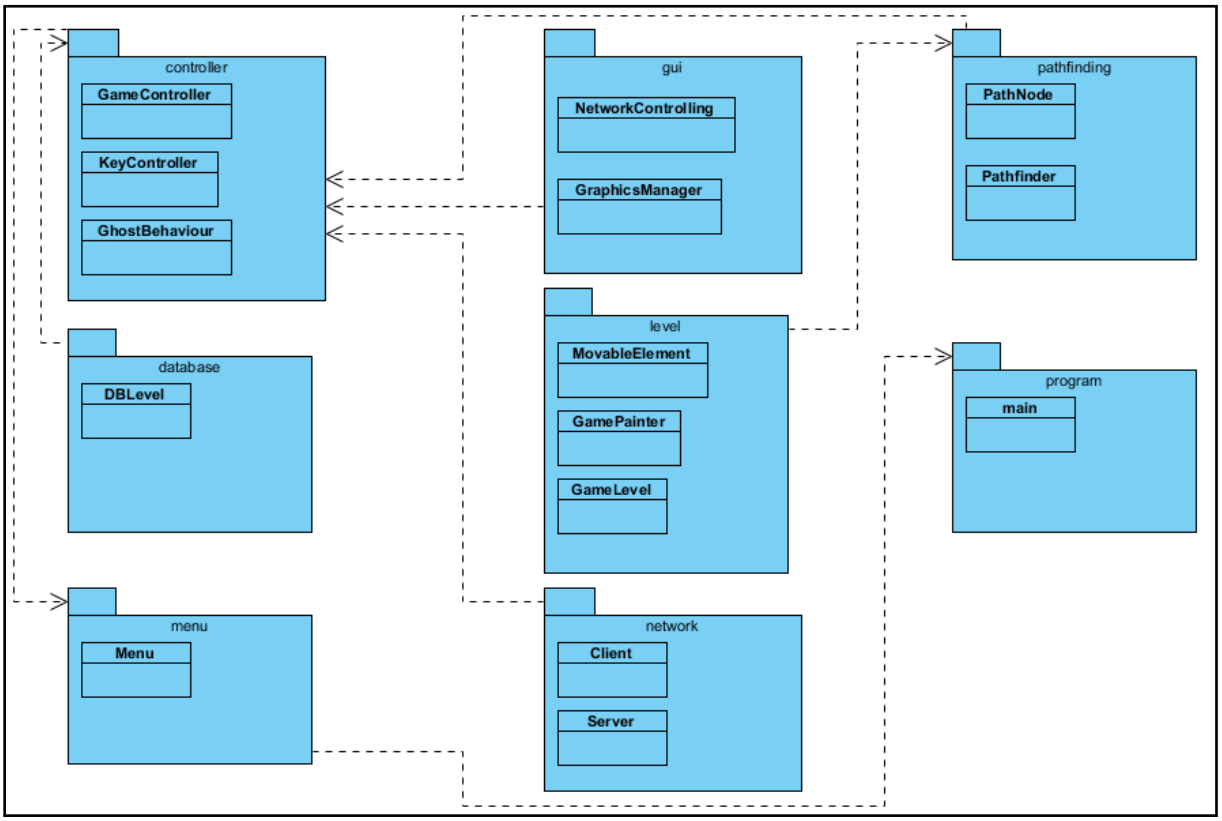


Abb. 2: UML Diagramm Pacman

4.1 Klassendiagramm

Die Analyse soll einen Überblick verschaffen, wie die Struktur im Wesentlichen aufgebaut ist. Hier werden keine Details demonstriert. Dies erfolgt dann erst in der eigentlichen Aufgabe des Projektes ab Punkt 5 (Synchronisation/Netzwerk). Weiter werden auch nur die wesentlichen/wichtigen Klassen der entsprechenden Packages aufgezeigt.

4.2 Package: controller

4.2.1 Klasse: GameController

Diese abstrakte Klasse ist für den gesamten Ablauf des Spiels zuständig und wird durch die jeweiligen Klassen abgeleitet um die verschiedenen Spielmodi zu repräsentieren, da es gewisse Unterschiede gibt ob ein PacMan alleine auf dem Spielfeld umher irrt oder gegen andere spielt.

- Steuerung der Geister
- Steuerung der Palmins

- Handling von Kollisionen
  Wer hat wen gefressen
- Handling der Specials
  Starten und beenden der Specials
  - Pacman kann die Geister fressen
  - Pacman bekommt verschiedene Spielgeschwindigkeiten annehmen (sehr langsam oder sehr schnell).
  - Steuerung wird beim Pacman vertauscht (Taste rauf wird gedrückt und er geht runter).

4.2.2 Klasse: KeyController

Die abstrakte Klasse KeyController wird durch drei Klassen abgeleitet und ist für die Tastatureingabe des Benutzers zuständig und behandelt diese entsprechend:

- KeyControllerSingle
  Diese Klasse ist zuständig für ein Single Game. Sie handelt die gewöhnliche Tastensteuerung.
- KeyControllerMultiLocal
  Diese Klasse ist zuständig für die Steuerung von zwei Spielern auf einem Gerät.
- KeyControllerMultiNetwork
  Diese Klasse ist zuständig für die Steuerung über das Netzwerk oder Internet mit bis zu vier Spielern. Die Steuerung wird via den zu Verfügung stehenden Protokollen welche in der Klasse ServerTCP und ServerUDP implementiert wurden versendet.

4.2.3 Klasse: GhostBehaviour

Diese Klasse ist zuständig für das gesamte Verhalten der einzelnen Geister. Der Pacman und die Geister können verschiedene Zustände erlangen.

- Geister sind im Startblock gefangen und müssen zuerst hinausfinden.
- Geister können durch den Pacman gefressen werden und kehren zum Startpunkt zurück.
- Geister konzentrieren sich auf den Spieler mit den meisten Punkten und folgen alle diesem.
- Geister können in einem „eingefrorenen“ Zustand sein und laufen jeweils in eine unterschiedliche Ecke vom Spielfeld.

Um die verschiedenen Geister zu implementieren wurden die speziellen Eigenschaften in eigene Klassen ausgelagert:

- GhostBlinky: Implementiert „Blinky“, den aggressivsten Geist der immer den zu verfolgenden PacMan direkt im Visier hat.
- GhostPinky: Implementiert „Pinky“ um dem Pacman den Weg abzuschneiden in dem er immer das Feld, welches 4 Felder vor dem Pacman liegt, in Angriff nimmt.
- Gnostik: Implementiert „Inch“, welcher immer als Ziel das Felde 3 Felder neben Pacman hat.
- GhostClyde: Implementiert „Clyde“, den verlorenen Geist, welcher immer ein bisschen im Zeug umherirrt.

4.3 Package: Gui

4.3.1 Klasse: NetworkController

Diese Klasse wurde ursprünglich zum Testen des Netzwerkes verwendet (Client-Server-Verbindung). Der *NetworkController* hat jedoch zum finalen Zeitpunkt keine Funktion mehr und wurde gelöscht, wird aber trotzdem erwähnt, um den Ablauf im Klassendiagramm zu zeigen.

Dies war der erste Schritt um das Pacman-Spiel Richtung Multiplayerspiel zu verändern. Mit diesem Fenster wurde man automatisch an einem fest vorgegebenen Server angemeldet und man konnte die gesamte Steuerung des Spiels von einem beliebig anderen Computer im Netzwerk übernehmen.

4.3.2 Klasse: GraphicsManager

Diese Klasse ist ein Zwischenspeicher für die Bilder. Durch diesen Vorgang müssen die Bilder nicht immer neu geladen werden und stehen so schneller für den Ablauf des Spiels zur Verfügung. Er hat lediglich die Funktion einer Performance Verbesserung.

4.4 Package: pathfinding

4.4.1 Klasse: PathNode

Diese Klasse ist eine kleine Hilfsklasse für den Wegfindungsalgorithmus und um Positionen mit einer bestimmten Richtung zu versehen. So kann diese verwendet werden um die Positionen und Richtung der einzelnen beweglichen Elemente auf dem Spielfeld verschickt werden um so die Synchronisierung zu vereinfachen.

4.4.2 Klasse: PathFinder

Diese Klasse ist zuständig, dass die verschiedenen Geister den kürzesten Weg zu dem gewünschten Ziel (Pacman) finden. Zuerst wollten wir einen vorhanden Algorithmus für die Wegfindung benutzen, wir hatten dann aber relativ schnell einen eigenen kleinen Algorithmus mit einer Queue zusammengestellt welcher immer alle Wege gleicher Länge

absucht (zuerst alle Wege mit der Distanz eins, dann mit der Distanz zwei, usw.) So sind wir sicher dass es der kürzeste Weg ist sobald die gesuchte Position erreicht wurde. Zur Hilfe benötigt dieser Algorithmus eine Queue.

4.5 Package: database

4.5.1 Klasse: DBLevel

Diese Klasse ist zuständig für die Speicherung des Levels in der Datenbank. Dieses ist in einer als String hinterlegt und kann durch den levelBuilder gelesen und interpretiert werden.

4.6 Package: level

4.6.1.1 Klasse: GameLevel

Diese Klasse ist das 1:1 Abbild des Status des Levels. Hier wird unter anderem definiert, wo die Geister ihre Startpositionen haben; dass man nur einem gewissen Weg entlang gehen kann (dieser wurde errechnet aus der Map vom DbLevel) und nicht durch die Mauern kommt und zuletzt auch wie die Blöcke aufgebaut sind, damit daraus ein Level entsteht. Der Level beinhaltet auch alle Objekte wie Coins, Geister und Pacmans und weitere nötige Details für die Spielsteuerung.

4.6.2 Klasse: Builder

Der Builder bekommt für die Erstellung eines Levels ein Bleuel aus der Datenbank. Daraus generiert er dann den Level mit allen nötigen Objekten und Einstellungen.

4.6.3 Klasse: MovableElements

Diese Klasse ist zuständig für alle beweglichen Elemente (Geister und Pacmans). Bewegt werden diese dann allerdings via Aufruf aus dem *GameController.*

4.6.4 Klasse: GamePainter

Diese Klasse ist für die graphische Darstellung auf dem Bildschirm des Levels zuständig. Dazu benötigt diese den aktuellen Level der durch den level.Builder erstellt wurde. Der GamePainter sorgt dann nur für die graphische Darstellung, in dem er auf die Objekte des Levels zugreift.

4.7 Package: program

4.7.1 Klasse: main

Über diese Klasse wird das Spiel gestartet. Ebenfalls beinhaltet diese Klasse einige Hilfsfunktionen um das eigentliche Spiel zu initialisieren damit dann gespielt werden kann.

4.8 Package: menu

4.8.1 Klasse: Menu

Diese Klasse bindet sämtliche Untermenu-Klassen wie zum Beispiel unten (PrintScreen) ein. Die Aktionen der Menupunkte wurden mit dem Command-Pattern erstellt im Package menu.actions.

Abb. 3: Bildausschnitt Sprach-Menu

Abb. 4: Bildausschnitt Menu

4.9 Package: network

In der gesamten Packlage ist alles Nötige für das Netzwerk implementiert. Dazu mehr Details im folgenden Abschnitt.

5 Design Netzwerk

5.1 Client-Server Konzept

Damit ein Spieler einem bestehenden Spiel beitreten kann, welches noch nicht gestartet muss er (der Client) dem Server eine Anfrage schicken. Diese besteht aus der UserId und seiner IP-Adresse. Klappt alles, so kriegt der Client eine **PlayerId** und wird in die Listeners des Servers aufgenommen. Anschliessend schickt der Server allen bereits angemeldeten Listeners (Clients) ein Update der PlayerList. Diese enthält alle Listeners inkl. UserId, PlayerId, Charakter und IP. Somit sind immer alle Clients auf dem gleichen Stand. Sobald dann genügend Spieler angemeldet sind kann das Spiel gestartet werden.

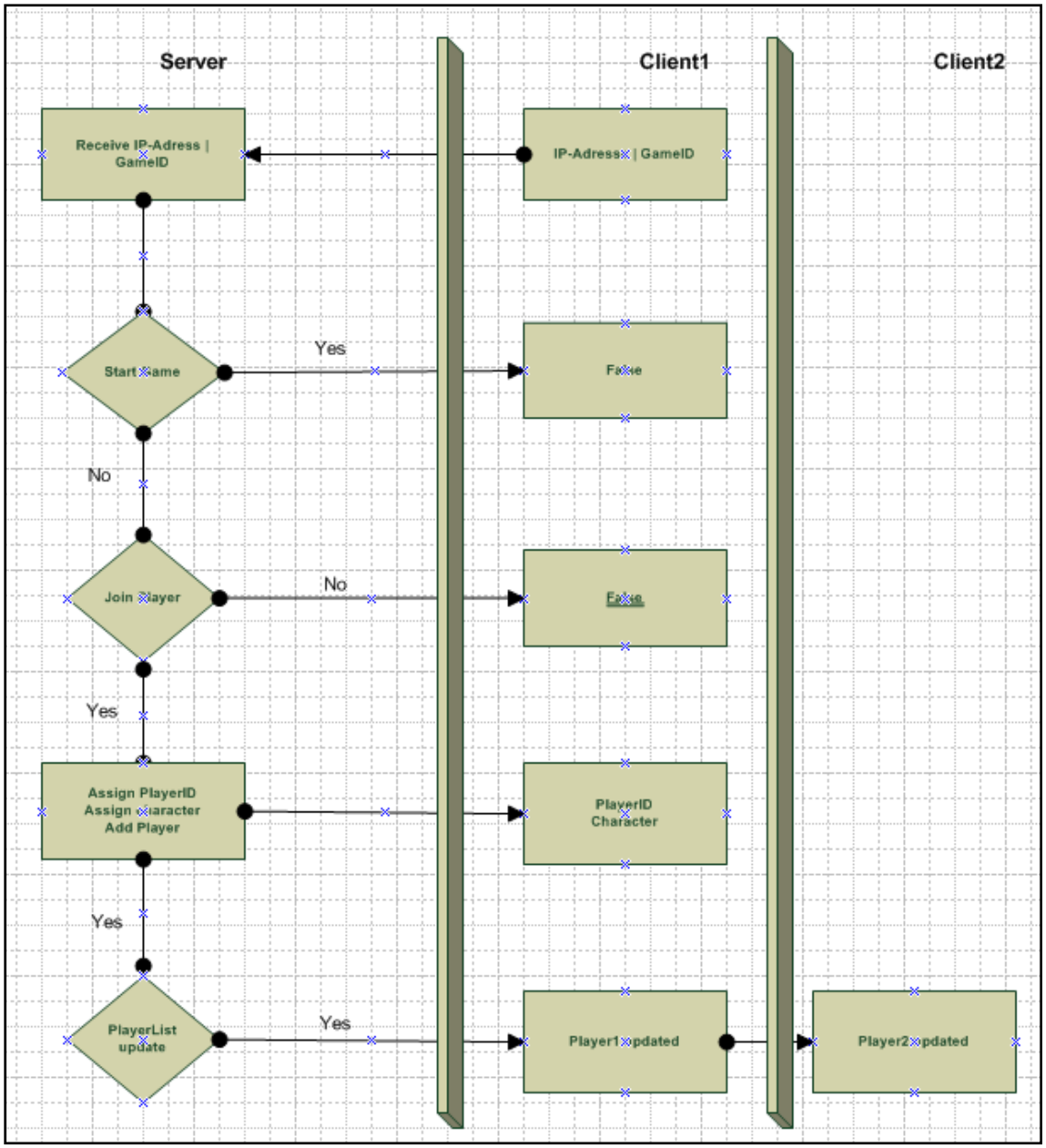


Abb. 5: Client-Server Konzept

5.2 Klassendiagramm Netzwerkumgebung

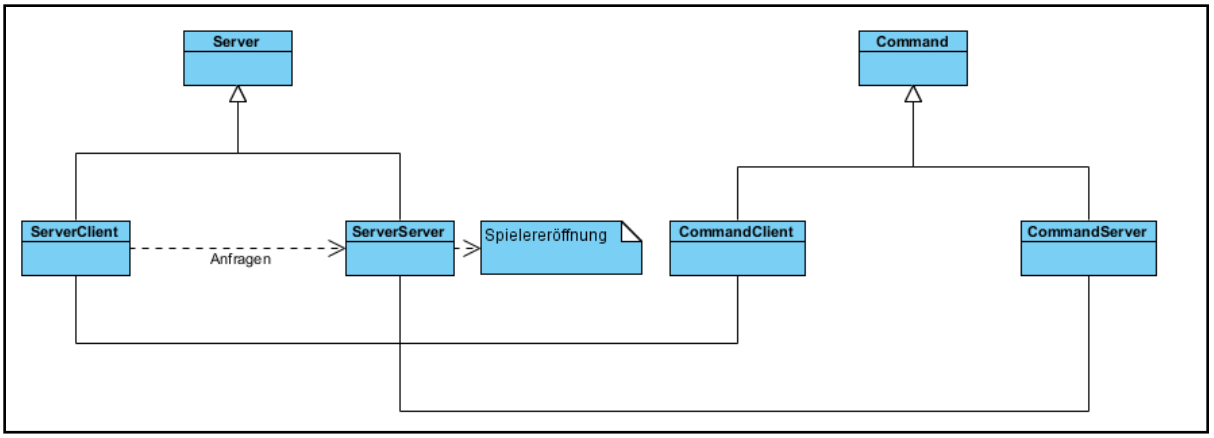


Abb. 6: Klassendiagramm Netzwerkumgebung

Zuerst wollten wir alles Peer2Peer machen. Während des Projektes haben wir uns aber entschlossen, einige Sachen auf ein Server-Client System auszulegen. Um in Java eine Art Perr2Peer zu bewerkstelligen müssen auf allen Clients-Server laufen, welche die Anfragen abhören, aufnehmen und bearbeiten. Diese Server laufen auf einem anderen Port als der GameServer (zum An- und Abmelden), damit es keine Socket-Konflikte gibt in Java.

5.2.1 Vererbung

Wir haben uns bei den Klassen für das Server, ServerServer und ServerClient Konzept entschieden, weil so strikte unterschieden werden kann, welche Anfragen für den Server und welche für den Client zuständig sind. Somit können Verwechslungen ausgeschlossen werden.

5.2.2 Klassen: ServerClient/CommandClient

Die Klasse *ServerClient* sieht, wenn eine Anfrage gestellt wird. Sobald dies der Fall ist, startet die Klasse *CommandClient.* Diese ist dann verantwortlich für die Abarbeitung aller anstehenden Anfragen und bearbeitet diese in Threads. Beide Klassen *ServerClient* und *CommandClient* sind bei jedem Spieler existent, welche sich am Spiel beteiligen.

5.2.3 Klassen: ServerServer/CommandServer

Die Klasse *ServerServer* ist verantwortlich für das An- und Abmelden am Spiel. Diese Klasse ist nur auf dem Spielerzeuger verfügbar. Sobald dieser das Spiel startet oder beendet, kommt die Klasse *CommandServer* zum Einsatz und handelt diese Anfragen.

5.3 Client-Server vs. Peer2Peer

Die jeweiligen Aufgaben haben wir aus folgenden Gründen so implementiert:

- Anmeldung / Abmeldung / Spiel-Server
  Um alle Clients mit den gleichen Informationen über Spieler und IPs zu updaten und damit es klar ist wo sich neue Spieler beziehungsweise verbundene Spieler

abmelden ist es einfacher eine Server-Client Struktur zu nehmen da die Listen nur an einem Ort verwaltet und dann nur mit allen synchronisiert werden müssen.

- Key-Events von Spielern im Spiel und Änderungen des Spielstatus (Pause, Weiterspielen, Starten)
  Damit die Anfragezeiten möglichst kurz sind und die Anfragen nicht zuerst über einen Server gehen müssen der diese einfach nur an alle Clients weiterleitet haben wir uns für ein Peer2Peer entschieden. So schicken sich die Events jeweils alle Clients gleich allen anderen.
- Synchronisierung der Spielstände
  Die Spielstände des Levels und somit aktuellen Spieles zu synchronisieren haben wir auch eine Server Client Struktur geeinigt. So müssen die Clients nur überlegen, ob sie die Pakete bearbeiten wollen oder ob sie schon zu alt sind und somit verwerfen. Die Idee war auch den Spielstand im Voraus zu berechnen und so zu synchronisieren, damit die Zeitdifferenz für die Netzwerkübertragung ausgebügelt werden kann.
  Spielstände sind z.bsp:
    - Für Geister:
      - Position
      - Richtung
      - Welcher PacMan wird gejagt
      - In welchem Modus ist der Geist (Jagen, fliehen, tot...)
    - Für Pacmans:
      - Position
      - Richtung
      - Geschwindigkeit

Wichtige Erkenntnis:

- Auf dem PC-Client, welcher das Spiel eröffnet, läuft sowohl ein Client als auch ein Server. Damit müssen keine Sonderfälle beachtet werden (Client-Server muss nicht manuell gestartet oder beendet werden).
- Wenn der Spielerzeuger das Spiel vorzeitig verlässt oder beendet, ist somit auch für alle Anderen das Spiel zu ende.

5.4 Zustandsmeldungen/Spielgenerierung

Derjenige Spieler, welcher das Spiel eröffnet, erreicht den Zustand des Servers. Die restlichen Spieler (Clients), welche dem Spiel beitreten wollen, müssen sich über den Server registrieren lassen. Dieser vergibt anschliessend eine **Typenbezeichnung**, eine **UserId** sowie die Rückgabe der jeweiligen **IP-Adresse**. Bei der Typenbezeichnung wird automatisch ein Spielertyp (siehe Tabelle unter 5.6) zugeordnet.

Zusätzlich startet aber pro Spieler ebenfalls ein Server, welcher die Anfragen von allen Spielern direkt entgegennimmt und alle Anfragen an alle Spieler sendet bei Statusänderungen. Somit geht keine Zeit verloren wenn eine Nachricht über einen Server geht und zuerst von diesem verarbeitet werden muss.

5.5 Spielterminierung

5.5.1 Anmeldung/Abmeldung am Spiel

Anmelden:

Die Klasse *ServerServer* erhält eine Anfrage betreffend einer Anmeldung von einem Client *(ServerClient) und erstellt ein CommandServer.* Dieses wird mit dem Socket der Anfrage verknüpft und bearbeitet diese. So wird der Client am Spiel angemeldet. Es wird unter anderem geprüft ob sich ein User zweimal anzumelden versucht und so weiter.

```
public class ServerServer extends Server {

	public static void main(String[] args) {
		GameState.setPlayers(1);
		GameState.gameServer = new ServerServer(4);

		new Thread(GameState.gameServer).start();
	}

	private ArrayList<DbUser> loggedUsers = new ArrayList<DbUser>();
	private int maxPlayers = 0;

	public int addUser(int userId, IP ip) {
		for (DbUser user : loggedUsers) {
			if (user.getId()==userId) {
				return -1;
			}
		}

		DbUser newUser = new DbUser(userId);
		loggedUsers.add(newUser);
		newUser.setPlayerId(loggedUsers.indexOf(newUser));
		newUser.setIP(ip);
		return newUser.getPlayerId();
	}
```

Abb. 7: Anfrage betreffend eines neuen User am *ServerServer*

```java
@Override
public Message connect(Message message) {
        System.out.println(message);
        Message answer = new Message(Message.Type.OK);
        int userId = Integer.parseInt(message.getParameter(0).toString());
        IP ip = new IP(message.getParameter(1).toString());

        try {
                if (GameState.gameServer.canLogin(ip)) {
                        int playerId = GameState.gameServer.addUser(userId, ip);
                        GameState.gameServer.addListener(ip);
                        System.out.println("already " + GameState.gameServer.getUser().size() + " of " + GameState.getPlayers() + " players");
                        answer.addParameter(0);
                        answer.addParameter(playerId);

                        GameState.gameServer.sendPlayerList();
                } else {
                        throw new RuntimeException("Can't login. To many players or you're ip is already in use.");
                }

                if (GameState.gameServer.addListener(ip)) {
                        throw new RuntimeException("IP already logged in");
                }
        } catch (RuntimeException ex) {
                answer = new Message(Message.Type.NOK);
                answer.addParameter(ex.getMessage());
        }
        System.out.println(answer);

        return answer;
}
```

Abb. 8: Handling der Anfrage für das Anmelden *(CommandServer)*

```java
@Override
protected void handleSocket(Socket socket) {
        CommandServer command = new CommandServer(socket);
        Thread t = new Thread(command);
        t.start();
}

public void sendPlayerList() {
        Message message = new Message(Message.Type.PLAYER_LIST);
        message.addParameter(GameState.gameServer.loggedUsers.size());

        for (DbUser dbUser : GameState.gameServer.loggedUsers) {
                message.addParameter(dbUser.getId() + "-" + dbUser.getPlayerId() + "-" + dbUser.getIP().toString());
        }

        notifyAll(message);
}
```

Abb. 9: *ServerServer* sendet IP-Adresse und UserList an alle angemeldeten Spieler

Zusammengefasst:

1. Der GameServer *(ServerServer)* erhält eine Request
2. Wenn die Anfrage erfolgreich war, antwortet der *ServerServer* mit der Nachricht OK mit der neuen PlayerID. Ansonsten NOK.
3. Der GameServer sendet die IP sowie die UserList an alle angemeldeten Spieler.
4. **Alle Server pro Spieler sind so auf dem aktuellen Stand.**

Abmelden:

Vom Spielerzeuger:

- Das Spiel wird beendet und an alle angemeldeten Spieler eine Meldung gesendet.

Vom einem Client:

- Das Spiel läuft weiter -> Spieler sowie die Punkte werden entfernt.

Zusammengefasst:

1. Der GameServer *(ServerServer)* erhält einen Request
2. Die UserList auf dem GameServer wird aktualisiert
3. Alle ClientServer *(ServerClient)* erhalten einen Request betreffend dem Update der IP-Listener's womit überall der entsprechende Spieler gelöscht werden kann.
4. Nachdem alle Clients up to date sind kann der Client die Server herunterfahren und das Spiel schliessen.

Dieses konnten wir leider nicht mehr umsetzen, die Voraussetzungen sind aber vorhanden.

5.5.2 Spielbeginn/Spielende (Multiplayer)

1. Ein Spiel ist ein Level. Sobald alle „Coins" weg sind, ist das Spiel zu Ende.
2. Spieler verlässt das Spiel:
    a. Server (Spielerzeuger): Message an GameServer --> Spielende
    b. Client (Spieleinsteiger): Message an GameServer --> Playerlist update, Spielende für den jeweiligen Client, welcher das Spiel frühzeitig beenden will.

5.6 Deklaration

Um den Status unter 5.3 zu erreichen, wird das ganze folgendermassen gehandhabt. Wir haben die Klasse Message im Package network so definiert, dass immer diese verwendet werden kann zum Schicken der Nachrichten. Sie enthält folgende Informationen:

- **Typ**

  Der Typ ist in 9 Variationen für die Datenübermittlung eingeteilt. Das heisst, pro Status 1 – 9 wird je 1 Byte übertragen, welche für das Handling des Netzwerkes, Zustandsmeldungen sowie der Spielgenerierung für die Zustandsbeschreibung verwendet wird. Anbei die verschiedenen Deklarationen:

  1. *CHECK_SERVER*
  2. *CONNECT*
  3. *DISCONNECT*
  4. *PLAYER_LIST*
  5. *GAME_STATE*
  6. *CONTROLL*
  7. *SYNCHRONISATION*
  8. *CHAT*
  9. *OK*
  10. *NOK*

- **TimeStamp**

  Der Timestamp wird dazu verwendet um sicherzustellen dass die Nachrichten in der richtigen Reihenfolge verarbeitet werden. So können Nachrichten, die zu viel Zeit gebraucht haben verworfen werden damit das Spiel nicht auf einen alten Spielstand gesetzt wird.

- **Parameter**

  Die Parameter sind je nach **Typ** unterschiedlich. So enthält zum Beispiel eine CONNECT-Anfrage folgende Parameter:

  1. **UserId**

     Die UserID ist die ID des angemeldeten Users aus der Datenbank.

  2. **IP-Adresse**

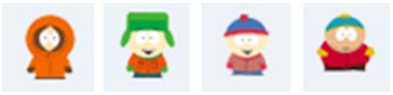
     Um die entsprechende IP-Adresse für das Netzwerk zu ermitteln, wenn sich ein neuer Spieler anmelden will; er meldet sich automatisch bei der Anmeldung unter www.whatismyipaddress.com an. Die ermittelte IP wird in der Datenbank beim entsprechenden User hinterlegt und kann dann jeweils bei der Anmeldung verwendet werden. So ist das Problem (externe IP-Adresse) sehr einfach gelöst und es entstehen keine weiteren Komplikationen diesbezüglich.

5.6.1 Beispiel einer CONNECT-Anfrage und der Bestätigung

Message-String: 2;TIMESTAMP;2;147.87.143.162;

Typ	**UserId**	**IP-Adresse**
1 = CONNECT	*2 = Thomas Häni*	147.87.143.162 = IP vom PC wo der User jetzt angemeldet ist (hier: Thomas Häni)

Message-String: 9;TIMESTAMP;1;0;

Typ	**PlayerId**	**Charakter**
9 = OK	1 = Steuerung wird für zweiten Spieler erzeugt.	0 = In dem Fall ist dies der erste Charakter 

Oder: 10;TIMESTAMP;Fehlermeldung;

Typ	**Fehlermeldung**
10 = NOK	Fehlermeldung wieso Anmeldung fehlgeschlagen ist.

6 Synchronisation des GameState

6.1 Synchronisation des Network

Es wurde schnell bemerkt, dass das Spiel falsch aufgebaut wurde, um die Synchronisation des Netzwerkes sauber zu implementieren. Folgendes wurde dabei neu gehandelt.

1. Package network
2. Kompletter Aufbau des Levels neu überarbeitet.
3. Server können einen grossen Einfluss auf die Funktionalität der Network-Synchronisation haben, deshalb wurde die folgende Lösung unter Punkt 5.2 „Klassendiagramm Netzwerkumgebung“ implementiert.

Leider hatten wir schon viel Implementiert und wir hatte nicht mehr genügend Zeit das ganze so umzustrukturieren, damit es besser umgesetzt werden konnte. Wir haben uns zwar im vorherein Gedanken darüber gemacht, leider aber haben wir aber gewisse Umstände falsch überlegt. Da dieses Projekt ja als Lerneffekt gelten soll, haben wir dann versucht das Beste daraus zu machen und an der bestehenden Lösung weiterzuarbeiten.

6.2 Implementation TCP-und UDP-Protokoll

Um den bestmöglichsten Nutzen dieser Netzwerkprotokolle für uns zu erzielen, haben wir uns dafür entschieden, beide der obengenannten Protokolle zu verwenden. Unsere Entscheidungskriterien waren die folgenden:

Events mit Wichtigkeit -> TCP	**Events mit Geschwindigkeit -> UDP**
Spielpause	Positionen der Pacman’s sowie der Geister
Spielstart	Kompletter Spielstatus (Coins, Rangliste, Levelabgleich)
Steuerung (links, rechts, rauf, runter)	
Anmeldung (im Netzwerk)	

1. Bei den Events mit Wichtigkeit ist sich darauf zu verlassen, dass die gesendeten Pakete beim Gegenüber auch ankommen. Deshalb haben wir für diese Teile der Synchronisation das Protokoll TCP gewählt.
2. Bei den Events mit Geschwindigkeit ist es wichtig, dass man nichts merken sollte, wenn Synchronisiert wird. Bei diesem Protokoll kann nicht garantiert werden, dass die gesendeten Protokolle auch ankommen oder in der gesendeten Reihenfolge ankommen. Es gehört daher in die Gruppe der verbindungslosen, nicht-zuverlässigen Übertragungsdienste.

6.2.1 Protokollfamilien/Programmausschnitt TCP- und UDP

TCP (Transmission Control Protocol)					
Familie:	Internetprotokollfamilie				
Einsatzgebiet:	Zuverlässiger bidirektionaler Datentransport				
TCP im TCP/IP-Protokollstapel:					
Anwendung	HTTP	SMTP	...		
Transport	TCP				
Internet	IP (IPv4, IPv6)				
Netzzugang	Ethernet	Token Bus	Token Ring	FDDI	...

Abb. 10: TCP-Protokoll

UDP (User Datagram Protocol)					
Familie:	Internetprotokollfamilie				
Einsatzgebiet:	Verbindungslose Übertragung von Daten über das Internet				
UDP im TCP/IP-Protokollstapel:					
Anwendung	DNS	DHCP	...		
Transport	UDP				
Internet	IP (IPv4, IPv6)				
Netzzugang	Ethernet	Token Bus	Token Ring	FDDI	...

Abb. 11: UDP-Protokoll

```
        public final Message sendMessage(Message message, String ip, int port) {
                Message answer = new Message(Message.Type.OK);
                Socket sock = null;
                BufferedReader in = null;
                PrintWriter out = null;

                try {
                        sock = new Socket(ip, port);
                        in = new BufferedReader(new InputStreamReader(sock.getInputStream()));
                        out = new PrintWriter(sock.getOutputStream(), true);

                        out.println(message);

                        answer = new Message(in.readLine());
                        out.close();
                        in.close();
                        sock.close();
                } catch (IOException ex) {
                        System.err.println(ex);
                }

                return answer;
        }
}
```

Abb. 12: *ServerTCP* sendMessage() [Protokollübertragung]

```
        public final Message sendMessage(Message message, String ip, int port) {
                try {
                        InetAddress ia = InetAddress.getByName(ip.toString());
                        byte[] data = message.getBytes();
                        DatagramPacket packet = new DatagramPacket(data, data.length, ia, port);
                        DatagramSocket toSocket = new DatagramSocket();
                        toSocket.send(packet);

                        return Message.OK;
                } catch (UnknownHostException ex) {
                        Logger.getLogger(ServerServer.class.getName()).log(Level.SEVERE, null, ex);
                        return Message.NOK;
                } catch (SocketException ex) {
                        Logger.getLogger(ServerServer.class.getName()).log(Level.SEVERE, null, ex);
                        return Message.NOK;
                } catch (IOException ex) {
                        Logger.getLogger(ServerServer.class.getName()).log(Level.SEVERE, null, ex);
                        return Message.NOK;
                }
        }
        public final Message sendMessage(byte[] message, String ip, int port) {
                try {
                        InetAddress ia = InetAddress.getByName(ip.toString());
                        DatagramPacket packet = new DatagramPacket(message, message.length, ia, port);
                        DatagramSocket toSocket = new DatagramSocket();
                        toSocket.send(packet);

                        return Message.OK;
                } catch (UnknownHostException ex) {
                        Logger.getLogger(ServerServer.class.getName()).log(Level.SEVERE, null, ex);
                        return Message.NOK;
                } catch (SocketException ex) {
                        Logger.getLogger(ServerServer.class.getName()).log(Level.SEVERE, null, ex);
                        return Message.NOK;
                } catch (IOException ex) {
                        Logger.getLogger(ServerServer.class.getName()).log(Level.SEVERE, null, ex);
                        return Message.NOK;
                }
        }
}
```

Abb. 13: *serverUDP* sendMessage() [Protokollübertragung]

6.3 Synchronisation Spiel

Um die Absichten zu demonstrieren, wie sich unser Spiel am besten synchronisieren lässt, zeigen wir anhand der nachfolgenden Schritte:

1. Nur die Key Event werden gesendet (Spielsteuerung):
    a. **Erkenntnis:** Die Spielzustände waren komplett unterschiedlich
        i. **--> KeyEvents**
2. Nur die Key Events und die Positionen werden gesendet
    a. **Erkenntnis 1:** die Geister sind sehr unterschiedlich
    b. **Erkenntnis 2:** sprunghafter Spielabgleich
        i. **--> KeyEvents + Position**
3. UDP-Server
    a. **Erkenntnis 1:** Synchronisation pro Frame (Ansonsten merkt man wenn ein PC schneller läuft)

b. **Erkenntnis 2:** UDP ist schneller, dafür unzuverlässiger als TCP
c. **Erkenntnis 3:** Der Abgleich der Geister erfolgt ohne grosse Verzögerung (flüssiger Ablauf).
d. **Erkenntnis 4:** die Pakete kommen zum Teil falsch an
    i. **-->** **KeyEvents + Position + Direction**

4. Timestamp-System
    a. Wird gebraucht, weil:
        i. Die Zustände sind nicht immer eindeutig
        ii. Verschiedene Punkte
        iii. Wer „frisst“ zuerst
            1. **--> KeyEvents + Position + Direction + Time**

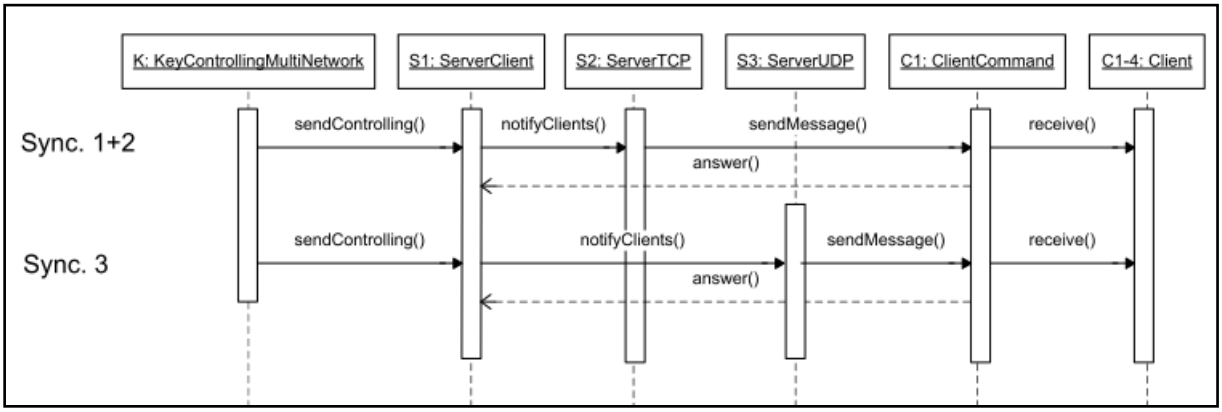


Abb. 14: *Zusammenhang der Funktionsweise im Programm*

7 Fazit/Eindrücke

- Das Projekt hatte nur positive Lerneffekte und somit keine Nachteile
- Wichtiges Wissen wurde angeeignet für ein nächstes Projekt, was man von Anfang an anders/besser machen kann. Oder wie man das Vorgehen eines solchen Projektes noch besser kalkulieren und planen kann.
- Wichtige Erkenntnisse wie in Java eine Datenbankanbindung funktioniert, wie man Daten ein- und auslesen kann und wie man eine User-Authentifizierung realisiert.
- Wichtige Erkenntnisse wie eine Server/Client Architektur sowie eine Peer2Peer Lösung funktioniert und wie die beiden Seiten zusammen harmonieren.
- Wichtige Erkenntnisse was es für verschiedene Netzwerkprotokolle gibt, wie diese funktionieren und welches die Vor- und Nachteile derjenigen sind. Wie zum Beispiel; UDP ist schneller als TCP. Dafür ist TCP zuverlässiger. Diesen Nutzen konnten wir in unserem Projekt bei der Synchronisation des GameStates optimal zu unseren Gunsten nutzen.
- Wichtige Erkenntnisse wie die Synchronisation über ein Netzwerk funktioniert, wie man dabei am besten Schritt für Schritt sich dem Endergebnis nähern kann, wie unter Punkt 6.3 beschrieben und so den stetigen Fortschritt erkennt.
- Wir hatten festgestellt dass es mit den verschiedenen benötigten Threads und den verschiedenen Clients und Servern oftmals sehr schwierig und kompliziert war, gewisse Fehler oder Fehlverhalten nachvollziehen und finden zu können.
- Wir haben auch gelernt, wie wir optimal Bilder aus dem JAR laden um so das Spiel schnell genug laufen zu lassen um so den Spielstatus schnell genug auszugeben.

Für uns war dieses Projekt eine super gute Lernerfahrung. Wir können nur positive Nutzen daraus ziehen und würden es sofort wieder machen, wenn auch vielleicht von Anfang an bereits im grossen Rahmen an das Netzwerk denken. Sobald wir die Protokolle und Architekturen vorhanden hatten war es nur noch ein zusammenstellen von Nachrichten, welche über das Netzwerk geschickt wurden und ausgeführt werden. Leider konnten wir nicht die beste Lösung für ein Synchronisierungsprotokoll wählen, da wir zeitlich am Limit waren. So wäre es sinnvoll, alle Clients gleich schnell laufen zu lassen damit der Server nicht so viel den Takt ausmacht. Dies haben wir gemerkt, sobald wir den Server auf einem alten langsamen Rechner laufen liessen. Wie gesagt ist es aber zeitlich nicht mehr möglich gewesen, alles neu zu überdenken und anzupassen und wir haben das Beste aus unserem Projekt gemacht. Das Spiel funktioniert für uns erfreulich gut und schnell und wir hatten viel Spass diese verschiedenen Themen zu erlernen, auszukunden und natürlich auszutesten. Wir hoffen dass der Spass auch rüberkommt wenn man sich im Spiel befindet und dieses spielt.

8 Abbildungsverzeichnis