



Projekt: JBombberman  
Software Architektur

Pascal Kistler  
Silvan Adrian  
Fabian Binna

## 1 Änderungshistorie

Datum	Version	Änderung	Autor
01.04.15	1.00	Erstellung des Dokuments	Gruppe
04.04.15	1.01	Logische Architektur	Fabian Bin
05.04.15	1.02	Logische Architektur Client	Fabian Bin
05.04.15	1.03	Ziele und Einschränkungen	Silvan Adri
06.04.15	1.04	Logische Architektur Server	Fabian Bin
06.04.15	1.05	Zustandsdiagramm Workflows	Fabian Bin
06.04.15	1.06	Systemübersicht Grafik und Beschreibung	Silvan Adri
06.04.15	1.07	Schnittstellen	Fabian Bin
06.04.15	1.08	Grössen und Leistung + Externes Design	Silvan Adri
07.04.15	1.09	Datenübertragung	Fabian Bin
07.04.15	1.10	Prozesse und Threads	Fabian Bin
07.04.15	1.11	Korrekturen	Fabian Bin
07.04.15	1.12	Korrekturen	Silvan Adri
16.04.15	1.13	Sequenz Diagramm Action	Fabian Bin
05.05.15	1.14	Anpassungen	Fabian Bin
05.05.15	1.15	Anpassungen an Client-, ServerController und LobbyCommunication	Pascal Kist
06.05.15	1.16	Externes Design erweitert + Grössen Leistung angepasst	

## Inhaltsverzeichnis

<b>1</b>	<b>Änderungshistorie</b>	<b>2</b>
<b>2</b>	<b>Einführung</b>	<b>5</b>
2.1	Zweck . . . . .	5
2.2	Gültigkeitsbereich . . . . .	5
2.3	Referenzen . . . . .	5
2.4	Übersicht . . . . .	5
2.4.1	Glossar . . . . .	5
<b>3</b>	<b>Systemübersicht</b>	<b>5</b>
<b>4</b>	<b>Architektonische Ziele &amp; Einschränkungen</b>	<b>6</b>
4.1	Ziele . . . . .	6
4.2	Einschränkungen . . . . .	6
<b>5</b>	<b>Logische Architektur: Applikationsübergreifend</b>	<b>7</b>
5.1	Time/time . . . . .	8
5.1.1	Klassenstruktur . . . . .	8
5.2	Domain/game . . . . .	9
5.2.1	Klassenstruktur . . . . .	9
5.3	Utilities/utls . . . . .	11
5.3.1	Klassenstruktur . . . . .	11
5.4	Wichtige Abläufe . . . . .	13
<b>6</b>	<b>Logische Architektur: Client</b>	<b>14</b>
6.1	Schnittstellen . . . . .	14
6.2	Presentation/view . . . . .	15
6.2.1	Klassenstruktur . . . . .	15
6.3	Workflow/application.client . . . . .	16
6.3.1	Klassenstruktur . . . . .	16
6.4	Domain/game.client . . . . .	18
6.4.1	Klassenstruktur . . . . .	18
6.5	Network/network.client . . . . .	20
6.5.1	Klassenstruktur . . . . .	20
6.6	Wichtige Abläufe . . . . .	21
6.6.1	Zustandsdiagramm Client Workflow . . . . .	21
6.6.2	Sequenzdiagramm GameLoop Client . . . . .	22
<b>7</b>	<b>Logische Architektur: Server</b>	<b>23</b>
7.1	Schnittstellen . . . . .	23
7.2	Workflow/application.server . . . . .	24
7.2.1	Klassenstruktur . . . . .	24

7.3	Domain/game.server . . . . .	25
7.3.1	Klassenstruktur . . . . .	25
7.4	Network/network.server . . . . .	27
7.4.1	Klassenstruktur . . . . .	27
7.5	Wichtige Abläufe . . . . .	28
7.5.1	Zustandsdiagramm Server Workflow . . . . .	28
<b>8</b>	<b>Prozesse und Threads</b>	<b>29</b>
8.1	Dispatcher Thread . . . . .	29
<b>9</b>	<b>Deployment</b>	<b>30</b>
<b>10</b>	<b>Datenübertragung</b>	<b>31</b>
10.1	RabbitMQ Broker . . . . .	31
10.2	Actions . . . . .	32
10.3	ControllerCommunication . . . . .	34
10.3.1	Client to Server . . . . .	34
10.3.2	Server to Client . . . . .	34
<b>11</b>	<b>Externes Design</b>	<b>36</b>
11.1	StartupFrame . . . . .	36
11.2	LobbyFrame . . . . .	36
11.2.1	Timer . . . . .	37
11.2.2	Fehlermeldungen . . . . .	37
11.3	GameFrame . . . . .	38
<b>12</b>	<b>Größen und Leistung</b>	<b>38</b>

## 2 Einführung

### 2.1 Zweck

Dieses Dokument beschreibt die Software Architektur für das Projekt JBombberman.

### 2.2 Gültigkeitsbereich

Dieses Dokument ist während des ganzen Projekts gültig und wird laufend aktualisiert.

### 2.3 Referenzen

### 2.4 Übersicht

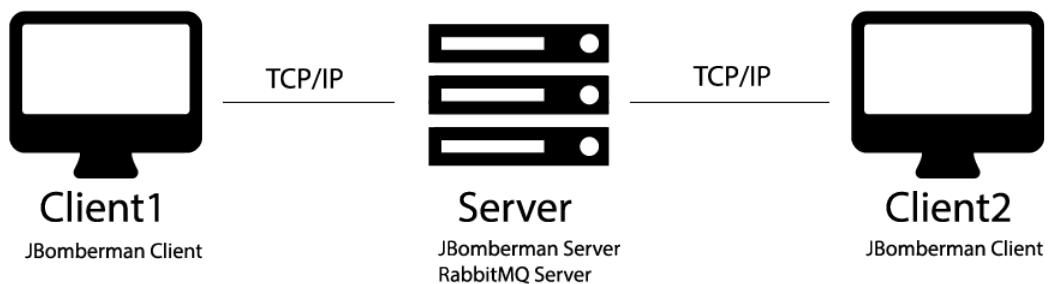
Die LogischeSicht wurde in drei Teile unterteilt: Applikationsübergreifend, Client und Server. Die Schnittstellen werden anhand einer kompletten Packageübersicht gezeigt. Im Kapitel Datenübertragung wird auf die RabbitMQ-Technologie eingegangen.

#### 2.4.1 Glossar

Siehe Dokument Glossar.pdf

## 3 Systemübersicht

JBombberman kann nur als Mehrspielerspiel gespielt werden, daher wird immer ein verfügbarer Server benötigt + muss ein RabbitMQ Server verfügbar sein. Zudem braucht es mindestens 2 Clients, damit ein Spiel überhaupt gespielt werden kann. Zur Veranschaulichung der Austausch zwischen 2 Clients und 1 Server:



## **4 Architektonische Ziele & Einschränkungen**

### **4.1 Ziele**

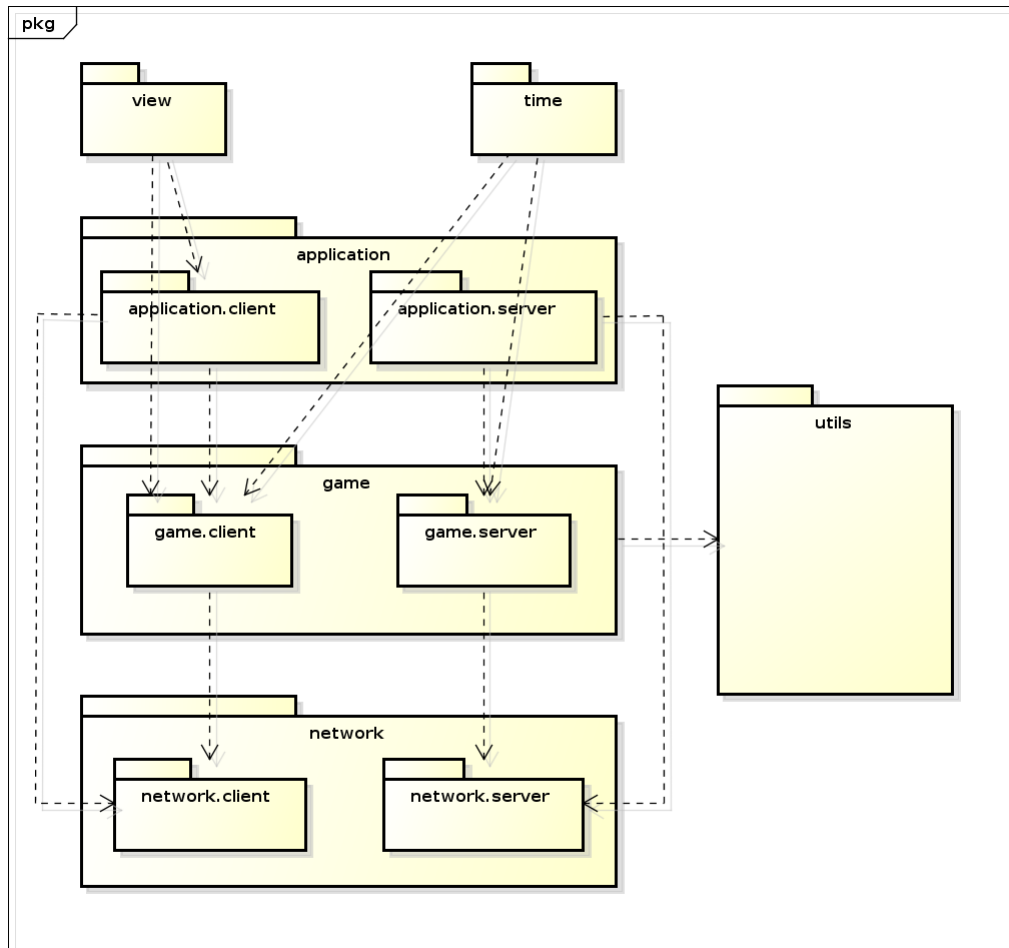
- Die Spielumgebungen müssen mit allen Clients synchronisiert werden (über den dedizierten Server)
- Möglichkeit weitere PowerUp's einzubauen.
- Austausch zwischen den Clients und dem Server wird über JMS umgesetzt.

### **4.2 Einschränkungen**

- Es wird keinen Live Server geben, jeder der das Spiel spielen will muss einen eigenen Server starten.
- Die Clients können nur zum Server Verbindung aufnehmen, wenn diese die IP des Servers kennen.

## 5 Logische Architektur: Applikationsübergreifend

Dieses Package Diagramm zeigt sowohl Client, als auch Server. Client und Server sind zwei eigenständige Applikationen, die getrennt ausgeführt werden. Sie verwenden jedoch teilweise die gleichen Packages.



powered by Astah

## 5.1 Time/time

Im Package time befindet sich eine Klasse, die für das timing der GameLoops sorgt.

### 5.1.1 Klassenstruktur



powered by Astah

#### Timer

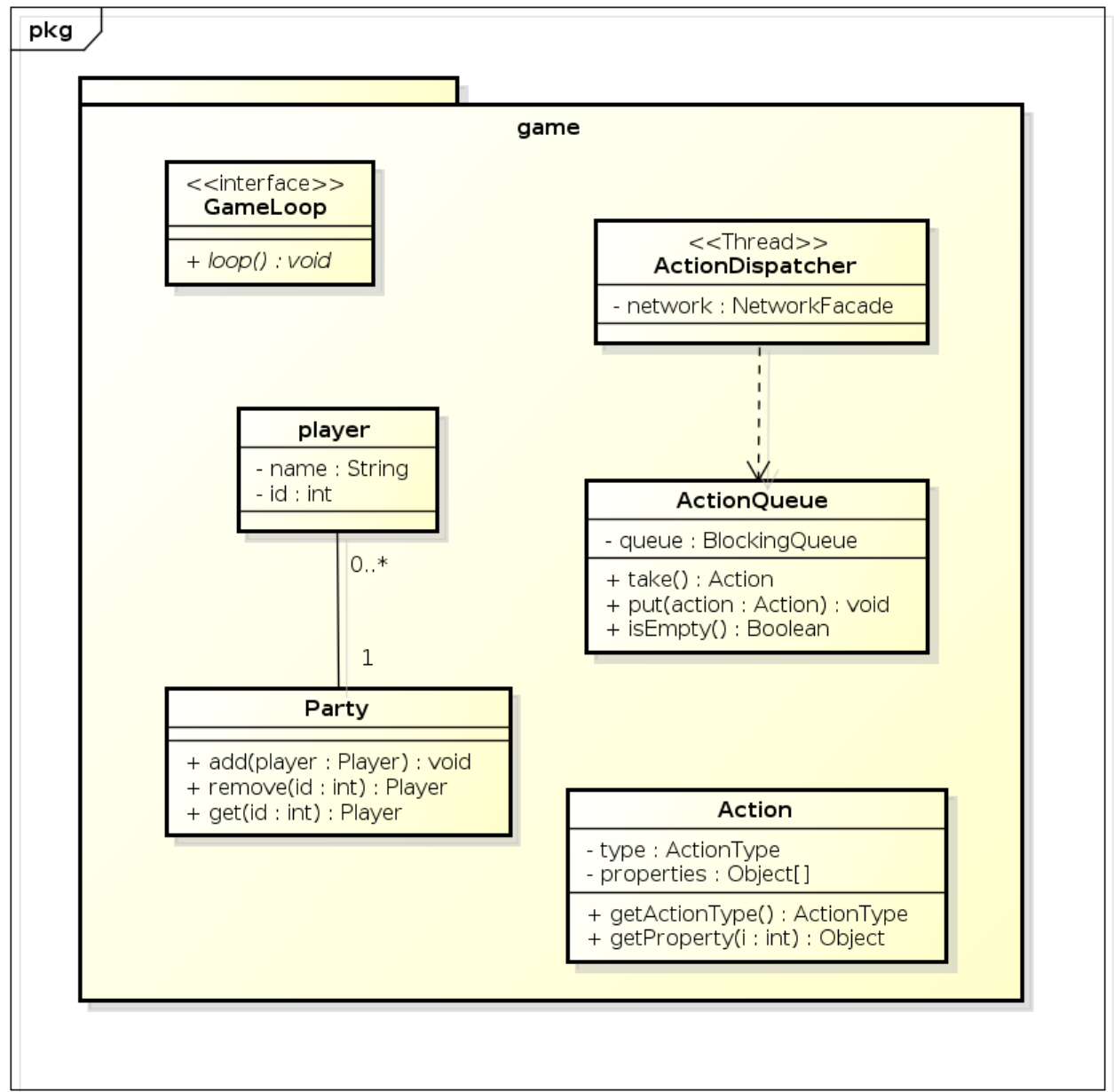
Der Timer besitzt eine Referenz auf einen GameLoop, bei dem er regelmässig die Methode loop() aufruft.



## 5.2 Domain/game

Im Package game befinden sich Klassen und Interfaces, die von Server und Client gemeinsam genutzt werden.

### 5.2.1 Klassenstruktur



powered by Astah

**Action**

Die Action Klasse wird über das Netzwerk zwischen Server und Client versendet und enthält Statusnachrichten, sowie Aktualisierungsdaten für die Objekte.

**ActionQueue**

Die ActionQueue speichert die Actions. Die Actions werden bei jedem loop aus der Queue entfernt und verarbeitet. Die ActionQueue muss Threadsafe sein, da der ActionDispatcher die Queue füllt.

**ActionDispatcher**

Der ActionDispatcher ist ein Thread und ist ausschliesslich damit beschäftigt auf eingehende Messages zu warten und diese als Action in der Queue einzureihen.

**Player**

Der Player speichert die Daten eines Spielers. Die Daten werden für die Zuweisung von Actions und das Darstellen des Scoreboard benötigt.

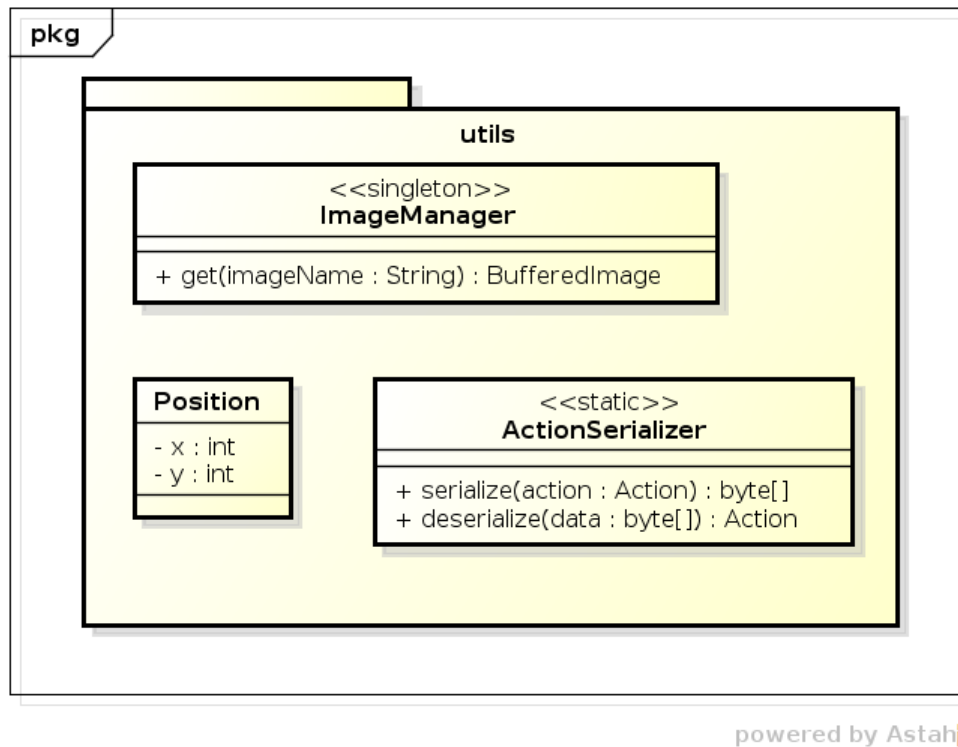
**Party**

Die Party beinhaltet alle Player eines Spiels. Ein Spiel kann nur mittels einer Party instanziiert werden.

### 5.3 Utilities/utils

Im Package utils befinden sich Hilfsklassen.

#### 5.3.1 Klassenstruktur



#### ImageManager

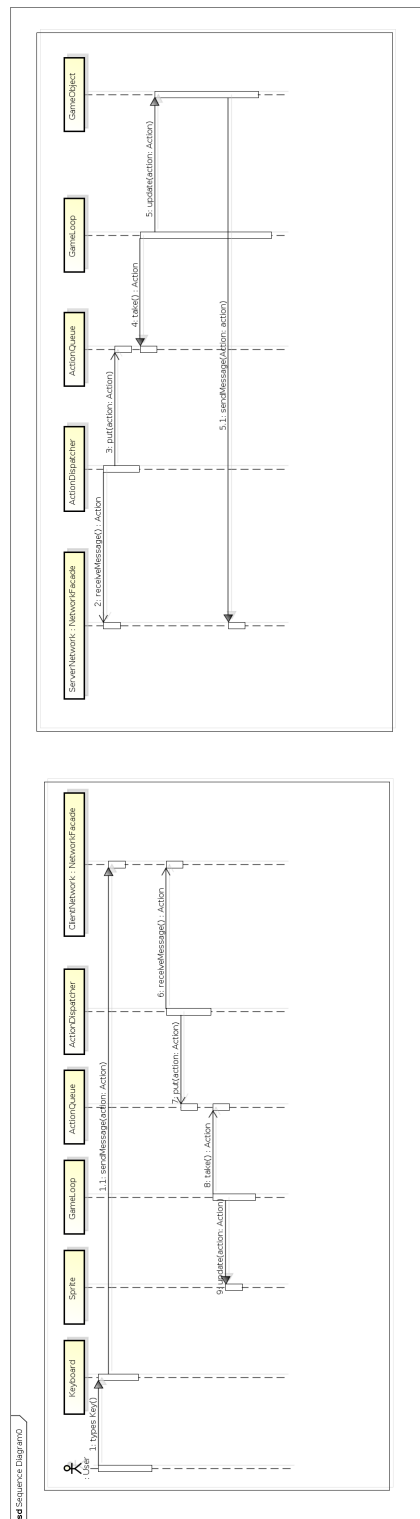
Der ImageManager lädt die Bilder. Jedes Sprite holt sich von da die Bilddaten.

#### ActionSerializer

Die Methoden des ActionSerializer sind statisch und werden benötigt um die Actions für das Netzwerk zu serialisieren bzw. deserialisieren.

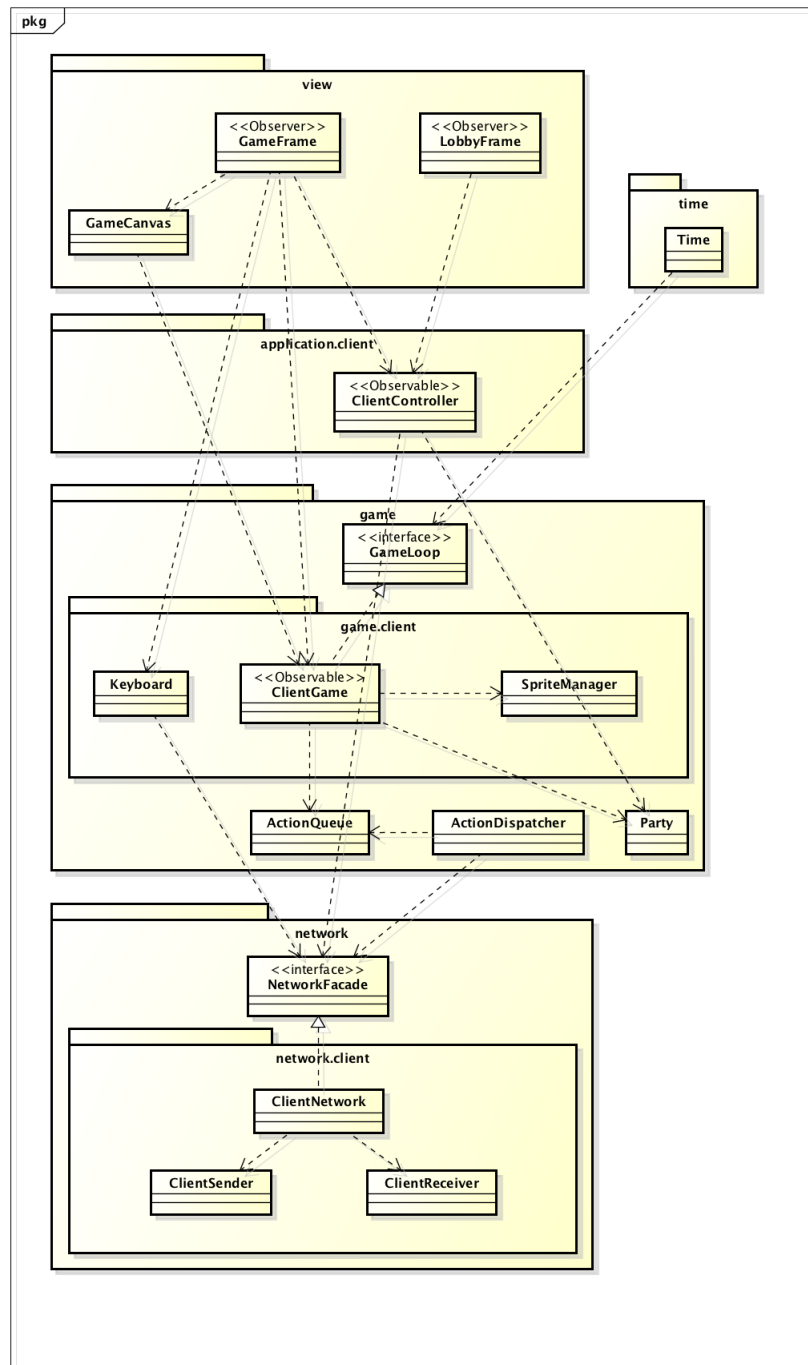


## 5.4 Wichtige Abläufe



## 6 Logische Architektur: Client

### 6.1 Schnittstellen

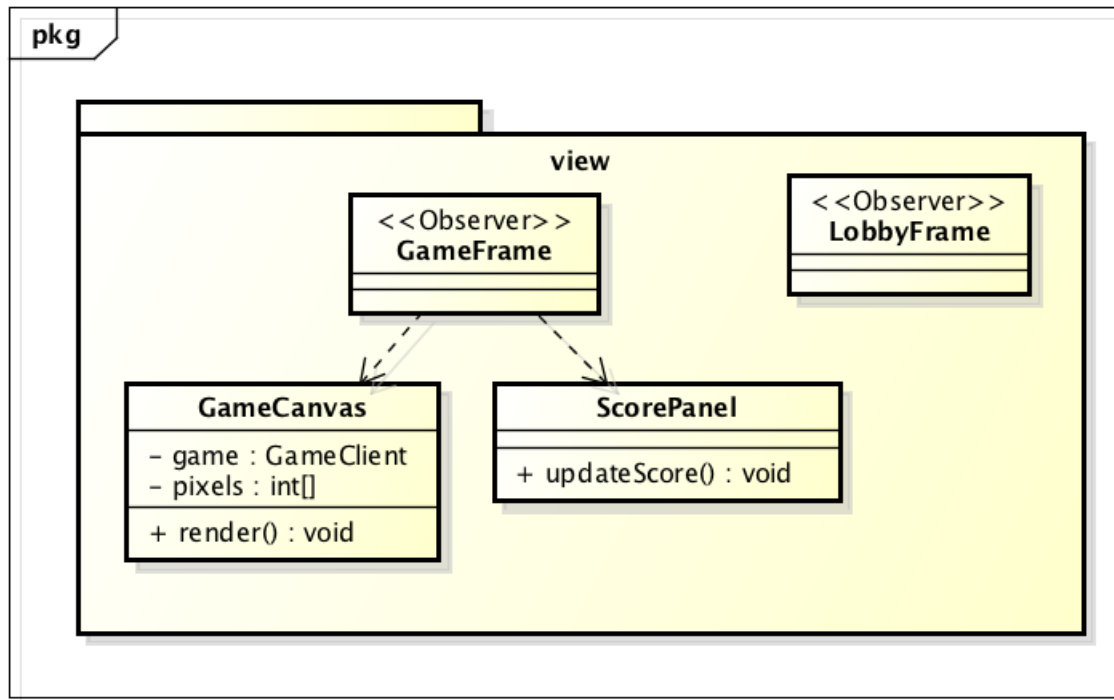


powered by Astah

## 6.2 Presentation/view

Im Package view befinden sich Frames und Canvas, die für die Presentation des Clients notwendig sind.

### 6.2.1 Klassenstruktur



powered by Astah

#### GameFrame

Der GameFrame ist ein Observer und delegiert die Notifies an das zugehörige Panel, die sich dann selber auf den neusten Stand bringen.

#### GameCanvas

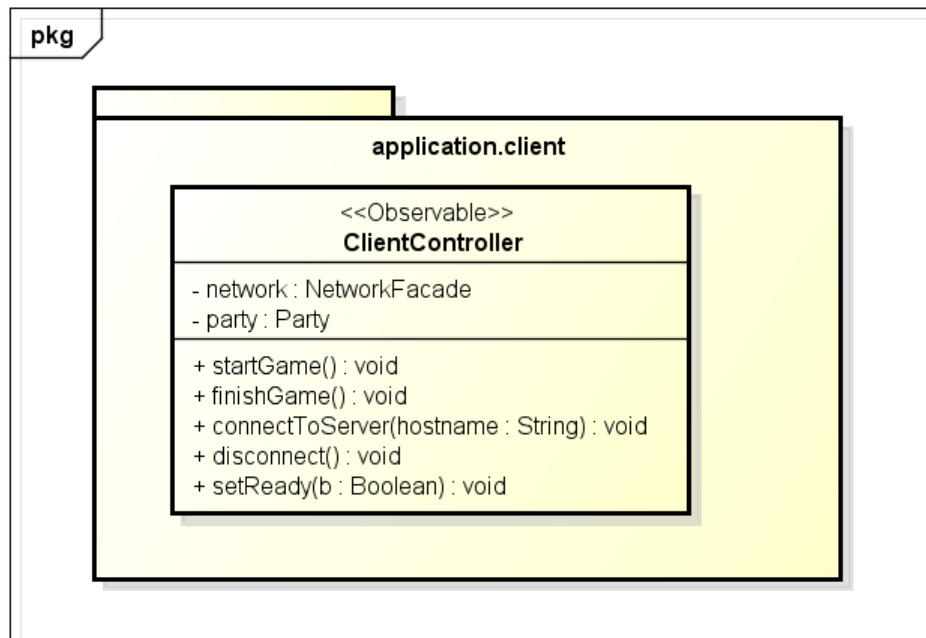
Der GameCanvas kümmert sich nur um das Rendering, also das Zeichnen der Szene. Dabei delegiert er jedoch nur die pixels[] an alle Sprites, welche sich dann eigenständig zeichnen. Der GameCanvas kümmert sich dann um das performante Buffering.

Methode	Beschreibung
render():void	Kümmert sich um die BufferStrategy und delegiert das zeichnen der Sprites direkt an die Sprites selbst.

### 6.3 Workflow/application.client

Im Package application.client befindet sich der workflow des Clients. Er kontrolliert unter anderem wann der LobbyFrame und der GameFrame sichtbar ist.

#### 6.3.1 Klassenstruktur



powered by Astah



**ClientController**

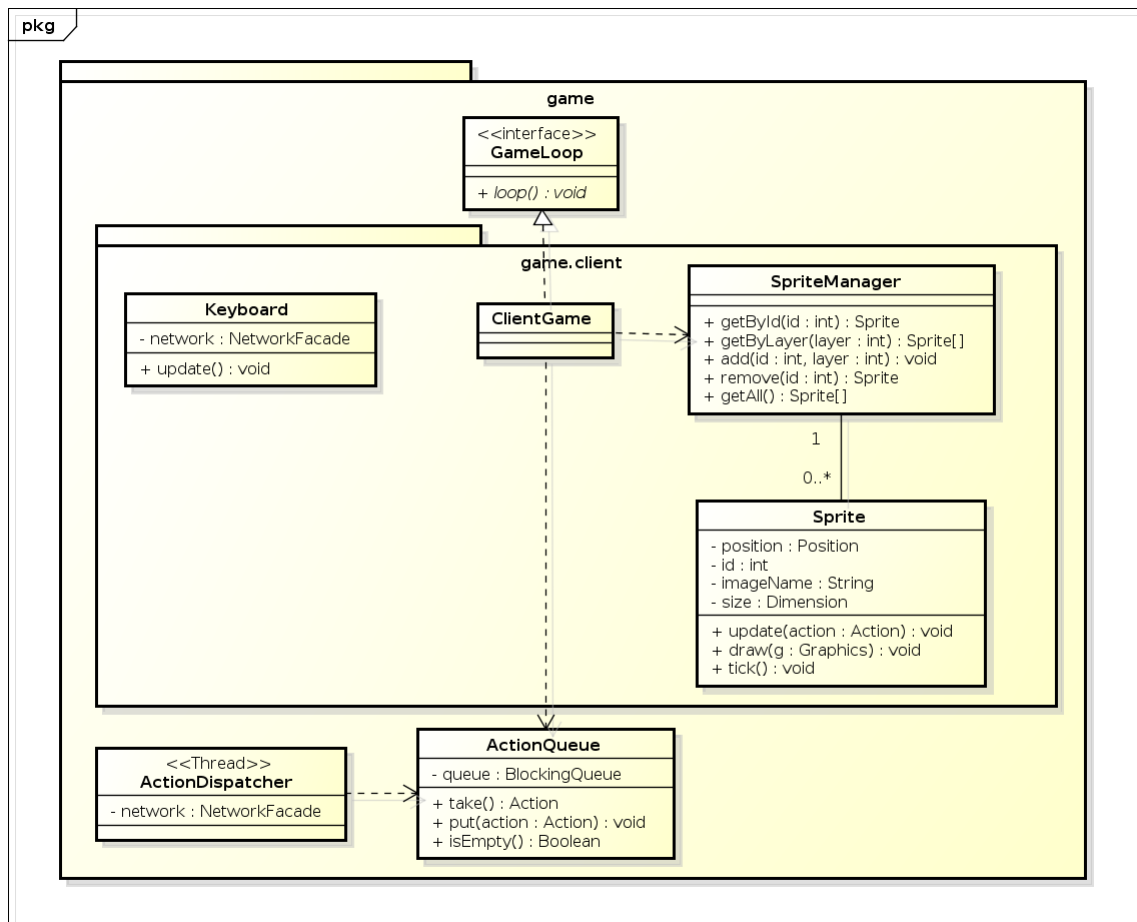
Der ClientController kontrolliert den Zustand der Client-Applikation. Er notifiziert nur den LobbyFrame, nicht aber den GameFrame.

<b>Methode</b>	<b>Beschreibung</b>
startGame():void	Erstellt eine GameFrame, welches die ClientGame Klasse instanziert und somit das Spiel startet.
finishGame():void	Informiert den LobbyFrame darüber, dass das Spiel beendet wurde.
connectToServer(hostname: String) : void	Verbindet den Client mit einem RabbitMQ Broker.
disconnect() : void	Schliesst die Verbindung mit dem RabbitMQ Broker.
setReady(b: Boolean): void	Setzt den Spieler auf bereit.

## 6.4 Domain/game.client

Im Package game.client werden die Actions vom Server interpretiert und die Sprites auf den neusten Stand gebracht. Die Sprites werden in einer Layer-Logik gespeichert, damit sie korrekt gezeichnet werden können.

### 6.4.1 Klassenstruktur



powered by Astah

#### ClientGame

Die Methode `loop` wird unter "Wichtige Abläufe" beschrieben.

#### SpriteManager

Der `SpriteManager` speichert alle Sprites in einer Schichten-Logik. Dies wird benötigt, damit die Sprites korrekt gezeichnet werden können. Zudem können die Sprites per id gefunden werden, damit das Aktualisieren der Positionen und Zustände einfacher wird. Die Methoden des `SpriteManager` sind ähnlich wie bei normalen Datenstrukturen und werden deshalb nicht weiter erläutert.

### **Sprite**

Die Sprite-Klasse beinhaltet alle Informationen die für das Zeichnen der Spielobjekte benötigt wird.

<b>Methode</b>	<b>Beschreibung</b>
update(action : Action) : void	Interpretiert die Action und führt die nötigen Aktualisierungsschritte durch.
draw(g : Graphics) : void	Zeichnet sich selbst auf den screen. Diese Methode wird vom GameCanvas aufgerufen und liefert seine Graphics mit auf dem gezeichnet werden kann.
tick() : void	Aktualisiert das Sprite. Wird hauptsächlich für Animationen benötigt.

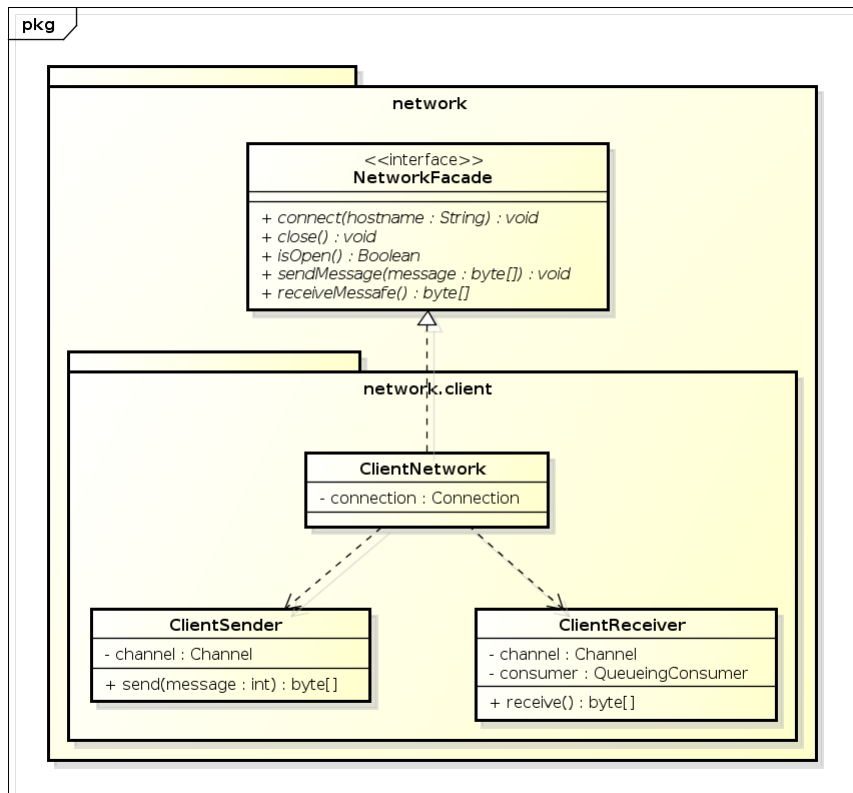
### **Keyboard**

Das Keyboard zeichnet die Tastatureingaben auf und sendet diese direkt über die Methode update an den Server.

## 6.5 Network/network.client

Der Client und der Server implementieren beide das Interface NetworkFacade. Die implementation ist jedoch grundverschieden.

### 6.5.1 Klassenstruktur



powered by Astah

#### ClientNetwork

Die Klasse ClientNetwork implementiert die NetworkFacade.

#### ClientSender

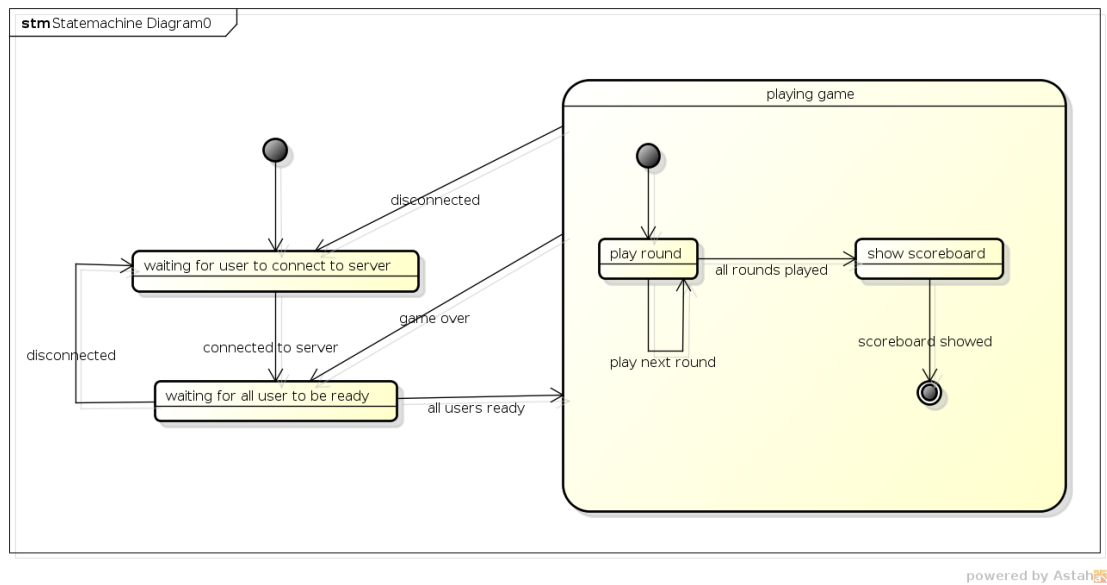
Der Client Sender sendet direkt auf eine RabbitMQ-Queue. Alle Clients senden auf die selbe Queue.

#### ClientReceiver

Der ClientReceiver holt Messages aus seiner eigenen RabbitMQ-Queue. Der Server sendet seine Updates auf jede einzelne Queue.

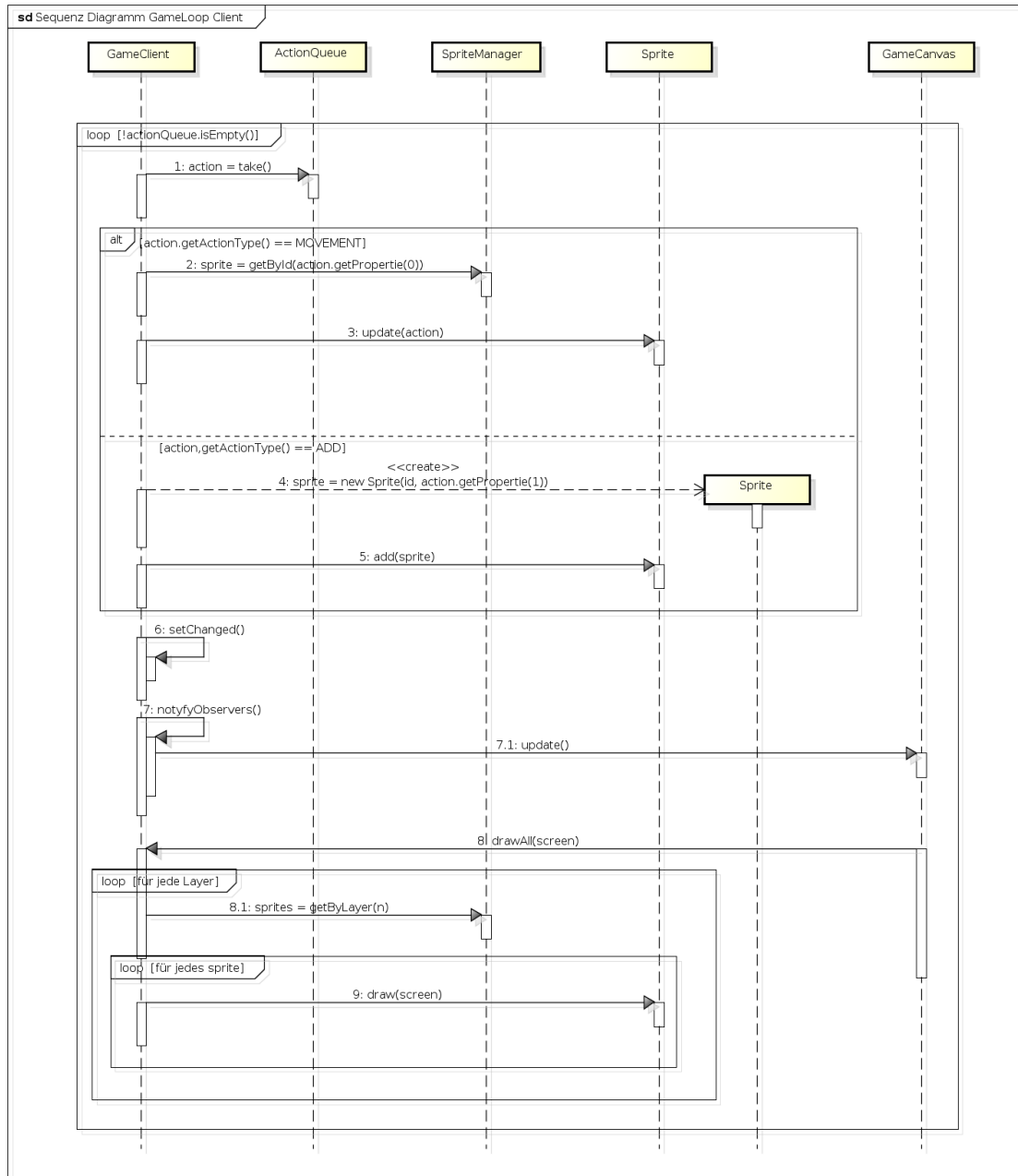
## 6.6 Wichtige Abläufe

### 6.6.1 Zustandsdiagramm Client Workflow



### 6.6.2 Sequenzdiagramm GameLoop Client

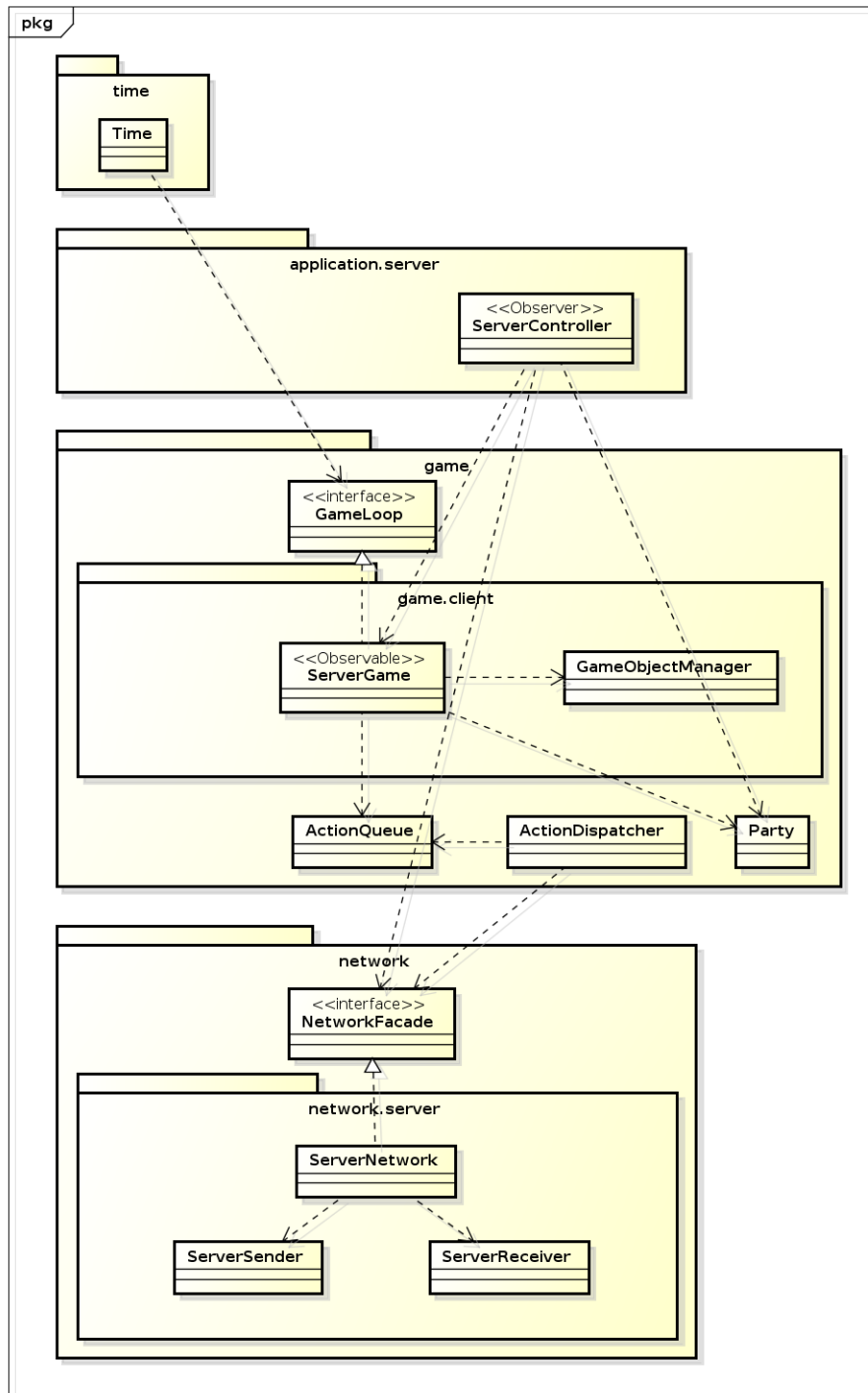
Das Sequenzdiagramm zeigt nur zwei alternative ActionTypes. Der GameLoop läuft beim Server grundsätzlich gleich ab, jedoch ohne das Rendern und anstelle der Sprites werden mit GameObjects gearbeitet.



powered by Astah

## 7 Logische Architektur: Server

## 7.1 Schnittstellen

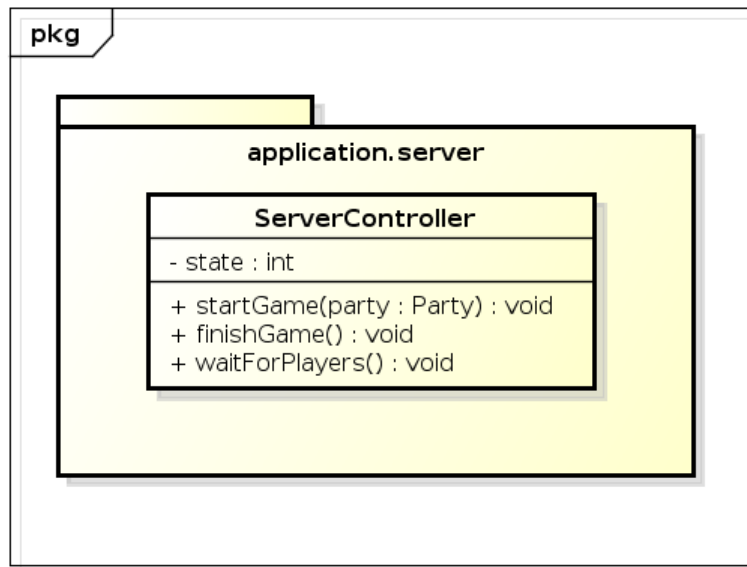


powered by Astah

## 7.2 Workflow/application.server

Der Workflow des Servers ist sehr simpel. Er wartet bis mehr als zwei Spieler verbunden und bereit sind. Darauf startet der Server dann das Spiel.

### 7.2.1 Klassenstruktur



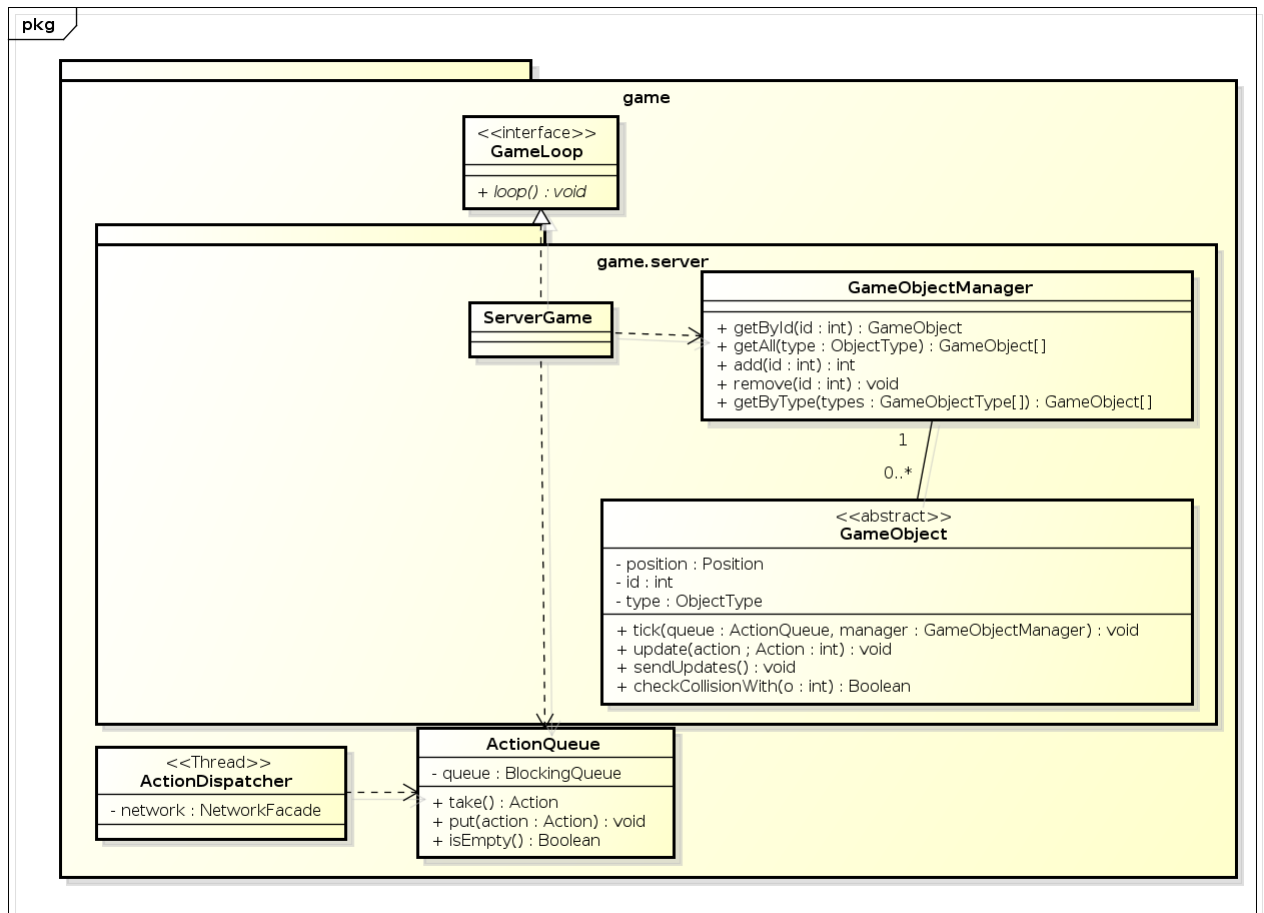
powered by Astah

Methode	Beschreibung
waitForPlayers() : void	Der Server befindet sich nach dem Starten in dieser Methode und wartet bis mehr als ein Spieler verbunden und bereit ist. Danach startet er das Spiel.
startGame(party : Party) : void	Erstellt die nötigen Klassen und startet das Spiel.
finishGame() : void	Entkoppelt die Spielrelevanten Klassen und geht in die waitForPlayers Methode zurück.



## 7.3 Domain/game.server

### 7.3.1 Klassenstruktur



powered by Astah

#### ServerGame

Die Methode `loop()` funktioniert ähnlich wie die Implementation in der `ClientGame` Klasse und wurde dort schon beschrieben.

#### GameObjectManager

Der `GameObjectManager` speichert die `GameObject` nach Typen ab. Dies ist hilfreich, wenn die Objekte aktualisiert werden müssen. Die Methoden des `GameObjectManager` sind ähnlich wie bei normalen Datenstrukturen und werden deshalb nicht weiter erläutert.

**GameObject**

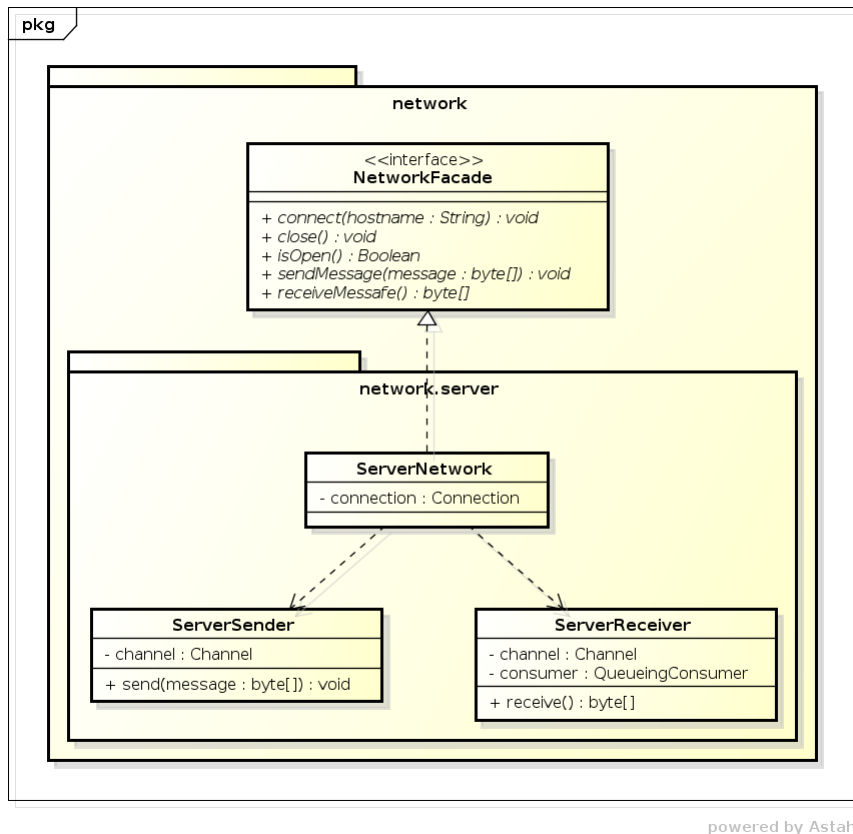
Das GameObject speichert alle nötigen Informationen um das Spielgeschehen und die Interaktionen der Gameobjects zu berechnen.

<b>Methode</b>	<b>Beschreibung</b>
tick(ActionQueue, GameObjectManager) : void	Aktualisiert Zustände und Variablen. Hauptsächlich Zeitbedingte Werte (Wann kann ich die nächste Bombe legen)
update(action : Action) : void	Interpretiert die Action und aktualisiert sich selbst.
sendUpdates() : void	Falls das Object neue Werte besitzt muss es die neuen Daten an die Clients senden.
checkCollisionWith(o : GameObject) : Boolean	Überprüft, ob das Object mit einem anderen kollidiert und führt dann die jeweiligen Korrekturen (z.B. beim Movement) oder Aktionen durch.

## 7.4 Network/network.server

Der Client und der Server implementieren beide das Interface NetworkFacade. Die implementation ist jedoch grundverschieden.

### 7.4.1 Klassenstruktur



powered by Astah

#### ServerNetwork

Die Klasse ServerNetwork implementiert die NetworkFacade.

#### ServerSender

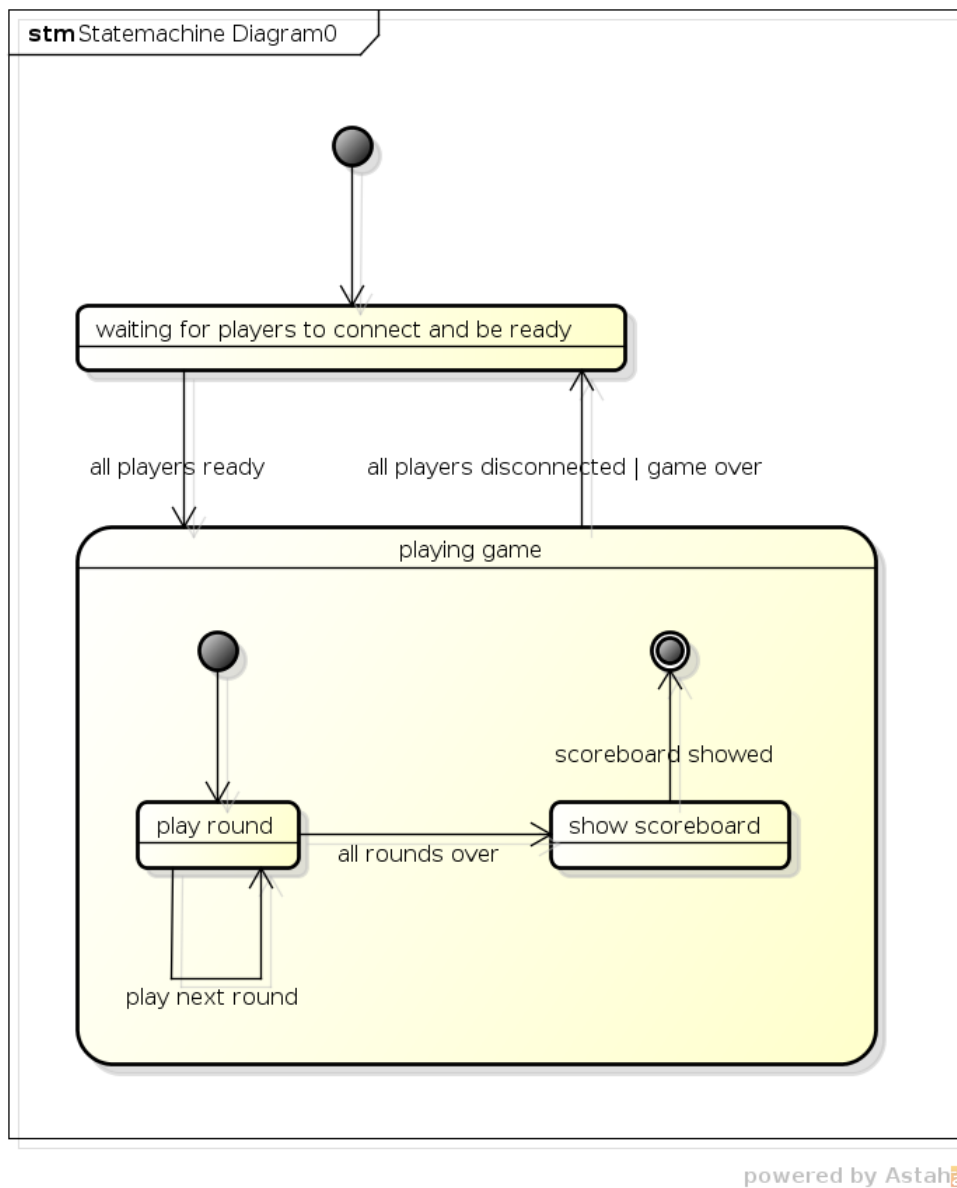
Der ServerSender sendet alle Daten die im übergeben werden direkt an alle angemeldeten Clients.

#### ServerReceiver

Der ServerReceiver holt alle Daten aus einer einzelnen RabbitMQ-Queue, auf welche alle Clients senden.

## 7.5 Wichtige Abläufe

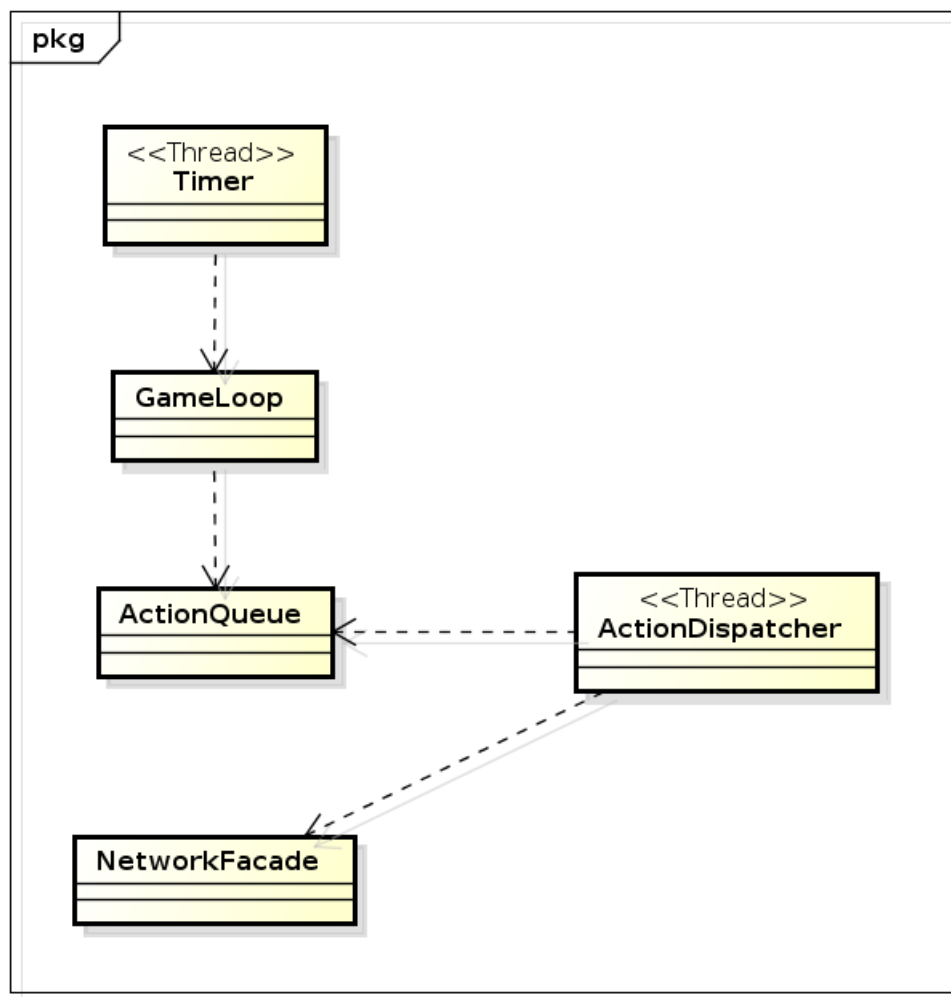
### 7.5.1 Zustandsdiagramm Server Workflow



## 8 Prozesse und Threads

### 8.1 Dispatcher Thread

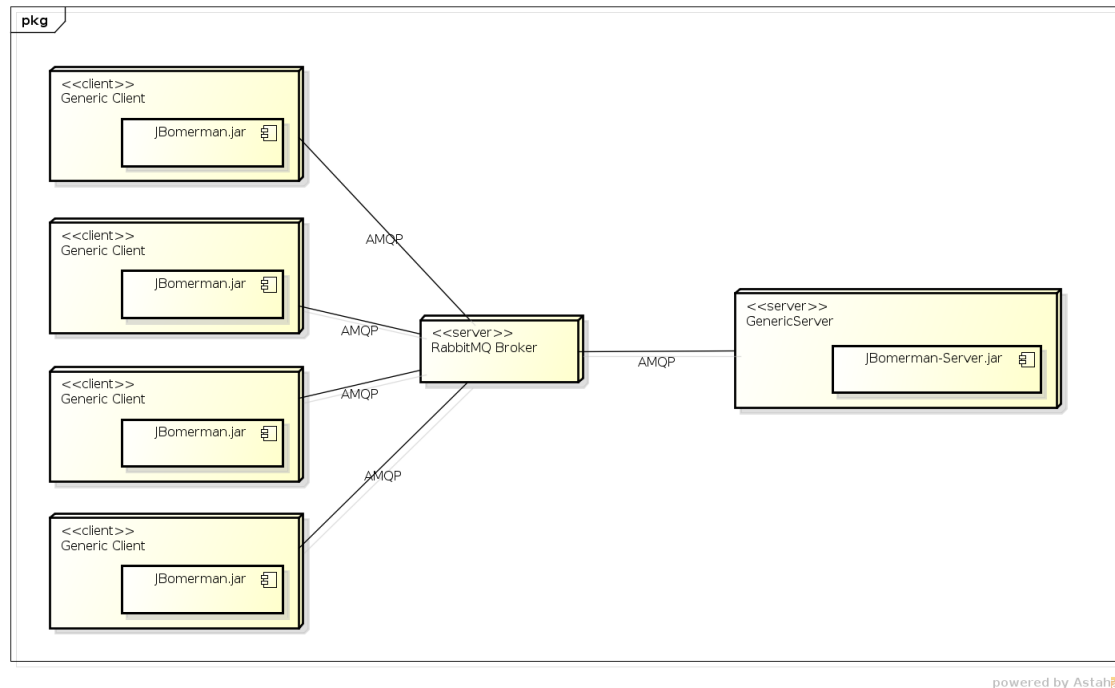
Der DispatcherThread(Klasse: ActionDispatcher) sorgt dafür, dass die Messages empfangen werden und in die ActionQueue abgelegt werden. Gleichzeitig holt der GameLoop die Action aus der Queue heraus. Die ActionQueue ist mittels einer LinkedBlockingQueue implementiert, und somit Threadsafte.



powered by Astah

## 9 Deployment

Das Messagerouting wird von einem RabbitMQ Broker übernommen, der meistens auf der gleichen Hardware wie der GameServer läuft, aber auch auf einer anderen Hardware betrieben werden kann. Die Übertragung läuft mittels AMQP (Advanced Message Queuing Protocol).

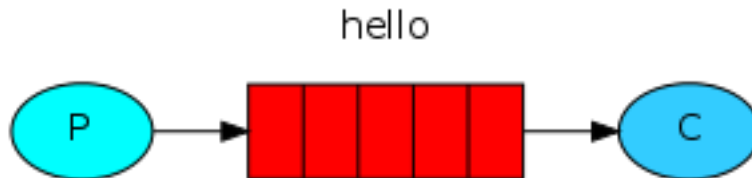


## 10 Datenübertragung

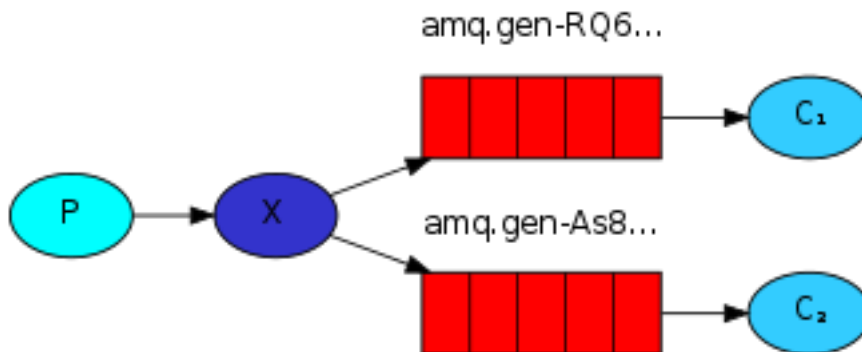
### 10.1 RabbitMQ Broker

Der RabbitMQ Broker speichert die erhaltenen Messages in Warteschlangen/Queues. Der Empfänger muss die Message also nicht sofort annehmen.

Die Clients(P) senden ihre Messages an die Queue BombermanInput(hello). Der Server(C) entnimmt dort regelmässig die Messages und kann sie anhand der IDs in den Actions identifizieren.



Die Messages die vom Server(P) an die Clients(Cx) gehen werden je in eine Client-Eigene Queue(amq.gen-XyZ...) gelegt. Dieser Vorgang übernimmt jedoch der RabbitMQ Broker(X) für uns.



## 10.2 Actions

Da die Informationen innerhalb einer Action in einem Object-Array gespeichert werden, müssen die verschiedenen ActionTypen dokumentiert werden.

### MOVEMENT

i	Klasse	Inhalt
0	int	id
1	Position	Neue Position
2	BombermanState	Animationsstatus

### CREATE\_<OBJECTNAME>

i	Klasse	Inhalt
0	int	id
1	Position	Anfangsposition

### DESTROY

i	Klasse	Inhalt
0	int	id

### PLAYER\_INPUT

i	Klasse	Inhalt
0	int	id
1	KeyCode	Taste
2	boolean	Taste gedrückt

### LOBBY\_COMMUNICATION

i	Klasse	Inhalt
0	String	Bezeichnung
1...3	Object	Properties



**Hinweis:** Weitere Actions werden während der Construction-Phase dokumentiert.

### 10.3 ControllerCommunication

Der Actiontype LOBBY\_COMMUNICATION wird für die Kommunikation zwischen Client- und ServerController verwendet. Neben der Bezeichnung werden je nach Message noch folgende Properties übermittelt.

#### 10.3.1 Client to Server

**Connect** hat keine weiteren Properties.

##### updateState

i	Klasse	Inhalt
0	int	PlayerId
1	boolean	Status (ready / not ready)

##### disconnect

i	Klasse	Inhalt
0	int	PlayerId

#### 10.3.2 Server to Client

**ServerFull** hat keine weiteren Properties.

##### lobbyList

i	Klasse	Inhalt
0	HashMap<Integer, Boolean>	PlayerStates
1	int	PlayerId

##### updateStates

i	Klasse	Inhalt
0	HashMap<Integer, Boolean>	PlayerStates

**countUpdate**

<b>i</b>	<b>Klasse</b>	<b>Inhalt</b>
----------	---------------	---------------

---

0	int	countdown
---	-----	-----------

**startGame**

<b>i</b>	<b>Klasse</b>	<b>Inhalt</b>
----------	---------------	---------------

---

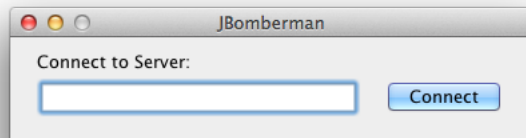
0	String[][]	Party
---	------------	-------

## 11 Externes Design

Es besteht lediglich ein GUI für den Client, der Server soll über das CLI gestartet werden.

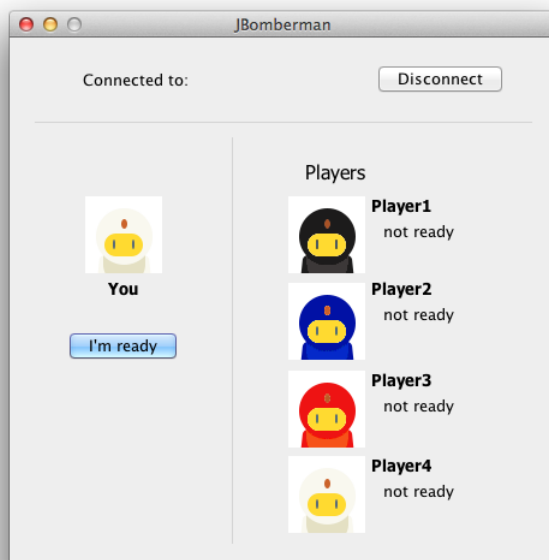
### 11.1 StartupFrame

Als StartupFrame kommt der Connect zum Server,dabei wird die IP oder Name des Servers benötigt um sich verbinden zu können. Der Connect Button ist hierbei der Standardbutton.



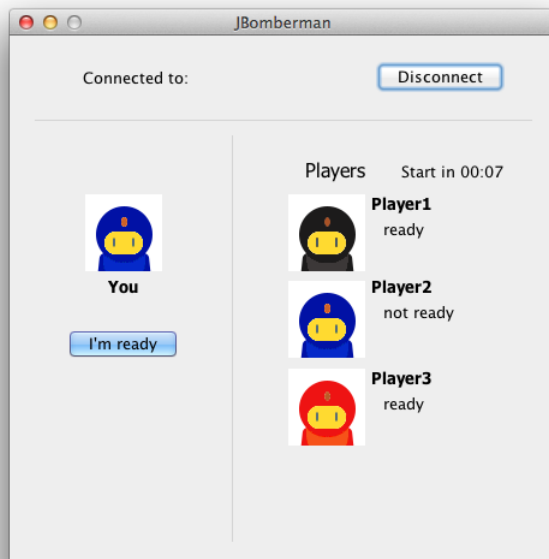
### 11.2 LobbyFrame

Sobald der Client sich mit dem Server verbunden hat erscheint das LobbyFrame in welchem alle Mitspieler, die sich auf den Server verbunden haben angezeigt werden. Dabei kann jeder Spieler sich über den Button I'm Ready ihren Status auf Ready setzen. Hier ist der I'm Ready Button als Standardbutton festgelegt.



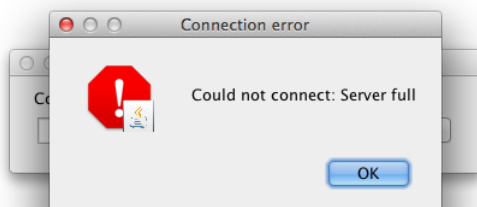
### 11.2.1 Timer

Sobald 2 Spieler auf Ready sind wird ein Timer gestartet (10 Sekunden), bis dann muss jeder Spieler der noch nicht Ready ist seinen Status ändern oder er/sie wird aus dem Spiel geworfen bzw.



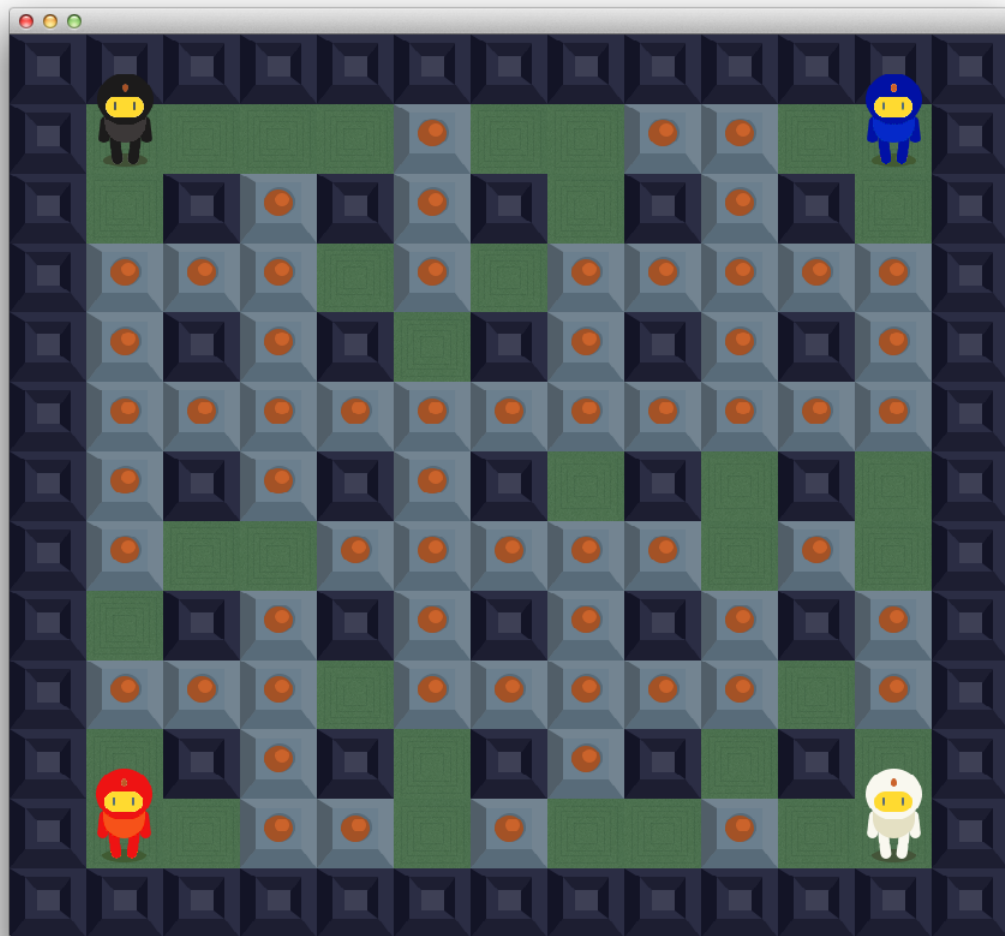
### 11.2.2 Fehlermeldungen

Falls auf dem Server bereits 4 Spieler vorhanden sind wird eine Fehlermeldung angezeigt, das der Server bereits Voll ist.



### 11.3 GameFrame

Sobald mindesten 2 Spieler ready waren und er Timer abgelaufen ist startet das Spiel.



## 12 Grössen und Leistung

JBombberman wird in zwei einzelne Applikationen unterteilt (Server und Client), dabei werden die Sprites auf dem Client gespeichert was diesen natürlich grösser macht ca.: 1 MB. Der Server selbst sollte nicht grösser als ca.: 0.5 MB werden. Das Spiel kann auf allen Clients gespielt werden, welche Java installiert haben und eine Tastatur als Eingabegerät haben. Die Grafiken sind für eine Grösse von 832px x 832px ausgelegt (kann jedoch auch verkleinert werden), wodurch eigentlich keine minimale Auflösung benötigt wird.