Projekt: JBomberman
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 Binna
05.04.15	1.02	Logische Architektur Client	Fabian Binna
05.04.15	1.03	Ziele und Einschränkungen	Silvan Adrian
06.04.15	1.04	Logische Architektur Server	Fabian Binna
06.04.15	1.05	Zustandsdiagramm Workflows	Fabian Binna
06.04.15	1.06	Systemübersicht Grafik und Beschreibung	Silvan Adrian

Inhaltsverzeichnis

1 Änderungshistorie **2**

2 Einführung **5**
2.1 Zweck . . . 5
2.2 Gültigkeitsbereich . . . 5
2.3 Referenzen . . . 5
2.4 Übersicht . . . 5

3 Systemübersicht **5**

4 Architektonische Ziele & Einschränkungen **6**
4.1 Ziele . . . 6
4.2 Einschränkungen . . . 6

5 Logische Architektur: Applikationsübergreifend **7**
5.1 Time/time . . . 8
5.1.1 Klassenstruktur . . . 8
5.2 Domain/game . . . 9
5.2.1 Klassenstruktur . . . 9
5.3 Utilities/utils . . . 11
5.3.1 Klassenstruktur . . . 11

6 Logische Architektur: Client **12**
6.1 Presentation/view . . . 12
6.1.1 Klassenstruktur . . . 12
6.2 Workflow/application.client . . . 13
6.2.1 Klassenstruktur . . . 13
6.3 Domain/game.client . . . 14
6.3.1 Klassenstruktur . . . 15
6.3.2 Wichtige interne Abläufe . . . 16
6.4 Network/network.client . . . 17
6.4.1 Klassenstruktur . . . 17
6.5 Wichtige Abläufe . . . 18
6.5.1 Zustandsdiagramm Client Workflow . . . 18

7 Logische Architektur: Server **19**
7.1 Workflow/application.server . . . 19
7.1.1 Klassenstruktur . . . 19
7.2 Domain/game.server . . . 20
7.2.1 Klassenstruktur . . . 20
7.3 Network/network.server . . . 22
7.3.1 Klassenstruktur . . . 22

7.4 Wichtige Abläufe 23
7.4.1 Zustandsdiagramm Server Workflow 23

8 Prozesse und Threads 23

9 Deployment 24

10 Grössen und Leistung 24

2 Einführung

2.1 Zweck

Dieses Dokument beschreibt die Software Architektur für das Projekt JBomberman.

2.2 Gültigkeitsbereich

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

2.3 Referenzen

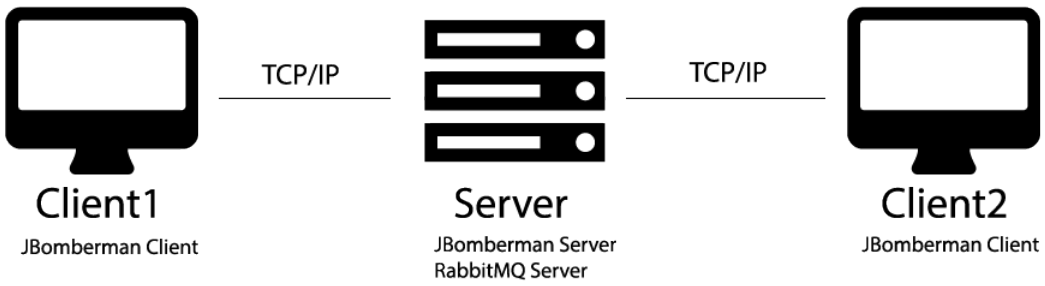
<Liste aller verwendeten und referenzierten Dokumente, Bücher, Links, usw.> <Referenz auf ein Glossar Dokument, wo alle Abkürzungen und unklaren Begriffe erklärt werden> <Die Quellen / Referenzen sollten mit dem Word Tool automatisch erstellt werden>

2.4 Übersicht

<Übersicht über den restlichen Teil dieses Dokumentes geben und dessen Aufbau erläutern>

3 Systemübersicht

JBomberman 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

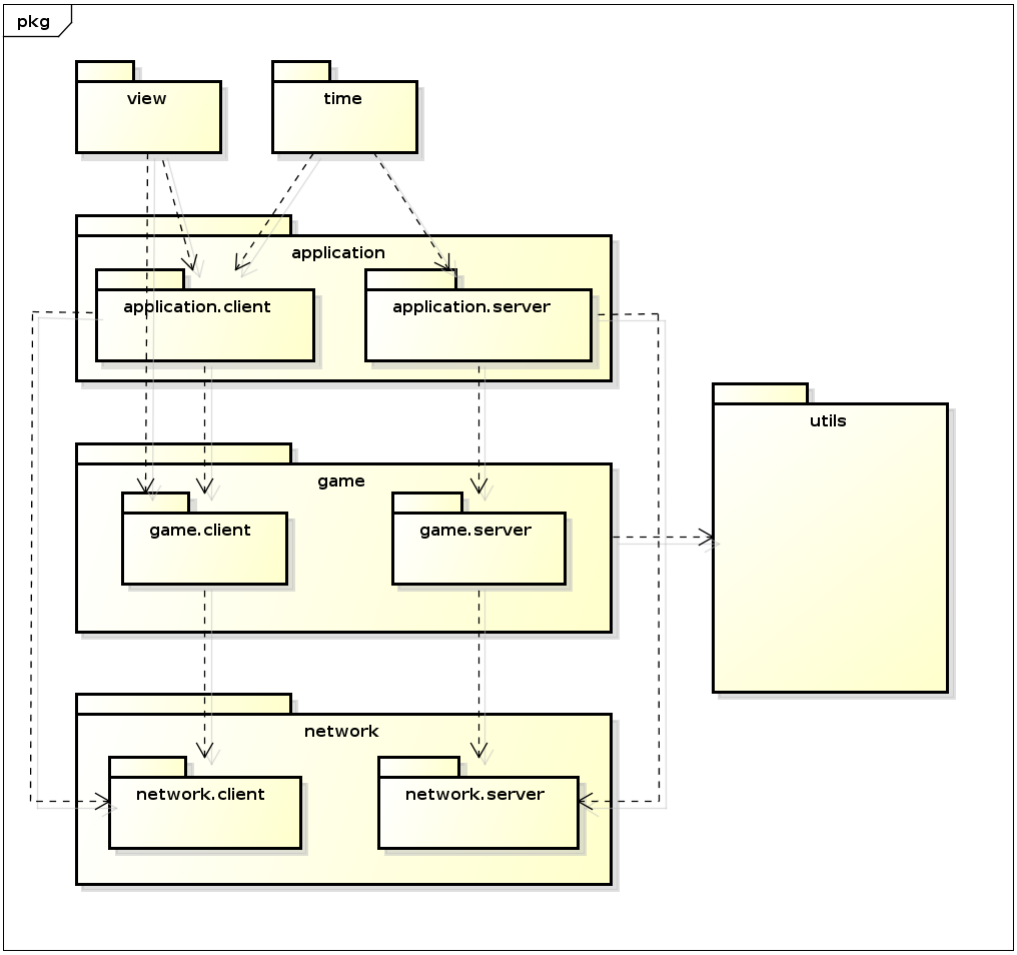
- Die Spielumgebungen müssen mit allen Clients synchronisiert werden (über den dedizierten Server)
- Möglichkeit weitere PowerUp's einzubauen.
- Austausch zwischen den Clients und dem Server wird über JMS umgesetzt.

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

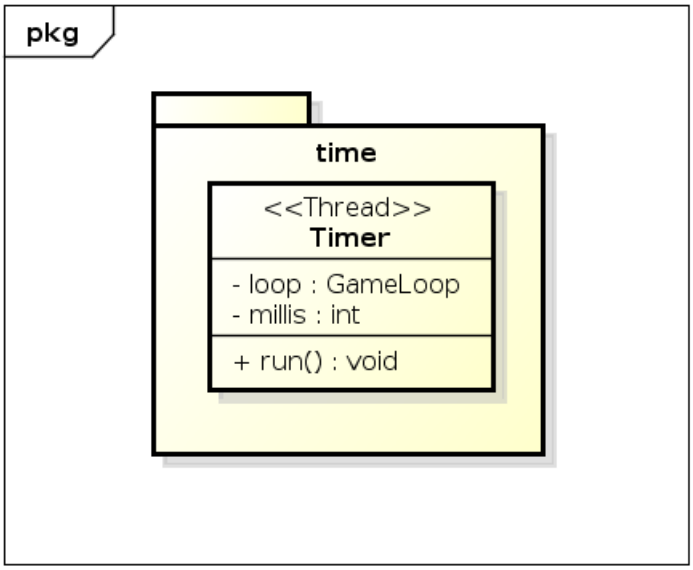
Dieses Package Diagramm zeigt sowol Client, als auch Server. Client und Server sind zwei eigenständige Applikationen, die getrennt ausgeführt werden. Sie verwenden jedoch teilweise die gleichen Packages.



5.1 Time/time

Im Package time sind Klasse, die für das timing der GameLoops sorgen.

5.1.1 Klassenstruktur

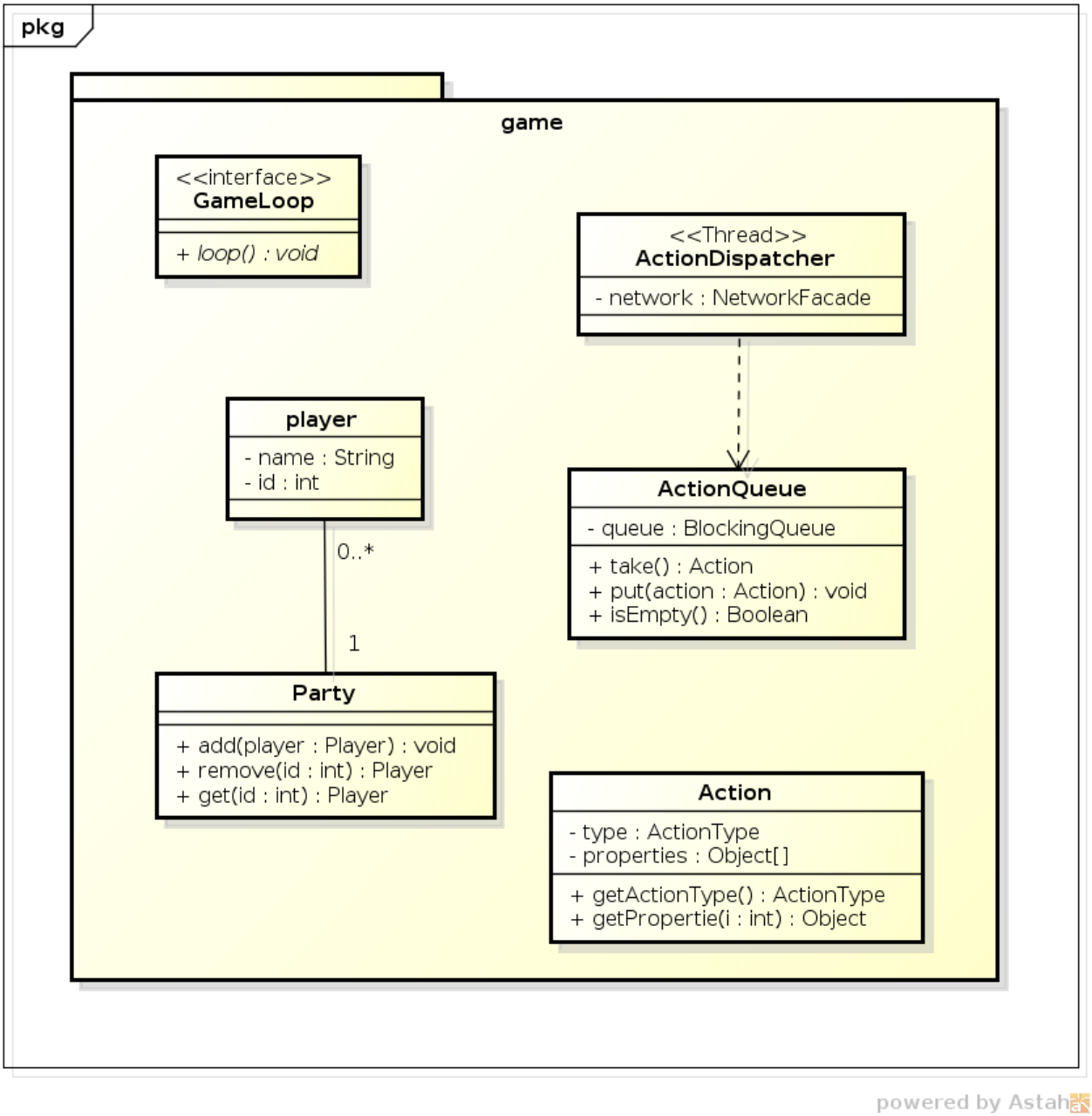


Timer
Der Timer besitzt eine Referenz auf einen GameLoop, bei dem er regelmässig die Methode loop() aufruft.

5.2 Domain/game

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

5.2.1 Klassenstruktur



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 beschäftig auf eingehende Messages zu warten und diese als Action in der Queue einzureihen.

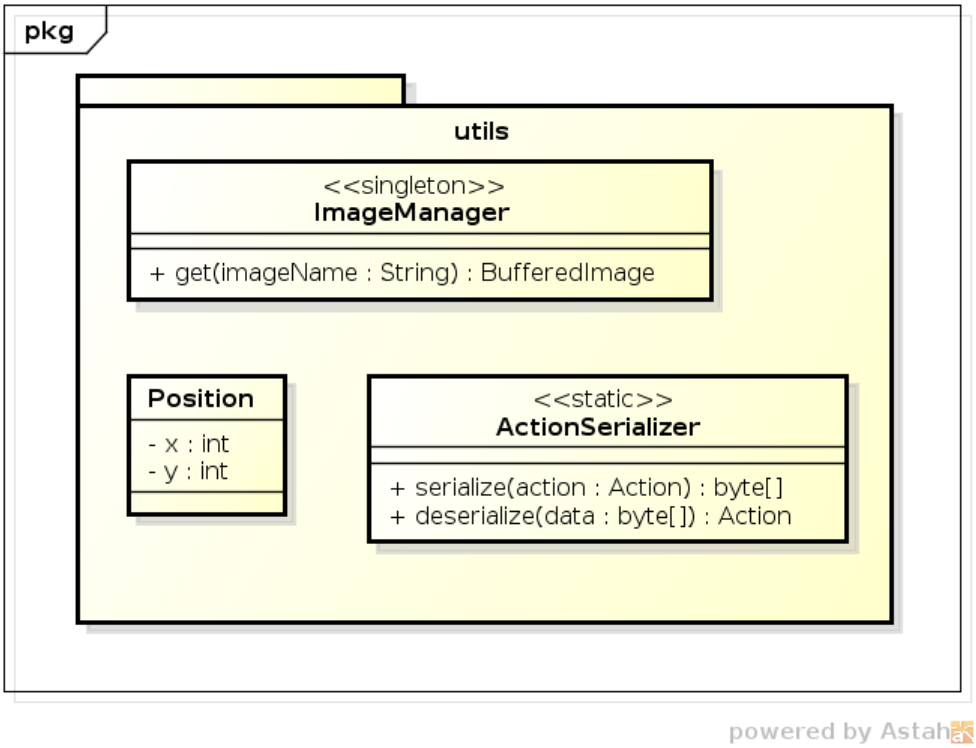
Player
Der Player speichert die Daten eines Spielers. Die Daten werden für die Zuweisung von Actions und das Darstellen des Scoreboard benötigt.

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

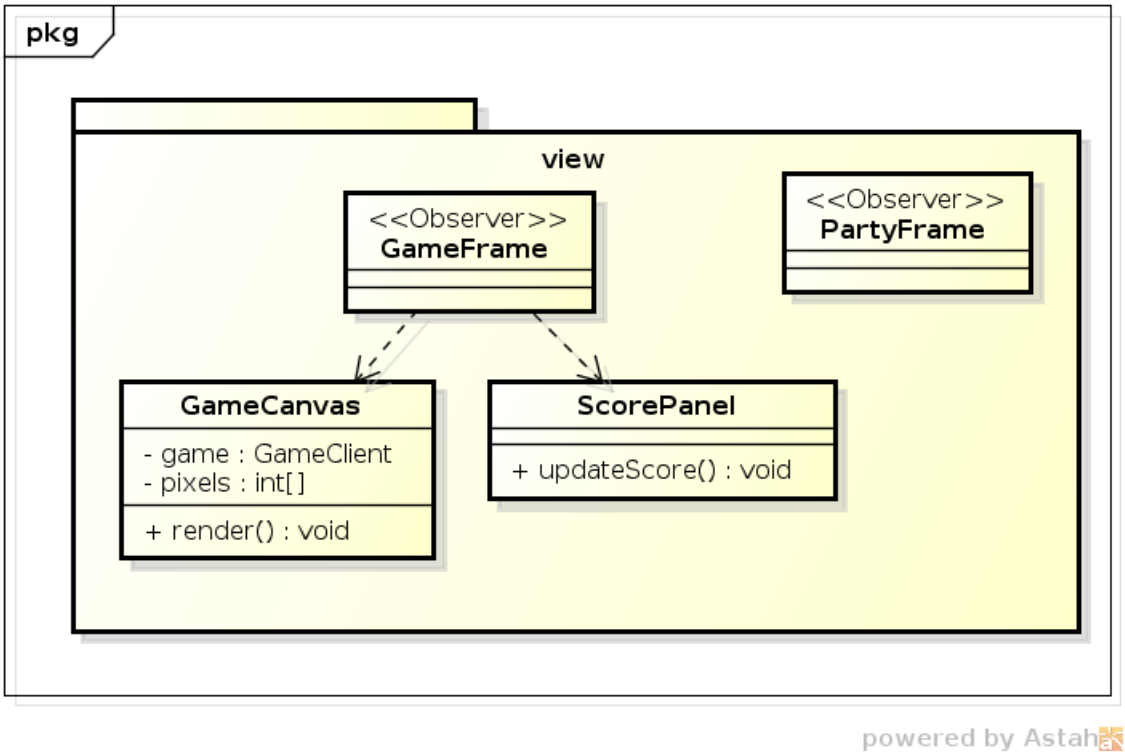
ActionSerializer
Die Methoden des ActionSerializer sind statisch und werden benötigt um die Actions für das Netzwerk zu serialisieren bzw. deserialisieren.

6 Logische Architektur: Client

6.1 Presentation/view

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

6.1.1 Klassenstruktur



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

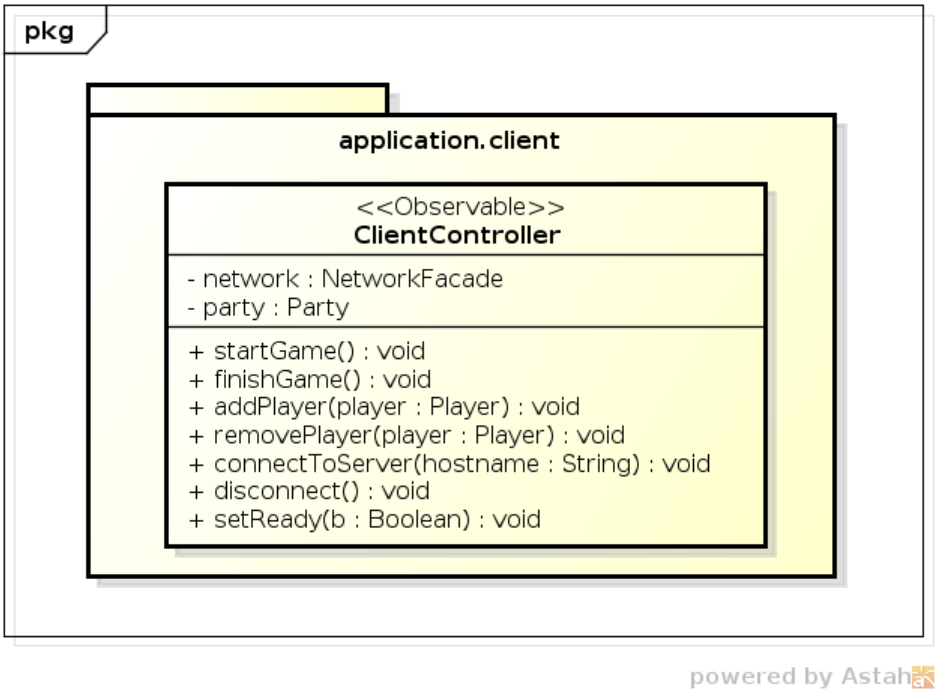
GameCanvas
Der GameCanvas kümmert sich nur um das Rendering, also das Zeichnen der Szene. Dabei delegiert er jedoch nur die pixels[] an alle Sprites, welche sich dann eigenständig zeichnen. Der GameCanvas kümmert sich dann um das performante Buffering.

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

6.2 Workflow/application.client

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

6.2.1 Klassenstruktur



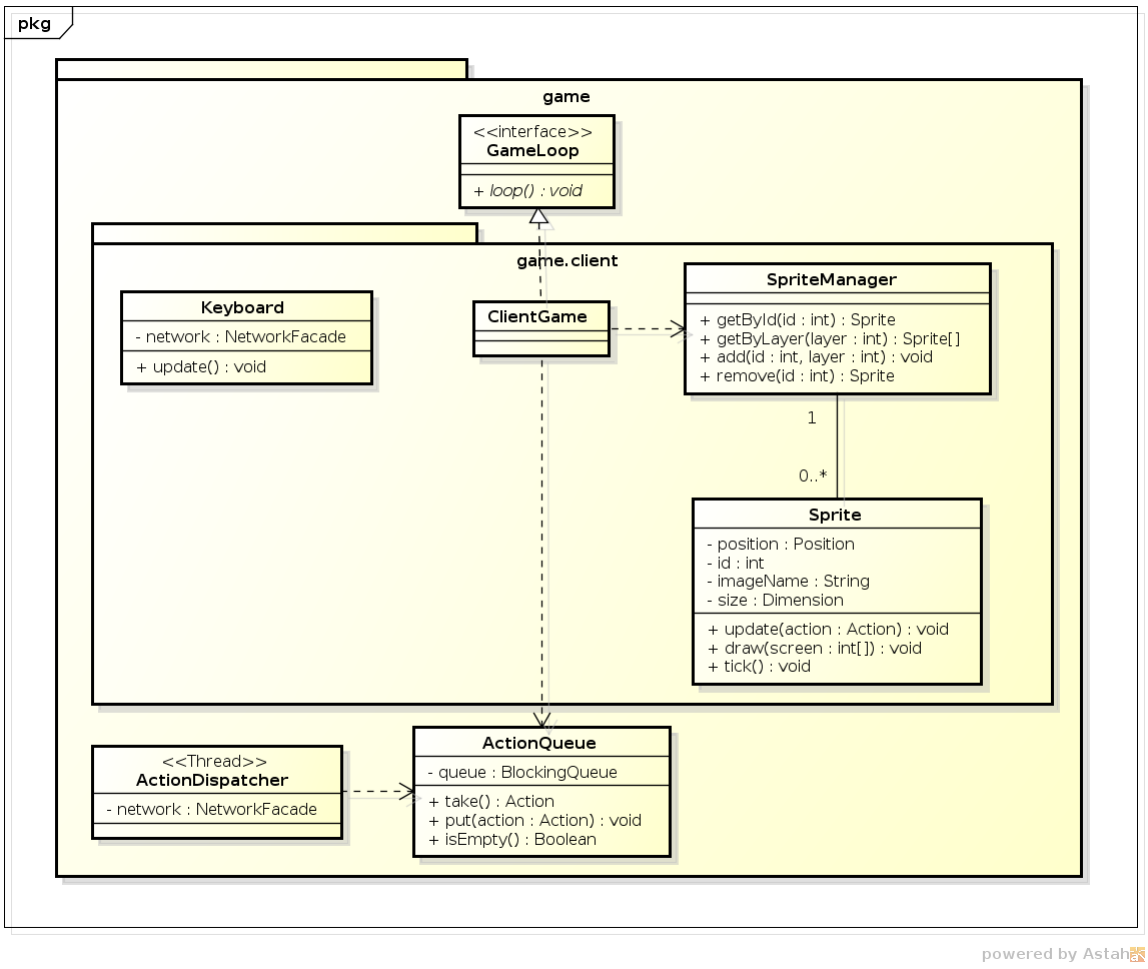
ClientController
Der ClientController kontrolliert den Zustand der Client-Applikation. Er notifiziert nur den PartyFrame, nicht aber den GameFrame.

Methode	Beschreibung
startGame():void	Erstellt eine GameFrame, welches die ClientGame Klasse instanziert und somit das Spiel startet.
finishGame():void	Informiert den PartyFrame darüber, dass das Spiel beendet wurde.
addPlayer(player: Player) : void	Fügt einen neuen Player in die Party ein.
removePlayer(player: Player) : void	Entfernt einen Player aus der Party.
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.

6.3 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 gespeichet, damit sie korrekt gezeichnet werden können.

6.3.1 Klassenstruktur



ClientGame

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

SpriteManager

Der SpriteManager speicher alle Sprites in einer Schichten-Logik. Dies wird benötig, 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.

Methode	Beschreibung
update(action : Acton) : void	Interpretiert die Action und führt die nötigen Aktualisierungsschritte durch.
draw(screen : int[]) : void	Zeichnet sich selbst auf den screen. Diese Methode wird vom GameCanvas aufgerufen und liefert seinen screen mit auf dem gezeichnet werden kann.
tick() : void	Aktualisiert das Sprite. Wird hauptsächlich für Animationen benötigt.

Sprite
Die Sprite-Klasse beinhaltet alle Informationen die für das Zeichnen der Spielobjekte benötig wird.
Keyboard
Das Keyboard zeichnet die Tastatureingaben auf und sendet diese direkt über die Methode update an den Server.

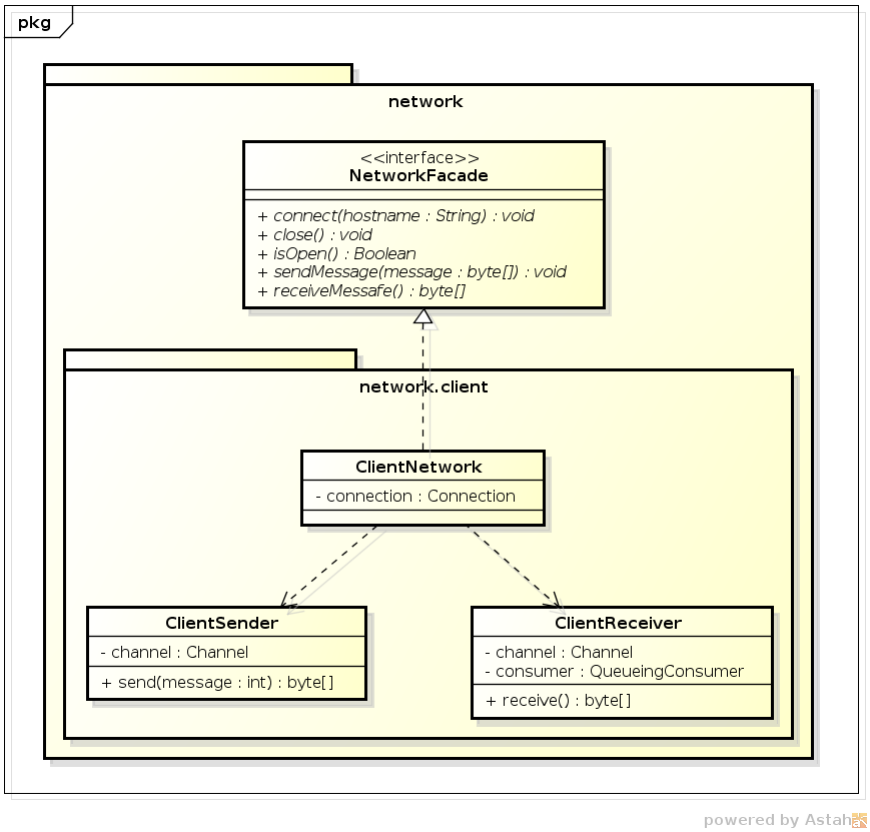
6.3.2 Wichtige interne Abläufe

//SSD zur GameLoop realisiertung einfügen.

6.4 Network/network.client

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

6.4.1 Klassenstruktur



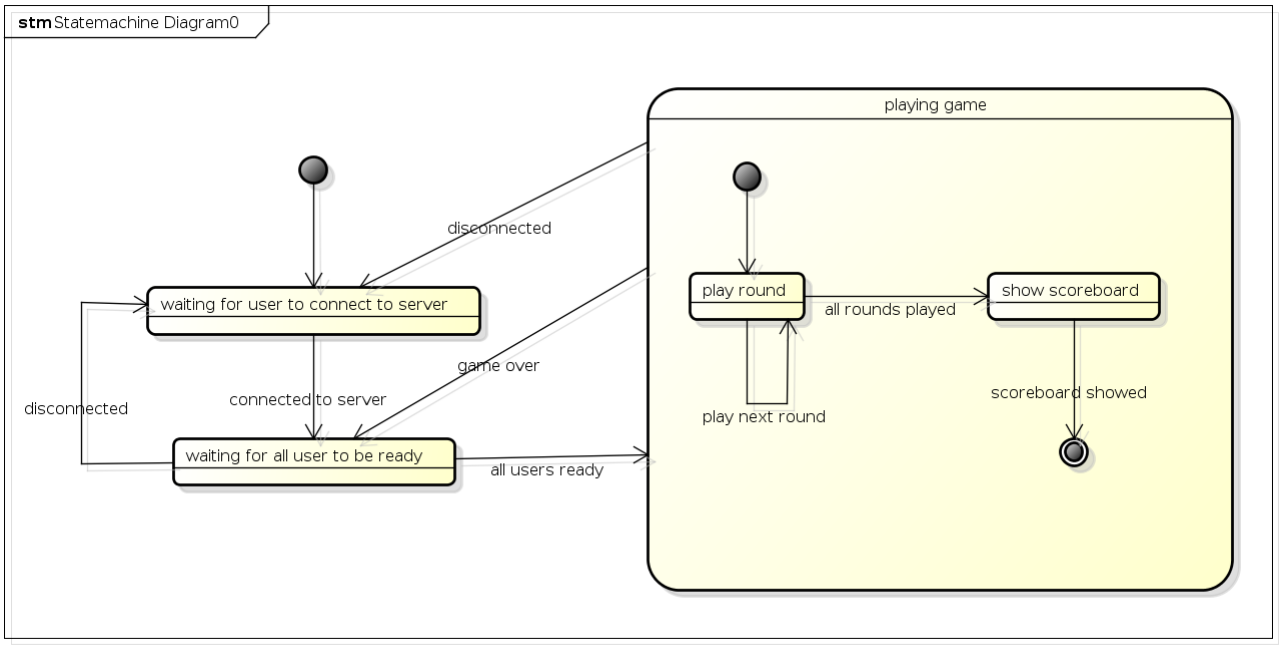
ClientNetwork
Die Klasse ClientNetwork implementiert die NetworkFacade.

ClientSender
Der Client Sender sendet direkt auf eine RabbtiMQ-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.

6.5 Wichtige Abläufe

6.5.1 Zustandsdiagramm Client Workflow

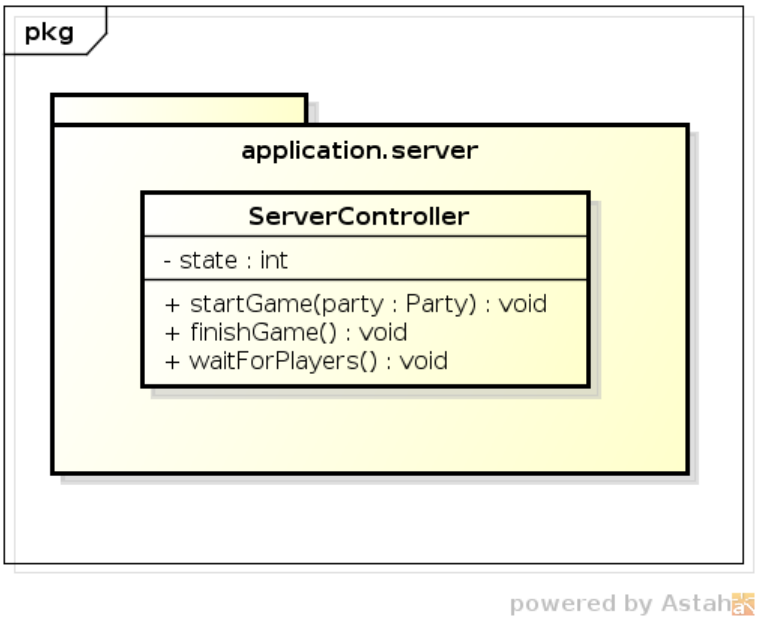


7 Logische Architektur: Server

7.1 Workflow/application.server

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

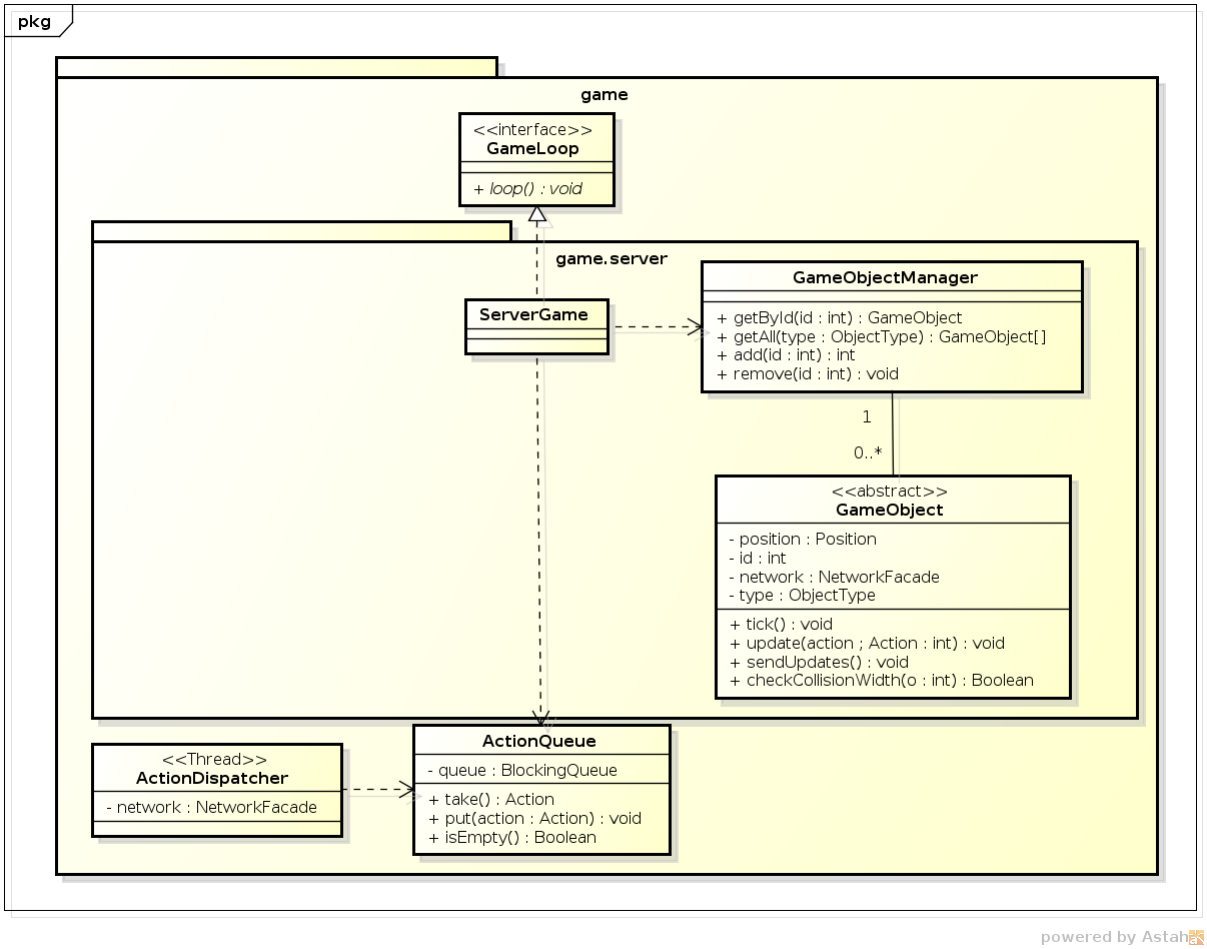
7.1.1 Klassenstruktur



Methode	Beschreibung
wairForPlayers() : 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.

7.2 Domain/game.server

7.2.1 Klassenstruktur



ServerGame

Die Methode loop() wird unter "wichtige Abläufe"genauer beschrieben.

GameObjectManager

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

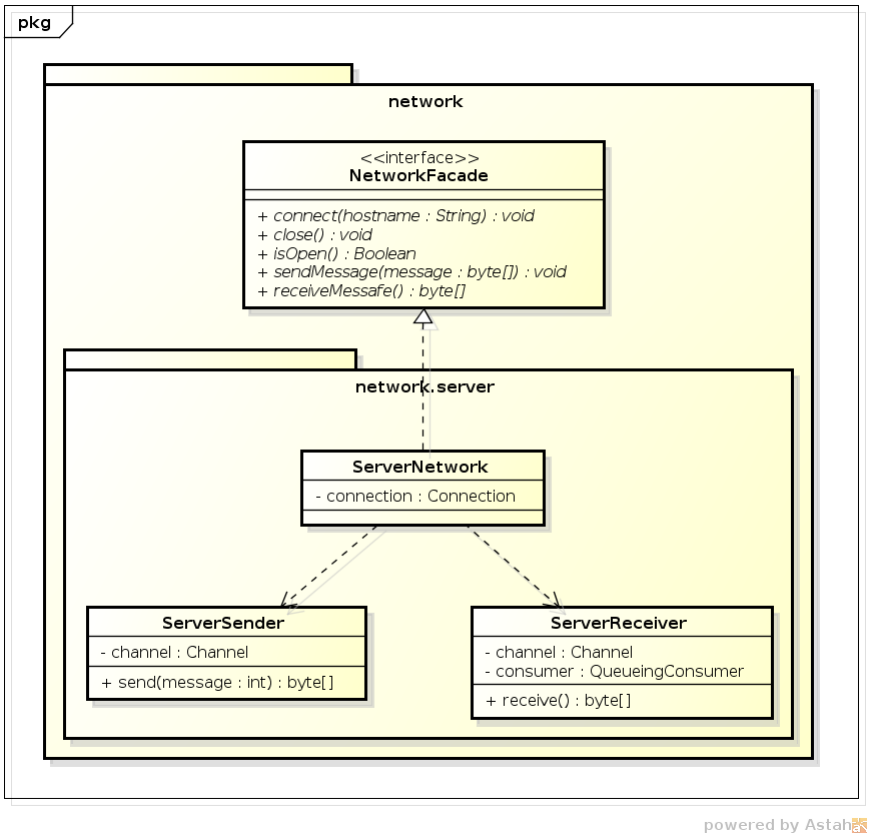
GameObject
Das GameObject speichert alle nötigen Informationen um das Spielgeschehen und die Interaktionen der Gameobjects zu berechnen.

Methode	Beschreibung
tick() : 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.
checkCollisionWidth(o : GameObject) : Boolean	Überprüft, ob das Object mit einem anderen kollidiert und führt dann die jeweiligen Korrekturen (z.B. beim Movement) oder Aktionen durch.

7.3 Network/network.server

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

7.3.1 Klassenstruktur



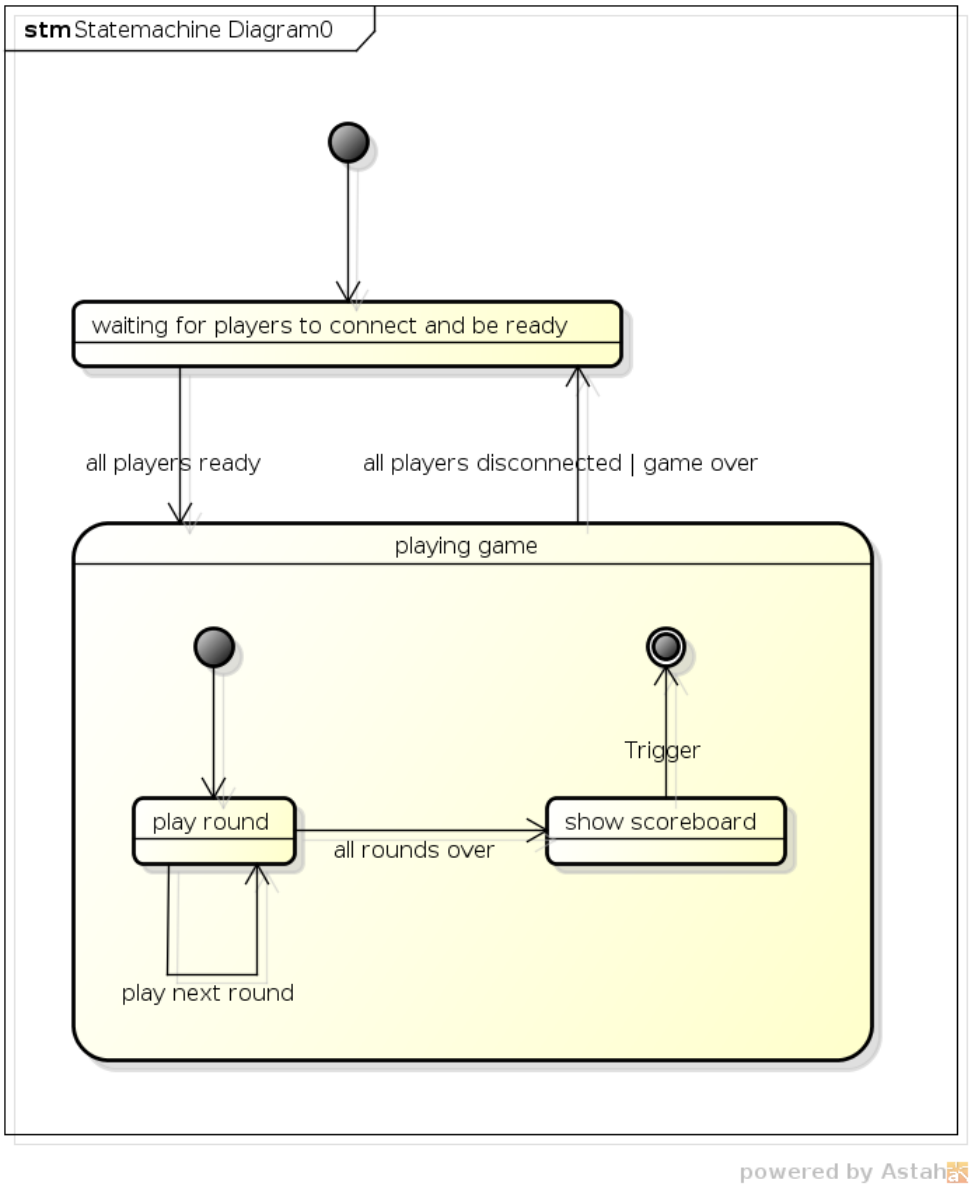
ServerNetwork
Die Klasse ServerNetwork implementiert die NetworkFacade.

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.

7.4 Wichtige Abläufe

7.4.1 Zustandsdiagramm Server Workflow



8 Prozesse und Threads

<Wenn mehrere Prozesse oder Threads eingesetzt werden wird hier beschrieben, wie diese ablaufen, miteinander funktionieren, Daten austauschen, sich synchronisieren, usw.>

9 Deployment

<Beschreibung der einzelnen Komponenten und deren Aufteilung (auf welchen Umgebungen, Servern, usw. laufen die Komponenten)>

10 Grössen und Leistung

<Einschränkungen der Applikation bezüglich Speicher, Leistung, etc.... (zum Beispiel: Verwaltung unterstützt maximal 20'000 Einträge)>