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
06.04.15	1.07	Schnittstellen	Fabian Binna
06.04.15	1.08	Grössen und Leistung + Externes Design	Silvan Adrian
07.04.15	1.09	Datenübertragung	Fabian Binna
07.04.15	1.10	Prozesse und Threads	Fabian Binna
07.04.15	1.11	Korrekturen	Fabian Binna
07.04.15	1.12	Korrekturen	Silvan Adrian
16.04.15	1.13	Sequenz Diagramm Action	Fabian Binna
05.05.15	1.14	Anpassungen	Fabian Binna
05.05.15	1.15	Anpassungen an Client-, ServerController und LobbyCommunication	Pascal Kistler
06.05.15	1.16	Externes Design erweitert + Grössen Leistung angepasst	Silvan Adrian
16.05.15	1.17	NetworkFacade und LobbyCommunication angepasst	Pascal Kistler
24.05.15	1.18	Packagediagramme angepasst	Pascal Kistler

Inhaltsverzeichnis

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

2.4 Übersicht

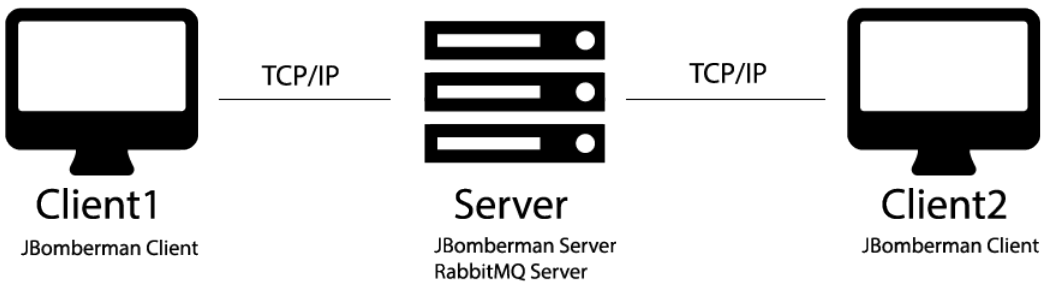
Die LogischeSicht wurde in drei Teile unterteilt: Applikationsübergreifend, Client und Server. Die Schnittstellen werden anhand einer kompletten Packageübersicht gezeigt. Im Kapitel Datenübertragung wird auf die RabbitMQ-Technologie eingegangen.

2.4.1 Glossar

Siehe Dokument Glossar.pdf

3 Systemübersicht

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

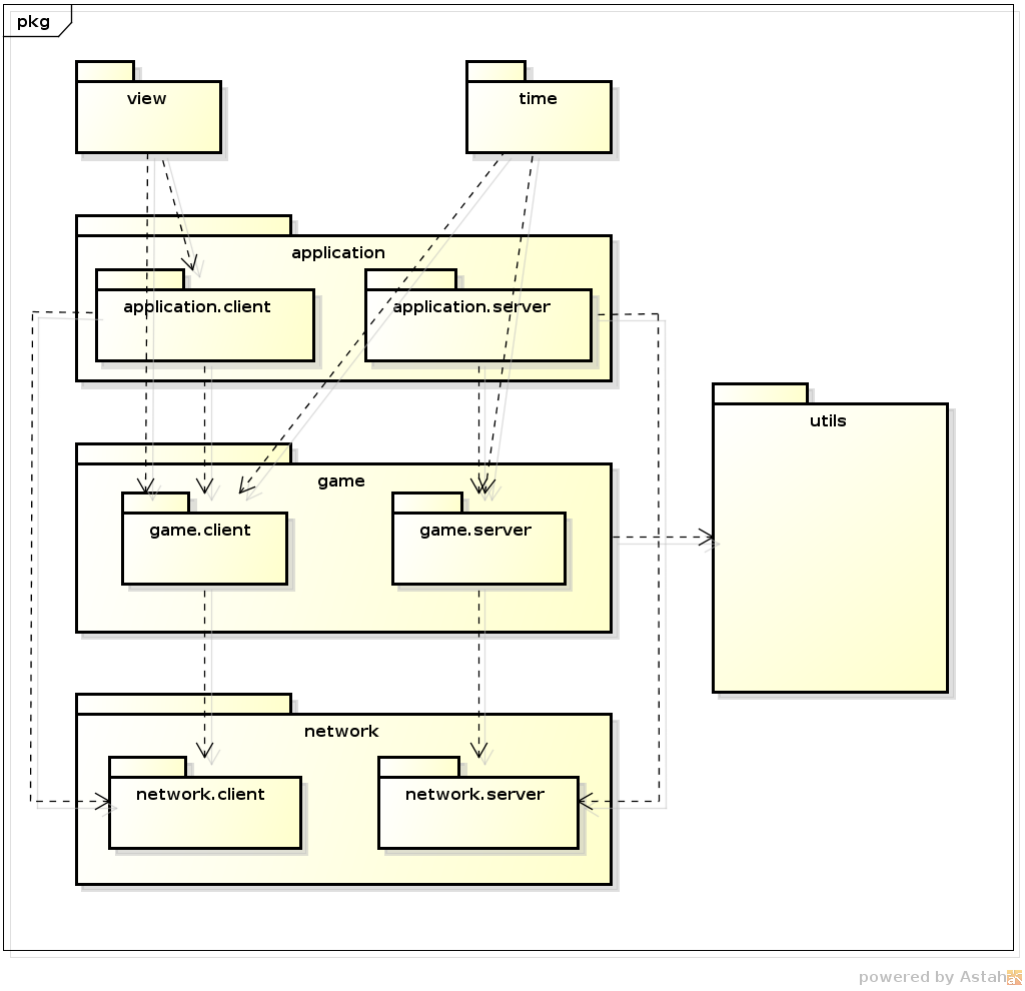
- Die Spielumgebungen müssen mit allen Clients synchronisiert werden (über den dedizierten Server)
- Möglichkeit weitere PowerUp's einzubauen.
- Austausch zwischen den Clients und dem Server wird über JMS umgesetzt.

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

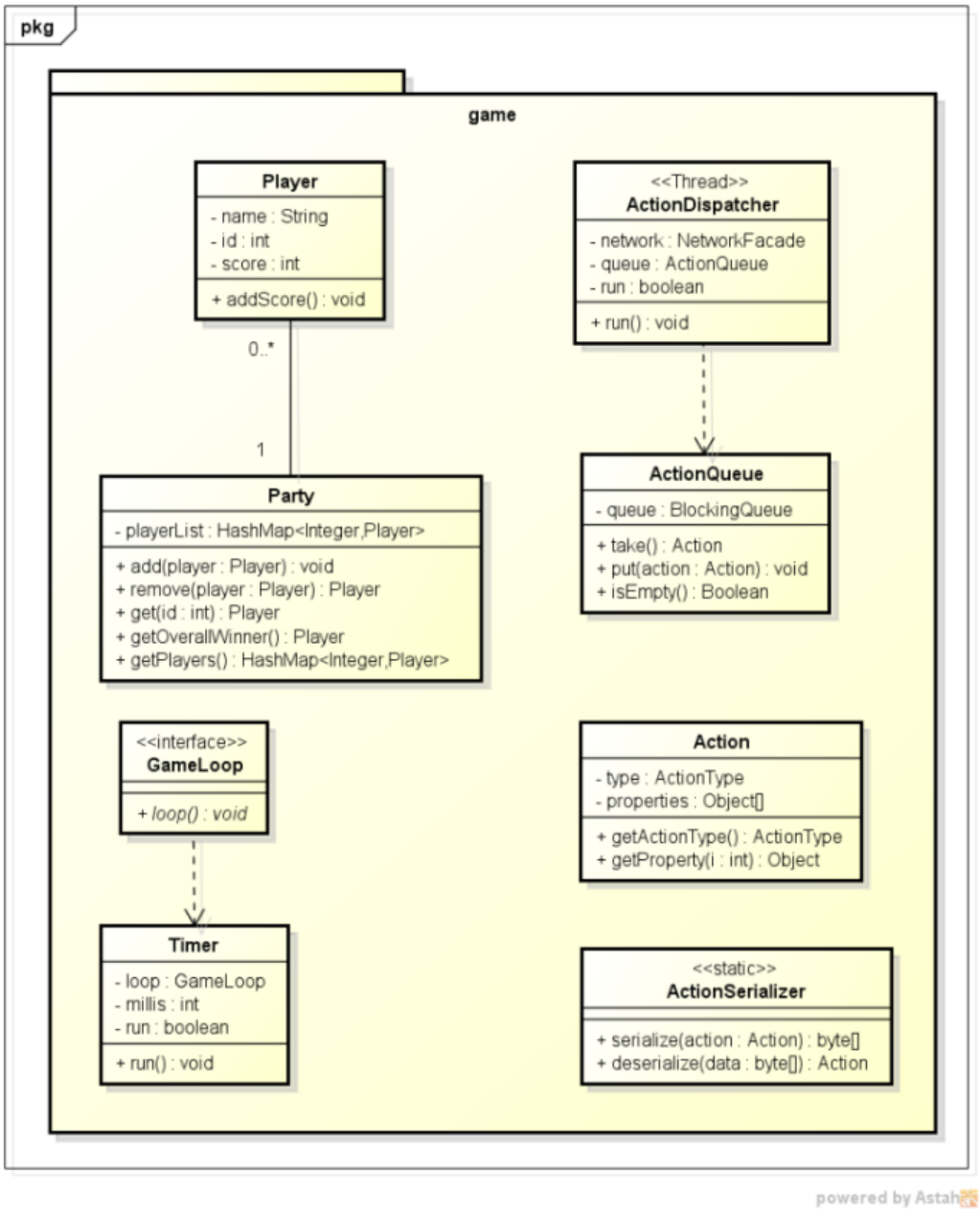
Dieses Package Diagramm zeigt sowohl Client, als auch Server. Client und Server sind zwei eigenständige Applikationen, die getrennt ausgeführt werden. Sie verwenden jedoch teilweise die gleichen Packages.



5.1 Domain/game

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

5.1.1 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.

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

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

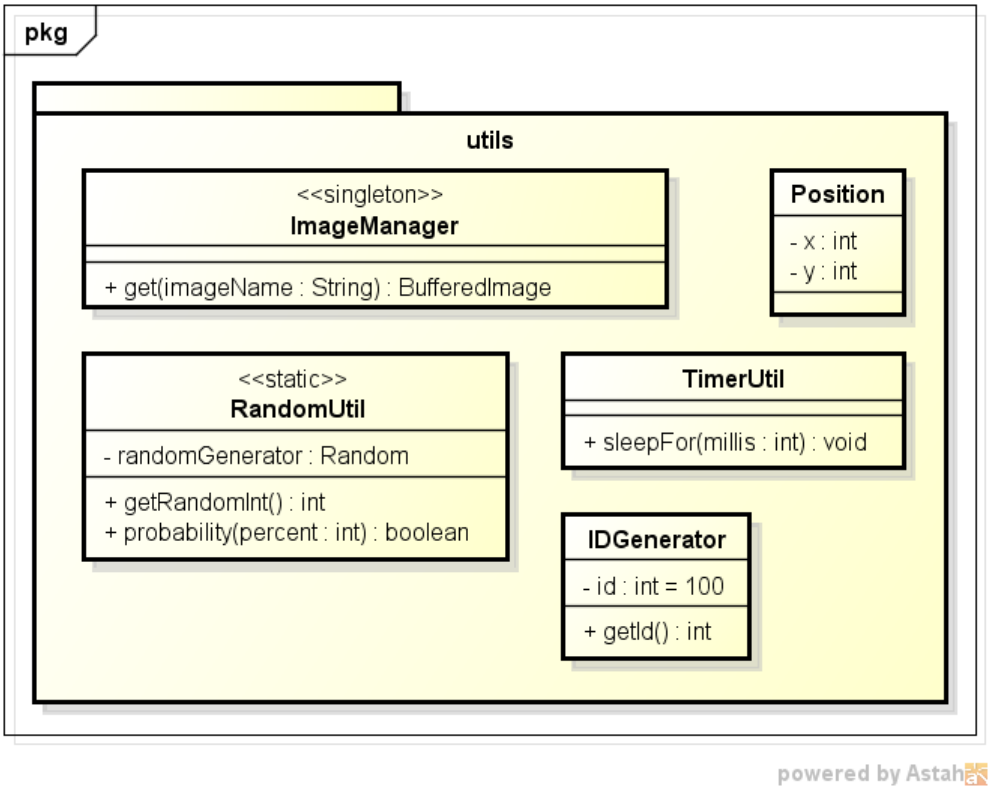
Party
Die Party beinhaltet alle Player eines Spiels. Ein Spiel kann nur mittels einer Party instanziert werden.

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

5.2 Utilities/utils

Im Package utils befinden sich Hilfsklassen.

5.2.1 Klassenstruktur



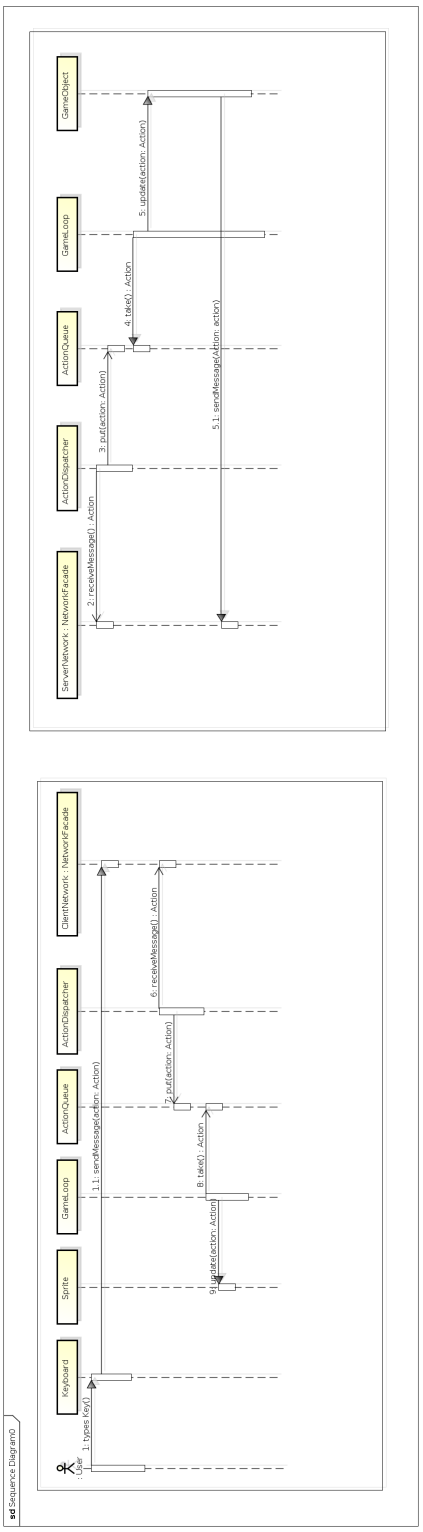
ImageManager
Der ImageManager lädt die Bilder. Jedes Sprite holt sich von da die Bilddaten.

IDGerenator
Der IDGenerator generiert ab 100 fortlaufende IDs

RandomUtil
Diese Klasse enthält alle Methoden, welche etwas mit Zufallszahlen zu tun haben, wie Zufallszahlen generieren und mit einer definierbaren Wahrscheinlichkeit true zurück zu geben

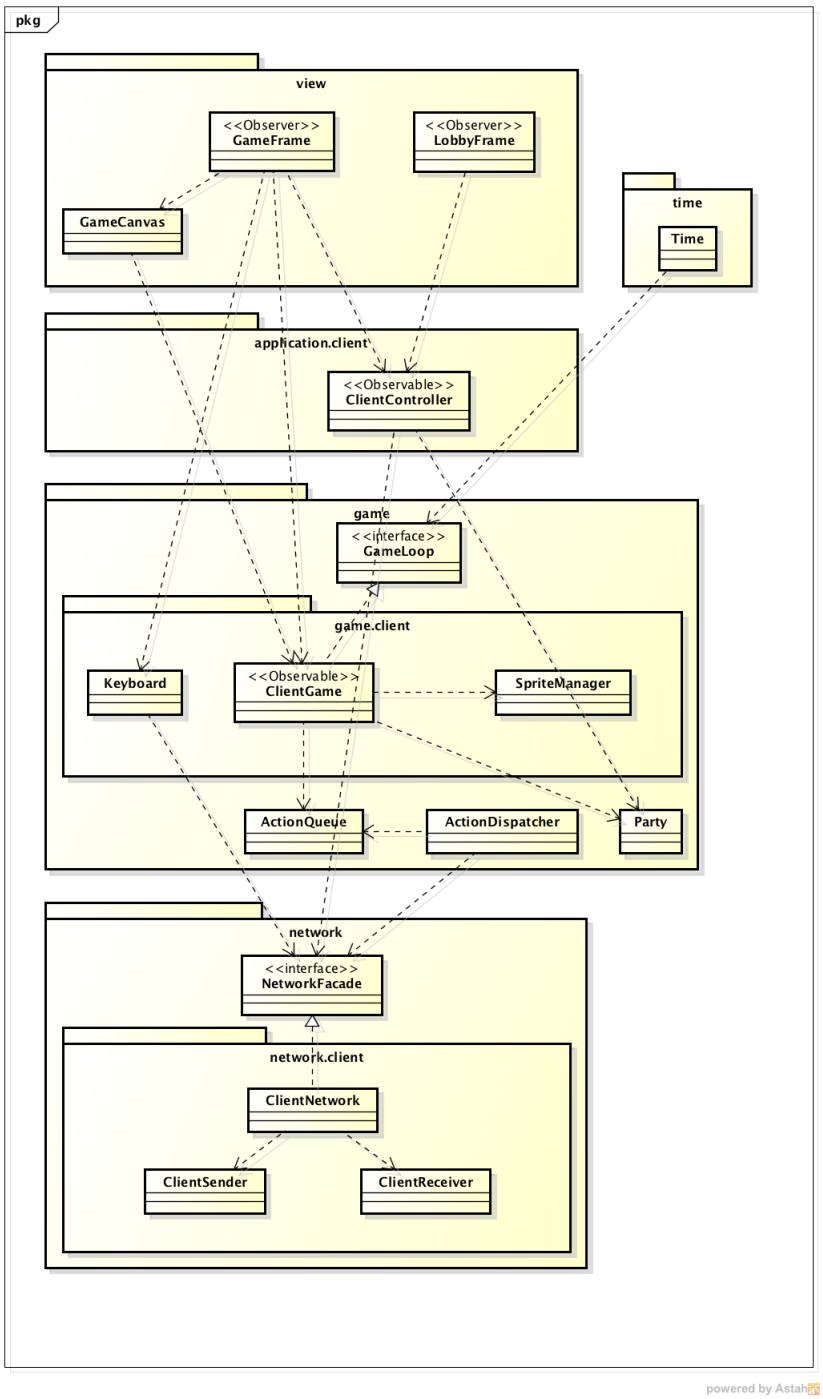
TimerUtil
Verzögert den aufrufenden Thread um die angegebene Zeit.

5.3 Wichtige Abläufe



6 Logische Architektur: Client

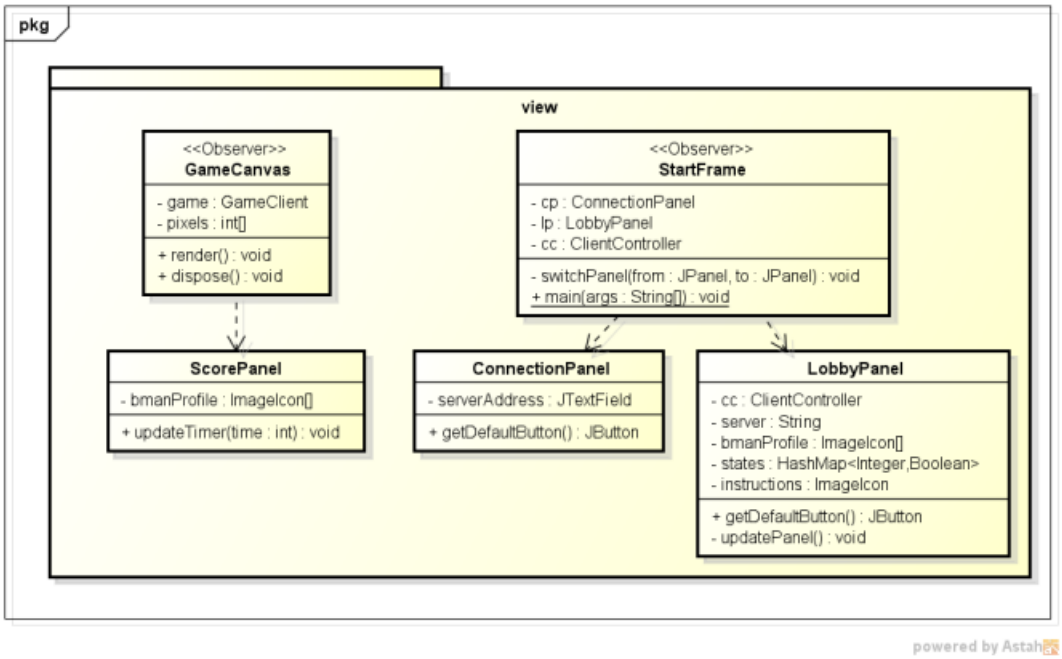
6.1 Schnittstellen



6.2 Presentation/view

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

6.2.1 Klassenstruktur



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.

ScorePanel
Das ScorePanel zeigt die aktuelle Runde und verbleibende Zeit an. Zudem werden auch die aktuellen Punktestände der einzelnen Spieler dargestellt.

StartFrame
Das StartFrame übernimmt die Koordination von Connection- und LobbyPanel und zeigt, falls vorhanden, Fehlermeldungen an.

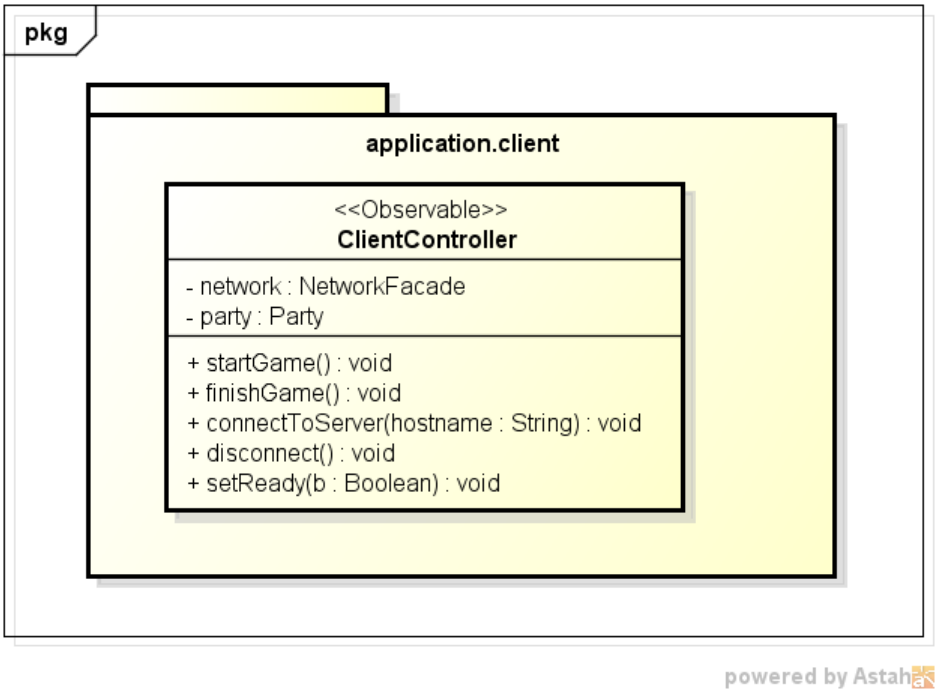
ConnectionPanel
Im ConnectionPanel wird die Serveradresse eingegeben und der Verbindungsaufbau gestartet.

LobbyPanel
Das LobbyPanel zeigt alle aktuell am Server angemeldeten Spieler an und deren Status.

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

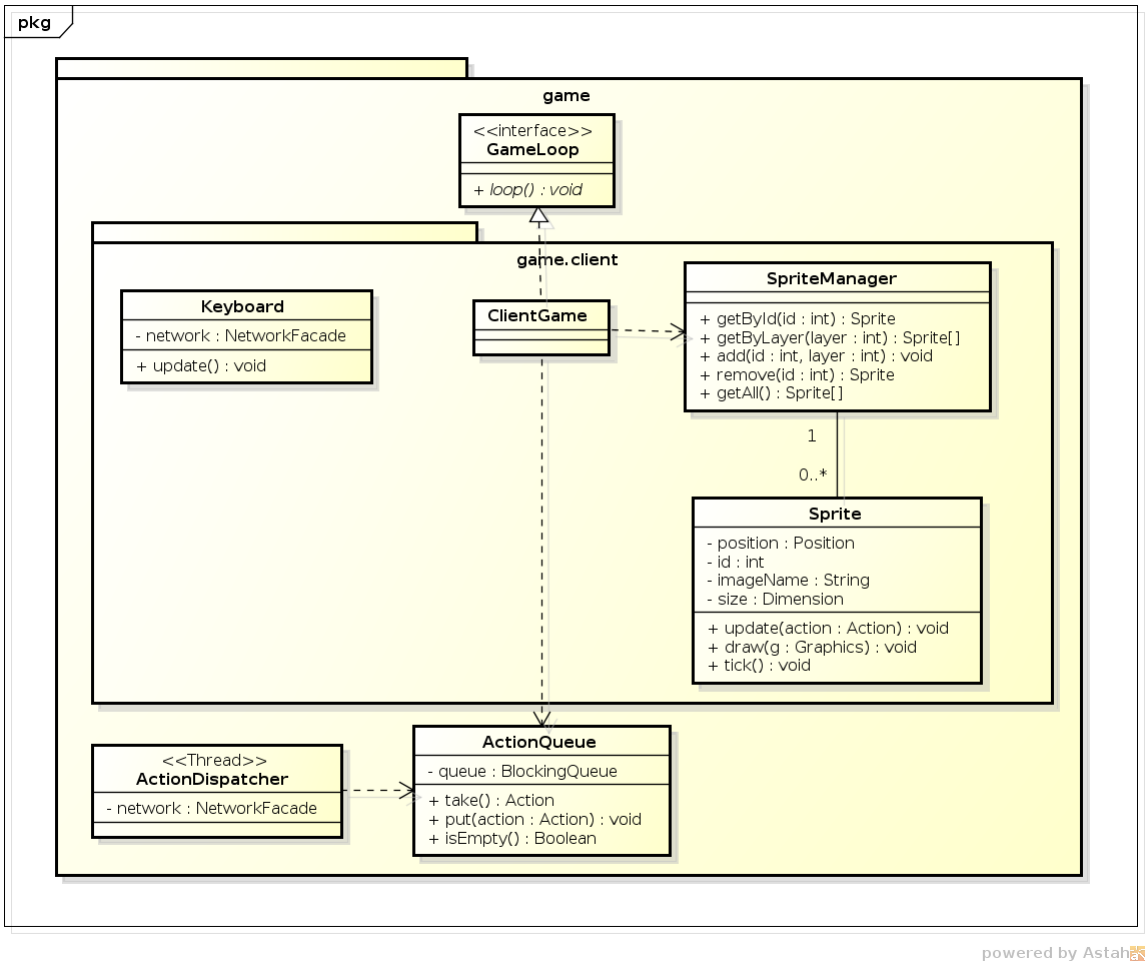


ClientController
Der ClientController kontrolliert den Zustand der Client-Applikation. Er notifiziert nur den LobbyFrame, nicht aber den GameFrame.

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 gespeichet, damit sie korrekt gezeichnet werden können.

6.4.1 Klassenstruktur



ClientGame

Die Methode loop wird unter "Wichtige 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.

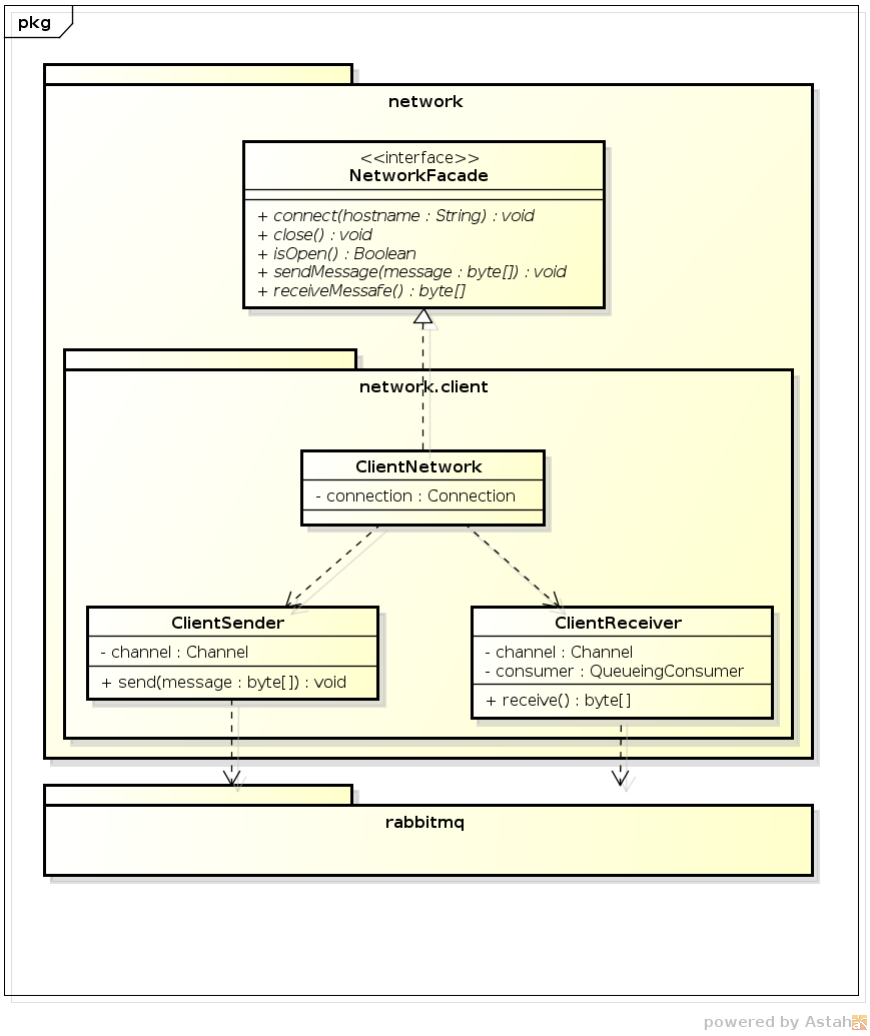
Sprite
Die Sprite-Klasse beinhaltet alle Informationen die für das Zeichnen der Spielobjekte benötig wird.

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

6.5 Network/network.client

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

6.5.1 Klassenstruktur



ClientNetwork
Die Klasse ClientNetwork implementiert die NetworkFacade.

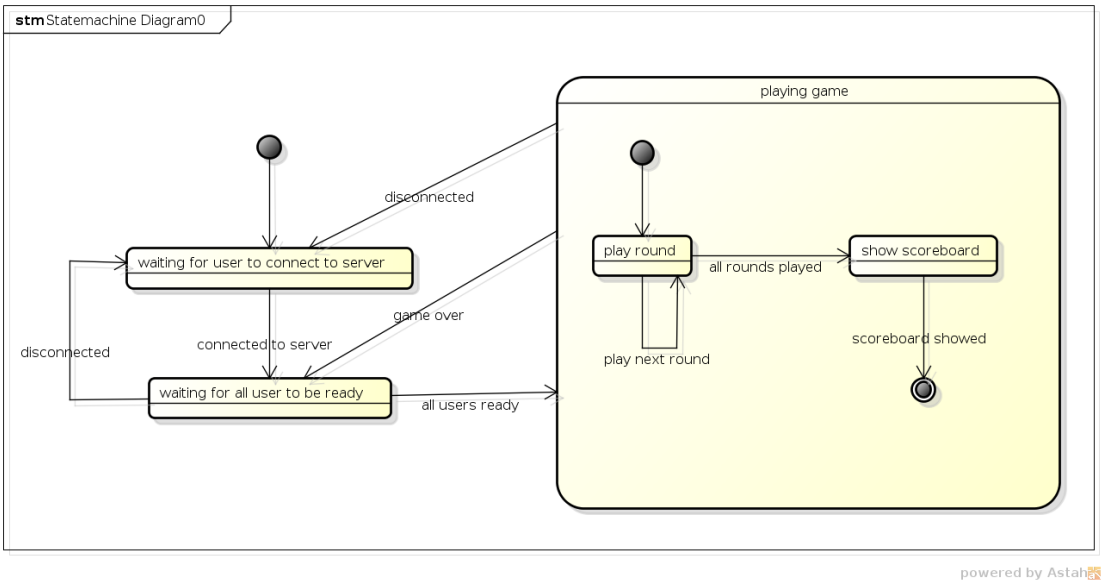
ClientSender
Der Client Sender sendet direkt auf eine RabbtiMQ-Queue. Alle Clients senden auf die selbe Queue.

ClientReceiver

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

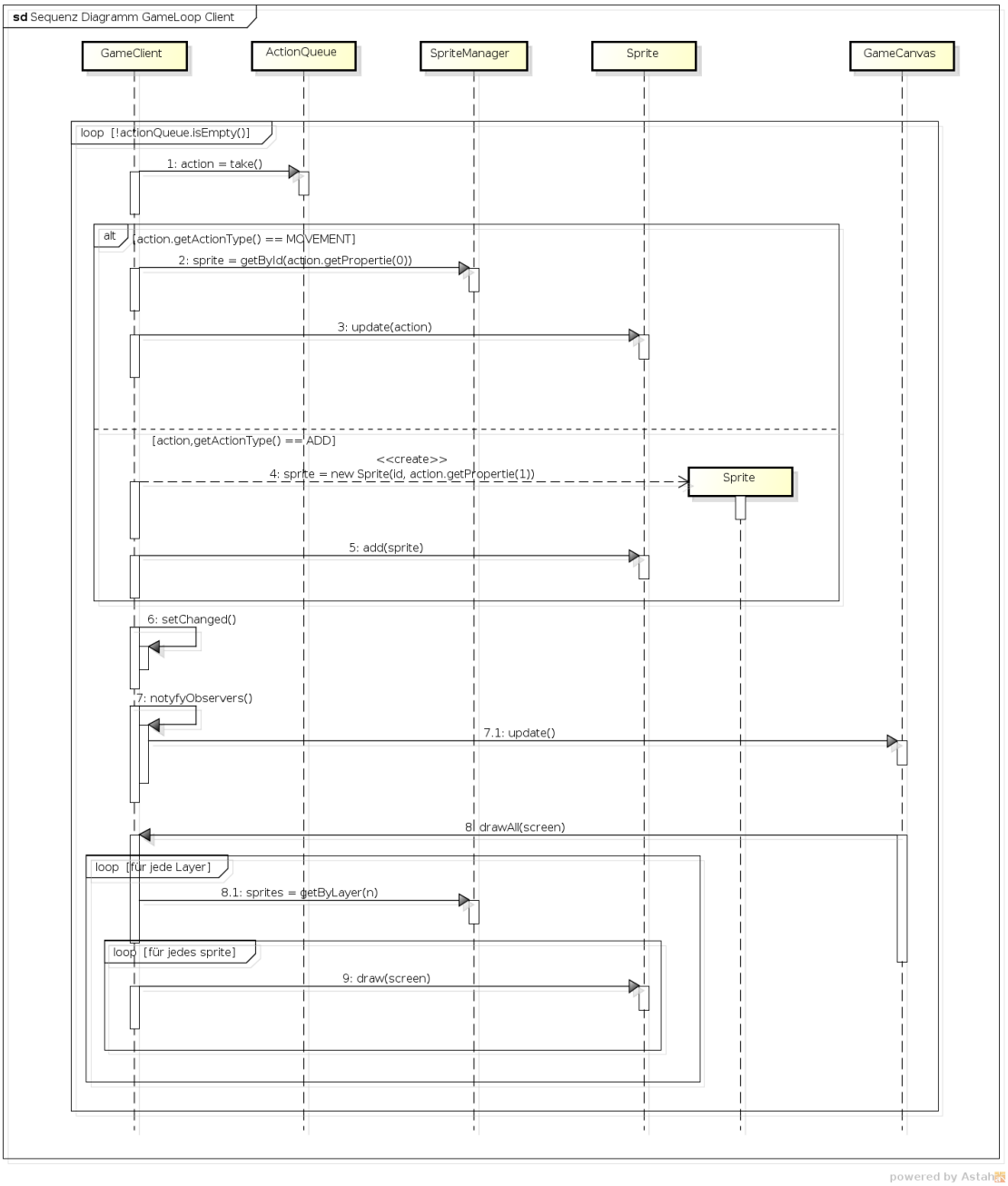
6.6 Wichtige Abläufe

6.6.1 Zustandsdiagramm Client Workflow



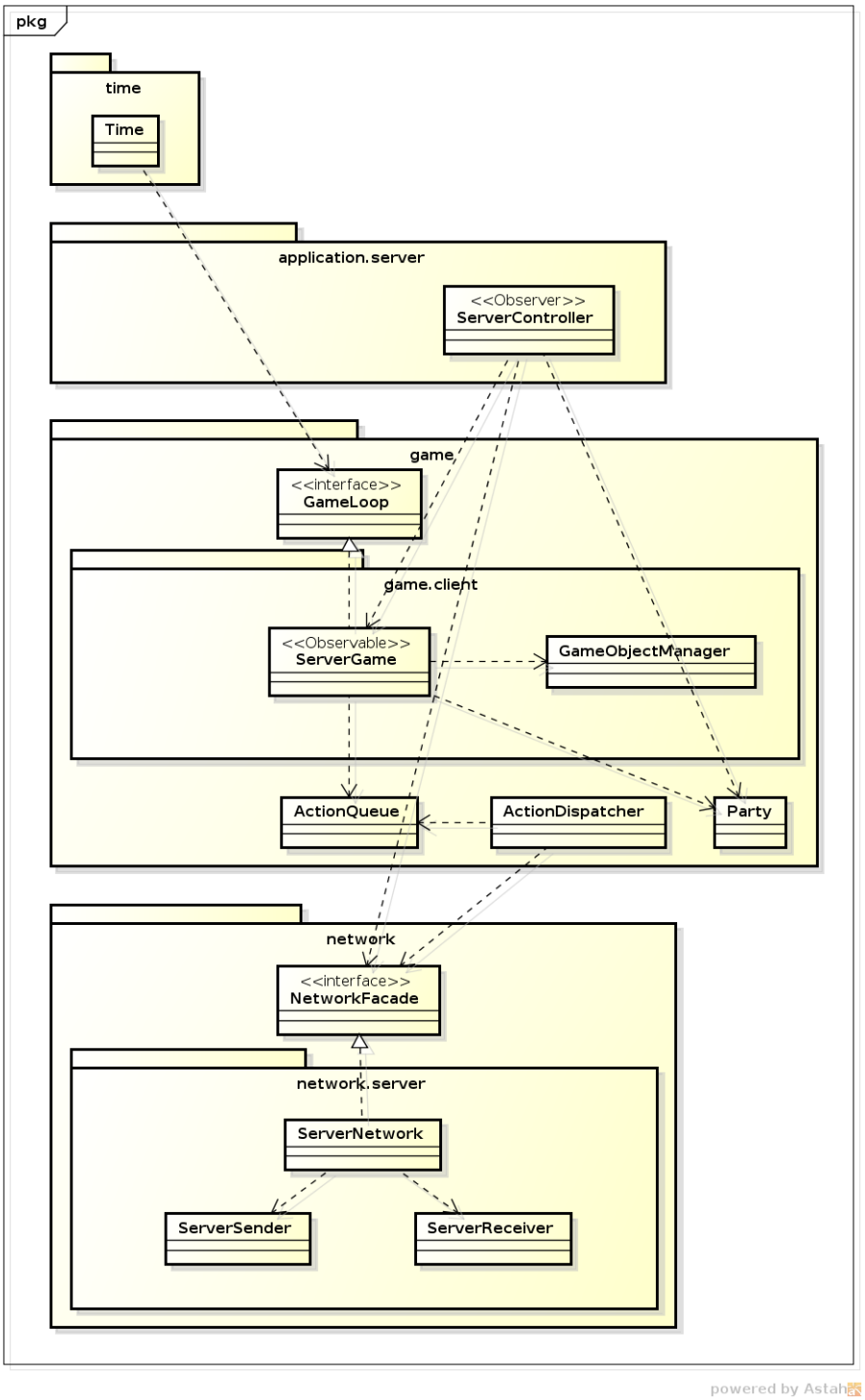
6.6.2 Sequenzdiagramm GameLoop Client

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.



7 Logische Architektur: Server

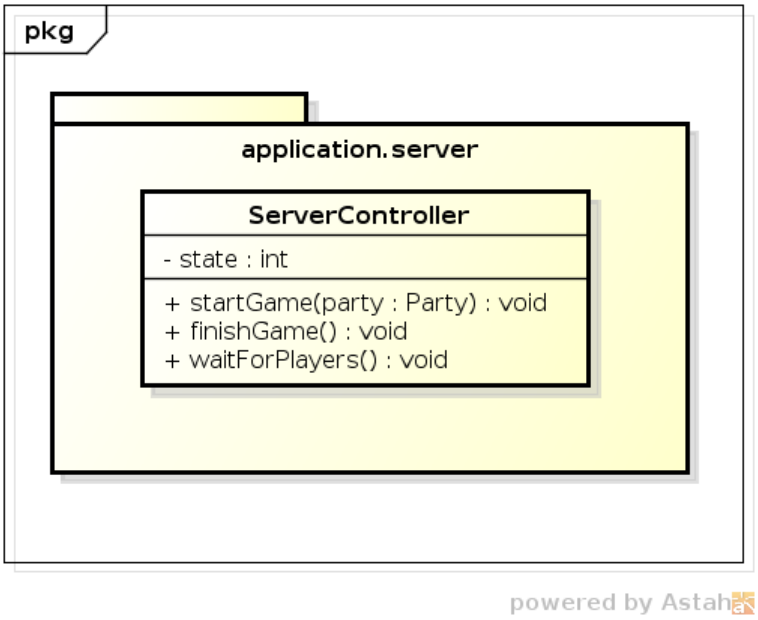
7.1 Schnittstellen



7.2 Workflow/application.server

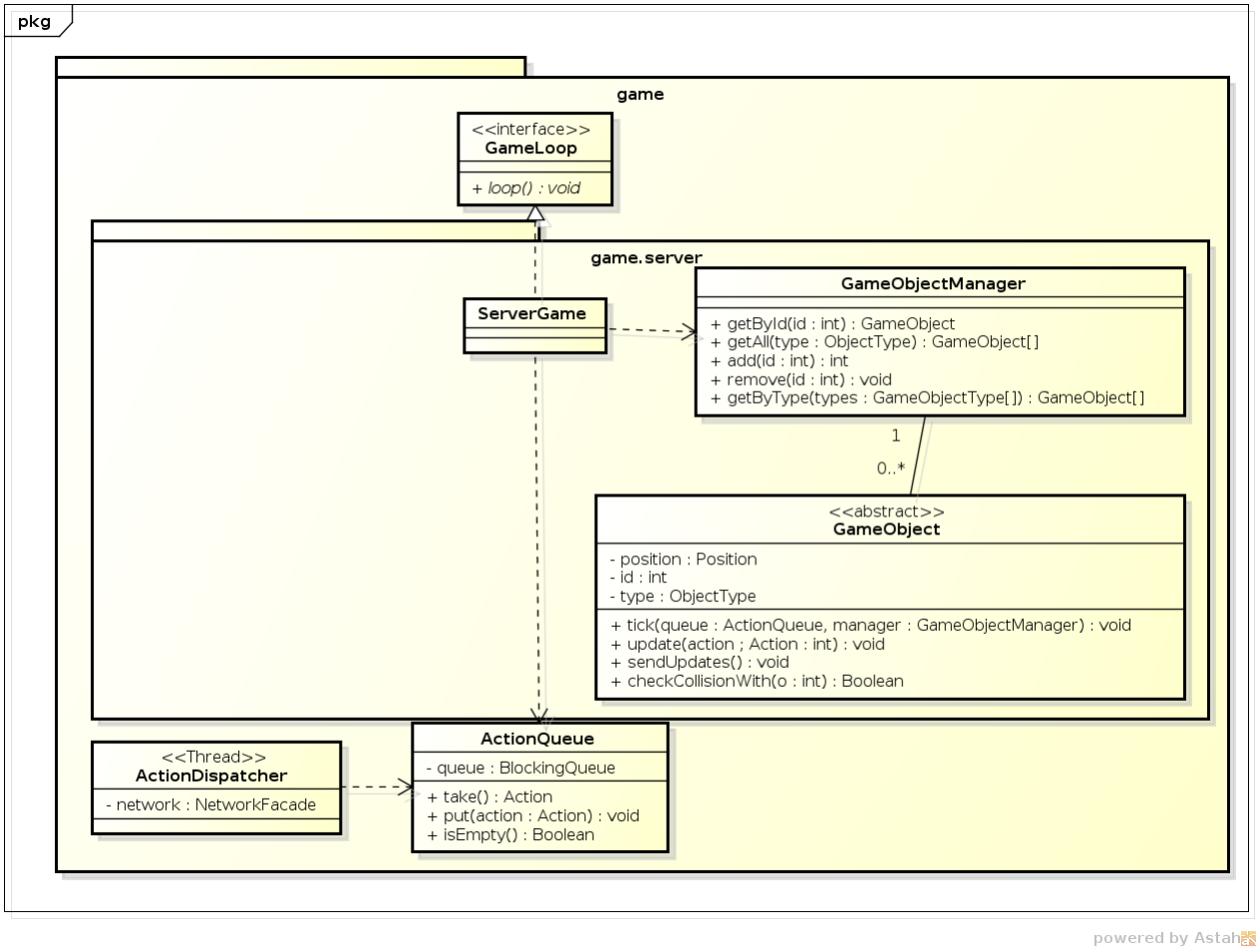
Der Workflow des Servers ist sehr simpel. Er wartet bis mehr als zwei Spieler verbunden und bereit sind. Darauf startet der Server dann das Spiel.

7.2.1 Klassenstruktur



7.3 Domain/game.server

7.3.1 Klassenstruktur



ServerGame

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

GameObjectManager

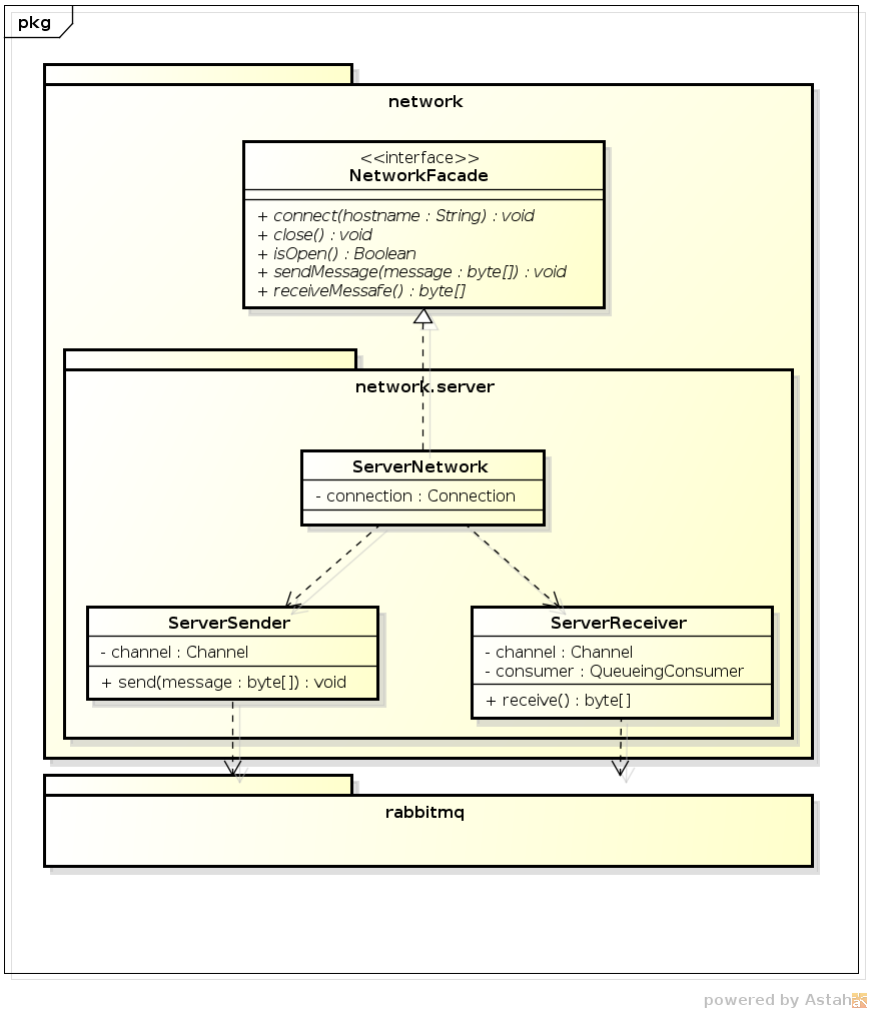
Der GameObjectManager speichert die GameObject nach Typen ab. Dies ist hilfreich, wenn die Objekte aktualisiert werden müssen. Die Methoden des GameObjectManagers sind ähnlich wie bei normalen Datenstrukturen und werden deshalb nicht weiter erläutert.

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

7.4 Network/network.server

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

7.4.1 Klassenstruktur



ServerNetwork
Die Klasse ServerNetwork implementiert die NetworkFacade.

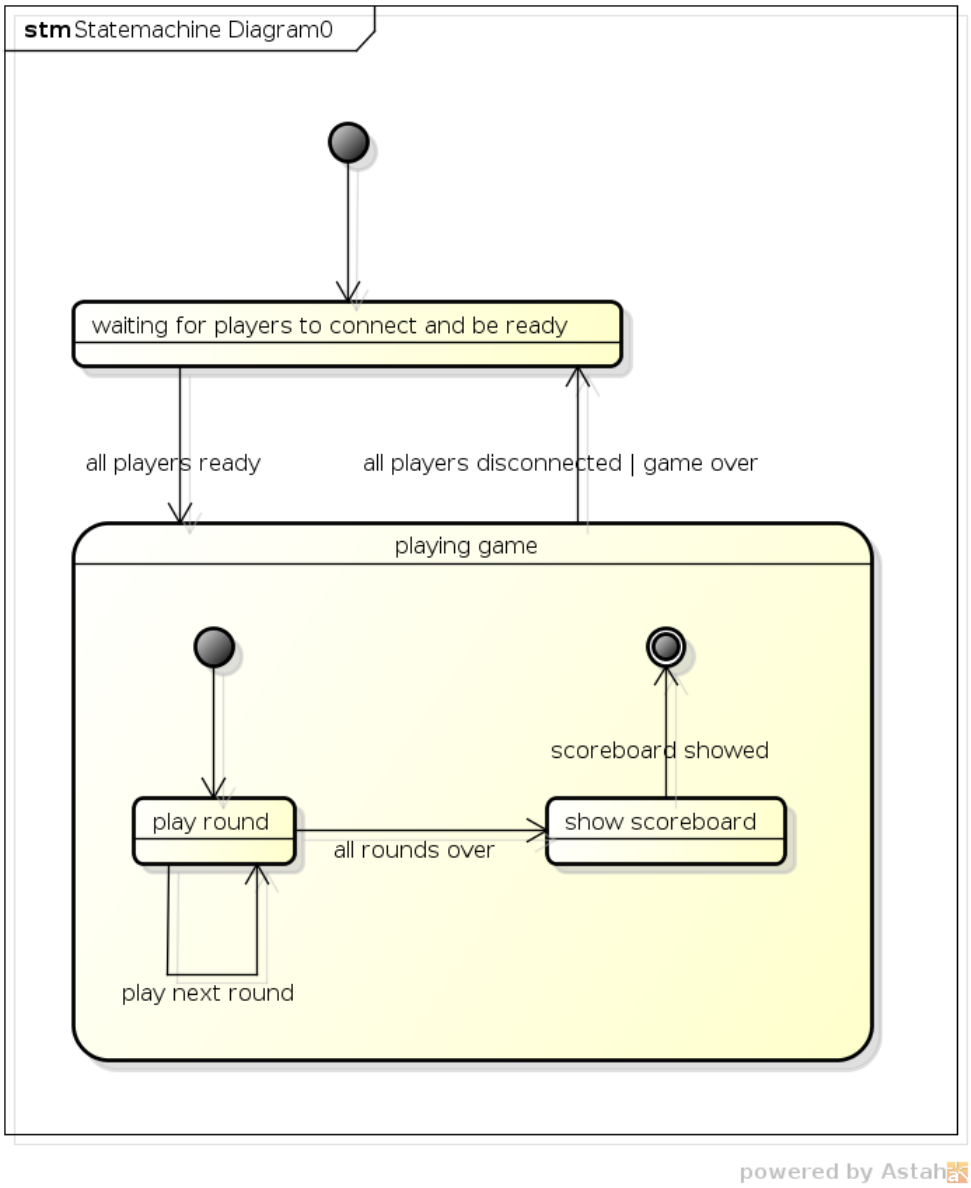
ServerSender
Der ServerSender sendet alle Daten die im übergeben werden direkt an alle angemeldeten Clients.

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

Clients senden.

7.5 Wichtige Abläufe

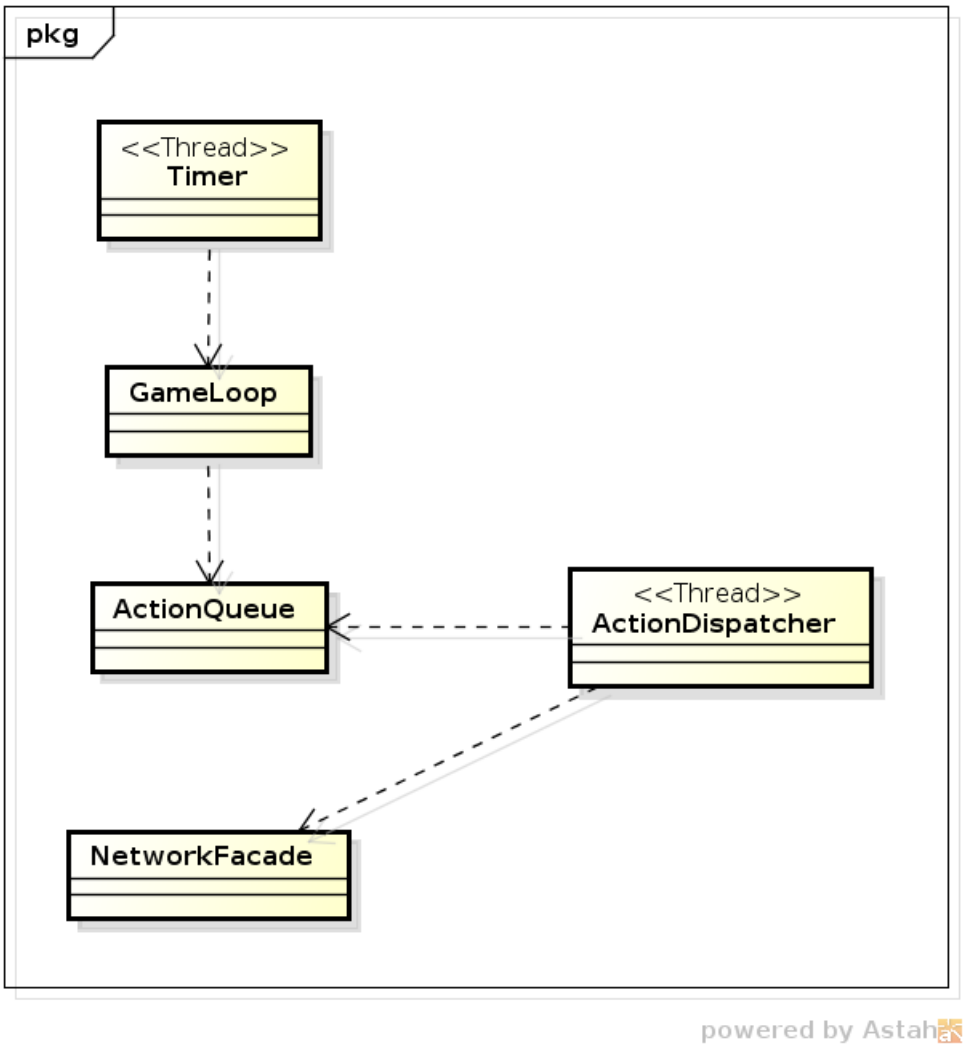
7.5.1 Zustandsdiagramm Server Workflow



8 Prozesse und Threads

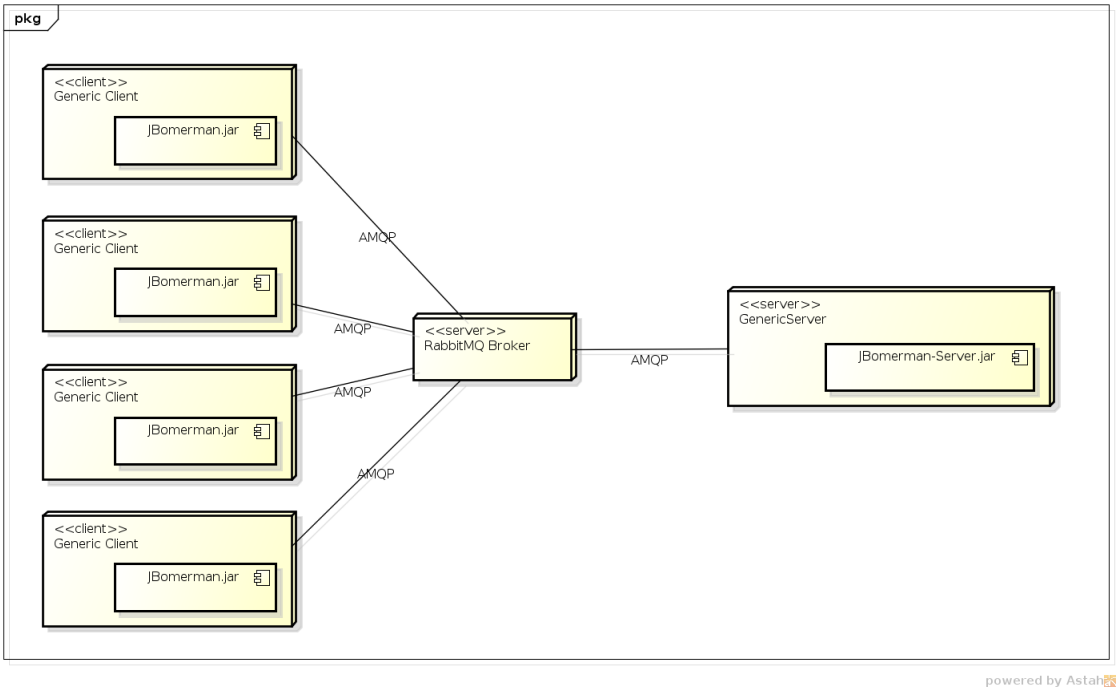
8.1 Dispatcher Thread

Der DispatcherThread(Klasse: ActionDispatcher) sorgt dafür, das 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 Threadsafe.



9 Deployment

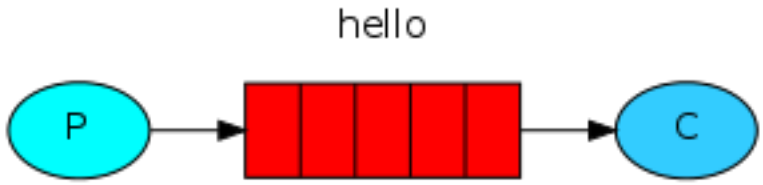
Das Messagerouting wird von einem RabbitMQ Broker übernommen, der meistens auf der gleichen Hardware wie der GameServer läuft, aber auch auf einer anderen Hardware betrieben werden kann. Die Übertragung läuft mittels AMQP (Advanced Message Queueing Protocol).



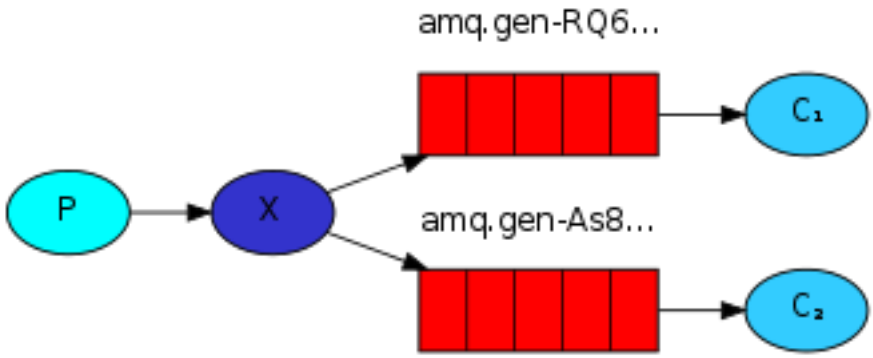
10 Datenübertragung

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)

alive

i	Klasse	Inhalt
0	int	PlayerId

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

i	Klasse	Inhalt
0	int	countdown

aliveCheck hat keine weiteren Properties

startGame

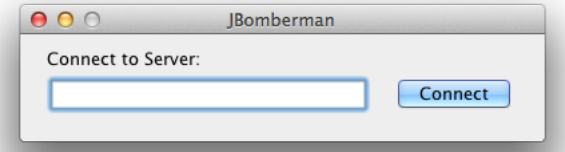
i	Klasse	Inhalt
0	String[][]	Party

11 Externes Design

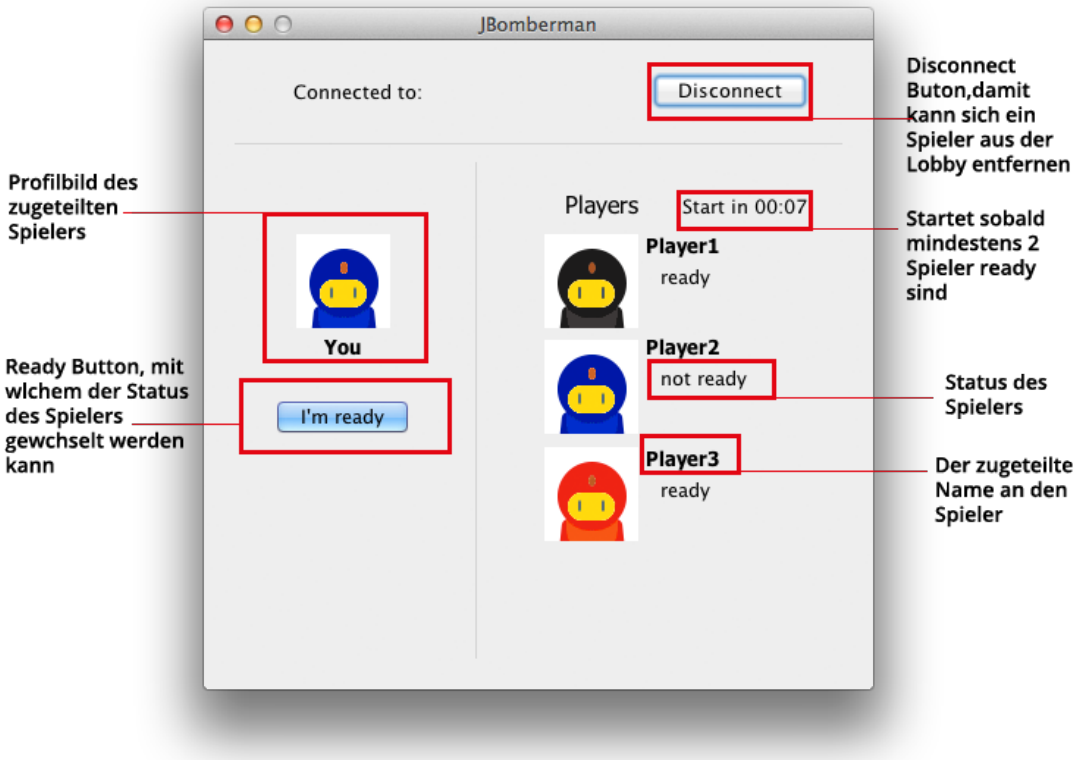
Es besteht lediglich ein GUI für den Client, der Server soll über das CLI gestartet werden.

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.

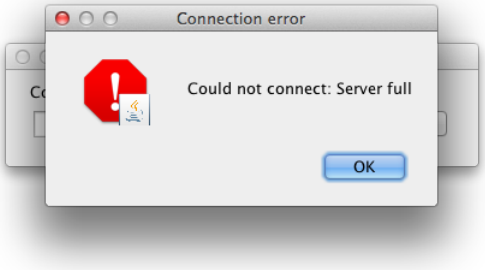


11.2 LobbyFrame



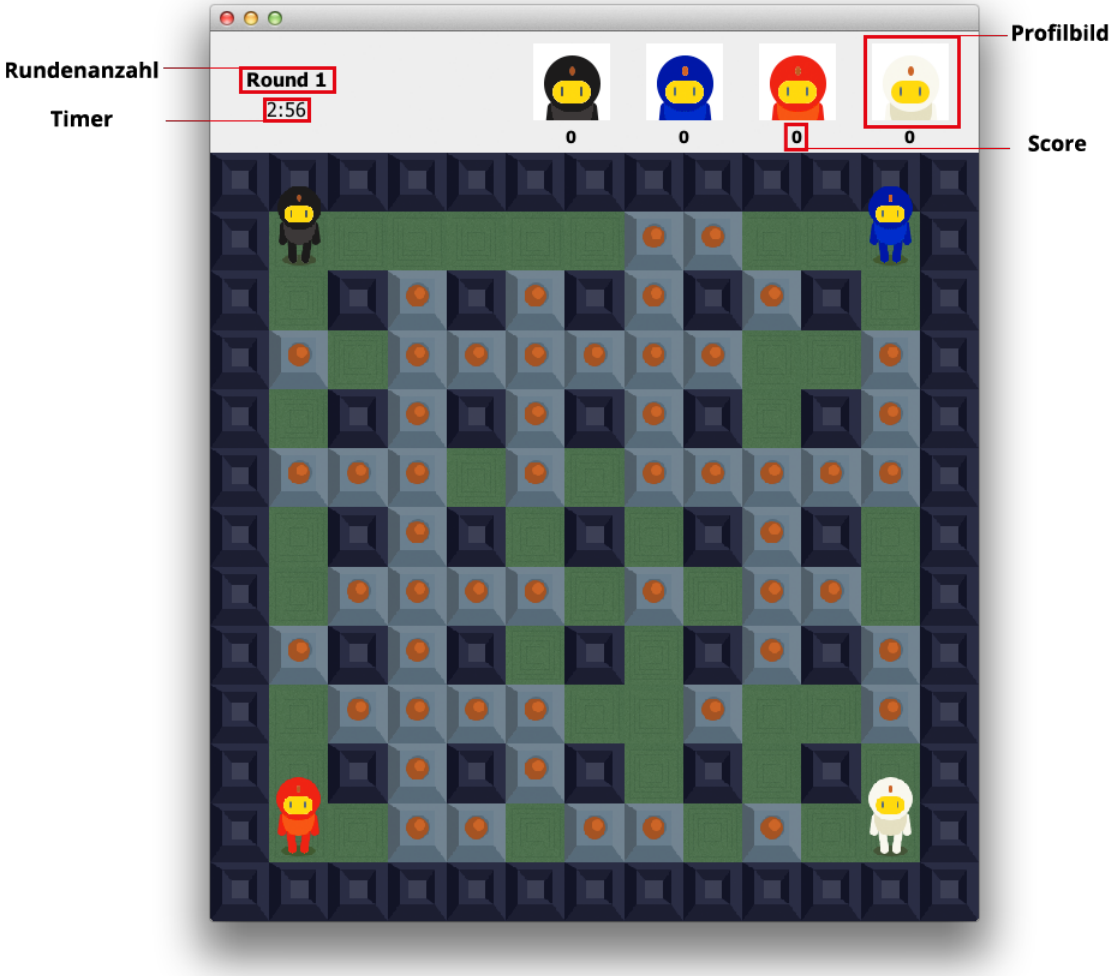
11.2.1 Fehlermeldungen

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



11.3 GameFrame

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



12 Grössen und Leistung

JBomberman 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.