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
05.04.15	1.01	Logische Architektur	Fabian Binna

Inhaltsverzeichnis

1 Änderungshistorie **2**

2 Einführung **4**
2.1 Zweck 4
2.2 Gültigkeitsbereich 4
2.3 Referenzen 4
2.4 Übersicht 4

3 Systemübersicht **4**

4 Architektonische Ziele & Einschränkungen **4**

5 Logische Architektur **5**

6 Logische Architektur Client **6**
6.1 Presentation/view 6
6.1.1 Klassenstruktur 6
6.2 Workflow/application.client 7
6.2.1 Klassenstruktur 7
6.3 Domain/game.client 8
6.3.1 Klassenstruktur 8
6.3.2 Wichtige interne Abläufe 9
6.4 Network/network.client 10
6.4.1 Klassenstruktur 10
6.5 Wichtige Abläufe 10

7 Prozesse und Threads **10**

8 Deployment **10**

9 Datenspeicherung **11**

10 Grössen und Leistung **11**

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

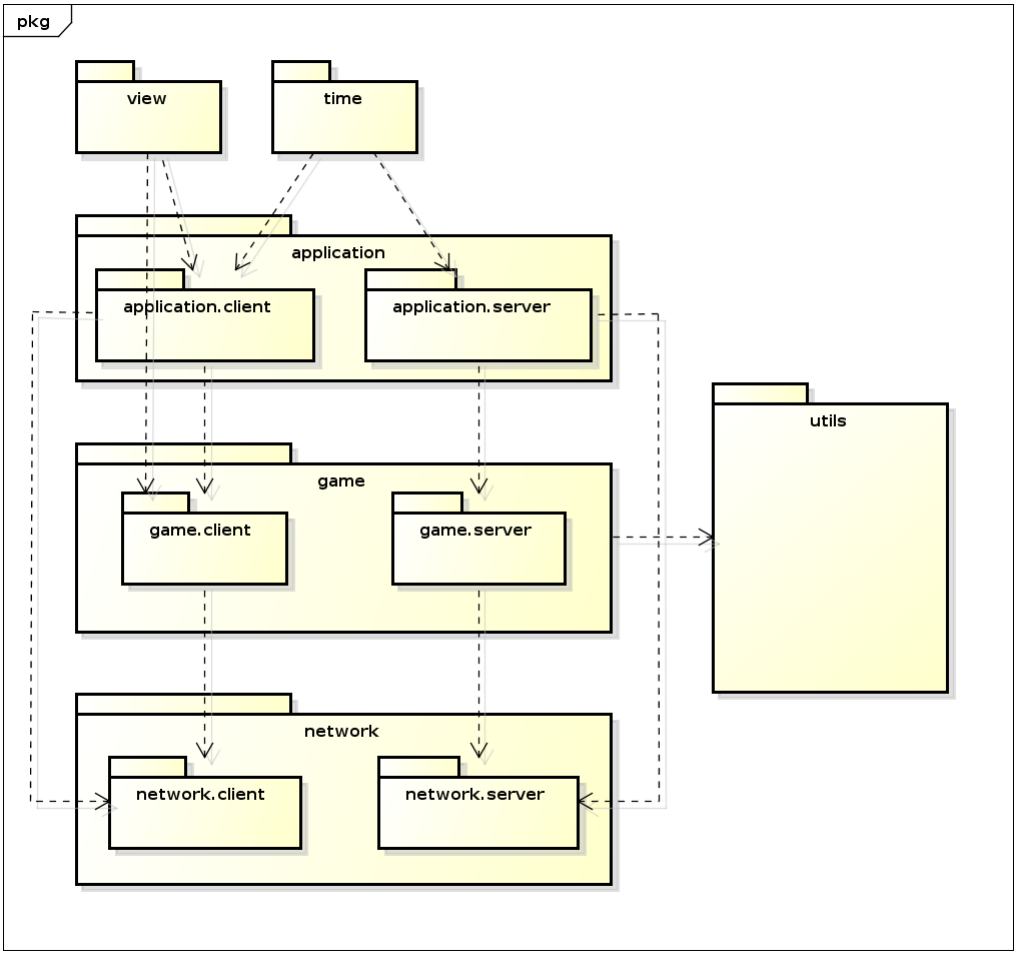
<Beschreibt die Softwarearchitektur eines Systems und wie sie sich präsentiert (am besten mit einem Bild um eine Übersicht zu ermöglichen) und einzelne Beschreibungen zu den einzelnen Elementen des Systems>

4 Architektonische Ziele & Einschränkungen

<Beschreibt die Softwareanforderungen und Objekte, welche einen Einfluss auf die Architektur haben (z.B. Safety, Security, Privacy, Distribution, usw.); Beinhaltet auch eine Beschreibung von Design und Implementationsstrategie, Entwicklungstools, usw.>

5 Logische Architektur

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.

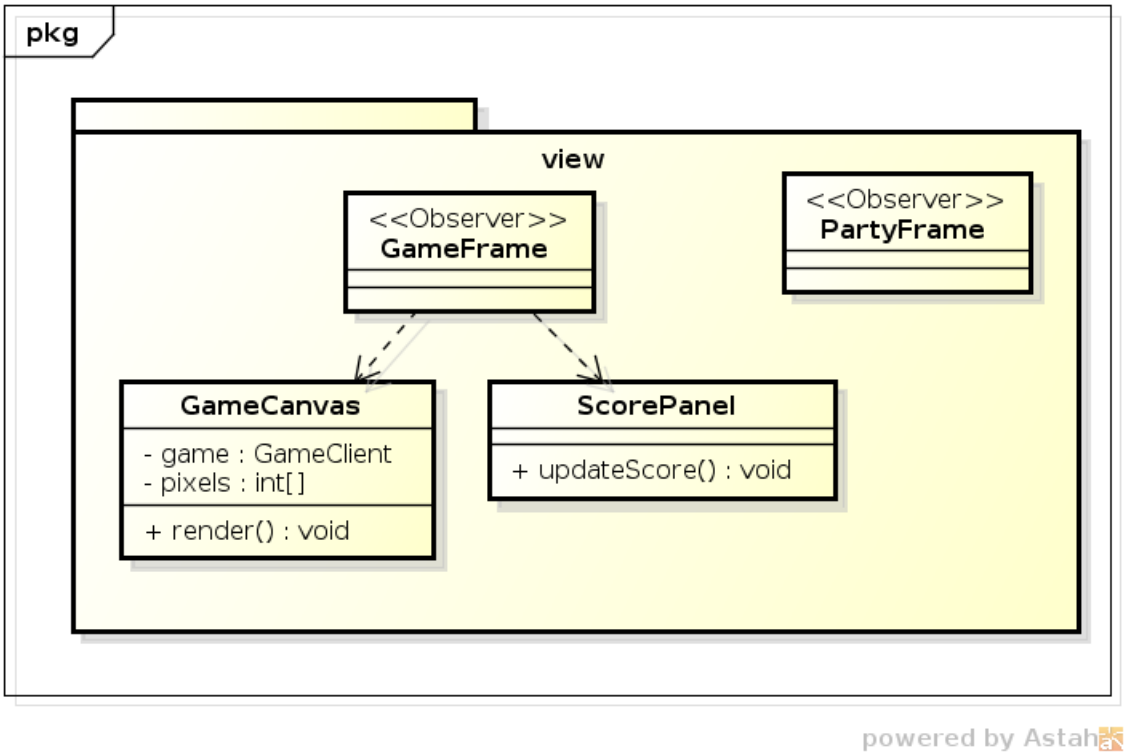


6 Logische Architektur Client

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.

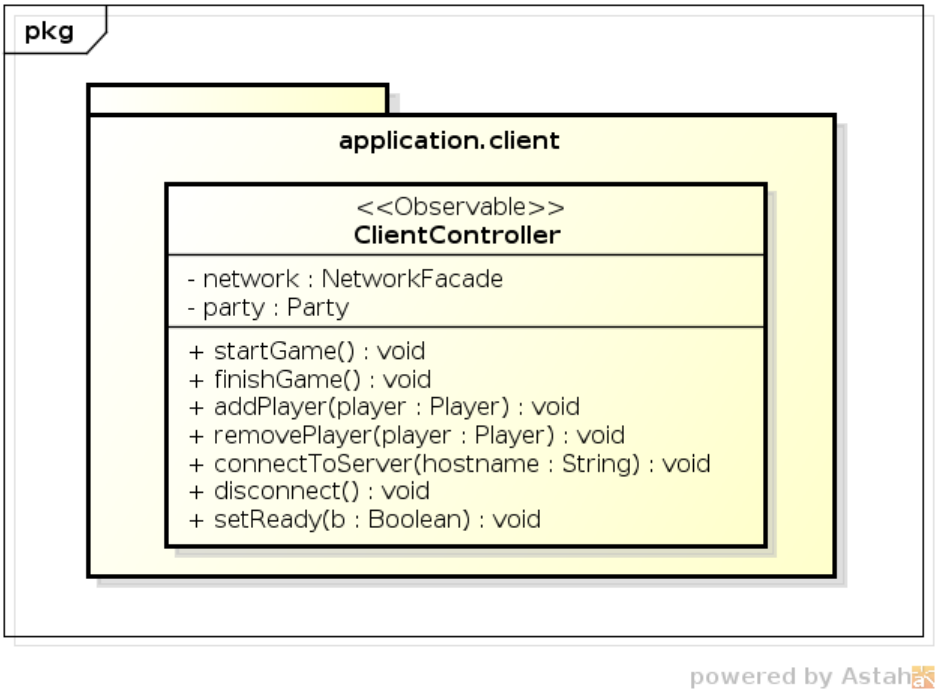
GameCanvas
Der GameCanvas kümmert sich nur um das Rendering, also das Zeichnen der Szene. Dabei delegiert er jedoch nur die pixels[] an alle Sprites, welche sich dann eigenständig zeichnen. Der GameCanvas kümmert sich dann um das performante Buffering.

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



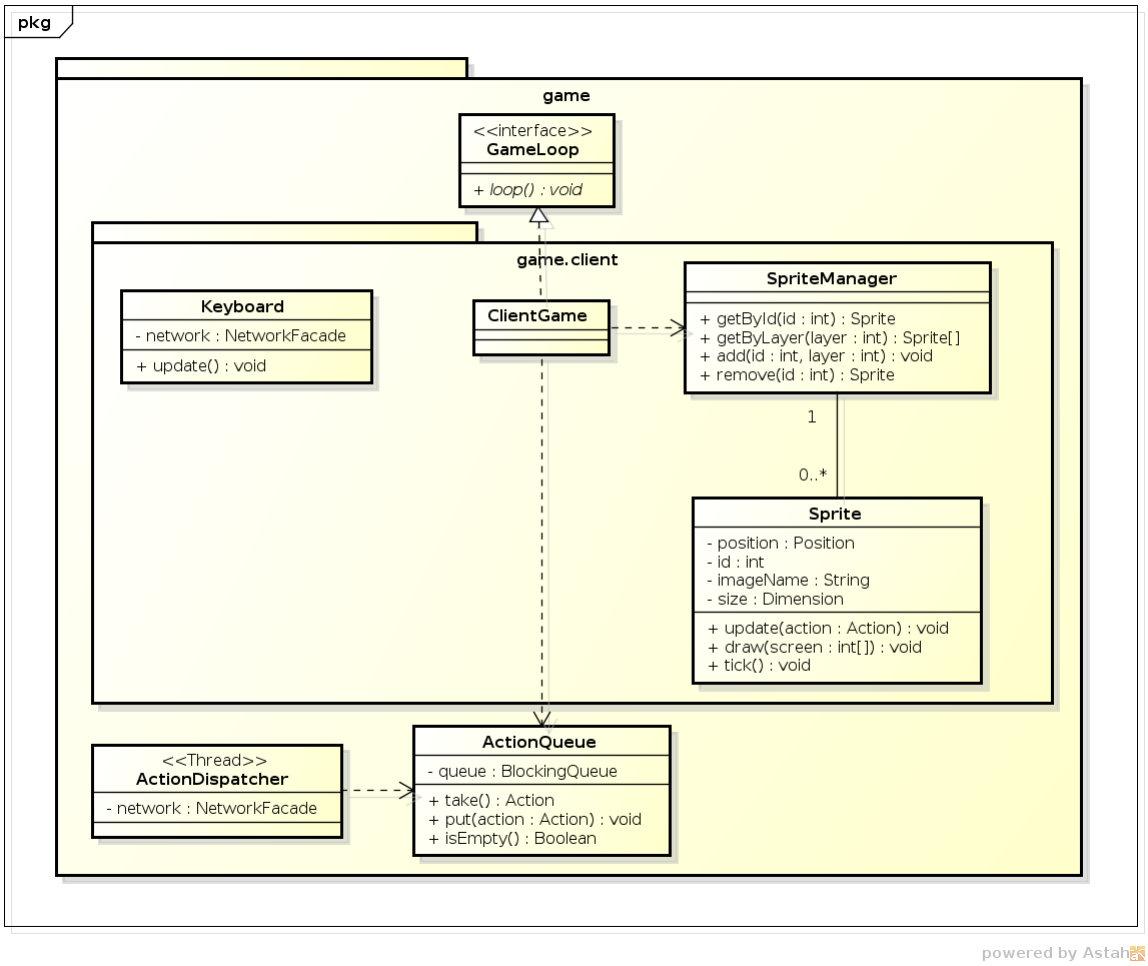
ClientController
Der ClientController kontrolliert den Zustand der Client-Applikation. Er notifiziert nur den PartyFrame, nicht aber den GameFrame.

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.

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

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.

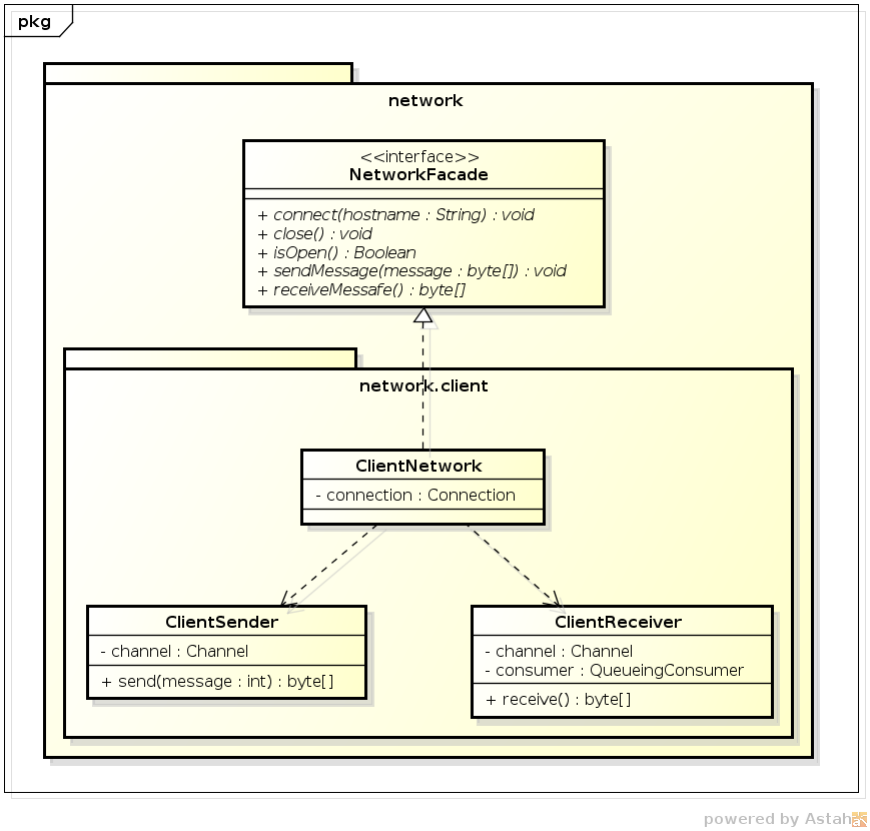
Keyboard
Das Keyboard zeichnet die Tastatureingaben auf und sendet diese direkt über die Methode update an den Server.

6.3.2 Wichtige interne Abläufe

//SSD zur GameLoop realisiertung einfügen.

6.4 Network/network.client

6.4.1 Klassenstruktur



6.5 Wichtige Abläufe

<Beschreibung von wichtigen Abläufen (packageübergreifend)>

7 Prozesse und Threads

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

8 Deployment

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

9 Datenspeicherung

<Beschreibung mit Diagramm der Datenspeicherung (Datenmodell, z.B. Datenbank)>

10 Grössen und Leistung

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