# Ein Programmierprojekt zur objektorientierten Programmierung mit Java: Lernziele, Teilaufgaben, Musterlösung

Studienarbeit

Célio Carreto

betreut von

Prof. Dr. Florian Matthes

Arbeitsbereich Softwaresysteme (4.02)
Technische Universität Hamburg-Harburg

29. Januar 1999

# Inhaltsverzeichnis

# Zusammenfassung

Diese Studienarbeit behandelt die Implementierung des internetbasierenden multi-player Spiels Q-Quadrat.
Der Anstoß zu dieser Arbeit war die Notwendigkeit Studierende das Faches Informatik für Ingenieure den erlernten Stoff anhand eines Programmierprojektes auch praktisch zu vermitteln.
Der Leser dieser Arbeit soll lernen, wie ein Client/Server-Problem mittels moderner Methoden analysiert und implementiert werden kann. Der Leser erfährt über die Möglichkeiten des Einsatzes der Modellierungssprache *UML* und erhält eine beispielhafte Implementierung eines Client/Server-Problems.
Ziel der Arbeit ist es eine Aufwandsabschätzung für ein Programmierprojekt, welches nachträglich von Studierenden erneut realisiert werden soll, abzugeben.
Im ersten Kapitel der Arbeit wird auf das Programmierprojekt und den in dieser Arbeit verwendeten Techniken eingegangen. Kapitel 2 dient der Analyse des Spieles. Dort werden die einzelnen Anwendungsfälle aufgezeigt. In Kapitel 3 wird das Design des Spieles betrachtet. Kapitel 4 zeigt einige Besonderheiten der Implementation auf. In Kapitel 5 betrachten wir das Gerüst für das Programmierprojekt und abschließend in Kapitel 6 wird ein Ausblick auf mögliche Erweiterungen bzw. Verbesserungen gegeben.

# Kapitel 1

# Einleitung

Der Anstoß zu dieser Arbeit war die Notwendigkeit Studierende das Faches Informatik für Ingenieure den erlernten Stoff anhand eines Programmierprojektes auch praktisch zu vermitteln. Dabei sollten folgende Lernziele im Vordergrund stehen:

- Erlernen des projektbezogenen Arbeitens
- Erwerben von praktischer Programmiererfahrung in der Programmiersprache java
- Verständnis von Datenstrukturen
- Einfache nebenläufige Programmierung
- Einfache verteilte Programmierung

Das in dieser Studienarbeit beschriebende Spiel Q-Quadrat ist für diese Punkte ein gutes Lernobjekt.
Die Studienarbeit bildet die Grundlage für das Programmierprojekt. Mittels der Analyse und der Implementierung des Spieles wurden zu lösende Probleme sichtbar und es konnte eine Abschätzung bezüglich des Aufwandes einer erneuten Implementierung durch Studierende gemacht werden. Hieraus ergibt sich ein Gerüst für das Programmierprojekts. Auf das Programmierprojekt selbst wird im Kapitel 5 eingegangen.

Neben dieser Aufwandsabschätzung, dient diese Studienarbeit ebenso als Beispiel und Grundlage für die Realisierung eines Client/Server-Modell für das Internet.

## 1.1 Spielursprung

Die ursprüngliche Idee des Spiels wurde von dem Mathematiker G. Keith Still 1979 während seiner College-Zeit entwickelt und 1996 im Scientific American veröffentlicht. In diesem Spiel setzen zwei Spieler abwechselnd Steine auf ein quadratisches, schachbrettartiges Spielfeld, mit dem Ziel, durch gezieltes Setzen der Steine ein Quadrat zu erstellen. Derjenige Spieler, der als erstes ein Quadrat erstellt, hat gewonnen. Um das Erstellen der Quadrate zu erschweren, hat jeder der Spieler ein gewisse Anzahl an Blockern, welche er zum Blockieren eines Feldes nutzen kann.
Im Jahre 1996 gab es eine Variante dieses Spieles im Onlinedienst AOL. Es hieß Metasquares und wurde von der Firma Metacreations geschrieben. Bei dieser Variante waren die Blocker weggefallen. Dafür mußte man solange Quadrate erstellen, bis eine gewisse Punktzahl erreicht war. Derjenige Spieler, der als erster diese Punktzahl erreicht, gewinnt.
In dieser Variante des Spieles werden beide oben vorgestellte Varianten miteinander kombiniert. Die Spieler müssen durch Erstellung von Quadraten eine gewisse Punktzahl erreichen und haben zum Blockieren von Feldern max. 3 Blocker zur Verfügung. Näheres dazu und zu den Spielregeln ist im Anhang A zu finden.

## 1.2 Verwendete Techniken

UML

Zur Modellierung des System wird in dieser Arbeit UML (Unified Modeling Language) genutzt. Dabei dient diese überwiegend grafische Notation dazu den Entwurf eines Systems auszudrücken. In dieser Arbeit werden verschiedene Diagrammtypen dafür verwendet. Diese werden in den entsprechenden Kapiteln eingeführt und erläutert. Nähere Informationen über UML ist z.B. in [FSc98] nachzulesen. Als Werkzeug wird Rational Rose verwendet, indem die Diagramme erstellt wurden.

### Java

Zur Implementation des Client/Server-Systems wird in dieser Arbeit die Sprache Java genutzt. Dabei erfolgt die Implementation in der Sprachversion 1.02, da diese Version von fast allen Browsern des Internets ohne zusätzliche Hilfsmittel unterstützt wird. Als Frontend kommt dabei ein Applet zu tragen. Auch das Backend (der Spielserver) ist in Java geschrieben. Zur Kommunikation werden Socketverbindungen mit entsprechenden Streams verwendet. Dies hat den Vorteil, daß man den Spielserver auch in einer anderen Sprache implementieren kann und trotzdem den Kommunikationsweg weiter nutzen kann. Einen groben Überblick über die Möglichkeiten der Kommunikation via Sockets und deren Implementation in Java ist in [MOS97] und [RSc98] zu finden.

## 1.3 Überblick über die weiteren Kapitel

In Kapitel 2 werden die Anwendungsfälle der einzelnen Benutzer betrachtet. Anwendungsfälle sind dabei Interaktionen zwischen den Benutzern und dem System. Diese werden in Anwendungsfalldiagramme dargestellt. Kapitel 3 zeigt die Modellierung des Systems mittels Klassendiagrammen. Dort wird der erste Schritt zur Implementierung getan. Kapitel 4 weist auf Besonderheiten der Implementation hin. In Kapitel 5 wird noch einmal auf das Programmierprojekt im einzelnen eingegangen und Kapitel 6 zeigt einen Ausblick auf Verbesserungen bzw. Erweiterungen des Systems.

# Kapitel 2

# Analyse

Die Analyse dient dazu, die Anwendungsfälle der einzelnen Benutzer zu betrachten. Anwendungsfällen sind Interaktionen zwischen Benutzer und dem System.
Im ersten Schritt der Analyse werden die Akteure (Benutzer) bestimmt. Danach werden für diese Akteure die Anwendungsfälle bestimmt und beschrieben.

## 2.1 Akteure

Nimmt man das klassische Q-Quadrat Spiel und stellt es sich als Brettspiel vor, so ergibt sich als ein Akteur der Spieler, welcher gegen einen anderen Spieler das Spiel Q-Quadrat spielt. Im Hinblick darauf, daß das Spiel über das Internet spielbar sein soll, ergibt sich als weiterer Akteur der Administrator, welcher für den reibungslosen Ablauf des Spieles verantwortlich ist.
Definitionen der einzelnen Akteure:

### Spieler

Ein Spieler ist eine Person, welche mit anderen Gleichgesinnten über das Internet sich im Spiel Q-Quadrat messen möchte, um ein möglichst hohes Ranking zu erhalten. Das Ranking bezeichnet damit die Spielstärke eines Spielers.

### Administrator

Ein Administrator ist eine Person, dessen Aufgabe es ist, den Spielserver funktionsfähig zu halten.

## 2.2 Anwendungsfälle

Nachdem die Akteure bestimmt sind, werden nun die Anwendungsfällen dieser betrachtet. Anwendungsfälle beschreiben Interaktionen zwischen den Benutzern und dem System. Zur Ermittlung der Anwendungsfälle, werden die Anforderungen der Benutzer an das System bestimmt. Die grafische Darstellung wird mittels Anwendungsfalldiagramme der Modellierungssprache *UML* verwirklicht. Abbildung 2.1 zeigt ein solches Diagramm. In diesem Diagramm werden die Akteure als Strichmännchen und die Anwendungsfälle als Ovale gezeichnet. Die Beziehung zwischen einem Akteur und einem Anwendungsfall wird mit dem Pfeil gekennzeichnet.

### 2.2.1 Anwendungsfälle (grob)

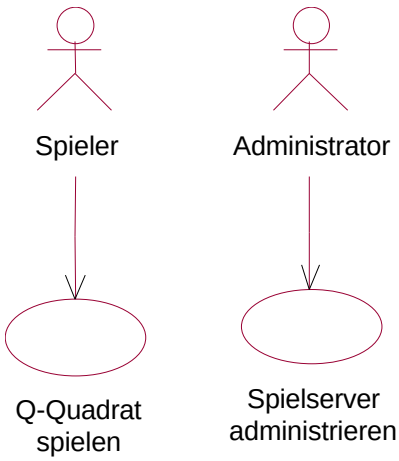


Abb. 2.1: Anwendungsfalldiagramm: Überblick

Zur Bestimmung der Anwendungsfälle interviewt man die Benutzer bezüglich der Erwartungen an das System. Auf diese Vorgehensweise wurde in diesem Fall verzichtet. Das System wurde ohne Mitarbeit der Benutzer entworfen. Die Anwendungsfälle ergeben sich aus der Analyse vergleichbarer Systeme im Internet.
Betrachtet man die Anwendungsfälle der Akteure, so kann man diese, wie in Abbildung 2.1 gezeigt, in zwei Anwendungsfälle zusammenfassen.
Den Anwendungsfall des Spielers kann man mit *'Q-Quadrat spielen'* betiteln. Dem Spieler geht es in erster Linie um das Messen des eigenen Könnens gegen andere Gleichgesinnte. Der Anwendungsfall des Administrators wird mit *'Spielserver administrieren'* betitelt. Der Administrator muß einen reibungslosen Ablauf des Spielens garantieren.

### 2.2.2 Anwendungsfälle (Detail)

Die im Abschnit 2.2.1 betrachteten Anwendungsfällen, beschreiben nur grob die Interaktion zwischen den Akteuren und dem System. Im Zuge der Analyse können die Anwendungsfälle feiner spezifiziert werden. Die Detailtiefe ist dabei von dem Programmierer selbst bestimmbar. Analysiert man das System tiefer und baut man die weiteren Voraussetzungen in die Diagramme ein, so kommt man für den Spieler zu der Abbildung 2.2 und für den Administrator zur Abbildung 2.3.

Spieler:

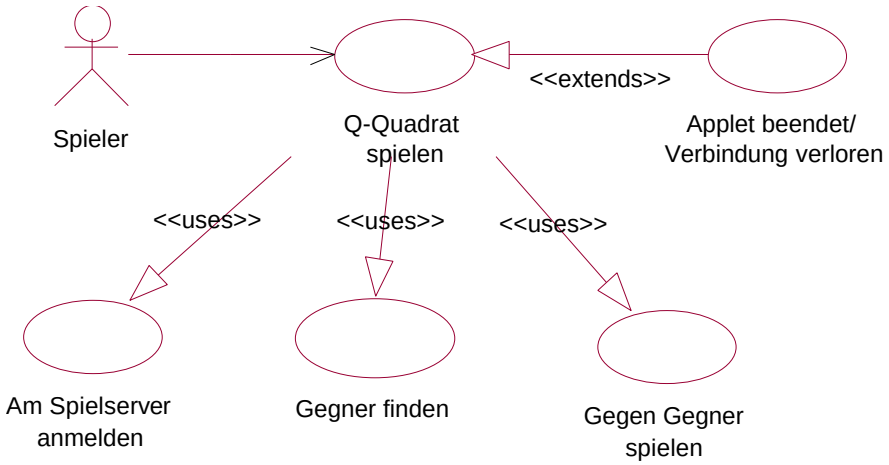


Abb. 2.2: Anwendungsfalldiagramm: Spieler

Im Gegensatz zu dem in Abschnitt 2.1 generellen Anwendungsfall 'Q-Quadrat spielen', ist in diesem Abschnitt dieser Anwendungsfall weiter aufgeteilt worden. Diese Anwendungsfälle stammen aus weiteren Überlegungen zu den Anforderungen an das System. Bei diesen Anforderungen handelt es sich schon um solche, welche man für die Realisierung des Systems benötigt. Folgende erweiterte Anforderungen werden von dem System verlangt:

- der Spieler soll eindeutig identifizierbar sein → Am Spielserver anmelden
- der Spieler soll aus eine Liste von Spielern sich den Gegner aussuchen können → Gegner finden
- falls das Applet geschlossen wird, soll der Spieler aus dem System entfernt werden → Applet beendet/Verbindung verloren

Aus diesen erweiterten Anforderungen an das System ergibt sich die Abbildung 2.2. In dieser Abbildung sind zwei neue Beziehungsarten (Pfeiltypen) hinzugekommen. Zum einen die *'extends'*- und zum anderen die *'uses'*- Beziehungsart. Die *'extends'*-Beziehung dient zum Zeigen, daß ein Anwendungsfall einem anderen ähnlich ist, aber diesen noch um einen gewissen Teil erweitert. Die *'uses'*-Beziehung wird verwendet, wenn ein Verhaltensanteil in verschiedenen Anwendungsfällen gleich ist oder ein Anwendungsfall so komplex ist, daß er separat beschrieben werden sollte.
In Abbildung 2.2 wird der Anwendungsfall *'Am Spielserver anmelden'* in einer *'uses'*-Beziehung mit dem Anwendungsfall *'Q-Quadrat spielen'* genutzt. Dies dient auf der einen

Seite zum besseren Verständnis bezüglich der Anforderungen an das System und zum anderen kann man diesen Anwendungsfall auch z.B. für Besucher nutzen, welche die Rankingliste des Systems Abfragen möchten.
In Abbildung 2.2 findet man auch eine *'extends'*-Beziehung. Diese soll darauf hinweisen, daß der Spezialfall *'Applet beendet/Verbindung verloren'* auch betrachtet werden muß. Dieser Anwendungsfall erweitert den *'Q-Quadrat spielen'*-Anwendungsfall um diesen Spezialfall. Daher wird eine *'extends'*-Beziehung genutzt.
In Anhang B wird die Abbildung 2.2 noch einmal um weitere Anwendungsfälle erweitert. Ebenso findet man dort eine detaillierte Beschreibung dieser Anwendungsfälle.

Administrator:

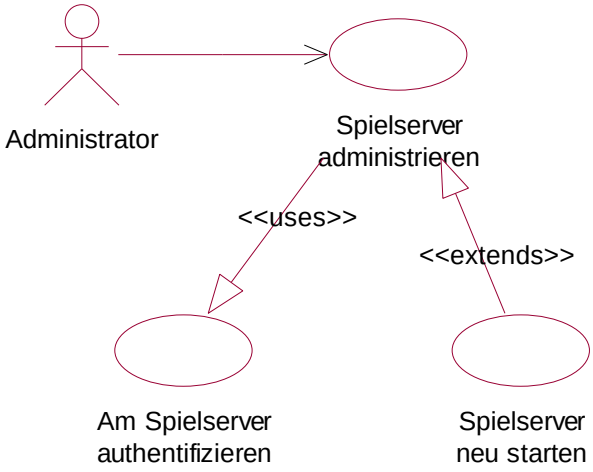


Abb. 2.3: Use Case Diagramm: Administrator

Abbildung 2.3 zeigt die Anwendungsfälle des Administrators. Auch in dieser Abbildung werden sowohl *'extends'*-Beziehung wie auch *'uses'*-Beziehung genutzt. Hier separat aufgezeigt wurde der *'Am Spielserver authentifizieren'*-Anwendungsfall, da dieses ein Problem bei der Implementation für das System stellen kann. Schließlich muß das System unterscheiden, ob der Benutzer Admin-Rechte besitzt oder ein Spieler, bzw. Besucher ist.
Ebenso wurde der Anwendungsfall *'Spielserver neu starten'* noch einmal getrennt aufegezeigt, da auch dieses in Betracht gezogen werden muß.
Detaillierte Informationen zu den einzelnen Anwendungsfällen sind im Anhang B zu finden.

## 2.3 Zusammenfassung der Analyse

In diesem Kapitel wurden die Anwendungsfälle der einzelnen Akteure betrachtet. Als Akteure wurden hier der Spieler und der Administrator betrachtet. Für jeden dieser Akteure wurden seine speziellen Anwendungsfälle betrachtet. Zusammengefaßt ergeben sich folgende Anwendungsfälle:

Spieler:
- möchte Q-Quadrat über das Internet spielen
- meldet sich am Spielserver an
- sucht und findet einen Gegner und baut einen Spielsitzung auf
- spielt gegen den Gegner
- beendet das Spiel/Applet

Administrator:
- meldet sich am Spielserver an
- administriert den Server
- meldet sich vom Spielserver ab

Die hier betrachteten Anwendungsfälle repräsentieren die Funktionalitäten des zu modellierenden Systems und bilden die Grundlage für die weiteren Schritte. Im nächsten Kapitel werden diese Anwendungsfälle zu Klassen modelliert, welche wiederum im darauf folgenden Kapitel implementiert werden.

# Kapitel 3

# Design

In Kapitel 2 haben wir die Anwedungsfälle des Systems betrachtet, um daraus zu erkennen, welche Anforderungen an das System gestellt werden. In diesem Kapitel wird das Design des Systems beschrieben. Dabei möchten wir die Anwendungsfälle des Kapitels 2 in passende Klassen, mit entsprechenden Objekten, Attributen und Methoden, umwandeln. Für diesen Zweck nutzen wir Klassendiagramm. Ein Klassendiagramm beschreibt die Typen von Objekten im System und die verschiedenen Arten von statischen Beziehungen zwischen diesen. Zusätzlich zeigen Klassendiagramme auch die Atrribute und Operationen der einzelnen Klassen.
Die hier vorgestellten Klassendiagramme sind sehr nahe an die Implementierung angelehnt, da die Eckdaten des System zu Beginn des Projektes bereits bekannt waren. Diese Eckdaten sind:

- Clients sollen als Applets realisiert werden.
- Der Spielserver soll in java realisiert werden.

## 3.1 Spielerapplet

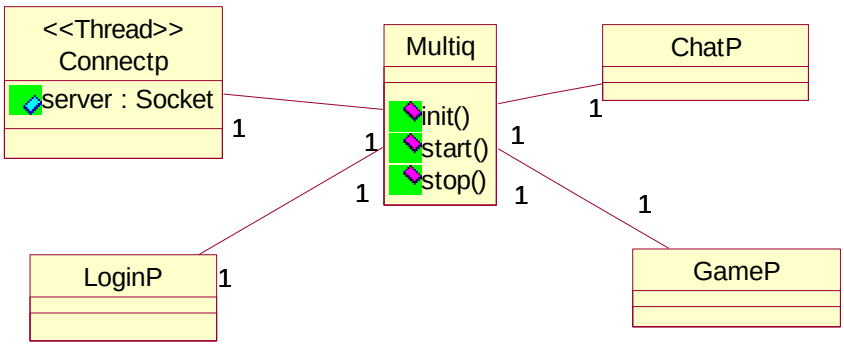


Abb 3.1: Klassendiagramm: Spielerapplet

Aus Kapitel2 sind die Anwendungsfälle des Spielers bekannt. Diese sind:

- möchte Q-Quadrat über das Internet spielen
- meldet sich am Spielserver an
- sucht und findet einen Gegner und baut einen Spielsitzung auf
- spielt gegen den Gegner
- beendet das Spiel/Applet

Btrachtet man diese Anwendungsfälle und denkt dabei an die Implementation dieser, so ergibt sich daraus, daß der Spieler 3 unterschiedliche Ansichten benötigt. Diese ergeben folgende Klassen:

- *LoginP* = Spielr meldet sich am Spielserver an
- *ChatP* = sucht und findet einen Gegner und baut einen Spielsitzung auf
- *GameP* = spielt gegen den Gegner

Um alle Klassen zu verwalten wird die Klasse *Multiq* genutzt. Um mit dem Spielserver kommunizieren zu können, wird eine Kommunikationsklasse benötigt. Hier *ConnectP* genannt. Da diese Kommunikation über Sockets mittels Streams realisiert werden soll, ergibt sich die Notwendigkeit, daß diese Klasse als *Thread* realisiert wird, welche als Attribut die Socket besitzt. Ebenso ist aus den Eckdaten bekannt, das der Client als Aqpplet realisiert werden soll. Daraus ergibt sich, daß *Multiq* die Operationen *init()*, *start()* und *stop()* besitzt. Wird die Implementation witer als Grundlage für das Klassendiagramm genommen, so kann das Diagramm um weitere Punkte erweitert werden. Das vollständige Diagramm und die Beschreibung diese ist in Anhang C zu finden.

## 3.2 Administratorapplet

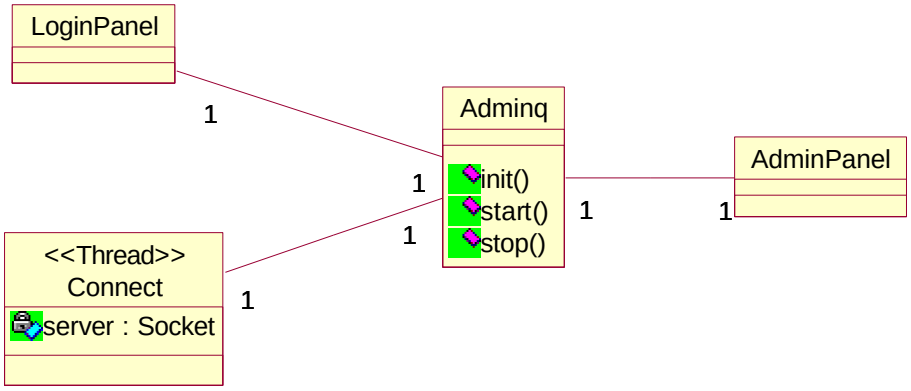


Abb. 3.2 Klassendiagramm: Administartorapplet

Für das Administratorapplet gelten die selben Überlegungen, wie für das Spielerapplet. Das Administartorapplet hat genau wie das Spielerapplet unterschiedliche Ansichten, welche als eigene Klassen moduliert wurden. Dazu gehört die Anmelde-Ansicht (*LoginPanel*) und die Administrationsansicht (*AdminPanel*). Ebenso wird eine Kommunikationsklasse (*Connect*) und eine Verwaltungsklasse (*Adminq*) benötigt. Auch hier führt eine Überlegung bezüglich der Implementation zu weiteren Attributen und Operationen. Diese sind in Anhang C beschrieben und dargestellt.

## 3.3 Spielserver

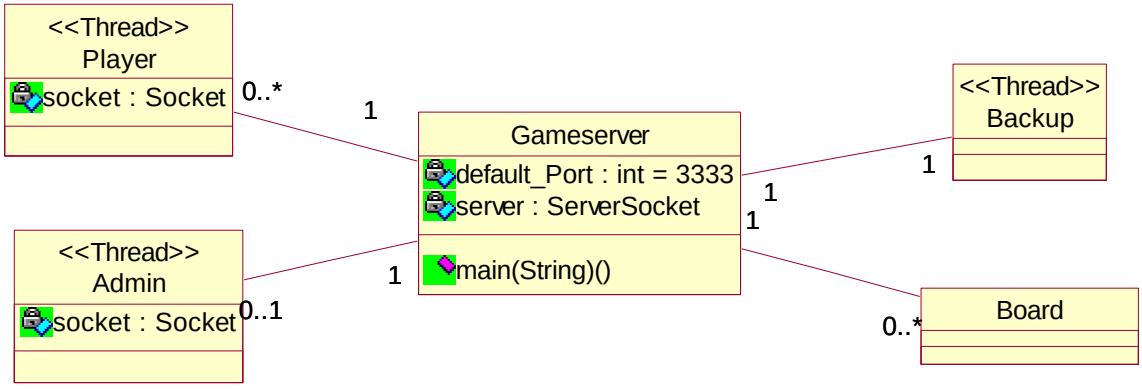


Abb. 3.3: Klassendiagramm: Spielserver

Abbildung 3.3 zeigt das Klassendiagramm für den Spielserver. Auch dieses ist an die Implementation angelehnt. Aufgrund der Realisierung mittels Sockets und entsprechenden Streams, benötigt der Spielserver für die Kommunikation Threads für den Administrator und die Spieler. Diese unterscheiden sich in der Interpretirungslogik der ankommenden Strings. Daher sind diese in zwei unterschiedliche Klassen aufgeteilt. Zusätzlich wurde eine Spiellogik eingebaut, welche die Spielzüge an den einzelnen Tischen auf Richtigkeit überprüft (*Board*). Ein weiterer Thread wurde hizugefügt (Backup), welcher die Spielerdaten nach einem gewissen Zeitraum in eine vorgegebene Datei speichert. Als Attribut besitzt die Klasse *Gameserver* die Serversocket. Das ist die Socket, an der der Spielserver auf neue Clients wartet. Eine detailliertere Darstellund der Klassen ist in Anhang C zu finden.

## 3.4 Interaktionsdiagramm - Konkurrierende Herausforderung

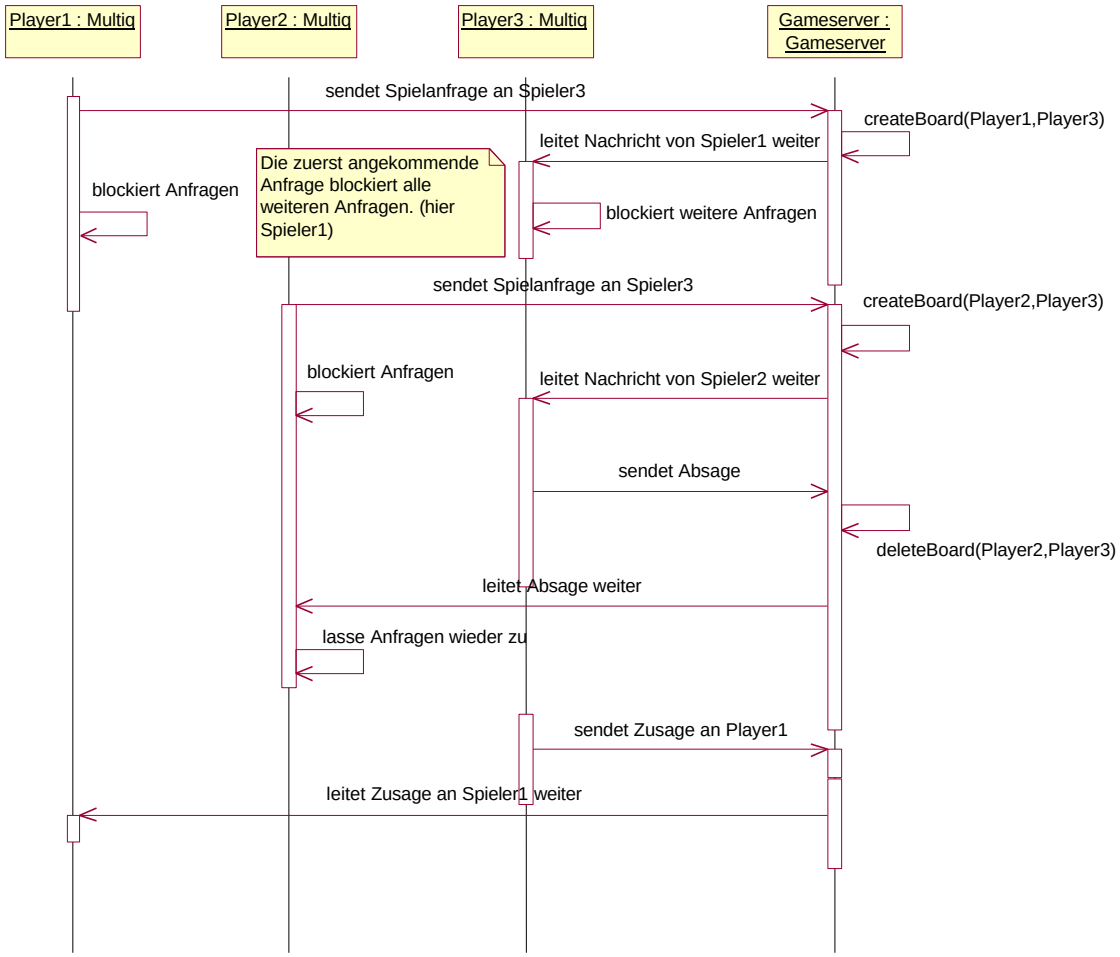


Abb. 3.4 Sequenzdiagramm: Konkurrierende Herausforderungen

Abbildung 3.4 zeigt ein Sequenzdiagramm. Mittels dieses Sequenzdiagramm soll erläutert werden, wie eine Herausforderung abläuft. Sobald ein Spieler eine Herausforderung abschickt oder eine Herausforderung bekommt, wird dieser für weitere Herausforderungen blockiert. Dieser Zustand ändert sich erst, falls er eine Absage bzw. Zusage bekommt oder er die Herausforderung zurücknimmt.

## 3.5 Zusammenfassung

Die in diesem Kapitel dargestellten Diagramme, dienen als Grundlage für die Implementation. Nicht alle der dargestellten Klasse werden auch separat implementiert. Einige werden in anderen Klasse integriert. Kapitel 4 zeigt die Implementation und einige Besonderheiten dieser.

# Kapitel 4

# Implementierung

In diesem Kapitel wird die Implementation und deren Besonderheiten beschrieben. Als Grundlage für die Implementation dienen die im Kapitel 3 erstellten Klassendiagramme.
Als Implementationssprache wird wie in Kapitel 1 beschrieben die Sprache Java in der Version 1.02 genutzt.

## 4.1 Spielerapplet

Aus dem Klassendiagramm 3.1 im Kapitel 3 ist ersichtlich, daß das Spielerapplet aus folgenden Klassen besteht.

- Multiq - Die Applet-Klasse
- Connectp - der Thread, welcher die Verbindung zum Server aufrecht hält
- Loginp - das Anmeldefenster
- Chatp - das Chatfenster
- Gamep - das Spielfenster

Zur Verbesserung der Attraktivität des Spieles werden Sounddateien verwendet. Ist dieses der Fall, so muß ausgeschlossen werden, daß mehrere unterschiedliche Programme auf das Gerät der Soundausgabe gleichzeitig zugreifen. Der gleichzeitige Zugriff mehrerer Quellen auf die Soundausgabe, hat jedenfalls unter Windows95 die Folge, daß das Applet im Browser abstürzt und nicht mehr reagiert. Um diesem entgegenzuwirken wird bei der Implementation eine Sounddatei genutzt, die einen nicht hörbaren Ton beinhaltet. Diese Sounddatei wird über den Methode *loop()* permanent an das Soundausgabegerät geschickt und blockiert somit den Zugriff anderer Programme. Diese Sounddatei wird sofort nach dem Laden des Applets gestartet.

Zusätzlich zu den Sounddateien werden auch Grafiken genutzt, welche normalerweise bei Start geladen werden. Da die Schnelligkeit der Internetverbindungen von einzelnen Spieler nicht besonders hoch ist, ist die Dauer zum Ladens der entsprechenden Dateien häufig lang und Spieler könnten das Laden des Spieles abbrechen. Um diesem entgegenzuwirken, werden die Grafiken erst dann geladen, wenn die Spieler sich schon in der Chatansicht befinden. So können sich die Spieler unterhalten und die Wartezeit der Spieler wird überbrückt.

Wie aus dem Klassendiagramm 3.1 ersichtlich, besteht das Applet aus verschiedenen Ansichten. Diese sind von der Klasse *java.awt.Panel* abgeleitet. Das Anzeigen bzw. Ausblenden einzelner Panels erfolgt mittels der Befehle *show()* bzw. *hide()*. Dies muß in der Hauptklasse (*Multiq*) realisiert werden.
Das Login-Panel (*LoginP*) dient zum Anmelden an dem Spielserver. Dies wurde mittels zweier *Textfields* realisiert, die auf maximal 10 Zeichen eingeschränkt sind.
Hat der Spieler sich angemeldet, so wechselt dieser in das Chat-Panel (*ChatP*). Dort kann der Spieler sich mit anderen Spielern unterhalten oder einen Spieler herausfordern. Das Herausfordern erfolgt durch Doppelklicken auf eine Spielernamen aus der Liste spielbereiter Spieler. Zu beachten ist, daß diese Spielerliste als *List* realisiert wurde und permanent aktualisiert wird. Bei der Implementation stellte sich heraus, daß der gleichzeitige Aktualisieren der Liste von mehreren Methoden einen Absturz verursachen, so daß die Methode *aktualisieren() synchronized* sein muß.
Das Game-Panel (*Gamep*) dient zur Anzeige des Spielfeldes und Entgegennahme von Spielzügen des Spielers.
Die Klasse *ConnectP* ist mit die wichtigste Klasse im Applet. Diese Klasse ist als Thread realisiert, welcher die Verbindung zum Spielserver öffnet und dann am Eingang permanent auf neue Nachrichten lauscht. Als ausgehenden Stream nutzen wird der *PrintStream* genutzt, welcher auf einfache Art erlaubt Nachrichten zu versendet. Dafür wird die Methode

*println()* genutzt, welche die Nachricht mit einem Zeilenvorschub abschließt. Dieses kommt einlesenden Stream, welcher als *DataInputStream* realisiert ist, entgegen, da dieser Streamtyp die Methode *readLine()* beinhaltet, welche immer eine Mitteilung genau bis zum Zeilenvorschub ließt. Dadurch ist stets bekannt, wo eine Nachricht aufhört und wann die neue Nachricht anfängt. Die ankommenden Nachrichten werden grob analysiert und dann an die entsprechenden Ansichten weitergeleitet.

## 4.2 Administratorapplet

Im Administratorapplet werden nur Daten angezeigt bzw. Kommandos an den Server geschickt. Von daher wird auf Grafiken und Sounds verzichtet. Das Administratorapplet besteht aus vier Klassen:

- Adminq - die Applet-Klasse
- LoginPanel - das Anmeldefenster
- AdminPanel - das Administrationsfenster
- Connect - Thread welcher die Verbindung aufbaut und am Eingang lauscht

In der Implementation der einzelnen Klassen unterscheidet sich das Administratorapplet nur unwesentlich vom Spielerapplet.
Die Adminq-Klasse verwaltet das umschalten der einzelnen Panels.
Das Login-Panel (*LoginPanel*) ist ähnlich dem des Spielerapplets realisiert. Es wurde allerdings auf die Beschränkung auf 10 Zeichen verzichtet.
Das Admin-Panel (*AdminPanel*) dient dem Anzeigen der einzelnen Spieler und der Spieltische. Die Abrufe dieser beiden Informationen ist mittels zweier Buttons realisiert. Ebenso existiert ein *Textfield*, welches zu Eingabe von Server-Kommandos dient.
Die Klasse *Connect* ist ähnlich der Klasse *ConnectP* aus dem Spielerapplet. Der Unterschied besteht nur in der Auswertung der eingehenden *Strings*.

## 4.3 Spielserver

Die Aufgabe des Spielservers ist es eine Verbindung zwischen mehreren Clients gleichzeitig zu ermöglichen. Dieses wird dadurch realisiert, daß für jeden Client ein eigener Thread abläuft, welcher die Kommunikation verwaltet. Dieser Thread hält die Verbindung zu einem Client aufrecht und lauscht nach seinen Nachrichten. Ankommende Nachrichten werden interpretiert und dann an die entsprechenden Methoden des Servers weitergeleitet, so daß die Aufgabe des Spielserver nur in der Verwaltung von Daten, Ermöglichen einer Kommunikation zwischen Clients und warten auf neue Clients besteht. Aus folgenden Klassen besteht der Spielserver:

- Gameserver - die ausführbare Klasse
- Player - ein Thread für jeden Spieler
- Admin - ein speziellen Thread für den Administrator
- Backup - Thread zum Sichern der Userdaten
- Board - Die Kontrollinstanz einer Spielsitzung

Zusätzlich ist ein weiterer Thread (*Ping.class*) im Spielserver realisiert, dessen Aufgabe es ist, alle Clients regelmäßig zu überprüfen und nicht mehr vorhandene an den Server zu melden, welcher diese dann entfernt.
Die Klasse *Gameserver* ist die Hauptklasse des Spielservers. Seine Aufgabe besteht darin:

- auf neue Clients zu warten
- für diese Clients einen entsprechenden Thread zu erstellen
- wenn die Client nicht mehr ansprechbar sind, diese wieder zu löschen
- spezielle Nachrichten an den Administrator zu schicken
- Administartor-Befehle auszuführen
- Kontollinstanzen für Spielsitzungen erstellen bzw. löschen

Die benötigten Spielrdaten werden mittels einer Hilfsklasse *AllUser* in einem Vektor gehalten. Wird eine neue Socket zu Spielserver aufgebaut, so erwartet dieser als nächstes den Anmeldestring, welcher ein bestimmtes Format hat. Zum Schutz des Spielservers wird die Länge und das Format des Strings geprüft und bei Fehler die Verbindung sofort gelöst. Diese Daten werden mit denen aus dem Vector mit den Spielerdaten verglichen. Ist die Anmeldung erfolgreich wird dem Client eine eindeutige Id zugewiesen und ein Thread für ihn erstellt. Dies erfolgt mittels der Methode *addUser()*. Zum Entfernen eines Users wird die Methode *delUser()* genutzt. Alle aktiven Clients werden ebenso in einem Vektor gespeichert, in dem man nach bestimmten Clients suchen kann. So wird z.B. wenn eine Nachricht an einen speziellen Client geschickt werden soll, dieser Vektor nach dem passenden Thread durchsucht. Die Zugriffe auf die Vektoren sind über *synchronized* geschützt.
Weitere Aufgaben der *Gameserver-Klasse* ist die Unterstützung der Arbeit des Administrators. Dieser kann spezielle Befehle an den Spielserver schicken, welche dann vom Spielserver ausgeführt werden.
Jeder Spieler und Administartor bekommt einen eigenen Thread. Die Aufgabe dieses Threads, ist es auf Nachrichten des Clients zu lauschen, diese zu interpretieren, an den Spielserver weiter zu geben und andere Nachrichten an den Client zu senden. Bereits oben wurde gesagt, daß ein Thread existiert, welcher die Verfügbarkeit der Clients regelmäßig prüft. Dazu wird an den Client der String "ping" geschickt. Dieser muß mit "OK" quittiert werden. Ist dies nicht der Fall, aber die Verbindung noch vorhanden, so wird ein Zeitzähler aktiviert, welcher nach 3 min Inaktivität dem Spielserver mitteilt, daß der Client entfernt werden soll. Ein großen Teil der Klasse besteht aus dem Algorithmus zur Interpretation der eingehenden Nachrichten, da diese Strings zur weiteren Nutzung in Teilstrings geteilt werden müssen.
Eine weitere wichtige Klasse ist der *Backup-Thread*. Diese Klasse ist ein selbständiger Thread, welcher in vorgegebenen Zeitabständen die Spielerdaten des Spielservers in eine vorgegebene Datei sichert.
Als Letzes soll die *Board-Klasse* erwähnt werden. Diese Klasse dient als Kontrollstruktur einer Spielsitzung auf dem Spielserver. Es geht darum, die Manipulation einer Spielsitzung und damit das Verändern des Rankings zu erschweren. Dazu wird nur von dem Spielserver der Gewinner bzw. Verlierer bestimmt, und nur der Spielserver bestimmt das neue Ranking und ändert es in der Spielerliste. Um den Gewinner bzw. Verlierer bestimmen zu können, verfolgt eine Instanz dieser Klasse den Spielverlauf, überprüft die Spielzüge und teilt am Ende den Gewinner mit.

## 4.4 Kommunikation

Die Kommunikation zwischen Clientapplets und dem Spielserver erfolgt mittels Sockets und zugehörige Streams. Über eine Socket wird eine feste Verbindung von einem Rechner zu einem anderen aufgebaut, die solange bestehen bleibt, bis diese Socket geschlossen wird. Ist eine Socketverbindung vorhanden, kann über entsprechende Streams eine bidirektionale Kommunikation hergerstellt werden. Damit ein Client mit dem Spielserver kommunizieren kann, muß ein Protokoll entworfen werden, welches die zu übermittelten Daten definiert, damit garantiert ist, daß sowohl der Server wie auch der Client genau wissen, welche Aktionen bei welcher Nachricht ausgeführt werden müssen. Dieses Protokoll wird in Anhang D erläutert.

# Kapitel 5

# Resultierendes Programmierprojekt

In diesem Kapitel wird auf das aus der Realisierung entstandene Programmierprojekt eingegangen.

Die Lernziele des Programmierprojektes waren durch den Inhalt der Vorlesung vorgegeben. Sie sind:

- Erlernen des projektbezogenen Arbeitens
- Erwerben von praktischer Programmiererfahrung in der Programmiersprache java
- Verständnis von Datenstrukturen
- Einfache nebenläufige Programmierung
- Einfache verteilte Programmierung

In dem Programmierprojekt wird das hier erarbeitete System von Studierenden erneut implementiert. Dazu ist ein Zeitplan erarbeitet worden und das Gesamtsystem in einzelne Teilprojekte, welche aufeinander aufbauend sind, geteilt. Für das Programmierprojekt ist der Zeitraum von 13 Wochen (ein Semester) angesetzt. In dieser Zeit werden die Studierenden von Tutoren betreut, die in praktischen Übungen den Studierenden Anleitungen zu den einzelnen Teilaufgaben geben.

Das Projekt ist in folgende Teilaufgaben eingeteilt.

1. Einfaches Zeichnen von Bildern, Implementierung eines einfachen Eventhandling
2. Programmieren eines Zwei-Spieler-Modus Spiel
3. Erweitern des Spieles um einen künstlichen Gegner
4. Implementierung eines Client/Server-Modells

Da die Studierenden unterschiedliche Voraussetzungen bezüglich der Vorkenntnisse in der Programmiersprache Java haben, wird in der ersten Teilaufgabe relativ wenig implementiert. In dieser Stufe des Programmierprojekts sollen die Studierenden den Umgang mit der Programmiersprache und der Programmierumgebung erlernen. Zusätzlich erlernen sie die Basis der Programmierung eines Applets. Gefordert wird in dieser Teilaufgabe, daß die Studierenden die für das Spiel benötigten Grafiken laden, diese auf den Bildschirm anzeigen und ein *Eventhandling* für einen Mausklick implementieren.

In der zweiten Teilaufgabe wird ein Zwei-Spieler-Modus programmiert. Dabei erlernen die Studierenden einfache Daten- und Programmstrukturen. Für einen Zwei-Spieler-Modus muß das Spielfeld in eine geeignete Datenstruktur modelliert, eine Unterscheidung zwischen beiden Spielern, die Anzeige der aktuellen Punktzahl, das Finden von Quadraten inklusive der Berechnung der Punkte für diese und das Spielende bzw. der Neustart eingebunden werden.

In der dritten Teilaufgabe wird das Spiel um die Möglichkeit erweitert, gegen einen künstlichen Gegner zu spielen. Dabei wird der künstliche Gegner vorgegeben. Den fortgeschrittenen Studierenden wird die Möglichkeit gegeben den Gegner zu erweitern oder neu zu schreiben.

Die vierte Teilaufgabe beschäftigt sich mit der Realisierung eines Client/Server-Modells. Das Spiel soll um die Möglichkeit erweitert werden, gegen einen Gegner über das Internet zu spielen. Hierfür wird ein Server zu Verfügung gestellt, welcher zwei Spieler miteinander verbindet und die Kommunikation zwischen diesen handhabt. Auch hier kann von den Studierenden ein eigener Server geschrieben werden. In dieser Teilaufgabe lernen die Studierenden das Programmieren von verteilten und nebenläufigen Systemen. Auf der Client-Seite muß ein Thread implementiert werden, welcher an seiner Socket nach neuen Nachricht lauscht, doch bevor dieses funktioniert muß erst eine Verbindung zu dem Server erstellt werden.

Für alle Teilaufgaben werden entsprechende Hilfen zur Verfügung gestellt. Die geanu Aufgabenstellung, Hilfen und Musterlösung ist unter [Mat98] zu finden.

# Kapitel 6

# Ausblick

In diesem Kapitel wird ein Ausblick auf die Erweiterungs- bzw. Verbesserungsmöglichkeiten bei der Implementation eingegangen.

## 6.1 RMI

Im Zuge der Weiterentwicklung der Sprache Java, ergeben sich neue Möglichkeiten in der Implementierung eines Client/Server-Modells. Im Unterschied zu der in dieser Arbeit genutzten Kommunikation mittels Streams und Sockets, kann die Remote Method Invocation (RMI) Technologie zur Kommunikation mit einem Server genutzt wird.
Mittels RMI kann in einem reinem in Java geschriebenem Client/Server-Modell die Kommunikation gelöst werden. Dies ist wiederum die Schwachstelle. Bei einer Lösung mittels Sockets und Streams kann der Server in jeder anderen Programmiersprache geschrieben sein. Dafür gibt es wiederum auch Vorteile. Der Vorteil liegt darin, daß man das System nicht auf der Protokollebene implementieren muß, sondern dieses von speziellen java-Klassen übernommen wird. Mittels RMI können so direkt Methoden auf dem Server aufgerufen werden, was eine Vereinfachung der Implementation bedeutet. Weitere Informationen zu diesem Thema sind in [HoC98] zu finden.

## 6.2 JDBC

Eine weitere Erweiterungsmöglichkeit des Systems ist durch die Nutzung einer Datenbank für die Verwaltung der Spielerdaten gegeben. Dazu kann die JDBC-Technologie genutzt werden. Durch die Verlagerung der Spielerdaten in eine Datenbank wird der eigentliche Spielserver entlastet und die gesamte Verwaltung der Spielerdaten auf die Datenbank verschoben. Bei einer großen Anzahl an unterschiedlichen Spielern kann dies zu einer Leistungssteigerung führen.
JDBC (Java Database Conectivity) ist ein generisches SQL-Datenbank-Interface, das von einem speziellen Datenbanksystem unabhängig ist. Java Programmierern steht somit eine Schnittstelle zur Verfügung, welche in einheitlicher Weise auf unterschiedliche Datenbanken zugreifen können. JDBC gestattet die direkte Ausführung von SQL-Kommandos und die Verarbeitung der Ergebnisse. Weitere Informationen zu diesem Thema sind in [HCF97] zu finden.

## 6.3 Weitere Erweiterungsmöglichkeiten

In der jetzigen Realisierungsvariante des Systems ist noch keine Rankingliste implementiert. Es gibt zwar eine Datei mit den Spielerdaten. Diese wird allerdings als Sicherungskopie der Spielerdaten von dem Server genutzt. Diese Datei wird dementsprechend nicht minütlich aktualisiert. Daraus ergibt sich das Problem der Aktualität. Sind die Spielerdaten in einer Datenbank implementiert, so hat man dort immer die aktuellen Spielerdaten und kann eine nach Spielstärken sortierte Rankingliste mittels eines entsprechendem HTTP-Aufrufes abrufen.

Ein weiterer Gesichtspunkt ist die Prozessorauslastung von Java-Servern. Hat der Spielserver eine hohe Auslastung, kann es erforderlich sein, die Spieleranzahl auf dem Server einzuschränken und mehrere Server mit dem selben Spiel aufzusetzen. Dafür muß allerdings eine für alle Spielserver gemeinsame Spielerdatenbank erstellt und eingebunden werden.

Ein weiterer Gesichtspunkt, welcher in dieser Version der Realisierung noch nicht implementiert ist ein Funktion eines Log-Files. So kann dieses Logfile die Aktionen der Spieler protokollieren und zur Fehlerdiagnose genutzt werden. Aufgrund der möglichen immensen Größe des Logfile, sollte dieses an und ausgeschaltet werden können.

Abschließend wird der Computergegner angesprochen. Dieser wäre ein sinnvolle Erweiterung des Systems, für den Fall, daß sich ein Spieler allein im System befindet und sich kein weiterer menschlicher Gegner finden läßt.

# Literaturverzeichnis


[MSS96] S.Middendorf, R. Singer, S. Strobel. *Java Programmierhandbuch und Referenz*. Dpunkt,1996.

[Fla98] D. Flanagan. *Java in a nutshell*. O'Reilly, 1998.

[Mos97] W. Moßner. *Ein Kommunikationsframework für Java*. Java Spektrum, November 1997.

[RSc98] P. Roßbach, H. Schreiber. *Server-Programmierung in Java*. Java Spektrum, Mai 1998.

[FSc98] M. Fowler, K. Scott. *UML konzentriert*. Addison-Wesley,1998.

[HCF97] G. Hamilton, R. Cattell, M. Fischer. *JDBC Database Access with Java*. Addison Wesley, 1997.

[HoC98] C.S. Horstmann, G. Cornell. *Core Java 1.1 Volume II Advanced Features*. Prentice Hall Computer Books, 1998.

# Anhang A

# Spielregeln

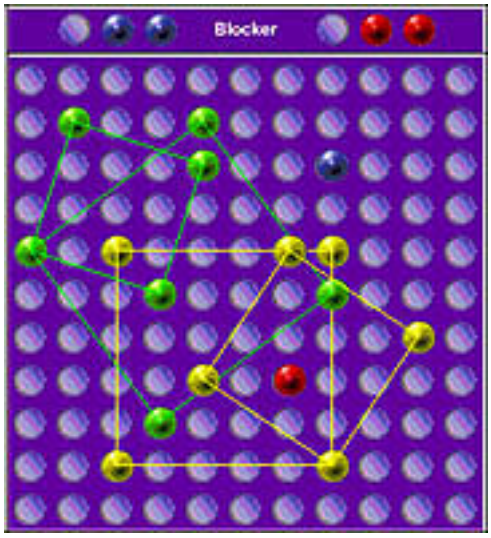


Abbildung A.1 - Spielfeld mit Quadraten

Zwei Spieler setzen abwechselnd je einen Stein auf ein rechteckiges, 11x11 Felder großes Spielbrett, bis

- entweder ein Spieler das Spiel abbricht,
- ein Spieler 200 Punkte erreicht hat oder
- das Spielfeld vollständig mit Steinen belegt ist.

Im Falle eines vollständig mit Steinen belegten Spielfelds gewinnt der Spieler mit der höheren Punktzahl.
Die Steine dürfen nach dem Setzen nicht wieder aufgenommen oder bewegt werden. Jeder Spieler besitzt Steine einer Farbe (Gelb bzw. Grün). Zusätzlich gibt es entsprechende Blocker, die die Farbe Blau bzw. Rot haben. (siehe Abbildung A.1)
Der Herausforderer startet das erste Spiel und setzt dabei den ersten Stein. In den nachfolgenden Spielen setzt jeweils der Verlierer des vorangegangenen Spiels den ersten Stein.
Das Ziel des Spieles ist es, durch geschicktes Setzen von Steinen möglichst viele Punkte zu erreichen. Punkte werden durch die Bildung von Quadraten auf dem Spielbrett erzielt. Ein Quadrat wird durch genau vier gesetzte Steine eines Spielers gebildet, die auf dem Spielbrett den Ecken dieses Quadrats entsprechen. Dabei kann ein Spielstein mehrere Quadrate gleichzeitig vervollständigen (siehe Abbildung A.1). Die Punktzahl ergibt sich dabei aus der Summe der Punktzahlen der Einzelquadrate.

### Punktzahl für ein Quadrat

Mit Quadraten sind Rechtecke mit gleicher Kantenlänge, die beliebig auf dem Brett rotiert sein können (siehe Abbildung 1.1) gemeint. Als Faustregel gilt: Je größer und gedrehter ein Quadrat ist, desto mehr Punkte gibt es dafür.

Die Punkte eines Einzelquadrats berechnet sich folgendermaßen:

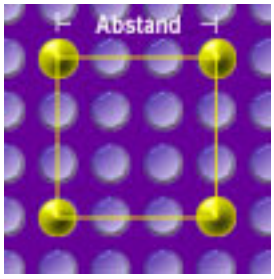


Abbildung A.2 - einfaches Quadrat

Abbildung A.2 zeigt ein Quadrat welches nicht gedreht ist. Für ein solches Quadrat berechnet sich die Punktzahl wie folgt:

Punktzahl = 4*Abstand.

Im Falle der Abbildung A.2: Punktzahl = 12

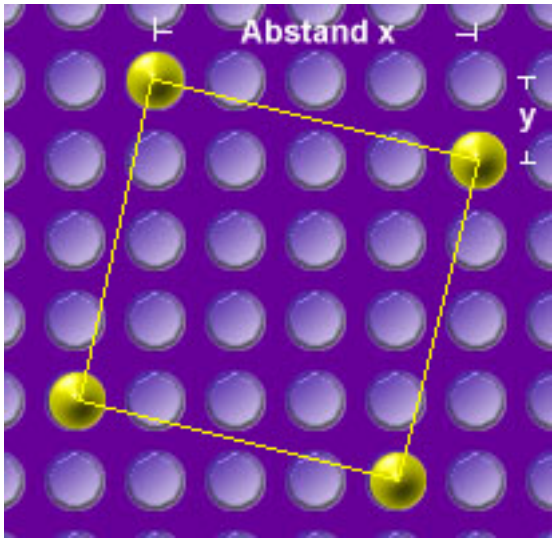


Abbildung A.3 zeigt ein Quadrat welches gedreht ist. Für ein solches Quadrat berechnet sich die Punktzahl wie folgt:

Punktzahl = 4*max(Abstand x, Abstand y)*2

Im Falle der Abbildung A.3: Punktzahl = 32

Abbildung A.3 - gedrehtes Quadrat

Aus der oben gezeigten Berechnung ergibt sich, daß das vorrangige Ziel des Spiels ist, die Steine so zu legen, daß dadurch möglichst viele große Quadrate erstellt werden. Dabei muß beachtet werden, daß dies auch das Ziel des Gegners ist. Um den Gegner aufzuhalten, kann man entweder einen Blocker (Blau bzw. Rot) oder einen Spielstein setzen. Das Setzen eines Blockers hat den Vorteil, daß man nach dem Setzen weiterhin am Zug bleibt, also einen weiteren Spielstein oder Blocker setzen kann. Allerdings hat jeder Spieler insgesamt nur drei Blocker zur Verfügung. Die gesetzten Blocker können nicht zur Quadratbildung genutzt und nicht vom Spielfeld wieder entfernt werden. Der belegte Platz wird damit auch für die eigene Strategie wertlos.

## Ranking

Um die Erfahrenheit eines Spielers zu messen, wird bei dieser Variante des Spieles noch ein Ranking eingeführt. Dieses Ranking wird durch eine Zahl repräsentiert, welche mit jedem Sieg steigt und mit jeder Niederlage wieder fällt. Jeder neue Spieler erhält ein Grundranking von 100. Bei jedem Sieg oder jeder Niederlage errechnet sich das Ranking wie folgt neu:

Falls das Ranking des Spieler1 größer oder gleich dem Ranking des Spieler2 ist und Spieler1 gewinnt, dann errechnet sich das neue Ranking beider Spieler wie folgt:

Spieler1-Ranking=Spieler1-Ranking+(0.03*(int)Spieler2-Ranking)+0.8

Spieler2-Ranking=Spieler2-Ranking-(0.02*(int)Spieler2-Ranking)

Dabei bedeutet das *(int),* daß der ganzzahlige Wert des Ranking zur Berechnung genutzt wird.

Beispiel:

Spieler1-Ranking=102,8; Spieler2-Ranking=98,2;

Daraus ergibt sich als neue Werte für das Ranking der einzelnen Spieler:

Spieler1-Ranking=102.8+(0.03*98)+0.8=106.54
Spieler2-Ranking=98.2-(0.02*98)=96.24

Falls das Ranking des Spieler1 größer oder gleich dem Ranking des Spieler2 ist und Spieler2 gewinnt, dann errechnet sich das neue Ranking beider Spieler wie folgt:

Spieler2-Ranking=Spieler2-Ranking+(0.03*(int)Spieler1-Ranking)
+0.2*((int)Spieler1-Ranking - (int)Spieler2-Ranking)+0.8

Spieler1-Ranking=Spieler1-Ranking-(0.02*(int)Spieler1-Ranking)
-0.2*((int)Spieler1-Ranking - (int)Spieler2-Ranking)

Beispiel:

Spieler1-Ranking=102,8; Spieler2-Ranking=98,2;

Daraus ergibt sich als neue Werte für das Ranking der einzelnen Spieler:

Spieler1-Ranking=102.8-(0.02*102)-0.2*(102-98)=99.96
Spieler2-Ranking=98.2+(0.03*102)+0.2*(102-98)+0.8=102.86

# Anhang B

# Anwendungsfälle

## Anwendungsfälle Spieler

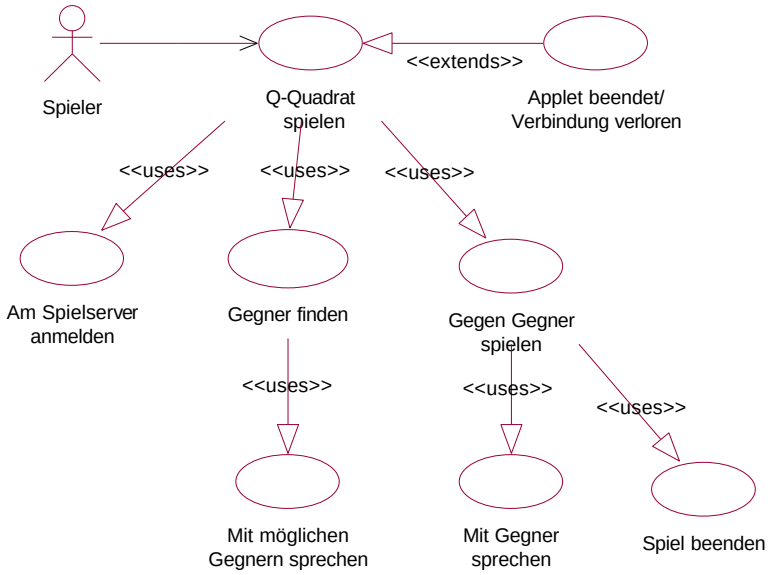


Abb. B.1: Use Case Diagramm: Spieler

Q-Quadrat spielen

Damit der Spieler sich mit anderen Gleichgesinnten im Spiel Q-Quadrat messen kann, ist folgende Vorgehensweise nötig:

1. Der Spieler meldet sich am Spielserver an.
2. Der Spieler findet einen Gegner.
3. Der Spieler spielt gegen den Gegner das Spiel Q-Quadrat.
4. Der Spielserver aktualisiert das Ranking des Spielers und Gegners bei Spielende.
5. Der Spieler beendet das Spiel.

Anwendungsfälle:

**uses Am Spielserver anmelden**

**uses Gegner finden**

**uses Gegen Gegner spielen**

Aktoren:

**Spieler (als Kunde)**

Am Spielserver anmelden

Bevor der Spieler zum eigentlichen Spielen übergehen kann, muß dieser sich am Spielserver anmelden. Der Spielserver hält eine Liste von allen Spielern, die sich jemals angemeldet haben. Um den Reiz am Spiel zu erhöhen, erhält jeder Spieler ein Ranking. Diese Zahl repräsentiert die Spielstärke eines Spielers. Mit jedem Sieg steigt und mit jeder Niederlage sinkt dieses Ranking. Um eine eindeutige Zuordnung zu

gewährleisten, wird die Anmeldung mittels des Namens und eines Paßwortes gesichert.

Vorgehensweise:

1. Der Spieler trägt seinen Namen und Paßwort ein.

2. Der Spieler drückt zum Abschicken auf einen Button und wartet auf die Rückmeldung.

3. Der Spielserver weist ihm das passende Ranking zu.

Gegner finden

Möchte ein Spieler eine Spielsitzung starten, so muß er dafür einen Gegner herausfordern.

Vorgehensweise:

1. Der Spieler wählt einen Gegner aus der Liste der spielbereiten Gegner.

2. Der Gegner wird über den Spielwunsch informiert.

3. Der Gegner akzeptiert den Spielewunsch oder lehnt ihn ab.

4. Der Spieler wird über die Antwort informiert.

5. Ist die Antwort positiv, kommt es zu einer Spielsitzung.

Anwendungsfälle:

**uses Mit möglichen Gegnern sprechen**

Gegen Gegner spielen

Sind sich zwei Spieler einig, so kommt es zu einer Spielsitzung. Diese läuft wie folgt ab:

1. Der Herausforderer fängt das erste Spiel an. Danach jeweils der Verlierer der Spielpartie.

2. Der Spieler am Zug setzt beliebig viele Blocker aus seinem Vorat an Blockern (max. 3). Jedes Setzen verringert seinen Vorrat.

3. Der Spieler setzt einen Spielstein.

4. Der Gegner wird über das Setzen der Spielsteine informiert.

5. Der Spielserver protokolliert das Setzen der Spielsteine, aktualisiert den Punktestand und zeigt gegebenenfalls den Sieg an.

Das Spiel ist zu Ende, falls:

- einer Spieler das Spiel abbricht
- ein Spieler 200 Punkte erreicht hat
- das Brett vollständig belegt ist

6. Der Gegner ist am Zug.

Anwendungsfälle:

**uses Mit Gegner sprechen**

**uses Spiel beenden**

Mit möglichen Gegnern sprechen

Zu jedem Zeitpunkt können der Spieler und die spielbereiten Gegner asynchron Nachrichten austauschen.

Vorgehensweise:

1. Ein spielbereiter Spieler gibt einen Text in das Eingabefeld ein und drückt danach die Eingabetaste.

2. Der Spielserver leitet den Text mit Angabe des Namen des Senders an alle anderen spielbereiten Spieler weiter.

Mit Gegner sprechen

Zu jedem Zeitpunkt einer Spielsitzung können der Spieler und der Gegner asynchron Nachrichten austauschen.

Vorgehensweise:

1. Ein spielbereiter Spieler gibt einen Text in das Eingabefeld ein und drückt danach die Eingabetaste.

2. Der Spielserver leitet den Text mit Angabe des Namen des Senders an alle anderen spielbereiten Spieler weiter.

Spiel beenden

Jeder der beiden in der Spielsitzung befindlichen Spieler kann während oder nach einer Spielsitzung diese beenden.

Vorgehensweise:

1. Der Spieler drückt einen Button zum Abbrechen des Spieles.

2. Der Gegner wird informiert.

3. Der Spielserver aktualisiert die Rankingliste und die Liste der spielbereiten Spieler.

Applet beendet/Verbindung verloren

Zu jedem Zeitpunkt kann es passieren, daß die Verbindung zum Spielserver verloren geht oder der Spieler das Applet beendet. In diesem Fall trägt der Spielserver den Spieler aus der Liste der spielbereiten Spieler aus.

Anwendungsfälle:

**extends Q-Quadrat spielen**

## Anwendungsfälle Administrator

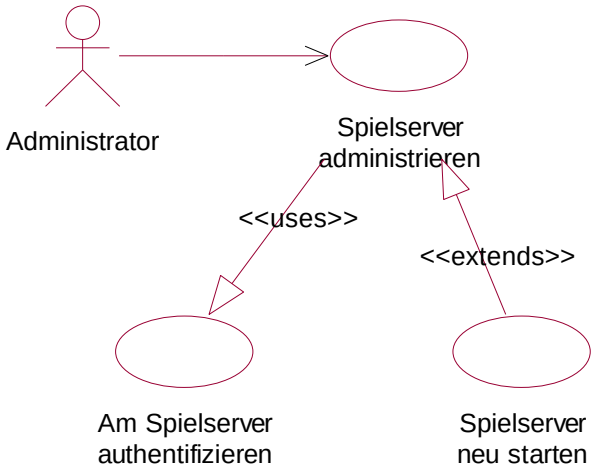


Abb. B.2: Use Case Diagramm: Administrator

Spielserver administrieren

Die Aufgabe des Administrators ist der reibungslose Ablauf des Spielservers. Dazu kann er von dem Spielserver Spieler entfernen, Spielsitzungen auflösen, Passwörter ändern und eine Sicherung der Spielerliste starten.

Vorgehensweise:

1. Der Administrator authentifiziert sich am Spielserver.

2. Der Administrator verschafft sich einen Überblick über die Spieler und Spielsitzungen des Spielserver.

3. Der Administrator löst vorhandene Probleme oder startet den Spielserver neu.

4. Der Administrator meldet sich ab.

Anwendungsfälle:

**uses Am Spielserver authentifizieren**

Aktoren:

**Administrator (als Kunde)**

Am Spielserver authentifizieren

Bevor der Administrator das spezielle Admin-Applet nutzen kann, muß er sich am Spielserver authentifizieren. Dadurch wird gewährleistet, daß nur autorisierte Benutzer dieses Applet nutzen können.

Vorgehensweise:

1. Der Administrator gibt sein Paßwort ein.

2. Der Spielserver erlaubt den Zugriff.

Spielserver neu starten

Es kann nötig sein, den Spielserver neu zu starten.

Anwendungsfälle:

**extends Spielserver administrieren**

# Anhang C

# Klassendiagramme

## Spielserver

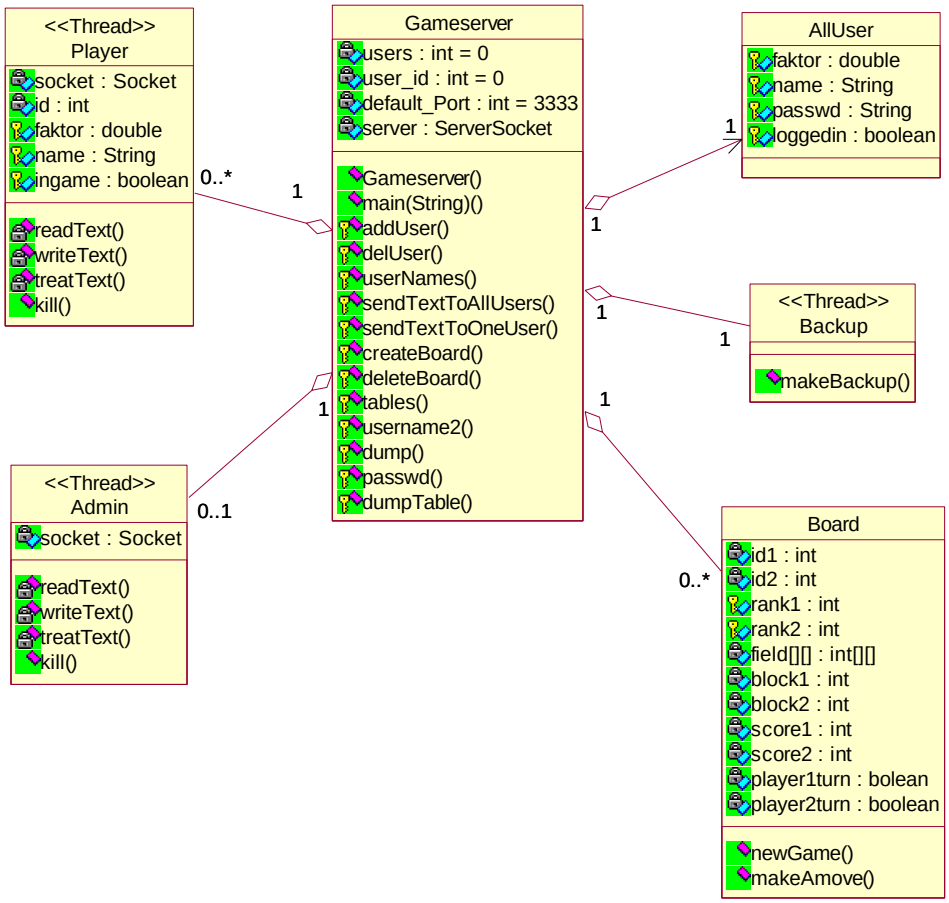


Abb. 3.1: Class Diagramm: Spielserver

Gameserver

Der Gameserver unterstüzt den Aufbau von Spielsitzungen und die Kommunikation zwischen angemeldeten Spielern. Außerdem kontrolliert er die Spielsitzungen, verwaltet die Rankingliste und sichert die Spielerliste in eine Datei.

Beim Start des Gameservers werden die aktuellen Spielerdaten aus der gesicherten Datei gelesen.

Private Attributes:

**users : int = 0**

Diese Zahl gibt die Anzahl der Spieler wieder, die am Server angemeldet sind. Maximal sollen 50 Spieler gleichzeitig angemeldet sein.

**user_id : int = 0**

Zur Identifizierung der einzelnen Spieler wird ihnen eine eindeutige Id zugewiesen.

**default_Port : int = 3333**

Port an dem der Gameserver lauscht.

**server : ServerSocket**

Socket an der der Server auf neue Clients wartet.

Public Operations:

**Gameserver (int port) :**

Methode, welche den Gameserver initialisiert.

**main(String args) : void**

Methode zum Starten einer Applikation.

**addUser (Socket s) : void**

Erstellt einen neuen Thread Player .

**delUser (int i) : void**

Entfernt einen Spieler vom Spielserver.

**userNames () : void**

Sendet die aktualisierte Spielerliste an alle Spieler.

**sendTextToAllUsers (String s) : void**

Sendet eine Nachricht an alle angemeldeten Spieler.

**sendTextToOneUser (String s , int i) : void**

Sendet eine Nachricht an einen Spieler.

**createBoard () : void**

Erstellt einen Tisch an dem eine Spielsitzung geleitet wird.

**deleteBoard () : void**

Löscht einen Spieltisch.

**tables () : void**

Gibt dem Administrator eine Liste der Spieltische zurück. (Administratorbefehl)

**username2 () : void**

Schickt dem Administrator eine Liste aller angemeldeten Spieler. (Administratorbefehl)

**dump (int i) : void**

Entfernt einen Spieler mit der Id i vom Spielserver. (Administratorbefehl)

**passwd () : void**

Ändert das Passwort eines Spielers. (Administratorbefehl)

**dumpTable () : void**

Löscht einen Spieltisch vom Gameserver. (Administratorbefehl)

## AllUser

Klasse welche die Daten eines einzelnen Spielers enthält.

Protected Attributes:

**faktor : double**

Ranking eines Spielers.

**name : String**

Login-Name des Spielers.

**passwd : String**

Paßwort des Spielers

**loggedin : boolean**

Diese Variable wird als Flag genutzt. Ist ein Spieler am Spielserver angemeldet so ist der Wert true sonst false. Dies wird benötigt, damit jeder Spielername zu einem Zeitpunkt nur einmal auf dem Spielserver angemeldet ist.

## Backup

Da die Informationen der Spieler nur im Arbeitsspeicher vorhanden sind, sichert dieser Thread in regelmäßigen Abständen die Liste aller Spieler in eine vorgegebene Datei.

Public Operations:

**makeBackup () : void**

Diese Methode benötigt man für den Administrator, damit er in der Lage ist eine Sicherung von Hand aus zu starten.

### Player

Die Verbindung zu einem Spieler muß in einem Thread realisiert werden, da am InputStream permanent gelauscht werden muß.

Protected Attributes:

**faktor : double**

Das Ranking des Spielers.

**name : String**

Login-Name des Spielers.

**ingame : boolean**

Dieses Flag wird gesetzt, falls der Spieler sich in einer Spielsitzung befindet. Damit ist er in der Liste der spielbereiten Spieler nicht mehr sichtbar.

Private Attributes:

**socket : Socket**

Entspricht der Socket des Spielers worüber kommuniziert wird.

**id : int**

Eindeutige Id des Spielers, damit er auf dem Gameserver identifizierbar ist.

Public Operations:

**kill () : void**

Beendet den Thread.

Private Operations:

**readText () : String**

Methode zum lesen des eines Streams.

**writeText () : String**

Methode zum schreiben des eines Streams.

**treatText () : void**

Methode zum interpretieren der Nachricht.

### Admin

Die Verbindung zum Administrator muß in einem Thread realisiert werden, da am InputStream permanent gelauscht werden muß.

Private Attributes:

**socket : Socket**

Entspricht der Socket des Administrators.

Public Operations:

**kill () : void**

Beendet den Thread.

Private Operations:

**readText () : String**

Methode zum Lesen des eines Streams.

**writeText () : String**

Methode zum Schreiben des eines Streams.

**treatText () : void**

Methode zum Interpretieren der Nachricht.

### Board

Das Board bildet die Kontrollinstanz einer Spielsitzung. Ist es zu einer Spielsitzung zwischen zwei Spielern gekommen, so werden alle Spielzüge in dieser Klasse geprüft und erst bei Richtigkeit an den Gegner weitergeleitet. Ebenso wird der Sieger auf dem Spielserver ermittelt. So wird das Manipulieren von Spielsitzungen erschwert.

Protected Attributes:

**score1 : int**

Punktzahl von Spieler1.

**score2 : int**

Punktzahl von Spieler2.

**id1 : int**

Id Spieler1.

**id2 : int**

Id Spieler2.

**rank1 : int**

Ranking Spieler1.

**rank2 : int**

Ranking Spieler2.

**field[][] : int[][]**

field[][] ist ein array der einzelnen Felder des Spielfeldes. Die Felder werden durch Spalte und Reihe identifiziert. Der Zustand eines Feldes wird durch ein Integer moduliert.
leeres Feld = 0
Spieler1-Spielstein = 1
Spieler1-Blocker = 2
Spieler2-Spielstein = 3
Spieler2-Blocker = 4

**block1 : int**

Gibt die Anzahl der Blocker von Spieler1 wieder. Maximal hat jeder Spieler 3 Blocker zur Verfügung. Sind diese verbraucht, kann der Spieler keine weiteren mehr setzen.

**block2 : int**

Gibt die Anzahl der Blocker von Spieler2 wieder. Maximal hat jeder Spieler 3 Blocker zur Verfügung. Sind diese verbraucht, kann der Spieler keine weiteren mehr setzen.
Ist Spieler1 am Zug so wird der Wert true gesetzt. Alle weiteren Spielzüge, welche vom Spieler2 an den Spielserver geschickt werden, werden mit einer Fehlermeldung quittiert.

**player2turn : booleanplayer1turn : bolean**

Ist Spieler2 am Zug so wird der Wert true gesetzt. Alle weiteren Spielzüge, welche vom Spieler1 an den Spielserver geschickt werden, werden mit einer Fehlermeldung quittiert.

Public Operations:

**newGame () : void**

Initialisiert das Spielfeld und die Blocker der Spieler.

**makeAmove (int : , String : ) : void**

Interpretiert einen Spielzug, überprüft die Richtigkeit, berechnet die neue Punktzahl und leitet den Spielzug an den Gegenspieler weiter.

## Administratorapplet

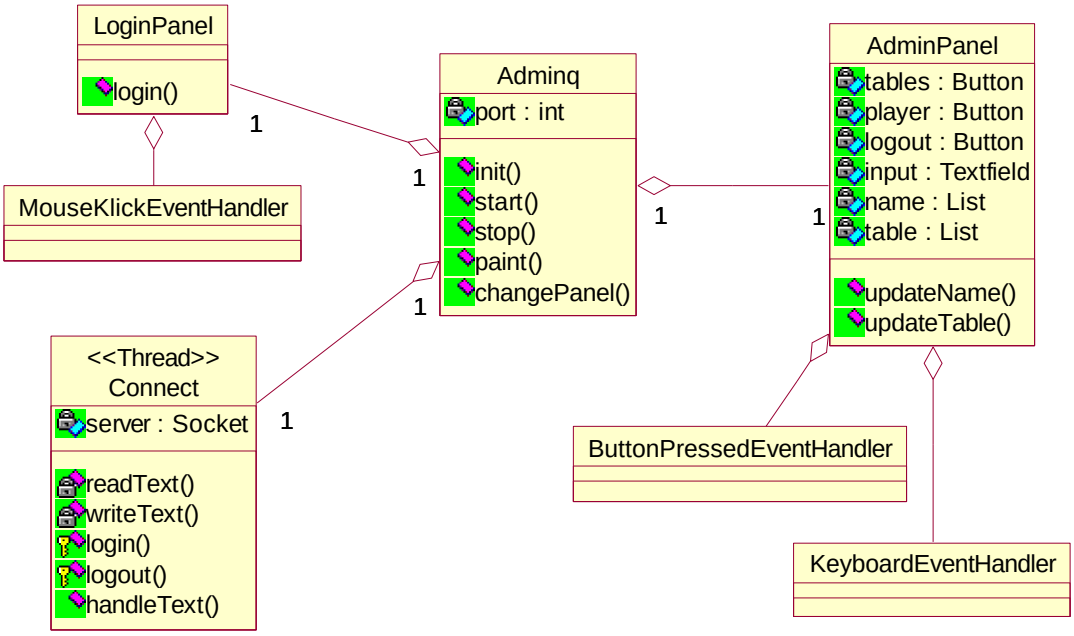


Abb. C.2 Class Diagramm: Administartor

### Adminq

Die Applet-Klasse des Administrators.

Public Attributes:

**port : int**

Der Port an dem der Spielserver lauscht.

Public Operations:

**init () : void**

Initialisierung der Variablen.

**start () : void**

Starten vom Thread.

**stop () :**

Stoppen des Threads Connect.

**paint () :**

Wechsel in das LoginPanel.

**changePanel () : void**

Methode, welche zwischen den Panels umschaltet.

### LoginPanel

Dies ist die Klasse, welche die grafische Ansicht des Anmeldefensters beschreibt.

Public Operations:

**login () : void**

Methode, welche die Anmeldung realisiert.

### Connect

Der Thread wird für die Verbindung zum Spielserver benötigt. Es wird permanent am Eingang auf neue Nachrichten gehorcht.

Private Attributes:

**server : Socket**

Socket über welche die Verbindung erstellt wird.

Public Operations:

**handleText () : void**

Behandelt die Nachrichten, die vom Spielserver angekommen sind.

**login () : void**

Vollzieht die Anmeldung.

**logout () : void**

Führt die Abmeldung von Spielserver durch.

Private Operations:

**readText () : String**

Methode zum Lesen des eines Streams.

**writeText () : String**

Methode zum Schreiben des eines Streams.

### AdminPanel

Das AdminPanel ist dass Arbeitsfensters des Administrators.

Private Attributes:

**tables : Button**

Button zum Abruf der Spieltische.

**player : Button**

Button zum Abrufen der Spieler.

**logout : Button**

Button zum Abmelden.

**input : Textfield**

Eingabefeld für Kommandos.

**name : List**

Liste der Spielernamen.

**table : List**

Liste der Spieltische.

Public Operations:

**updateName () : void**

Methode zum Aktualisieren der Spielerliste.

**updateTable () : void**

Methode zum Aktualisieren der Spieltischliste.

### MouseKlickEventHandler

Wird beim LoginPanel der grafische Button gedrückt, muß eine entsprechende Nachricht an den Spielserver geschickt werden.

### ButtonPressedEventHandler

Für die angegebenen Buttons sollen entsprechende Nachrichten an den Spielserver geschickt werden.

### KeyboardEventHandler

Wird in dem Eingabefeld die ENTER-Taste gedrückt so soll der dort geschriebene Inhalt an den Spielserver geschickt werden.

## Spielerapplet

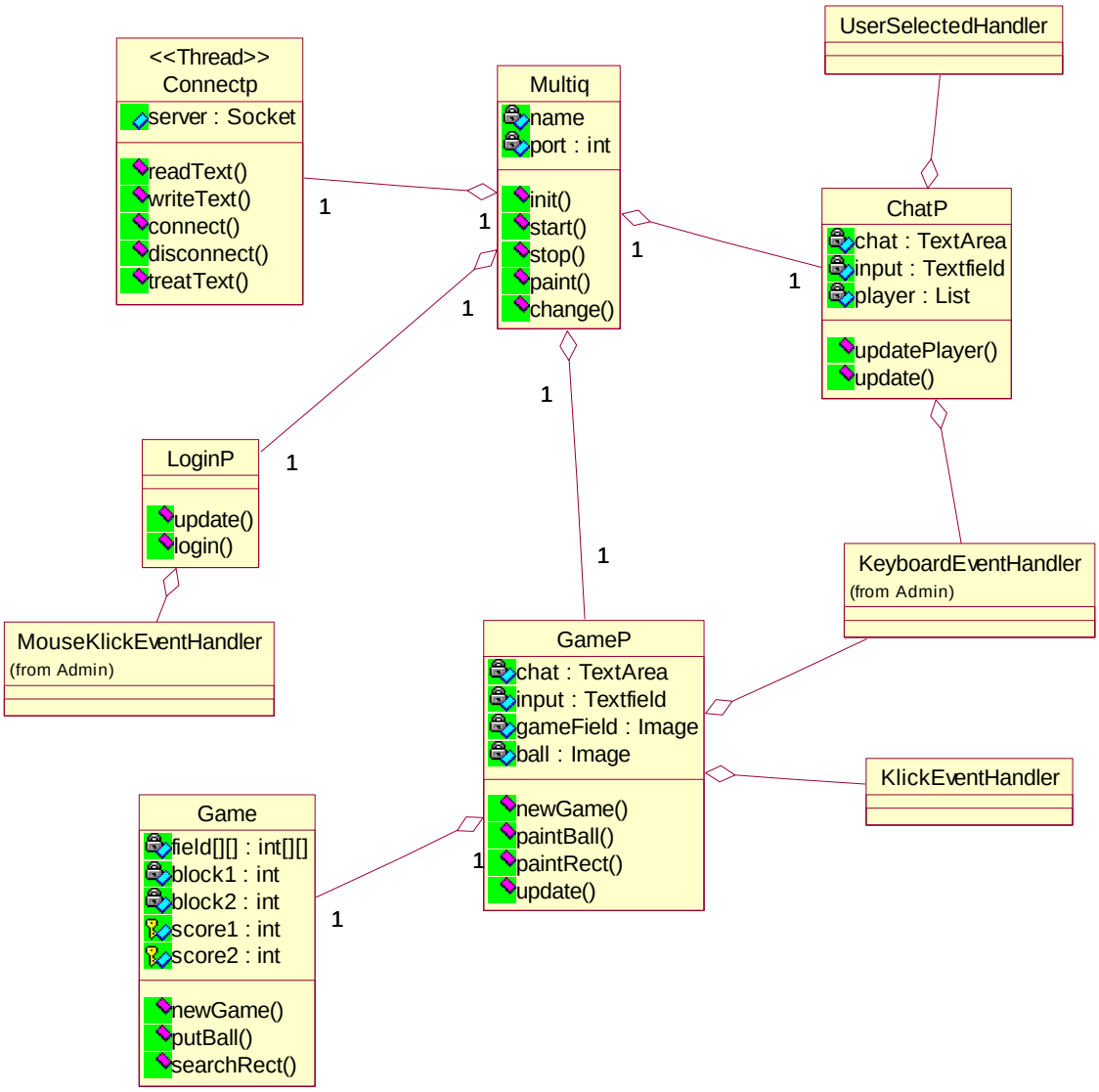


Abb C.3: Class Diagramm: Spieler

### Connectp

Thread welcher die Verbindung zum Spielserver auf- bzw. abbaut und am Eingang lauscht.

Public Attributes:

**server : Socket**

Socket über die mit dem Server kommuniziert wird.

Public Operations:

**readText () : String**

Methode zum lesen des eines Streams.

**writeText () : String**

Methode zum schreiben des eines Streams.

**connect () : void**

Vollzieht die Anmeldung.

**disconnect () : void**

Führt die Abmeldung von Spielserver durch.

**treatText () : void**

Interpretiert den angekommenden Stream.

### LoginP

Panel zum Anmelden am Spielserver.

Public Operations:

**update () : void**

Zeigt Meldungen, die vom Server kommen, an.

**login () : void**

Methode, welche die Anmeldung realisiert.

### Multiq

Die Appletklasse des Spielers.

Private Attributes:

**port : int**

Der Port an dem der Spielserver lauscht.

Public Operations:

**init () : void**

Initialisierung der Variablen. Laden von Sounds und Grafiken.

**start () : void**

Starten vom Thread.

**stop () : void**

Stoppen des Threads Connectp.

**paint () : void**

Wechsel nach LoginP.

**change () : void**

Methode zum Wechsel zwischen den einzelnen Ansichten.

### ChatP

Diese Klasse enspricht dem Fenster, in dem sich spielbereite Spieler unterhalten und herausfordern können.

Private Attributes:

**chat : TextArea**

Textarea für den Chat

**input : Textfield**

Eingabefeld für den Chat

**player : List**

Liste der spielbereiten Spieler.

Public Operations:

**updatePlayer () : void**

Methode, die die Liste der spielbereiten Spieler aktualisiert.

**update (Graphics g : ) : void**

Malt eventuelle Nachrichten.

### GameP

Diese Klasse entspricht dem Fenster, in der eine Spielsitzung angezeigt wird.

Private Attributes:

**chat : TextArea**

Textarea für den Chat.

**input : Textfield**

Eingabefeld für den Chat.

**gameField : Image**

Grafik des Spielfeldes.

**ball : Image**

Grafik aller Bälle.

Public Operations:

**newGame () : void**

Aktualisiert alle Grafiken. Malt z.B. ein leeres Spielfeld.

**paintBall (int x : , int y : , int b : ) : void**

Malt einen Ball vom Typ b an die Position (x,y).

**paintRect (Polygon P : ) : void**

Malt ein Quadrat mit den übergebenen Koordinaten.

**update (Graphics g : ) : void**

Aktualisiert die Bildschirmanzeige.

## Game

Diese Klasse hält alle Informationen die in einem Spiel benötigt werden.

Protected Attributes:

**score1 : int**

Punktzahl von Spieler1.

**score2 : int**

Punktzahl von Spieler2.

Private Attributes:

**field[][] : int[][]**

field[][] ist ein array der einzelnen Felder des Spielfeldes. Die Felder werden durch Spalte und Reihe identifiziert. Der Zustand eines Feldes wird durch ein Integer moduliert.
leeres Feld = 0
Spieler1-Spielstein = 1
Spieler1-Blocker = 2
Spieler2-Spielstein = 3
Spieler2-Blocker = 4

**block1 : int**

Gibt die Anzahl der Blocker von Spieler1 wieder. Maximal hat jeder Spieler 3 Blocker zur Verfügung. Sind diese verbraucht, kann der Spieler keine weiteren mehr setzen.

**block2 : int**

Gibt die Anzahl der Blocker von Spieler2 wieder. Maximal hat jeder Spieler 3 Blocker zur Verfügung. Sind diese verbraucht, kann der Spieler keine weiteren mehr setzen.

Public Operations:

**newGame () : void**

Initialisiert alle Variablen für ein neues Spiel.

**putBall (int x : , int y : , int b : ) : void**

Setz ein Ball an die Stelle (x,y) in field. Der Typ des Balles wird durch ein int moduliert.
Spieler1-Spielstein = 1
Spieler1-Blocker = 2
Spieler2-Spielstein = 3
Spieler2-Blocker = 4

**searchRect () : void**

Sucht nach einem Quadrat, nachdem ein Stein gesetzt wurde. Ein Quadrat besteht aus vier unterschiedlichen Spielsteinen eines Spielers. Die Formel zur Berechnung der Punktzahl für ein Quadrat ist in Kapitel 1 beschrieben.

## UserSelectedHandler

Ist ein Spieler aus der Liste der spielbereiten Spielern ausgewählt worden, so muß dieser über den Spielwunsch entsprechend benachrichtigt werden.

## KlickEventHandler

Ist der Spieler am Zug umd setzt einen Spielstein oder Blocker so muß dieses an den Spielserver weitergeleitet werden und grafisch angezeigt werden.

# Anhang D

# Kommunikationsprotokoll

### Kommunikationsprotokoll Spielerapplet - Spielserver

Zum Anmelden an dem Spielserver sendet das Spielerapplet folgende Nachricht:

| | |
|---|---|
| *Name,passwd,* | Anmelde-String des Spielerapplets |

Wird dieser String unmittelbar nach der Kontaktaufnahme eines Spielerapplets zum Spielserver geschickt, weiß der Spielserver, daß es sich um eine Anmeldung handelt überprüft den Namen und das Paßwort, und gibt eine entsprechende Meldung zurück. Als Rückmeldung von dem Spielserver erhält das Spielerapplet eine der folgenden Nachrichten:

| | |
|---|---|
| *Voll* | Diese Meldung bedeutet, daß der Spielserver ausgelastet ist. |
| *Spieler* | Diese Meldung bedeutet, daß der sich anmeldende Spieler bereits auf dem Spielserver angemeldet ist. |
| *Passwd* | Diese Meldung bedeutet, daß das eingegebene Paßwort nicht mit dem in der Spielerliste gesicherten übereinstimmt. |
| *ok* | Diese Meldung bedeutet, daß die Anmeldung erfolgreich war. |

Ist die Anmeldung nicht erfolgreich gewesen, bleibt das Spielerapplet in der Anmeldeansicht und zeigt eine entsprechende Meldung an. War die Anmeldung dagegen erfolgreich wechselt das Spielerapplet in die Chatansicht. In dieser Ansicht kann der Spieler sich über den eingebauten Chatraum mit anderen spielbereiten Spielern unterhalten oder einen Gegner herausfordern. Bevor er dieses machen kann fordert das Spielerapplet einen Id an. Dieses ist die eindeutige Kennung des Spielers, welche er für eine direkte Kommunikation mit anderen Spielern benötigt. Dafür sendet das Spielerapplet den String:

| | |
|---|---|
| *!!ID!!* | Diese Nachricht fordert die Id des Spieler an, welche von dem Spielserver jedem Spieler zugeordnet wird. |

Der Server sendet daraufhin die Id wie folgt zurück:

| | |
|---|---|
| *befehle!!,ID,Id* | Das Spielerapplet weiß, daß es sich hierbei um die Id des Spielers handelt. |

Damit der Spieler erkennen kann, wen er herausfordern kann, benötigt er die aktuelle Liste der anwesenden Spieler. Diese wird über den folgenden String angefordert.

| | |
|---|---|
| *!!namen!!* | Nachricht zum anfordern der Spielerliste. |

Der Server antwortet mit der angeforderten Liste, die wie folgt in einem String zusammengefaßt ist:

| | |
|---|---|
| *!!namen!!aktuelleAnzahlderSpieler,Spieler1Id~Spieler1Faktor!Spieler1Name?Spieler1Fertig\|Spieler2Id....* | Mit dem Anfangsstring !!namen!! weiß das Spielerapplet, daß es sich um eine Spielerliste handelt. Als nächstes kommt die Anzahl der Spieler im System. Darauffolgend kommen die Daten des Spieler1, Spieler2 usw. Die Daten eines Spielers bestehen aus folgenden Komponenten:<br>Id - wird benötigt um eine direkte Aufforderung zu schicken<br>Faktor - Ist das Maß für die Spielstärke eines Spielers<br>Name - Spielername der in der Liste der Chatansicht angezeigt wird<br>Fertig - gibt an, ob der Spieler die benötigten Grafiken geladen hat. Dieses wird in der Liste mittels einer Raute gekennzeichnet. |

Die für das Spiel benötigten Grafiken werden erst in der Chatansicht geladen werden. Dies hat den Vorteil, daß der Spieler sich schon mit anderen Spielern unterhalten kann und die Wartezeit auf die Grafiken überbrückt wird. In dieser Zeit kann der Spieler weder einen anderen Spieler herausfordern noch kann er herausgefordert werden. Hat der Spieler alle

Grafiken geladen, gibt er dem Spielserver Bescheid und dieser schickt die aktualisierte Spielerliste an alle Spieler. Der dazu benötigte String sieht wie folgt aus:

| | |
|---|---|
| *!!fertig!!,eigen_id* | Teilt dem Spielserver mit, daß alle Grafiken geladen sind und der Spieler spielbereit ist. |

Währen der Zeit des Ladens der Grafiken und danach, kann der Spieler sich mit anderen Spielern unterhalten. Schickt der Spieler eine Nachricht an alle anderen Teilnehmer, so wird dieser String an den Server geschickt: Der Server erkennt, daß es sich um keinen Befehl handelt, fügt den Namen des Spielers dazu und schickt diesen String an alle sich im Chatraum befindlichen Spieler.

Neben dem Unterhalten, kann ein Spieler auch einen anderen zu einem Spiel auffordern. Möchte ein Spieler einen anderen Spieler zu einem Spiel auffordern, so wählt dieser durch zweimaliges Anklicken den Namens des Gegners aus der Spielerliste aus. Das Spielerapplet fragt den Spieler noch einmal nach der Korrektheit des Gegners. Wird dieses bestätigt, sperrt das Spielerapplet weitere Anfragen und sendet folgenden String an den Spielserver:

| | |
|---|---|
| *!!befehle!!,geger_id,Spiel,eigen_id,name,* | Stringfolge zum Herausfordern eines Gegners mit der Id *gegner_id* von dem Spieler mit der Id *eigen_id.* |

Fängt ein String mit *!!befehle!!* an, gefolgt von gegner_id, weiß der Spielserver, daß es sich um eine Nachricht an den Spieler mit der Id *gegner_id* handelt. Der Spielserver interpretiert den darauffolgenden Teilstring und weiß in diesem Falle, daß er ein neues Board für die Spielsitzung erstellen muß. Hat er dieses getan, sendet er die Nachricht nur an den Spieler mit der Id *gegner_id*. Der vom Server ausgehende String sieht wie folgt aus:

| | |
|---|---|
| *!!befehle!!,Spiel,gegner_id,name,* | Hier ist *gegner_id* die Id und name der Name des Herausforderers. |

Erhält das Spielerapplet den oben genannten String, so weiß es, daß es sich um eine Herausforderung handelt. Als erstes wird geprüft, ob der Spieler nicht schon jemanden anderes herausgefordert hat oder schon eine andere Herausforderung vorher angekommen ist. In diesem Falle ist der Spieler für eine weitere Herausforderung gesperrt. Ist der Spieler gesperrt schickt das Spielerapplet automatisch eine Absage der folgenden Form:

| | |
|---|---|
| *!!befehle!!,gegner_id,Quitt,* | String für eine Absage aufgrund einer Blockierung |

Bei dem oberen String löscht der Spielserver das erstellte Board und sendet die Absage an den Spieler mit der Id *gegner_id*. Ist der Spieler nicht blockiert, so wird diesem die Herausforderung in der Chatansicht angezeigt und er kann einem Spiel zustimmen oder absagen. Sagt der Spieler ab, so wird folgender String an den Gegner gesendet:

| | |
|---|---|
| *!!befehle!!,gegner_id,Quitten,* | String für die Absage einer Herausforderung. |

In diesem Falle löscht der Spielserver das zuvor erstellte Board. Stimmt der Spieler einer Spielsitzung zu, so sendet das Spielerapplet folgenden String an den Spielserver und wechselt in die Spielfeldansicht.

| | |
|---|---|
| *!!befehle!!,gegner_id,Ok,name,* | String für das Annehmen einer Herausforderung |

Der Spielserver interprtiert den String, fügt den zweiten Spieler zum Board hinzu und sendet folgenden String an den Gegner:

| | |
|---|---|
| *!!befehle!!,OK,name,* | Bestätigung einer Herausforderung. |

Wird eine Herausforderung bestätigt, so wechselt das Spielerapplet in die Spielfeldansicht. Dort erhält der Spieler vom Spielserver noch einen weiteren String, welcher alle Infos beider Spieler enthält. Der String sieht wie folgt aus:

| | |
|---|---|
| !!befehle!!,Faktor,spieler2Faktor,spieler1Faktor,spiele | In diesem String werden die beiden Faktoren der Spieler und die Id des Gegners geschickt. |

| | |
|---|---|
| r2_id | |

Dauert einem Herausforderer die Antwort zu lange, kann dieser jederzeit die Anfrage zurücknehmen. Dazu schickt er folgenden String zum Spielserver:

| | |
|---|---|
| *!!befehle!!,gegner_id,Quitt 3,* | Aufhebung einer Aufforderung. |

Wird diese Art der Absage dem Spielserver geschickt, so löscht dieser das Board und sendet die Absage weiter an den Gegner. Bei diesem erscheint daraufhin die Meldung der Zurücknahme der Herausforderung.

Befinden sich beide Spieler in der Spielfeldansicht und ist das Board erstellt, so steht dem eigentlichem Spielen nichts mehr im Wege. Das Spielerapplet schickt folgenden String an den Spielserver, damit dieser die Spieler aus der Liste spielbereiter Spieler entfernt.

| | |
|---|---|
| *!!spielen!!,eigen_id,true,* | String, der den Spielserver dazu veranlaßt, den Spieler aus der Liste der spielbereiten Spieler entfernt. |

In der Spielfeldansicht spielen die Spieler gegeneinander. Dies erfolgt durch Setzen von Spielsteinen bzw. Blockern. Wird ein Spielstein an die Stelle x,y gesetzt, so wird folgender String an den Spielserver geschickt:

| | |
|---|---|
| *!!befehle!!,gegner_id,Stein ,x,y,* | Durch diesen String wird dem Gegner mit der Id *gegner_id* das Setzen eines Steines in die Spalte x und Zeile y mitgeteilt. |

Der Spielserver nimmt diese Mitteilung entgegen, protokolliert diese und sendet den folgenden String an den Spieler mit der Id gegner_id:

| | |
|---|---|
| *!!befehle!!,Stein,x,y,* | String, welcher vom Server zum Spieler geschickt wird. |

Analog dazu erfolgt das Setzen eines Bolckers durch folgenden String:

| | |
|---|---|
| *!!befehle!!,gegner_id,Block er,x,y,* | Durch diesen String wird dem Spieler mit der Id gegner_id das Setzen eines Blockers in die Spalte x und Zeile y mitgeteilt. |

Auch dieser String wird protokolliert und dann wie folgt weitergeleitet:

| | |
|---|---|
| *!!befehle!!,Blocker,x,y,* | String, welcher vom Server zum Spieler geschickt wird. |

In der Spielfeldansicht befindet sich ein Chatfenster. Die Mitteilungen werden wie folgt kodiert:

| | |
|---|---|
| *!!befehle!!,gegner_id,Chat, text,* | Dieser String schickt einen Text an den Spieler mit der Id gegner_id. |

Dieser String wird einfach nur durchgereicht. Beide Spieler können jederzeit das Spiel abbrechen. Wird ein Spiel abgebrochen, so wird folgender String an den Spielserver geschickt:

| | |
|---|---|
| *!!befehle!!,gegner_id,Quit,* | Durch diesen String wird dem Spieler mit der Id gegner_id der Abbruch des Spieles mitgeteilt. |

Erreicht dieser String den Spielserver, so wird das Board aufgelöst und folgender String weitergeleitet:

| | |
|---|---|
| *!!befehle!!,Quit,* | String welcher den Abbruch der Spielsitzung bewirkt. |

Nach einem Spielabbruch wechseln beide Spieler in die Chatansicht zurück. Hat ein Spieler die 200 Punkte Grenze erreicht, so hat dieser gewonnen. Die Mitteilung des Sieges bzw. der Niederlage kommt dabei von dem Spielserver. Die Strings sehen wie folgt aus:

| | |
|---|---|
| *!!befehle!!,Sieg,spieler1Fa ktor,spieler2Faktor* | Dieser String wird zum Sieger geschickt . |
| *!!befehle!!,Lost,spieler1Fa* | Dieser String wird zum Verlierer geschickt. |

| | |
|---|---|
| *ktor,spieler2Faktor* | |

Nachdem ein Spiel beendet worden ist können sich beide Spieler zu einem neuen Spiel entscheiden. Will ein Spieler ein erneutes Spiel so sendet das Spielerapplet folgenden String an den Spielserver:

| | |
|---|---|
| *!!befehle!!,gegner_id,Ok,* | String zum Bestätigen eines neuen Spieles. |

Wird von beiden Spielerapplets dieser String zum Spielserver geschickt, so erstellt der Spielserver ein neues Spiel. Wird dagegen von einem der Spieler eine Absage geschickt, so wird das Board gelöscht und beide Spieler wechseln in die Chatansicht. In diesem Falle muß dem Spielserver mitgeteilt werden, daß er die Spieler wieder in die Liste der spielbereiten Spieler aufnehmen muß. Dies Erfolg durch das Senden des folgenden Strings:

| | |
|---|---|
| *!!spielen!!,eigen_id,false,* | Durch diesen String wird dem Spielserver mitgeteilt, daß der Spieler mit der eigen_id sich nicht mehr in der Spielansicht befindet |

Sind beide Spieler in der Chatansicht zurück, so können sie sich einen neuen Gegner suchen. Möchte ein Spieler das System komplett verlassen, muß er nur das Appletfenster schließen. Als letzen möglichen String existieren folgende beiden:

| | |
|---|---|
| *ping* | Schickt der Server zum Client. |
| *OK* | Automatische Antwort eines Clients. |

Die oben gezeigten String werden dazu genutzt, um zu schauen ob ein Client noch aktiv ist. Der Spielserver enthält eine Routine, welche den String *ping* an alle Clients schickt. Bekommt ein Client diesen String, so antwortet dieser automatisch mit dem unterem String. Im anderen Fall erkennt der Spielserver die Inaktivität und entfernt den Spieler von dem Spielserver.

### Kommunikation Administratorapplet - Spielserver

Im oberen Abschnitt haben wir das Protokoll zwischen einem Spielerapplet und dem Spielserver betrachtet. In diesem Abschnitt betrachten wir das Protokoll zwischen dem Administratorapplet und dem Spielserver. Das Administratorapplet hat gegenüber dem Spielerapplet administrative Aufgaben. Von daher werden  von dem Spielserver nur Informationen abgerufen oder spezielle Befehle an den Spielserver geschickt.

Bevor der Administrator seinen Aufgaben nachkommen kann, muß er sich authentifizieren. Dazu existiert genau wie beim Spielerapplet eine Anmeldeansicht. Dort kann der Administrator allerdings nur das Paßwort eingeben. Der Name ist vorgegeben. Der zum Spielserver geschickte String sieht dann wie folgt aus:

| | |
|---|---|
| *Admin,passwd,* | Anmelde-String des Administartorapplets |

Aufgrund des Admin zu Beginn des Strings, weiß der Spielserver, daß es sich um den Administrator handelt. Stimmt das Paßwort nicht wird analog zum Spielerapplet eine entsprechende Meldung zurückgegeben. War die Anmeldung dagegen erfolgreich so wechselt der Administrator in die Adminansicht. Diese besteht aus drei Knöpfen. Der Knopf "Namen" sendet folgenden String an den Server:

| | |
|---|---|
| *!!namen!!* | Dieser String fordert die aktuelle Spielerliste von dem Spielserver an. |

Das Ergebnis ist eine Liste aller auf dem Spielserver befindlichen Spieler mit den Informationen der Spielerid, des Namens, des Faktors und dem Wert, ob die Grafiken geladen sind. Der Antwortstring von dem Server sieht dann wie folgt aus:

| | |
|---|---|
| *!!Namen!!,Spieler1Id Spieler1Namen Spieler1Faktor Spieler1Fertig\|....* | Antwortstring des Spielservers nach der Anfrage der Spielerliste. |

Ein weiterer Knopf ist "Tische". Dieser sendet folgenden String an den Spielserver:

| | |
|---|---|
| !!tische!! | Mit diesem String fordert das Administratorapplet die aktuelle Liste der Spielsitzungen von dem Spielserver an. |

Das Ergebnis dieser Anfrage ist eine Liste aller momentanen Spielsitzungen mit den entsprechenden Ids beider Spieler. Der Antwortstring des Spielservers sieht wie folgt aus:

| | |
|---|---|
| *!!Tische!!,Spieler1Id Gegner1Id\|...* | Antwortstring des Spielservers nach der Anfrage der Spielsitzungen. |

Der letzte Knopf ist ein Knopf zum Abmelden. Dieser bricht die Verbindung zum Server ab und wechselt wieder in die Anmeldeansicht. Der dazu gehörende String sich wie folgt aus:

| | |
|---|---|
| *!!abmelden!!* | String zum Beenden einer Administratorsitzung. |

Zusätzlich zu den oberen Befehlen zur Abfrage von Information, kann man noch eine Reihe von Befehle an den Spielserver schicken, die wie folgt aussehen:

| | |
|---|---|
| *!!dump!!,id,* | Mit diesem Befehl kann der Administartor den Spieler mit der Id *id* von dem Spielserver entfernen. |
| *!!passwd!!,name,passwd,* | Dieser Befehl ändert das Paßwort für den Spieler *name.* |
| *!!tisch!!,idspieler1,idspieler 2,* | Mit diesem Befehl kann der Administrator ein Spielsitzung auf dem Spielserver entfernen. |
| *!!sichern!!* | Mittels diesem Befehl kann der Administrator die Spielerliste von Hand aus sichern. |