TECHNISCHE
UNIVERSITÄT
DARMSTADT

Fachbereich Informatik
Fachgebiet Knowledge Engineering
Prof. Dr. Johannes Fürnkranz

Eine Grafische Benutzeroberfläche für ein Poker-Spiel

Bachelorarbeit

Max Bank

Prüfer: **Prof. Dr. Johannes Fürnkranz**
Betreuer: **Sang-Hyeun Park**

Darmstadt, 18.04.2011

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe alle Stellen, die ich aus den Quellen wörtlich oder inhaltlich entnommen habe, als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, am 18.04.2011

Zusammenfassung

Verfahren des maschinellen Lernens und der künstlichen Intelligenz werden gerne in Computerprogrammen angewendet, welche in bekannten Gesellschaftsspielen gegeneinander antreten, etwa bei einem Pokerspiel. Aus diesem Grunde wurde im Fachgebiet Knowledge Engineering der Technischen Universität Darmstadt ein Framework entwickelt, welches die nötigen Mechanismen für solche Wettkämpfe bereitstellt.

Dieses Framework verfügte jedoch bisher nur über eine sehr rudimentäre grafische Benutzeroberfläche, welche die Spielinformationen für den Benutzer nicht sehr anschaulich darstellt. Im Rahmen dieser Bachelorarbeit wurde daher eine neue Grafische Benutzeroberfläche mit einer besser verständlichen Visualisierung entwickelt, die außerdem noch weitere Funktionen integriert.

Inhaltsverzeichnis

1 Einleitung

Brettspiele, Kartenspiele oder allgemein Gesellschaftsspiele erfreuen sich stetiger Beliebtheit. In den allermeisten Fällen werden sie sicherlich zum Zeitvertreib und zur Pflege der Geselligkeit, also zum reinen Vergnügen, gespielt. Im Bereich der Informatik gibt es jedoch zahlreiche Forschungsgebiete, für die sich eine Beschäftigung mit Spielen ebenfalls anbietet. So lassen sich neu entwickelte Verfahren häufig auf die von einem Spiel vorgegebene Aufgabenstellung anwenden und können so erprobt und teilweise auch mit alternativen Ansätzen verglichen werden. Durch diese Anwendung in einem Spielszenario kann das jeweilige Verfahren besonders anschaulich dargestellt werden. Außerdem sind viele Spiele in weiten Teilen der Gesellschaft wohlbekannt, somit können die Verfahren auf diesem Wege häufig auch fachfremden Personen verständlich näher gebracht werden.

Ein Beispiel für ein Kartenspiel, welches sich gut zur Erprobung von Konzepten der künstlichen Intelligenz eignet, ist Poker. So werden auch im Fachgebiet Knowledge Engineering der Technischen Universität Darmstadt sogenannte *Bots* programmiert, Computerprogramme, die selbstständig den Spielverlauf in einem Pokerspiel erfassen und sich danach für eigene Spielzüge entscheiden können. Als sportlichen Anreiz nimmt das Fachgebiet regelmäßig an der Annual Computer Poker Competition [ACPC11] teil, einem Wettbewerb, bei dem die an verschiedenen Universitäten weltweit entwickelten Pokerbots gegeneinander antreten.

Nachdem die Pokergruppe des Fachgebiets früher die Serversoftware der Computer Poker Competition verwendet hatte, wurde schließlich im Rahmen einer Bachelorarbeit [Zopf10] ein eigener Pokerserver entwickelt. Dieser verfügt bereits über eine einfache, textbasierte Anzeige des Spielzustandes. Des Weiteren werden Logdateien angelegt, mit deren Hilfe der Spielverlauf nachvollzogen und ausgewertet werden kann. Der besseren Anschaulichkeit wegen wäre jedoch noch eine *Grafische Benutzeroberfläche (GUI)* hilfreich, die den Spielzustand visuell darstellt, ähnlich dem Pokertisch in einer realen Pokerrunde. Die vorliegende Bachelorarbeit beschäftigt sich mit der Entwicklung einer ebensolchen GUI. Es werden zunächst einige Grundlagen und die bereits vorhandene Software dargestellt. Anschließend wird die neue GUI gezeigt und ihre Bedienung erläutert. Es folgen Beschreibungen der Architektur und der Implementierung, bevor zum Schluss über die Arbeit resümiert wird.

2 Limit Texas Hold'em Poker

Im folgenden Abschnitt soll ein kurzer Einblick in die im Framework [Zopf10] implementierte Pokervariante *Limit Texas Hold'em*, die entsprechend auch von der zu entwickelnden GUI unterstützt werden soll, gegeben werden. Die Beschreibung beschränkt sich jedoch hauptsächlich auf die Spieleigenschaften, die für das Verständnis der vorliegenden Arbeit nötig sind, weitergehende Informationen finden sich etwa in [Warren07].

2.1 Spielaufbau

Limit Texas Hold'em Poker wird, wie andere Pokervarianten auch, mit 52 Standard-Spielkarten gespielt. Jede dieser Spielkarten verfügt über eine *Farbe* und einen *Wert*. An jeden Spieler werden zu Beginn 2 Karten ausgeteilt, die er für sich verdeckt behält, die sogenannten *Hole Cards*. In der Mitte des Tisches werden im Verlauf des Spiels 5 Karten aufgedeckt, die *Board Cards*. Außerdem befindet sich auf dem Tisch noch der *Pot*, also die gesammelten Wetteinsätze aller Spieler.

2.2 Spielphasen

Beim *Texas Hold'em Poker* gibt es fünf Spielphasen, welche jeweils nach den Board Cards benannt sind, die zuletzt neu aufgedeckt wurden. Die erste Spielphase, bei der noch keine Board Cards vorhanden sind, nennt man *Preflop*. Die zweite Phase, die mit dem Aufdecken der gleichnamigen ersten drei Board Cards beginnt, heißt *Flop*. Es folgen *Turn* und *River*, wobei wieder jeweils eine Board Card dazu kommt. Die letzte Spielphase nennt sich *Showdown* und wird in einem eigenen Abschnitt erklärt.

2.3 Spielziel

Es gewinnt das Spiel der Spieler, aus dessen Hole Cards sich zusammen mit den Board Cards eine Kombination aus 5 Karten zusammenstellen lässt, die bezüglich ihres Wertes die Kombinationen aller anderen Spieler übertrifft. Das Regelwerk, welches die Rangfolge der Kombinationen festlegt, ist nicht Teil dieser Arbeit. Da die Spieler jedoch weder auf die Board Cards noch auf ihre Hole Cards Einfluss haben, können sie lediglich anhand der bereits sichtbaren Karten die Wahrscheinlichkeit abschätzen, später eine gute Kombination bilden zu können. Entsprechend dieser Einschätzung agieren sie dann in den *Wettrunden*.

2.4 Blinds und Spielerpositionen

Eine wichtige Größe beim *Limit Texas Hold'em Poker* ist der *Small Blind (SB)*. Er gibt an, welchen Betrag der Spieler auf der ebenfalls SB genannten Position nach dem Austeilen der Karten in der Preflop-Phase setzen muss. Hat er dies getan, muss der Spieler links von ihm den doppelten Betrag, den *Big Blind (BB)*, setzen. Seine Spielpo-

sition wird entsprechend mit BB bezeichnet. Anschließend geht die *Wettrunde* mit dem Spieler weiter, der links von BB sitzt, die Namen der nachfolgenden Positionen haben keine tiefer gehende Bedeutung.

2.5 Wettrunde

Ist ein Spieler an der Reihe, hat er die Wahl zwischen den Aktionen *call*, *raise* und *fold*. *Call* heißt, genau den Betrag zu setzen, der nötig ist, um im Spiel zu bleiben. Dies ist der Betrag, der zuletzt von einem Spieler in dieser Wettrunde gesetzt wurde. Muss ein Spieler nicht setzen, entweder weil in dieser Runde noch kein Betrag gesetzt wurde oder der Spieler den Betrag bereits gesetzt hat, so sagt man *check* anstatt *call.*

Die zweite Möglichkeit nennt sich *raise*. Hier erhöht der Spieler den Einsatz, beim *Texas Hold'em Poker* mindestens um den Big Blind. Die nachfolgenden Spieler müssen nun ihren Einsatz ebenfalls aufstocken, um im Spiel zu bleiben. Den ersten *raise*, also wenn vorher noch niemand gesetzt hatte, nennt man *bet.*

Schließlich kann ein Spieler, der an der Reihe ist, auch noch aussteigen, genannt *fold*. In diesem Fall gibt er seine Karten ab, hat keine Chance mehr, den Pot zu gewinnen und ist erst beim nächsten Preflop wieder mit im Spiel.

2.6 Showdown

Ist die Wettrunde nach dem River abgeschlossen, so folgt der *Showdown*. Alle Spieler, die nicht vorher ausgestiegen sind und den Pot für sich reklamieren wollen, zeigen ihre Hole Cards. Die gemäß der Rangfolge beste 5-Karten-Kombination jedes Spielers aus seinen Hole Cards und den Board Cards wird mit den Kombinationen der anderen Spieler verglichen, und der Spieler mit der besten Kombination erhält den Pot. Verfügen mehrere Spieler über gleich gute Kombinationen, so wird der Pot entsprechend aufgeteilt, man spricht von einem *Splitpot*.

3 Bereits vorhandene Software

In diesem Abschnitt wird Software beschrieben, die mit der zu entwickelnden GUI zusammenhängt oder die eventuell sogar für die GUI teilweise wiederverwendet werden kann. Hier ist natürlich in erster Linie das Poker-Framework zu nennen, für welches die GUI geschrieben wird und welches mit ihrer Hilfe bedient werden soll. Daneben existiert im Fachgebiet aber auch noch ein Statistiktool, aus dem nach Möglichkeit einige Funktionalitäten in die GUI integriert werden sollen.

3.1 Poker-Framework

Das Framework wird ausführlich in [Zopf10] beschrieben. Es enthält einen Spielserver sowie die gesamte Funktionalität zum Erstellen und Durchführen von Spielen. Ebenfalls stellt es eine Infrastruktur auf Clientseite bereit, die die Kommunikation mit dem Server ermöglicht. In der vorliegenden Arbeit wird nur auf die für die Entwicklung der GUI relevanten Bereiche des Frameworks eingegangen. Es folgt zunächst ein Überblick über die entsprechenden Frameworkklassen.

3.1.1 Übersicht über die relevanten Frameworkklassen

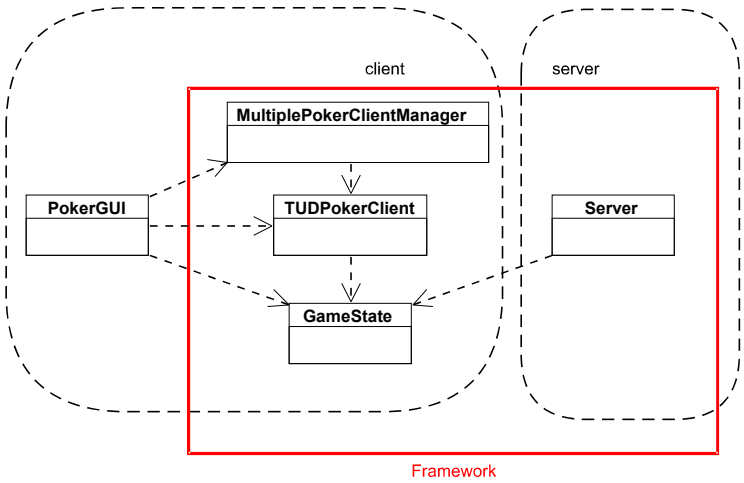

Abbildung 1: Relevante Frameworkklassen

Abbildung 1 zeigt die für die Entwicklung einer GUI wichtigen Klassen des Frameworks. Die Frameworkklassen sind rot eingerahmt, die Klasse `PokerGUI` steht stellvertretend für die GUI. Die Klasse `Server` ist zwar nicht wirklich relevant für die GUI, wurde aber zur Illustration des Client-Server-Konzepts trotzdem mit aufgenommen. Im Folgenden werden die Frameworkklassen genauer erläutert.

`MultiplePokerClientManager`: Dies ist die erste Anlaufstelle für jeden Client. Das Framework unterstützt nämlich zwei Arten von Clients, und es ist die Aufgabe des `MultiplePokerClientManagers`, wahlweise einen der beiden bereitzustellen.

`TUDPokerClient`: Der `TUDPokerClient` ist die Klasse, über die ein Client hauptsächlich mit dem Framework interagiert. Über den `TUDPokerClient` meldet sich ein Client am Server an und wird in der Folge über andere verbundene Spieler sowie laufende und beendete Spiele informiert. Außerdem lassen sich über ihn neue Spiele starten. Während eines Spiels erhält der Client vom `TUDPokerClient` Informationen über den aktuellen Spielzustand. Auch die Spielzüge, die der Client vornimmt, übermittelt er über diese Klasse an den Server.

`GameState`: Diese Klasse kapselt den aktuellen Spielzustand des laufenden Pokerspiels. Der Server sendet nach jeder Änderung des Spielzustands ein entsprechendes `GameState`-Objekt an jeden Client. `GameState` ist eine recht komplexe Klasse, die zur Abbildung des Spielzustandes zahlreiche weitere Klassen wie etwa `Player`, `Street` und `Cards` verwendet. Neben der reinen Speicherung des Spielzustands verfügt `GameState` aber auch über Hilfsfunktionen zur Auswertung. So lässt sich unter anderem der Nettogewinn jedes Spielers berechnen, vorausgesetzt, die aktuelle Spielphase ist der Showdown.

In den nachfolgenden Abschnitten wird näher auf die einzelnen Klassen eingegangen und ihre Benutzung erklärt.

3.1.2 MultiplePokerClientManager

Das Framework bietet die Möglichkeit, entweder den Pokerserver der Annual Computer Poker Competition [ACPC11] zu nutzen oder den frameworkeigenen Pokerserver. Der `MultiplePokerClientManager` verfügt über zwei Methoden, um den jeweils dazu passenden Pokerclient zu initialisieren. Da die GUI den frameworkeigenen Pokerserver verwenden soll, wird sie die Methode `initializeTUDPokerClient()` aufrufen. Dadurch wird eine Instanz der Klasse `TUDPokerClient` erzeugt, auf die anschließend durch die Methode `getClient()` zugegriffen werden kann.

3.1.3 TUDPokerClient

Der `TUDPokerClient` ist für den Client die Schnittstelle zum Server. Die Verbindung zu diesem wird über die Methode `init(String name, boolean active, InetAddress iaddr)` hergestellt. Beim Aufruf sind der gewünschte Spielername und die Serveradresse zu übergeben. Außerdem ist festzulegen, ob der Client aktiv oder passiv sein möchte. Nur aktive Clients können Spiele erstellen und starten, passive Clients müssen darauf warten, von einem aktiven Client als Mitspieler ausgewählt zu werden. Um Daten vom Server zu empfangen, startet die Methode einen eigenen Thread.

Abbildung 2 zeigt, welche Methoden jeweils in welchem Verbindungszustand zum Server aufgerufen werden können.

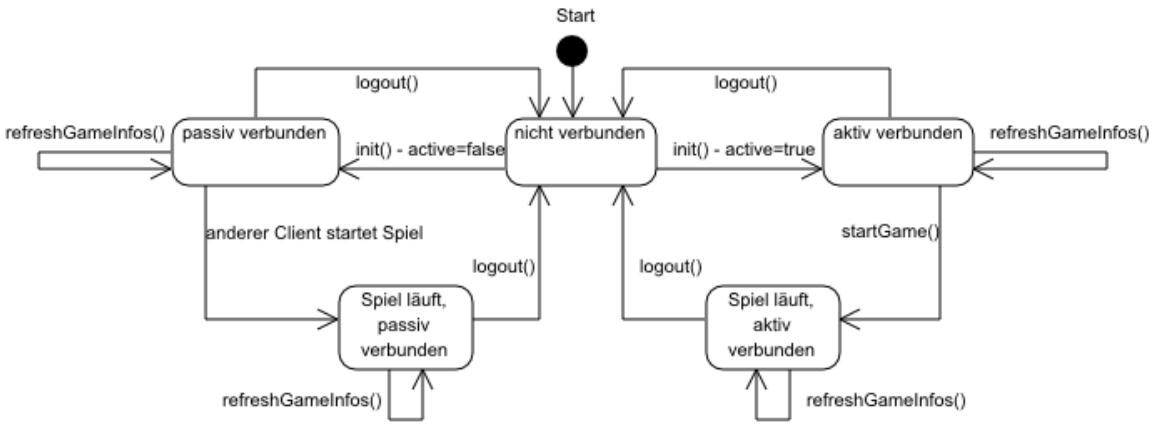

Abbildung 2: Verbindungszustände des TUDPokerClient

Ein Client, der mit einem Server verbunden ist, kann jederzeit die Methoden `refreshGameInfos()` oder `logout()` aufrufen. `refreshGameInfos()` erfragt beim Server die laufenden Spiele, die beendeten Spiele und die verbundenen Clients, die für neue Spiele zur Verfügung stehen. `logout()` beendet die Verbindung zum Server.

Hat sich der Client aktiv zum Server verbunden, so kann er mit Hilfe von `startGame(int rounds, LinkedList<Account> opponents, boolean spectateMode, boolean playPermutations, boolean showTruePlayernames)` ein Spiel starten. Dabei sind die Anzahl der zu spielenden Runden und die Accounts der Gegner anzugeben. Ein `true` als `spectateMode` bewirkt, dass der Client lediglich Zuschauer des Spiels ist, bei aktiviertem `playPermutations` werden alle Permutationen der Hole Cards erzeugt und als zusätzliche Runden gespielt. Damit das Framework die tatsächlichen Spielernamen an die Clients übermittelt, muss für `showTruePlayernames` `true` übergeben werden, ansonsten werden die Spielernamen anonymisiert.

3.1.4 GameState

Die Klasse `GameState` ist die für die Entwicklung einer GUI wichtigste Frameworkklasse, da sie alle Spielinformationen kapselt, die vom Server an den Client übertragen werden und von der GUI visualisiert werden sollen. `GameState` verwendet zur Repräsentation des Spielzustands einige Hilfsklassen, die nun zuerst vorgestellt werden, bevor schließlich die für diese Arbeit relevanten Methoden von `GameState` selbst erklärt werden.

`Action`: Dies ist ein Enum-Typ, der die möglichen Spielaktionen `RAISE`, `CALL` und `FOLD` repräsentiert. Es wird nicht zwischen call/check oder raise/bet unterschieden.

`Position`: Ebenfalls ein Enum-Typ, der zur Darstellung der verschiedenen Spielerpositionen verwendet wird. Mögliche Werte sind: `SB`, `BB`, `UTG_1`, `UTG_2`, `UTG_3`, `MP_1`, `MP_2`, `MP_3`, `CO`, `BU`, `INVALID`; `SB` und `BB` stehen für die Blinds, die nachfolgenden sind Abkürzungen der üblichen Positionsbezeichnungen. `INVALID` wird

etwa beim Showdown von der Methode `GameState.getPositionToAct()` verwendet, um anzuzeigen, dass gerade kein Spieler an der Reihe ist.

`Cards`: Diese Klasse dient der Repräsentation einer oder mehrerer Spielkarten. Dazu aggregiert die Klasse `Card`-Objekte, die jeweils für eine Spielkarte einer bestimmten Farbe und eines bestimmten Wertes stehen.

`Street`: Dieser Enum-Typ modelliert die aktuelle Spielphase eines Spielzustands. Mögliche Werte sind entsprechend `PREFLOP`, `FLOP`, `TURN`, `RIVER`, `SHOWDOWN` und `INVALID`. Letzteres wird vom Framework nur intern verwendet und tritt nicht in `GameState`-Objekten auf, die an den Client übertragen werden.

`Player`: In der Klasse `Player` werden alle spielerbezogenen Informationen gespeichert. Tabelle 1 zeigt die Methoden, die zur Visualisierung notwendige Attribute zurückgeben. So lassen sich der Name, die Hole Cards, die aktuelle Spielposition und die Wetteinsätze des Spielers erfragen. `isHero()` ergibt immer genau beim eigenen Spieler `true`. Nimmt man nur als Zuschauer an einem Spiel teil, so gibt es keinen Hero.

```
public double getInvestedThisStreet()
public boolean hasFolded()
public Position getPosition()
public Cards getHolecards()
public boolean isHero()
public String getPlayerName()
public double getInvested()
```

Tabelle 1: Wichtige Methoden der Klasse Player

`GameState`: Die Klasse `GameState` repräsentiert mit Hilfe von Instanzen der zuvor beschriebenen Klassen einen kompletten Spielzustand. Der Server sendet zu Beginn jeder Spielphase und nach jedem Spielzug ein `GameState`-Objekt an alle Clients. Tabelle 2 zeigt auch hier die für eine GUI relevanten Methoden. Spielerbezogene Daten müssen über `getPlayers()` oder `getPlayer()` direkt über die `Players`-Objekte bezogen werden. Eine Ausnahme davon ist die Methode `getOutcome()`, welche direkt am `GameState`-Objekt aufgerufen werden muss. Sie gibt den Nettogewinn beziehungsweise Nettoverlust eines Spielers zurück, also den Gewinnanteil am Pot abzüglich des Einsatzes. Bei einem `GameState`-Objekt der Spielphase Showdown sind die Spieler Gewinner, bei denen `getInvested()` und `getOutcome()` beide positive Werte liefern.

```
public double getPotsize()
public double getMaxBetsizeThisStreet()
public LinkedList<Action> getStreetAction(Street street)
public double getOutcome(String playerName)
public final Street getCurrentStreet()
public Position getPositionToAct()
public Player getPlayerToAct()
public Player getLastActingPlayer()
public int getActivePlayerCount()
public int getPlayerCount()
public boolean existsHero()
public Player getHero()
public boolean isHeroActing()
public Array List<Player> getPlayers()
public Player getPlayer(String playerName)
public Cards getFlop()
public Cards getTurn()
public Cards getRiver()
public Cards getBoard()
public final LinkedList<Action> getCurrentStreetAction()
public int getRoundIndex()
```

Tabelle 2: Wichtige Methoden der Klasse GameState

3.2 Analysewerkzeug

Neben dem Framework, das Spiellogik, Infrastruktur und den Server bereitstellt, ist noch eine weitere Software bereits im Fachgebiet vorhanden, die bei der Entwicklung der GUI mit einbezogen werden soll. Es handelt sich um ein Analysewerkzeug, welches noch für den Server der Computer Poker Competition [ACPC11] entwickelt wurde. Es wertet die vom Spielserver erstellten Logdateien aus und stellt sie grafisch dar. Abbildung 3 zeigt die tabellarische Ansicht von aus den Logdateien ermittelten Kennzahlen des ausgewerteten Spiels.

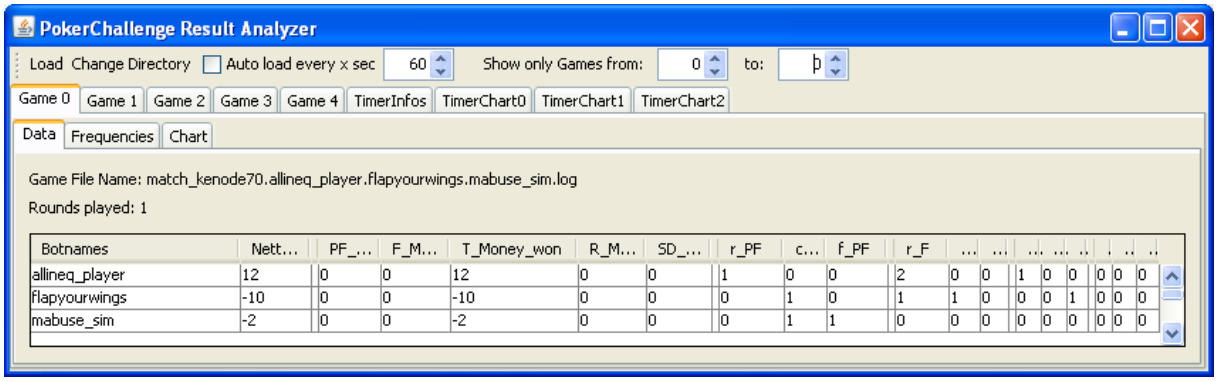

Abbildung 3: Datenansicht des Analysewerkzeugs

Hier werden die Gewinne und die Spielaktionen nach Spieler und nach Spielphase aufgeschlüsselt angezeigt. Die zweite Registerkarte enthält die Häufigkeiten der unterschiedlichen Spielereignisse, und auf einer dritten Registerkarte werden die akkumulierten Gewinne oder Verluste der Spieler in einem Graphen dargestellt. Das Analysewerkzeug bietet darüber hinaus noch die Möglichkeit, die Zugzeiten der Spieler

auszuwerten. Da diese jedoch vom neuen Framework nicht in der benötigten Form übermittelt werden, wird eine solche Funktionalität wohl nicht in die GUI mit übernommen werden. Wie man auf dem Bildschirmfoto erkennen kann, sollte das Layout der Tabelle noch überarbeitet werden, damit auch auf kleineren Bildschirmen möglichst alle Spaltenbezeichnungen sichtbar sind. Es sollten außerdem Bezeichnungen gewählt werden, die leichter verständlich sind.

4 Die Grafische Benutzeroberfläche und ihre Bedienung

Der besseren Anschaulichkeit wegen werden hier zunächst die grafischen Komponenten der Benutzeroberfläche dargestellt und ihre Bedienung erklärt, bevor in den Abschnitten 5 und 6 auf die Architektur und die Implementierung der GUI eingegangen wird.

4.1 Die Lobby

Nach dem Starten der GUI bekommt der Benutzer zuerst die sogenannte Lobby zu sehen. Wie in Abbildung 4 ersichtlich, ist sie in drei Bereiche unterteilt. In Bereich (1) wird die Verbindung zu einem Server hergestellt. Ist dies geschehen, werden in Bereich (2) Informationen über die zu diesem Server verbundenen Clients sowie auf dem Server laufende und abgeschlossene Spiele angezeigt. Ein aktiv verbundener Client kann zudem im Bereich (3) ein neues Spiel starten. Im Folgenden werden die drei Bereiche detailliert erklärt.

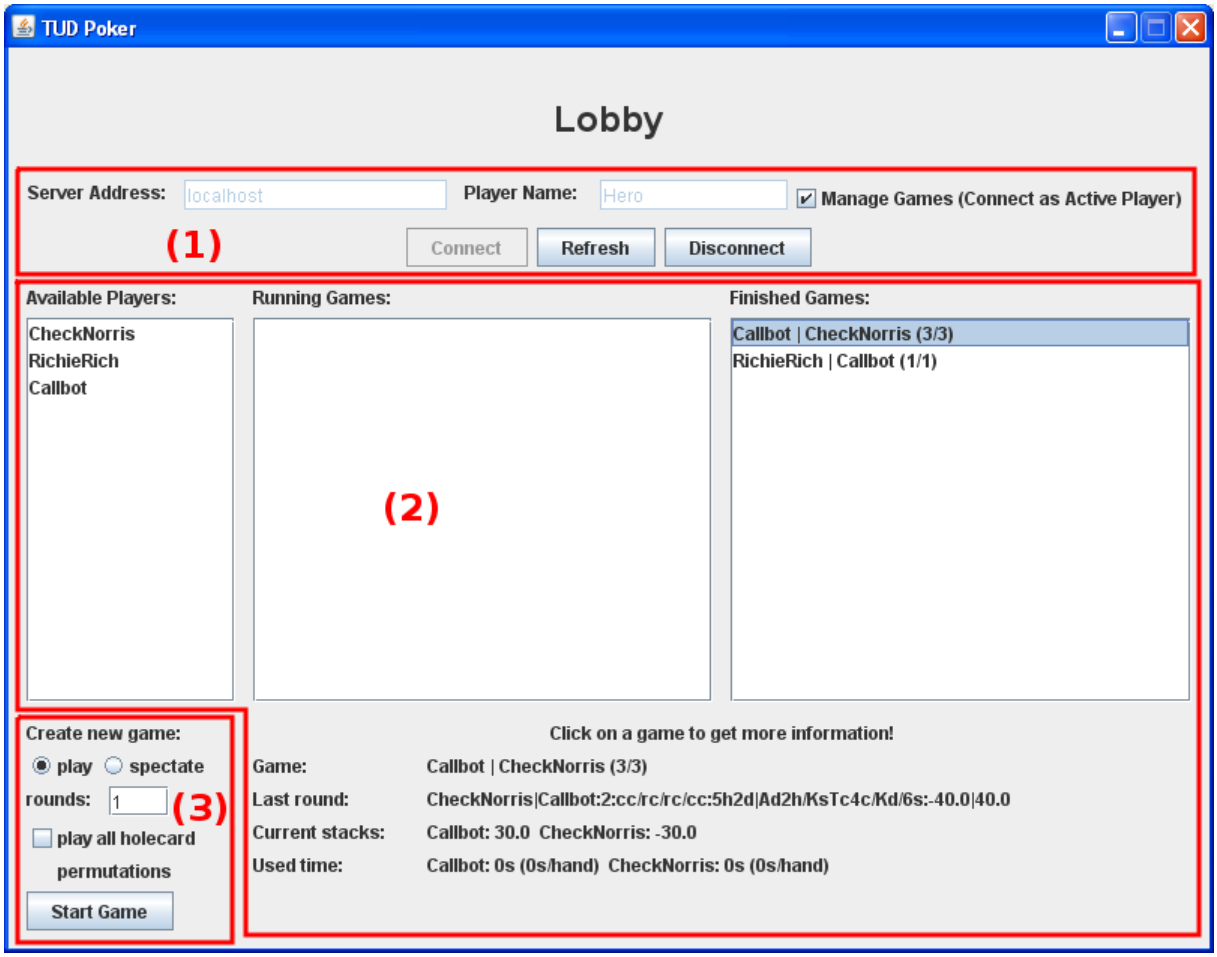

Abbildung 4: Die Lobby

4.1.1 Verbindung zum Server

Server Address: Hier ist die Adresse des Spielservers einzugeben, zu dem eine Verbindung hergestellt werden soll. Alternativ kann auch der Hostname angegeben werden.

Player Name: Hier ist ein frei wählbarer Name für den eigenen Spieler einzugeben. Der Name sollte jedoch eindeutig sein, falls schon ein Spieler mit demselben Namen mit dem Server verbunden ist, kann dies zu Problemen führen.

Manage Games (Connect as Active Player): Wenn dieser Haken gesetzt ist, wird im aktiven Modus zum Server verbunden. Dies ist die Voraussetzung, um eigene Spiele starten zu können.

Connect: Hiermit wird die Verbindung zum angegebenen Spielserver aufgebaut.

Refresh: Aktualisiert die Listen der Spieler sowie der laufenden und beendeten Spiele in Bereich (2).

Disconnect: Hiermit wird die Verbindung zum Spielserver getrennt.

4.1.2 Informationen zu Spielen und verbundenen Clients

Available Players: Hier werden alle passiven Clients angezeigt, die aktuell mit dem Server verbunden sind und für ein neues Spiel bereitstehen. Wurde im aktiven Modus zum Server verbunden, so können hier ein oder mehrere (durch Mausklick mit gedrückter STRG-Taste) Clients als Mitspieler für ein neues Spiel ausgewählt werden.

Running Games: Hier werden alle Spiele angezeigt, die aktuell auf dem Server laufen. Die Bezeichnung setzt sich zusammen aus den durch „|“ getrennten Namen der teilnehmenden Spieler, gefolgt von der Anzahl der gespielten Runden und der Gesamtrunden in Klammern.

Finished Games: Anzeige der beendeten Spiele auf dem Server, die Bezeichnung entspricht der der laufenden Spiele.

Beim Anklicken eines der laufenden oder abgeschlossenen Spiele werden die folgenden vom Framework gelieferten Detailinformationen zum jeweiligen Spiel angezeigt:

Game: Die Bezeichnung des markierten Spiels, entsprechend der Bezeichnung in der Liste.

Last Round: Hier wird der sogenannte *ResultString* der letzten abgeschlossenen Runde angezeigt. Dieser ist nach dem Schema

`Spielernamen:Rundenindex:Spielhistorie:Karten:Ergebnis` zusammengesetzt. `Spielernamen` entspricht den mit „|“ voneinander getrennten Namen der beteiligten Spieler. `Rundenindex` ist die Nummer der aktuellen Runde, beginnend bei null. In der `Spielhistorie` werden die Spielaktionen der gespielten

Wettrunden dargestellt, nach jeder Wettrunde folgt ein „/“. Dabei steht „c“ für ein call/check, „r“ für ein raise/bet und „f“ für ein fold. Bei den `Karten` werden erst die durch „|“ voneinander getrennten Hole Cards der Spieler angezeigt, dann die jeweils durch ein „/“ getrennten Flop-, Turn- und Riverkarten. Jede Karte wird dargestellt durch den Anfangsbuchstaben der englischen Wertbezeichnung gefolgt vom Anfangsbuchstaben der englischen Farbbezeichnung. Im `Ergebnis` schließlich stehen die Nettogewinne/-verluste der Spieler in deren vorheriger Reihenfolge, wieder getrennt durch „|“.

Current Stacks: Hier werden die aufsummierten Gewinne/Verluste der Spieler aus allen gespielten Runden angezeigt.

Used Time: Anzeige der Zeiten, die die Spieler bisher insgesamt zum Ziehen gebraucht haben.

4.1.3 Starten eines neuen Spiels

play/spectate: Hier ist auszuwählen, ob man in dem neuen Spiel selber spielen oder nur zuschauen möchte.

rounds: Die Anzahl der zu spielenden Runden.

play all holecard permutations: Ist hier ein Haken gesetzt, dann werden alle Permutationen der Hole Cards gespielt. Hierbei werden zunächst die gewählte Anzahl Runden wie gewöhnlich gespielt. Danach werden die Hole Cards und gleichzeitig parallel die Spielpositionen der Spieler getauscht, und es wird nochmal dieselbe Anzahl Runden gespielt. Dies geht solange, bis jeder Spieler alle Runden mit jedem der ausgeteilten Handkarten-Paare gespielt hat.

Start Game: Ein Klick auf diese Schaltfläche startet ein neues Spiel mit den zuvor gewählten Parametern.

4.2 Die Spielansicht

Sobald vom Server ein GameState-Objekt eines neu gestarteten Spiels empfangen wird, wird von der Lobbyansicht auf die Spielansicht gewechselt. Abbildung 5 zeigt ein Bildschirmfoto der Spielansicht. Sie ist einem realen Pokertisch nachempfunden. Das Hintergrundbild wurde von dem OpenSource-Pokerspiel poker-network[1] übernommen. In der Mitte des Tisches werden die Board Cards (1) angezeigt, links daneben (2) der aktuelle Pot. Die Bilder für die Spielkarten entstammen dem Pokerspiel PokerTH[2], welches ebenfalls unter einer OpenSource-Lizenz steht.

(3), (4) und (5) zeigen die Darstellung der Spieler. Hinter dem Spielernamen steht in Klammern die Abkürzung der Spielposition. Darunter wird der aktuelle Stack, also die über die bisher gespielten Runden aufsummierten Gewinne/Verluste des Spielers angezeigt. Unterhalb des Stacks erscheinen die Board Cards. Nimmt der Nutzer selber am Spiel teil, so bekommt er immer den Platz unten zugewiesen. Die

[1] http://pokersource.info

[2] http://www.pokerth.net

Abbildung 5: Die Spielansicht

restlichen Spieler werden in Reihenfolge ihrer Spielpositionen möglichst gleichmäßig um den Tisch verteilt. Bei dem gezeigten Beispiel ist gerade Spieler „Callbot“ (4) an der Reihe, was durch die gelbe Einfärbung dargestellt wird. Davor war der Nutzer selber an der Reihe und hat gesetzt, was durch das „bets“ über seinen Hole Cards visualisiert wird. Der Spieler „CheckNorris“ (5) ist bereits aus dem Spiel ausgestiegen, in diesem Fall werden die Hole Cards nicht mehr angezeigt. Während einer Wettrunde werden die Einsätze zuerst neben dem jeweiligen Spieler angezeigt (6), bevor sie am Ende der Wettrunde zum Pot addiert werden.

In dem Fall der aktiven Teilnahme am Spiel erscheinen rechts neben dem eigenen Spieler die Schaltflächen zum Tätigen der Spielaktionen (7). Diese sind jedoch nur anwählbar, wenn der Nutzer gerade an der Reihe ist.

Am unteren Rand der Spielansicht befindet sich noch eine Reihe von Steuerelementen (8), die im Folgenden erklärt werden.

go on: Hiermit kann das Spiel zu jedem Zeitpunkt pausiert oder fortgesetzt werden. Die Beschriftung der Schaltfläche wechselt je nach Zustand zwischen „pause“ und „go on“.

pause on showdown: Ist hier ein Haken gesetzt, so wird am Ende jeder Runde automatisch pausiert.

speed: Hier lässt sich einstellen, wie lange jeder Spielzug angezeigt werden soll.

statistics: Hierüber kann das Statistik-Fenster ein- und ausgeblendet werden, welches noch vorgestellt werden wird.

history: Hierüber kann das Fenster mit der Spielhistorie ein- und ausgeblendet werden. Dieses wird im nächsten Abschnitt vorgestellt.

quit: Hiermit verlässt man das aktuelle Spiel und kehrt zur Lobby zurück. Falls das Spiel auf dem Server zu diesem Zeitpunkt noch nicht beendet ist, wird automatisch die Verbindung zum Server getrennt, da das Framework eine anderweitige vorzeitige Beendigung eines Spiels nicht unterstützt.

Ist eine Runde zu Ende, so wird gemäß Abbildung 6 über den Board Cards eingeblendet, welcher Spieler wie viel gewinnt.

Abbildung 6: Gewinnanzeige beim Showdown

4.3 Die Spielhistorie

Damit der Nutzer während eines laufenden Spiels jederzeit den bisherigen Spielverlauf einsehen kann, bietet die GUI eine Spielhistorie, die in der Spielansicht ein- und ausgeblendet werden kann. Abbildung 7 zeigt, wie diese aussieht.

Zu Beginn jeder Runde wird eine allgemeine Informationszeile mit Rundennummer, Anzahl der teilnehmenden Spieler und der aktuellen Uhrzeit ausgegeben. Anschließend folgt die Ausgabe der einzelnen Spielphasen, jeweils durch eine Leerzeile voneinander getrennt. Die Zahlen in Klammern hinter der Bezeichnung der Spielphase geben den momentanen Stand des Pots an (beim Preflop sind die Blinds bereits enthalten). Nimmt der Nutzer sebst am Spiel teil, so werden beim Preflop noch die eigene Spielposition sowie die eigenen Hole Cards ausgegeben. Bei allen weiteren Spielphasen werden an dieser Stelle die neu aufgedeckten Board Cards sowie die Anzahl der Spieler angezeigt, die noch im Spiel sind. Die Spielaktionen des eigenen Spielers, falls vorhanden, werden stets in blau dargestellt, die der übrigen Spieler in rot.

Abbildung 7: Die Spielhistorie

4.4 Die Statistik

Das in Abschnitt 3.2 beschriebene Analysewerkzeug wurde überarbeitet und für die GUI angepasst und kann nun wie die Spielhistorie in der Spielansicht eingeblendet werden. Es wertet alle abgeschlossenen Runden des laufenden Spiels aus, Abbildung 8 zeigt zunächst die Tabellenansicht.

In der ersten Zeile wird angezeigt, wie viele Runden gespielt wurden und zu welchen Anteilen diese in den unterschiedlichen Spielphasen endeten. Es folgt eine Übersicht über die Nettogewinne/-verluste der einzelnen Spieler und eine Aufschlüsselung auf die einzelnen Spielphasen. Anschließend werden für jede Spielphase in einer eigenen Tabelle weitergehende Informationen angezeigt.

Statistics

Data | Chart

Rounds played: 5; 0,0% ended on Preflop, 0,0% on Flop, 0,0% on Turn, 0,0% on River, 100,0% on Showdown

Money:

Playername	Nettowin total	Preflop	Flop	Turn	River	Showdown
Callbot	100.0	0.0	0.0	0.0	0.0	100.0
CheckNorris	-175.0	0.0	0.0	0.0	0.0	-175.0
Hero	75.0	0.0	0.0	0.0	0.0	75.0

Preflop:

Playername	Played	Wins	% won	Raises	Avg. Raises	Calls	Avg. Calls	Folds	% folded
Callbot	5	0	0,0%	0	0,00	8	1,60	0	0,0%
CheckNorris	5	0	0,0%	2	0,40	5	1,00	1	20,0%
Hero	5	0	0,0%	5	1,00	1	0,20	0	0,0%

Flop:

Playername	Played	Wins	% won	Raises	Avg. Raises	Calls	Avg. Calls	Folds	% folded
Callbot	5	0	0,0%	0	0,00	7	1,40	0	0,0%
CheckNorris	4	0	0,0%	3	0,75	5	1,25	0	0,0%
Hero	5	0	0,0%	4	0,80	1	0,20	0	0,0%

Turn:

Playername	Played	Wins	% won	Raises	Avg. Raises	Calls	Avg. Calls	Folds	% folded
Callbot	5	0	0,0%	0	0,00	7	1,40	0	0,0%
CheckNorris	4	0	0,0%	0	0,00	6	1,50	1	25,0%
Hero	5	0	0,0%	5	1,00	0	0,00	0	0,0%

River:

Playername	Played	Wins	% won	Raises	Avg. Raises	Calls	Avg. Calls	Folds	% folded
Callbot	5	0	0,0%	0	0,00	6	1,20	0	0,0%
CheckNorris	3	0	0,0%	0	0,00	4	1,33	1	33,3%
Hero	5	0	0,0%	2	0,40	3	0,60	0	0,0%

Showdown:

Playername	Played	Wins	% won
Callbot	5	3	60,0%
CheckNorris	2	1	50,0%
Hero	5	3	60,0%

Abbildung 8: Die Tabellenansicht der Statistik

Die Spalten im Detail:

Played: Anzahl der Runden, in denen der Spieler diese Spielphase gespielt hat.

Wins: Anzahl der Runden, die der Spieler in dieser Spielphase gewonnen hat.

% won: Prozentsatz der gewonnenen Runden im Verhältnis zu allen Runden, in denen der Spieler diese Spielphase gespielt hat.

Raises: Anzahl der Raises in dieser Spielphase.

Avg. Raises: Durchschnittliche Anzahl der Raises pro gespielter Spielphase.

Calls: Anzahl der Calls in dieser Spielphase.

Avg. calls: Durchschnittliche Anzahl der Calls pro gespielter Spielphase.

Folds: Anzahl der Folds in dieser Spielphase.

% folded: Prozentsatz der Runden, in denen Spieler in dieser Spielphase ausgestiegen ist, im Verhältnis zu den Runden, in denen der Spieler diese Spielphase gespielt hat.

Auf der zweiten Registerkarte des Statistikfensters wird in einem Graphen dargestellt, wie sich die Stacks der Spieler entwickelt haben. Abbildung 9 zeigt ein entsprechendes Beispiel.

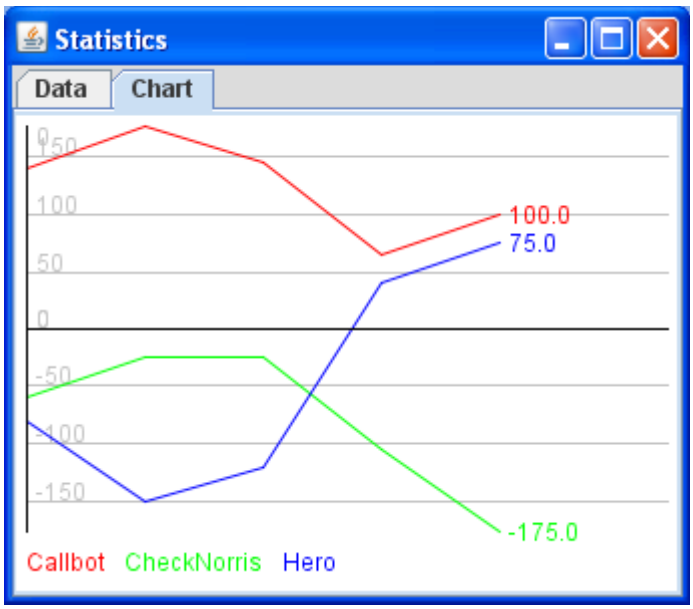

Abbildung 9: Die Graphansicht der Statistik

5 Architektur

Der folgende Abschnitt stellt zunächst kurz die Eigenschaften dar, welche für die technische Umsetzung der Grafischen Benutzeroberfläche als erstrebenswert befunden wurden. Anschließend werden zwei Entwurfsmuster erklärt und es wird gezeigt, wie diese bei der Architektur der Software angewandt wurden.

5.1 Designziele

Bei der technischen Umsetzung der Software wurde vor allem Wert auf einfache Verständlichkeit und Nachvollziehbarkeit gelegt. Um dies zu erreichen, wurden insbesondere drei Prinzipien angewandt, welche in den nachfolgenden Unterabschnitten erklärt werden.

5.1.1 Lose Kopplung

Das Prinzip der losen Kopplung (*low coupling*) wird etwa in [Larman05, 17.12] näher beschrieben. Die Kopplung einer Klasse ist demnach umso höher, je größer die Anzahl der Klassen ist, von denen die Klasse abhängig ist.

Eine lose Kopplung, also Abhängigkeit von nur wenigen Klassen, hat nun einige wünschenswerte Konsequenzen. Zum einen lassen sich Klassen mit nur wenigen Abhängigkeiten leichter wiederverwenden, da weniger weitere Klassen notwendig sind. Darüber hinaus müssen sie seltener an Änderungen in den Klassen angepasst werden, von denen sie abhängig sind, da die Wahrscheinlichkeit von Änderungen umso kleiner ist, je geringer die Anzahl dieser Klassen ist. Schließlich, und dieser Aspekt ist für das Erreichen der oben definierten Designziele besonders hilfreich, erleichtert lose Kopplung richtig angewandt auch das Verständnis einer Klasse. Häufig lässt sich das Verhalten einer Klasse nämlich nur dann nachvollziehen, wenn man außerdem über die Klassen Bescheid weiß, zu denen Abhängigkeiten bestehen. Lose Kopplung verringert somit die Anzahl der Klassen, die man gleichzeitig betrachten muss, um eine Klasse zu verstehen.

Es gilt allerdings, bei der losen Kopplung das richtige Maß zu finden, denn im Extremfall lässt sich auch die gesamte Funktionalität einer Software in einer einzigen Klasse implementieren, was zwar die Kopplung auf null reduziert, für die Übersichtlichkeit aber wenig förderlich ist.

5.1.2 Hoher logischer Zusammenhalt

Ein ähnlich allgemeines Prinzip, welches bei der Aufteilung von Verantwortlichkeiten auf Klassen hilfreich ist, ist das Prinzip des hohen logischen Zusammenhalts (*high logical cohesion*) [Larman05, 17.14].

Es besagt, dass alle in einer Klasse implementierten Methoden der Erfüllung einer wohldefinierten und abgegrenzten Aufgabe dienen sollten. Dass die Einhaltung die-

ses Prinzips eine leichte Verständlichkeit der Klasse begünstigt, ist offensichtlich. Da das Prinzip des hohen logischen Zusammenhalts gegen die im vorigen Abschnitt angesprochene Konzentration vieler voneinander unabhängiger Funktionalitäten in einer Klasse spricht, kann der Eindruck entstehen, lose Kopplung und hoher logischer Zusammenhalt seien gegensätzliche Prinzipien. Tatsächlich sind jedoch beide weitgehend unabhängig voneinander und müssen auch entsprechend einzeln angewendet werden.

5.1.3 Anwendung allgemein bekannter Entwurfsmuster

Besonders gut nachvollziehbar ist eine Implementierung, wenn sie sich an allgemein bekannte Schemata hält. So gibt es zahlreiche etablierte sogenannte Entwurfsmuster [Gamma09], welche jeweils für ein Problem eine standardisierte Lösungsstrategie liefern, die bereits vielfach erprobt wurde und sich bewährt hat. Sowohl das Entwurfsmuster als Ganzes als auch die Komponenten, aus denen es zusammengesetzt ist, besitzen eindeutige Bezeichnungen. Somit können die Stellen der Implementierung, an denen Entwurfsmuster angewendet wurden, einfach und eindeutig dokumentiert werden. Jeder, der anschließend die Dokumentation liest und die Entwurfsmuster kennt, wird die entsprechenden Zusammenhänge sofort erkennen und einordnen können.

In den folgenden zwei Abschnitten werden zwei Entwurfsmuster dargestellt, die die Architektur der GUI maßgeblich mitbestimmen.

5.2 Das Entwurfsmuster Observer

Das Entwurfsmuster Observer [Gamma09, S. 287ff.] liefert ein Schema, nach dem ein oder mehrere Objekte über Änderungen an einem anderen Objekt informiert werden können, und dies bei möglichst geringer Kopplung der Objekte untereinander. Abbildung 10 zeigt die Struktur des Entwurfsmusters.

Die Objekte `ConcreteObserver1` und `ConcreteObserver2` beobachten eine Instanz der Klasse `Subject`. Diese enthält in ihrem Feld `observers` Referenzen auf ihre Beobachter. Nach jeder Veränderung seines Zustands ruft `Subject` die Methode `notifyObservers()` auf. Dadurch wird jedem `Observer` aus `observers` durch Aufruf der Methode `update()` die Gelegenheit gegeben, den aktuellen Zustand von `Subject` zu erfragen.

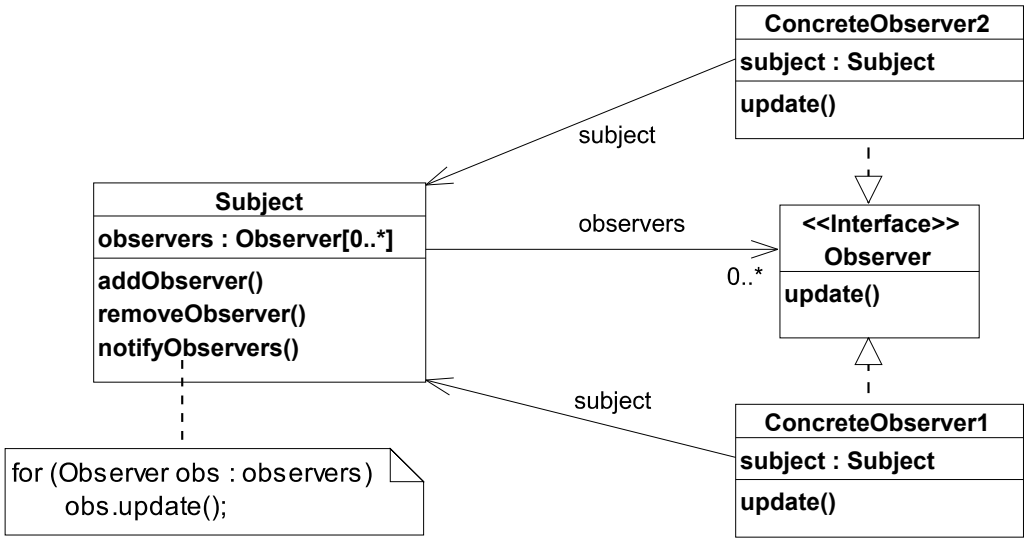

Abbildung 10: Die Struktur des Entwurfsmusters Observer

Um die Kopplung der zu beobachtenden Klasse zu minimieren, implementieren alle Beobachter die Schnittstelle `Observer`, so dass `Subject` nur eine Abhängigkeit darauf aufweist.

5.3 Das Entwurfsmuster Model-View-Controller

Das Entwurfsmuster Model-View-Controller geht zurück auf [Reenskaug79] und wurde dort noch als Thing-Model-View-Editor bezeichnet. Es definiert drei funktionale Bereiche, in die sich eine Software mit grafischer Benutzeroberfläche häufig gut unterteilen lässt. Abbildung 11 zeigt die drei Bereiche *Model*, *Controller* und *View*.

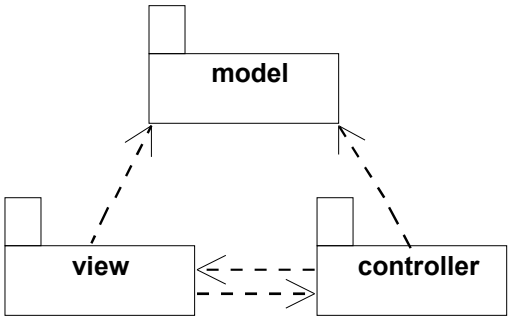

Abbildung 11: Die Struktur des Entwurfsmusters Model-View-Controller

Das ***Model*** beinhaltet alle Daten, mit denen die Software arbeitet, sowie die Logik, nach der die Daten verarbeitet werden.

Die ***View*** visualisiert das *Model* gegenüber dem Nutzer und stellt außerdem Schaltflächen für Benutzereingaben zur Verfügung.

Der ***Controller*** nimmt Benutzereingaben entgegen und ruft die entsprechenden Methoden im *Model* auf.

Während *View* und *Controller* in der Regel wechselseitig voneinander abhängig sind, sollte das *Model* zu keinem von beiden Abhängigkeiten aufweisen. Dies hat den Vorteil, dass das *Model* auch unabhängig von der *View* entwickelt und mit verschiedenen *View*-Implementierungen oder ganz ohne *View* etwa im Kommandozeilenmodus verwendet werden kann.

5.4 Die Architektur der Grafischen Benutzeroberfläche

Abbildung 12 zeigt anhand der wichtigsten Klassen, wie die Entwurfsmuster *Observer* und *Model-View-Controller* bei der Architektur der grafischen Benutzeroberfläche angewandt wurden.

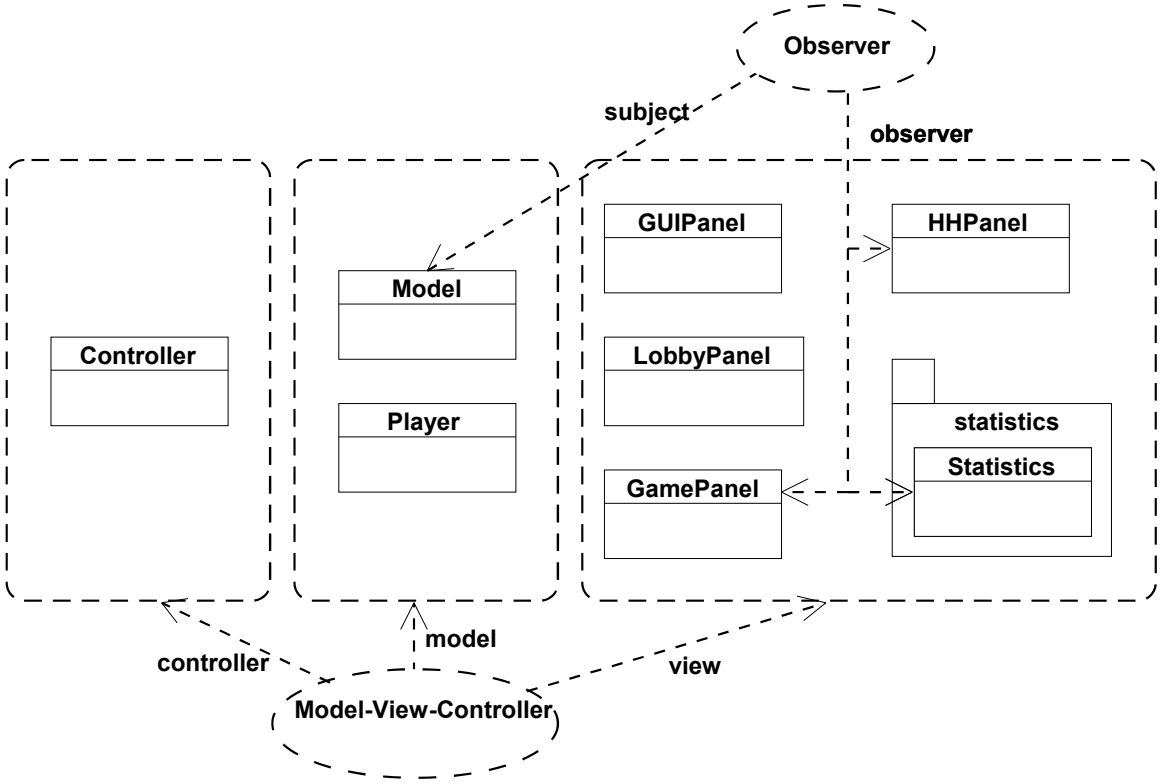

Abbildung 12: Die Architektur der Grafischen Benutzeroberfläche

Als *Controller* fungiert die gleichnamige Klasse, das *Model* besteht aus der Klasse `Model` und der Hilfsklasse `Player`, in der spielerbezogene Daten gespeichert werden. Die Klassen `GUIPanel`, `LobbyPanel`, `GamePanel`, `HHPanel` und `Statistics` sind für jeweils eine Ansicht zuständig und bilden zusammen die *View*. Für die Statistik-Funktionalität wurden mehrere Klassen aus dem in Abschnitt 3.2 beschriebenen Analysewerkzeug übernommen und einem eigenen Package `statistics` zugeordnet.

Gemäß dem Entwurfsmuster *Observer* sind die Klassen `GamePanel`, `HHPanel` und `Statistics` Beobachter der Klasse `Model`, welche den Spielzustand speichert, und werden über dessen Änderungen informiert.

6 Implementierung

Im nun folgenden Abschnitt werden die Implementierung der drei Subsysteme Model, Controller und View sowie zwei weitere Klassen zum Starten der Grafischen Benutzeroberfläche im Detail dargestellt.

6.1 Model

Das Model-Subsystem wurde in den drei Klassen `Model`, `Player` und `SeatAssigner` implementiert. Die Klasse `Model` speichert alle benötigten Spielinformationen, für die spielerbezogenen Informationen werden `Player`-Instanzen referenziert. `SeatAssigner` ist für die Verteilung der Spieler auf die Sitze zuständig.

6.1.1 Die Klasse Model

Die Klasse Model hat die Aufgabe, alle für die GUI relevanten Spielinformationen zu speichern und den grafischen Anzeigekomponenten zur Verfügung zu stellen. Dazu wird immer das zuletzt vom Server erhaltene `GameState`-Objekt im Feld `displayedGameState` gespeichert, siehe dazu das Klassendiagramm Tabelle 3.

Model
-controller : Controller -players : Player[] -resultStringHistory : List<String> -displayedGameState : GameState -oldGameState : GameState
+reset() : void +updateGamestate(gameState : GameState) : void -newGame() : void +getDisplayedGameState() : GameState +getLastActivePlayer() : Player +getPlayers() : Player[] -getPlayer(playerName : String) : Player +getLastAction() : Action +getHero() : Player +getLastActionString() : String +getBruttoWin(player : Player) : double +getResultStringHistory() : List<String>

Tabelle 3: Klassendiagramm der Klasse Model

Neben den Spielinformationen, die direkt vom aktuellen `GameState`-Objekt geliefert werden, benötigt die GUI aber noch weitere Informationen, wie etwa den Spieler, der zuletzt an der Reihe war oder die Summe aller Gewinne/Verluste jedes Spielers für die Anzeige der Stacks. Deshalb wird im Model zusätzlich der vorherige

GameState im Feld oldGameState gespeichert und es werden in dem Array players weitergehende Informationen zu den Spielern in einer eigenen Klasse Player vorgehalten. Der Zugriff auf diese Player-Objekte ist über die Methoden getLastActivePlayer(), getPlayers(), getPlayer() und getHero() möglich, als Hero wird immer der eigene Spieler des Nutzers bezeichnet. Daneben besitzt die Klasse Model noch einige weitere Methoden, die Spielinformationen aus displayedGameState oder oldGameState extrahieren und für die Anzeige bereitstellen. getLastAction() liefert die letzte Spielaktion als Action-Objekt, getLastActionString() gibt diese als String zurück, wobei der aktuelle Spielkontext berücksichtigt wird und gegebenenfalls „check“ statt „call“ beziehungsweise „bet“ statt „raise“ zurückgegeben wird. Da die Methode getOutcome() in der GameState-Klasse nur den Nettogewinn eines Spielers liefert, gibt es in Model noch eine Methode getBruttoWin(), die die Summe aus Nettogewinn und Einsatz zurückgibt, so wie sie etwa für die Gewinnanzeige benötigt wird. Im Feld resultStringHistory werden die Spielergebnisse aller abgeschlossenen Spielrunden in String-Form gespeichert und über die entsprechende get-Methode zugänglich gemacht, welche dann von der Statistik aufgerufen wird.

Wenn ein Spiel beendet wird, sollte die Methode reset() aufgerufen werden, welche die Datenfelder leert, beziehungsweise mit null überschreibt, um anzuzeigen, dass das Model aktuell keinen Spielzustand speichert. Die Methode newGame() schließlich wird zu Beginn eines neuen Spiels aufgerufen, legt im Feld players neue Player-Instanzen an und weist ihnen Sitzplätze am Pokertisch zu.

6.1.2 Die Klasse Player

Neben den Informationen, die schon zusätzlich im Model gespeichert werden und eher den Gesamt-Spielzustand betreffen, sind für die Visualisierung auch noch weitere Daten relevant, die sich auf den einzelnen Spieler beziehen. Konkret muss für jeden Spieler noch seine Sitzposition am Pokertisch und der Betrag seines Stacks gespeichert werden, diese Informationen sind in der pokertud.gamestate.Player-Klasse des Frameworks nicht enthalten. Um dabei dem Prinzip der Objektorientierung zu entsprechen, wurde hierfür eine eigene Klasse Player erstellt, die von der GUI an Stelle der frameworkeigenen Klasse verwendet wird.

Wie das Klassendiagramm in Tabelle 4 zeigt, enthält jede Player-Instanz im Feld gsPlayer einen Verweis auf das ursprüngliche pokertud.gamestate.Player-Objekt, welches als Teil des aktuellen GameState vom Server empfangen wurde. Bei jedem GameState-Update wird das bisher gespeicherte mit Hilfe der Methode setGsPlayer() durch das neu empfangene ersetzt. Alle Funktionalitäten, die die Frameworkklasse bereitstellt und die von der GUI benötigt werden, bietet die neue Klasse Player unter jeweils demselben Methodennamen an und leitet entsprechende Aufrufe an das interne gsPlayer-Objekt weiter.

Die einzigen Methoden, die gegenüber denen der Frameworkklasse neu hinzugekommen sind, sind außer setGsPlayer() nur die get- und set-Methoden, die für

Player
-seat : int -stack : double -gsPlayer : pokertud.gamestate.Player
+Player(gsPlayer : pokertud.gamestate.Player) +getPlayerName() : String +getPosition() : Position +hasFolded() : boolean +getSeat() : int +setSeat(seat : int) : void +setGsPlayer(gsPlayer : pokertud.gamestate.Player) void +getStack() : double +setStack(stack : double) : void +toString() : String +getInvested() : double +getInvestedThisStreet() : double +isHero() : boolean +getHolecards() : Cards

Tabelle 4: Klassendiagramm der Klasse Player

den Zugriff auf die Felder `seat` und `stack` zuständig sind. Das Feld `seat` enthält den Index des zugewiesenen Sitzplatzes am Pokertisch, also eine Ganzzahl zwischen null und sieben. Im Feld `stack` wird zu jeder Zeit die Summe der Gewinne und Verluste des Spielers aus allen abgeschlossenen Runden gespeichert. Der Wert ändert sich also immer nur beim Showdown am Ende jeder Runde.

6.1.3 Die Klasse SeatAssigner

Die Klasse `SeatAssigner` hat einzig und allein die Aufgabe, die Spieler gleichmäßig auf die Sitzplätze um den Pokertisch zu verteilten. Tabelle 5 zeigt das Klassendiagramm.

SeatAssigner

-usedSeats : int[][] = {
{0},
{0, 5},
{0, 4, 6},
{0, 3, 5, 7},
{0, 2, 4, 6, 8},
{0, 2, 4, 5, 6, 8},
{0, 2, 3, 4, 6, 7, 8},
{0, 2, 3, 4, 5, 6, 7, 8},
{0, 1, 2, 3, 4, 6, 7, 8, 9},
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
}

+assignSeats(players : Player[]) : void

Tabelle 5: Klassendiagramm der Klasse SeatAssigner

Das Feld `usedSeats` enthält die bei unterschiedlichen Spielerzahlen zugewiesenen Sitz-Indizes. Die Methode `assignSeats()` weist den übergebenen Spielern die entsprechenden Sitzplätze zu.

6.2 Controller

Das Subsystem Controller ist die Steuerzentrale der GUI und das Bindeglied zwischen dem Datenmodell, den grafischen Komponenten und dem Framework. Es wurde vollständig in der Klasse `Controller` implementiert, von der Tabelle 6 das Klassendiagramm zeigt.

Controller
-model : Model -gui : GUIPanel -playerName : String -active : boolean -paused : boolean -gameRunning : boolean -expectNewGame : boolean -nextGameStateUpdate : long -pauseBetweenHands : boolean -updateDelay : int = 1500 +MAX_UPDATE_DELAY : int = 5000
+Controller(model : Model) +connect(playerName : String, active : boolean, hostname : String) : void +start(rounds : int, selectionIndices : int[], spectateMode : boolean, playPermutations : boolean, showTruePlayerNames : boolean) : void +update(o : Observable, arg : Object) : void +action(action : Action) : void +quitGame() : void +disconnect() : void +refresh() : void -gameStarted() : void -possiblyNewGame(gameState : GameState) : boolean -optionallyWait() : void +isPaused() : boolean +pause(bool : boolean) : void +toogglePause() : void +getDelay() : int +setDelay(ms : int) : void +skipDelay() : void +isPauseBetweenHands() : boolean +setPauseBetweenHands(bool : boolean) : void +connectionAbort() : void +getRunningGames() : List<IGame> +getFinishedGames() : List<IGame>

Tabelle 6: Klassendiagramm der Klasse Controller

Gleich im Konstruktor wird die Methode `initializeTUDPokerClient()` der Framework-Klasse `MultiplePokerClientManager` aufgerufen. Dabei wird eine Instanz der Klasse `TUDPokerClient` erzeugt, welche von nun an als Schnittstelle zum Framework fungiert. So werden Aufrufe der Methoden `connect()`, `start()`, `disconnect()`, `refresh()`, `action()`, `getRunningGames()` und `getFinishedGames()` jeweils an die entsprechende Methode in `TUDPokerClient` weitergeleitet, gleichzeitig werden Model und View aktualisiert.

`Controller` implementiert die Schnittstelle `Observer` der Java-Klassenbibliothek und meldet sich bei Aufruf von `connect()` bei `TUDPokerClient` als *Observer* an. In der Folge wird jedes Mal, wenn vom Server ein Update des Serverzu-

stands oder des Spielzustands empfangen wird, die Methode `update()` aufgerufen. Diese aktualisiert entsprechend entweder die im `LobbyPanel` angezeigten Serverinformationen oder den Zustand des `Models`.

Nicht ganz trivial ist die Erkennung, wann ein neues Spiel gestartet wurde, insbesondere dann, wenn im Passiv-Modus zum Server verbunden wurde, da der Spielstart in diesem Fall von einem anderen Client ausgeht. Da das Framework keine eigne Nachricht für den Spielstart versendet, wird prinzipiell immer dann vom Start eines neuen Spiels ausgegangen, wenn ein `GameState`-Objekt empfangen wurde, bei dem der Rundenindex null und die aktuelle Straße der Preflop ist und bei dem noch keine Spielaktionen getätigt wurden. Die Methode `possiblyNewGame()` führt einen entsprechenden Test durch. Fällt dieser positiv aus, wird das interne Flag `gameRunning` auf `true` gesetzt und das `GamePanel` angezeigt. Solange nun das `gameRunning`-Flag gesetzt ist, leitet der Controller alle nachfolgenden Updates an die Spielansicht weiter. Verlässt der Client das Spiel, so wird in der Methode `quitGame()` `gameRunning` wieder auf `false` gesetzt. Jetzt eventuell noch ankommende `GameState`-Updates des abgebrochenen Spiels werden verworfen. Erst wenn wieder ein `GameState`-Update eintrifft, bei dem `possiblyNewGame()` `true` ergibt, wird dieses dem `Model` als neues Spiel übergeben. Diese Methode funktioniert zuverlässig, solange kein Spiel gestartet wird, bei dem auch die Permutationen der Hole Cards gespielt werden. Jede Permutationsrunde beginnt nämlich wieder mit dem Rundenindex null im `GameState`-Objekt und wird so immer als Beginn eines neuen Spiels gewertet werden, auch wenn sie eigentlich noch Teil eines schon abgebrochenen Spiels ist. Als bestmögliche Lösung des Problems wurde im Aktiv-Modus ein weiteres Flag `expectNewGame` eingeführt, welches auf `true` gesetzt wird, wenn der Client selber ein Spiel startet. Beim Empfang eines `GameState`-Updates, für das `possiblyNewGame()` `true` ergibt, veranlasst der `Controller` nun nur noch dann die Anzeige eines neuen Spiels, wenn auch `expectNewGame` auf `true` gesetzt ist, ansonsten wird das Update ebenfalls verworfen. Im Passiv-Modus fehlt diese Information leider, und es werden entsprechende `GameState`-Updates immer als Start eines neuen Spiels interpretiert, wenn `gameRunning` `false` ist.

Würde beim Empfang eines Spielupdates immer sofort die Spielansicht aktualisiert werden, so wäre es dem Nutzer kaum möglich, dem Spielablauf zu folgen, da insbesondere die Pokerbots oft in Sekundenbruchteilen ihre Spielzüge ausführen. Aus diesem Grunde ist eine Verzögerung zwischen den einzelnen Spielzügen sinnvoll, und diese Aufgabe wird ebenfalls von `Controller` übernommen. Die Länge dieser Verzögerung in Millisekunden wird in `updateDelay` gespeichert und kann in der Spielansicht über einen Schieberegler eingestellt werden, der Zugriff auf die Variable erfolgt über `setDelay()`. Bei jeder Aktualisierung des Models im Rahmen eines Spielupdates wird die momentan eingestellte Verzögerung `updateDelay` auf die aktuelle Systemzeit addiert und in `nextGamestateUpdate` gespeichert. Beim Eintreffen des nächsten Spielupdates ruft die `update()`-Methode die Methode `optionallyWait()` auf, die den Empfangs-Thread mit Hilfe von `Thread.wait()` solange anhält, bis der in `nextGamestateUpdate` hinterlegte Zeitpunkt erreicht ist,

dies führt zu der gewünschten Verzögerung. Manchmal ist es allerdings auch wünschenswert, sofort den nächsten Spielzustand anzuzeigen. Dies ist etwa der Fall, wenn der Nutzer selber eine Spielaktion getätigt hat, da man dann davon ausgehen kann, dass er den aktuellen Spielzustand bereits wahrgenommen hat. Hierfür gibt es die Methode `skipDelay()`, welche `nextGamestateUpdate` auf die aktuelle Systemzeit setzt, so dass das nachfolgende Spielupdate unverzüglich angezeigt wird.

Schließlich kann der Nutzer noch jederzeit über eine Schaltfläche in der Spielansicht den aktuell angezeigten Spielzustand pausieren oder weiterlaufen lassen, was zu einem Aufruf der Methode `pause()` führt. In der Variable `paused` wird gespeichert, ob das Spiel momentan angehalten ist. Standardmäßig wird das Spiel zudem bei jedem Showdown pausiert, was sich ebenfalls in der Spielansicht mit Hilfe der Methode `setPausedBetweenHands()` konfigurieren lässt. Wie auch bei der Verzögerung wird beim Pausieren der Empfangsthread angehalten, dies geschieht wieder in der `optionallyWait()`-Methode, die beim Spielupdate aufgerufen wird.

6.3 View

Das View-Subsystem besteht aus den Klassen `LobbyPanel`, `GamePanel`, `GUIPanel`, `HHPanel` und `Statistics`. Im Folgenden werden die implementierten Klassen näher beschrieben.

6.3.1 LobbyPanel

Die Klasse `LobbyPanel` ist eine Unterklasse von `JPanel` und stellt die Lobby dar. Im Klassendiagramm in Tabelle 7 wurden alle in der Lobby angezeigten grafischen Komponenten weggelassen, da diese ja aus Abschnitt 4.1 bekannt sind und deren Aufführung das Diagramm sehr unübersichtlich gemacht hätte.

LobbyPanel
-serialVersionUID : long = -4277379804366764433L -gui : GUIPanel -controller : Controller ...
+LobbyPanel(gui : GUIPanel, controller : Controller) -disconnectButtonPressed() : void -connectButtonPressed() : void -startButtonPressed() : void +refreshPlayersList(tudPokerClient : TUDPokerClient) : void +setConnectedStatus(isConnected : boolean, isActive : boolean) : void -runningGamesListElementSelected(evt : ListSelectionEvent) : void -finishedGamesListElementSelected(evt : ListSelectionEvent) : void -showResult(game : IGame) : void

Tabelle 7: Klassendiagramm der Klasse LobbyPanel

Die wesentliche Aufgabe von `LobbyPanel` ist es, dem Nutzer Eingabefelder und Schaltflächen zur Serververbindung und zum Starten eines Spiels zur Verfügung zu stellen und die Eingabedaten und Steuerkommandos an die jeweiligen Methoden von `Controller` weiterzuleiten. Dies geschieht mit Hilfe der Methoden `connectButtonPressed()`, `startButtonPressed()`, `disconnectButtonPressed()` und `refreshPlayersList()`. Klickt der Nutzer auf ein laufendes oder abgeschlossenes Spiel, so wird die entsprechende Methode `runningGamesListElementSelected()` beziehungsweise `finishedGamesListElementSelected()` aufgerufen, diese zeigen dann mit Hilfe von `showResult()` die Details zum Spiel im Informationsbereich der Lobby an.

6.3.2 GamePanel

Die Klasse `GamePanel` ist genau wie `LobbyPanel` von `JPanel` abgeleitet und stellt die Spielansicht dar. Im Klassendiagramm in Tabelle 8 wurden ebenfalls die zahlreichen Felder, die Verweise auf Schaltflächen, Fonts oder Layoutdaten enthalten, der besseren Übersicht wegen weggelassen.

GamePanel
-serialVersionUID : long = -6661420009933181693L -controller : Controller -model : Model -statistics : Statistics -history : JFrame -graphics : HashMap<String, BufferedImage> = new HashMap<String, BufferedImage>() ...
+GamePanel(controller : Controller, model : Model) -initialize() : void +update(Observable o, Object arg) : void -updateActionButtons(enabled : boolean, cAction : String, rAction : String) : void #paintComponent(g : Graphics) : void -drawStringCentHoriz(g : Graphics, string : String, y : int) : void -drawNextCard(cardsIterator: Iterator<Card>, posX : int, posY : int, g : Graphics) : void +setPaused(final boolean bool) -setButtonsEnabled(enabled : boolean) : void +historyClosed() : void +statisticsClosed() : void +setGraphics(graphics : HashMap<String, BufferedImage>) : void

Tabelle 8: Klassendiagramm der Klasse GamePanel

Immer dann, wenn ein neuer Spielzustand vom Server empfangen und das Model aktualisiert worden ist, wird die Methode `update()` aufgerufen. Diese blendet abhängig davon, ob der Nutzer gerade selber an der Reihe ist, die Schaltflächen zum Mitgehen, Erhöhen oder Aussteigen ein und stößt anschließend durch den Aufruf der von `JPanel` geerbten Methode `repaint()` die Neuzeichnung der Spielansicht an.

Dadurch wird automatisch vom Grafiksystem die Methode `paintComponent()` aufgerufen, welche für die Darstellung aller in der Spielansicht sichtbaren Elemente

außer den Schaltflächen zuständig ist. Die dazu nötigen Spielinformationen werden über die entsprechenden Methoden vom Model erfragt. Zunächst wird die im Feld `graphics` hinterlegte Rastergrafik des Pokertischs als Hintergrund der Spielansicht gezeichnet. Es folgt die Darstellung der Spieler mit Spielername, Spielposition und aktuellem Stack. Die Hole Cards der Spieler und die Board Cards werden durch die Funktion `drawNextCard()` gezeichnet, die nötigen Grafiken der Karten sind ebenfalls in `graphics` referenziert. Es wird der Betrag des Pots neben den Board Cards angezeigt und der zuletzt getätigte Spielzug durch einen entsprechenden Text auf den Hole Cards des jeweiligen Spielers visualisiert. Beim Showdown blendet die Methode `drawStringCentHoriz()` die Benachrichtigung, welcher Spieler wie viel gewonnen hat, in der Mitte des Tisches über den Board Cards ein.

Eine weitere Aufgabe des `GamePanels` ist die Anzeige der Spielhistorie und der Statistik. Beide können über entsprechende Checkboxen ein- oder ausgeblendet werden. Wird die Spielhistorie oder die Statistik direkt über das jeweilige Fenster geschlossen, so wird die Funktion `historyClosed()` beziehungsweise `statisticsClosed()` aufgerufen, die den Haken in der jeweiligen Checkbox entfernt und so die Anzeige konsistent hält.

6.3.3 GUIPanel

Die Klasse `GUIPanel` erbt von `JPanel` und dient als Container, der selbst nicht sichtbar ist, aber wahlweise das `LobbyPanel` oder das `GamePanel` anzeigt. Bei der Instanziierung wird zunächst das `LobbyPanel` eingebettet. Außerdem wird der `GraphicsLoader` gestartet, eine Unterklasse von `Thread`, die nur die Aufgabe hat, die von der Spielansicht benötigten Bilddateien in das Feld `graphics` zu laden, damit die Spielansicht später ohne größere Verzögerungen angezeigt werden kann.

GUIPanel
-serialVersionUID : long = -2221982342327087758L -controller : Controller -lobbyPanel : LobbyPanel -gamePanel : GamePanel -graphicsLoader : GraphicsLoader -graphics : HashMap<String, BufferedImage>
+GUIPanel(controller : Controller, model : Model) +showGamePanel() : void +showLobbyPanel() : void +setConnected(isConnected : boolean, isActive : boolean) : void +setPaused(bool : boolean) : void +refreshPlayersList(tudPokerClient : TUDPokerClient) : void +error(title : String, message : String) : void

Tabelle 9: Klassendiagramm der Klasse GUIPanel

Dies geschieht beim Start eines Spiels durch Aufruf der Methode `showGamePanel()` (siehe Klassendiagramm in Tabelle 9), die das `LobbyPanel` entfernt und stattdessen das `GamePanel` mit der Spielansicht anzeigt. Beim Verlassen eines Spiels wird umgekehrt `showLobbyPanel()` aufgerufen, und es wird wieder die Lobby eingeblendet. Die Methoden `setConnected()`, `refreshPlayersList()` und `setPaused()` werden von `Controller` aufgerufen und aktualisieren die Anzeige der Lobby beziehungsweise der Spielansicht. Fehlermeldungen können mit Hilfe der Methode `error()` ausgegeben werden, die ein einfaches Meldungsfenster mit dem übergebenen Nachrichtentext anzeigt.

6.3.4 HHPanel

Auch die Klasse `HHPanel` ist eine Unterklasse von `JPanel`, und ist für die Darstellung der Spielhistorie zuständig. Dazu enthält sie, wie aus Tabelle 10 ersichtlich ist,

HHPanel
-serialVersionUID : long = -4545486103299205934L -model : Model -hhArea : JTextPane -hhScrollPane : JScrollPane -BLUE : SimpleAttributeSet = new SimpleAttributeSet() -DEFAULT : SimpleAttributeSet = new SimpleAttributeSet() -RED : SimpleAttributeSet = new SimpleAttributeSet() -farbe : SimpleAttributeSet = pokertud.clients.swingclient.HHPanel.DEFAULT
+HHPanel(model : Model) -initialize() : void +update(o : Observable, arg : Object) : void -appendTextToHH(text : String, farbe : SimpleAttributeSet) : void -getDateTime() : String

Tabelle 10: Klassendiagramm der Klasse HHPanel

im Feld `hhArea` ein `JTextPane`, welches durch `hhScrollPane` um Bildlaufleisten erweitert wird. Mit Hilfe der Methode `appendTextToHH()` wird der Spielverlauf als Text in `hhArea` eingeblendet.

6.3.5 Statistics

Für die Statistikfunktionalität wurden einige Klassen aus dem vorhandenen Statistiktool übernommen, die hier nicht im Detail erläutert werden sollen. Sie befinden sich im Package `statistics`. Neu ist die Klasse `Statistics`, welche für die Anbindung an die Grafische Benutzeroberfläche verantwortlich ist. Tabelle 11 zeigt das Klassendiagramm.

Statistics
-gamePanel : GamePanel -statisticsFrame : MainFrame
+Statistics(gamePanel : GamePanel, model : Model) +update(arg0 : Observable, arg1 : Object) : void -parseResults(resultStrings : List<String>) : GameInfo +setVisible(bool : boolean) : void +statisticsClosed() : void

Tabelle 11: Klassendiagramm der Klasse Statistics

Die bei Aufruf der Methode `update()` vom Model empfangenen Spielinformationen werden durch die Methode `parseResults()` zur Darstellung weitergeleitet. `setVisible()` und `statisticsClosed()` dienen dazu, das Statistikfenster ein- und auszublenden und die entsprechende Schaltfläche in der Spielansicht synchron zu halten.

6.4 Klassen zum Starten der GUI

Die Grafische Benutzeroberfläche lässt sich wahlweise als Applet in eine Website einbetten oder als eigenständige Anwendung ausführen. Dazu sind zwei weitere Klassen notwendig, die sich nicht einer der drei Funktionsgruppen Model, View und Controller zuordnen lassen und im Folgenden beschrieben werden.

6.4.1 StandaloneClient

Die Klasse `StandaloneClient` wird verwendet, um die Benutzeroberfläche als eigenständige Anwendung zu starten. Wie das Klassendiagramm in Tabelle 12 zeigt, verfügt sie dafür lediglich über eine `main()`-Methode.

StandaloneClient
+main(args : String[]) : void

Tabelle 12: Klassendiagramm der Klasse StandaloneClient

6.4.2 ClientApplet

Statt als eigenständige Anwendung kann die Benutzeroberfläche auch in einem Applet ausgeführt werden. Für diesen Zweck erbt die Klasse `ClientApplet` von der Oberklasse `JApplet` für Swing-Applets und implementiert die erforderliche Methode `init()`, wie aus dem Klassendiagramm in Tabelle 13 hervorgeht.

ClientApplet
-serialVersionUID : long = -8709854905957909792L
+init() : void -createGUI() : void

Tabelle 13: Klassendiagramm der Klasse ClientApplet

In der Methode `createGUI()` wird das `GUIPanel` angezeigt, welches zunächst die Lobby enthält.

7 Fazit

In der vorliegenden Ausarbeitung zur Bachelorarbeit wurde die Entwicklung einer Grafischen Benutzeroberfläche zu einem Poker-Framework dokumentiert. Dazu wurde zunächst ein kurzer Einblick in das Spiel Poker gegeben, indem die wichtigsten und für das Programm relevanten Elemente des Spiels dargestellt wurden.

Anschließend wurde die Software beschrieben, auf welche die Grafische Benutzeroberfläche aufbaut. Dies ist in erster Linie das Poker-Framework, welches die Spiellogik implementiert und einen Server sowie eine Schnittstelle auf dem Client bereitstellt, an die die Spielinformationen übertragen werden. Es wurde gezeigt, welche Klassen des Frameworks für die Grafische Benutzeroberfläche wichtig sind und wie sie zu verwenden sind. Außerdem wurde kurz ein bereits existierendes Programm zur statistischen Auswertung von Logdateien des Spielservers vorgestellt, dessen Funktionalität in die Benutzeroberfläche mit integriert werden sollte.

In Kapitel 4 wurden die Dialoge gezeigt, aus denen die Grafische Benutzeroberfläche besteht. Es wurden die Lobby, die Spielansicht, die Spielhistorie und das Statistikfenster im Detail erläutert und ihre Bedienung erklärt. So wurde gezeigt, wie eine Verbindung zu einem Spielserver hergestellt wird, wie ein neues Spiel gestartet wird und wie Spielzüge getätigt werden.

Kapitel 5 befasste sich mit der Architektur der Grafischen Benutzeroberfläche. Hier wurden zunächst die Designziele lose Kopplung, hoher Zusammenhalt und leichte Verständlichkeit formuliert, welche die Software erfüllen sollte. Anschließend wurden die Entwurfsmuster Observer und Model-View-Controller zur Aufteilung der Software in Funktionsgruppen vorgestellt und es wurde gezeigt, wie diese bei der Architektur der Software angewendet wurden.

Schließlich wurden die Klassen im Detail dargestellt, in denen die Software implementiert wurde. Anhand von Klassendiagrammen wurde ein Überblick über die jeweiligen Felder und Methoden gegeben, und die wichtigsten Elemente wurden näher beschrieben.

Das Ziel, das Poker-Framework intuitiver bedienbar zu machen, wurde erreicht. Durch die Darstellung des Spielgeschehens an einem Pokertisch analog zu einem Pokerspiel in der Realität und die bildliche Darstellung der Spielkarten ist der aktuelle Spielzustand wesentlich einfacher zu erfassen als bei der textuellen Darstellung, welche die ursprüngliche Benutzeroberfläche des Frameworks verwendet. Die neue Grafische Benutzeroberfläche unterstützt nun den gesamten Ablauf vom Verbinden zum Spielserver über das Starten eines Spiels bis zur Interaktion während des Spiels. Hierfür mussten vorher zwei einzelne Anwendungen gestartet werden.

Eine weitere Verbesserung ist die Integration der Statistikfunktionalität in die Grafische Benutzeroberfläche. Mit ihrer Hilfe kann das Verhalten der anderen Spieler oder des eigenen Bots analysiert werden und mit den gewonnenen Erkenntnissen die eigene Taktik verbessert werden.

Erweiterungen der Grafischen Benutzeroberfläche sind sowohl im optischen wie auch im funktionalen Bereich denkbar. Sollten später weitere Spielmodi oder Varianten des Pokerspiels in das Framework integriert werden, so könnte auch die Benutzeroberfläche entsprechend erweitert werden. Für Spieler, die keinen Zugriff auf den Computer haben, auf dem der Server läuft, wäre es nützlich, wenn sie die Logdateien des Spiels über den Client beziehen könnten.

Im Bereich der Optik könnte die Spieldarstellung noch beliebig weiter verschönert werden, bis hin zu einer Darstellung des Spielgeschehens in 3D. Da die sich die Endgeräte, auf denen die Benutzeroberfläche verwendet wird, bezüglich Bildschirmgröße und Auflösung sicherlich teilweise deutlich voneinander unterscheiden, wäre es zudem von Vorteil, wenn die verwendeten Rastergrafiken durch Vektorgrafiken ersetzt würden und die Benutzeroberfläche frei skalierbar gestaltet würde.

Es bleibt festzuhalten, dass die Grafische Benutzeroberfläche die gesetzten Ziele erfüllt, es aber noch zahlreiche Ansatzpunkte für Verbesserungen gibt.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[ACPC11] **Annual Computer Poker Competition**. <http://www.computerpokercompetition.org>, Zugriff am 11.02.2011

[Gamma09] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:** *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley Verlag, München 2009.

[Larman05] **Craig Larman: Applying UML and Patterns:** *An Introduction to object-oriented analysis and design and iterative development*. Prentice Hall PTR, 3. Auflage, 2005.

[Reenskaug79] **Trygve M. H. Reenskaug:** *Thing-Model-View-Editor: An Example from a planningsystem*. Xerox PARC <http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf>, 1979.

[Warren07] **Ken Warren:** *Das große Pokerbuch*. Heel Verlag, Königswinter 2007.

[Zopf10] **Markus Zopf:** *Ein Framework zur Entwicklung und Evaluation intelligenter Pokeragenten*. Technische Universität Darmstadt, 2010.