Design einer Library für Elo-Rating-basiertes Matchmaking in Online-Multiplayer-Spielen

Tamara Nitsch

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interaktive Medien

in Hagenberg

im Dezember 2011

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 30. November 2011

Tamara Nitsch

Inhaltsverzeichnis

Kurzfassung

Matchmaking Algorithmen werden in der stetig wachsenden Welt der Online Multiplayerspiele immer wichtiger. Spieler wollen schnell und unkompliziert den bestmöglichen Matchpartner für ein Spiel finden. Im Falle dieser Arbeit bedeutet bestmöglich, einen möglichst gleichstarken Gegner zu finden, andere Aspekte werden nur am Rande erwähnt. Um abschätzen zu können ob ein Gegenspieler ungefähr gleich stark ist und welche Gewinnchancen man gegen ihn hat, müssen Rating Systeme eingesetzt werden. Vor allem das Elo-Rating-System, welches schon im Schach und einigen anderen Sportarten eingesetzt wird, liefert eine vielversprechende Grundlage welche auch in der Onlinewelt funktionieren kann. Es werden aber auch Weiterentwicklungen dieses Systems, insbesondere das Glicko-Rating-System, betrachet, um den bestmöglichen Rating Algorithmus zu finden. Die Arbeit versucht auf möglichst einfach verständliche Art diese Algorithmen aufzubereiten und zu erklären und ihre Vorteile und Probleme aufzuzeigen.

Aus den Anforderungen für das Matchmaking und den Erkenntnissen aus dem Rating Bereich wird in weiterer Folge eine Library erstellt welche grundsätzliche Funktionalität zur Verfügung stellt, um Spielstärken zu messen und gleichstarke Gegner zusammenzuführen. Dabei wird besonderer Wert darauf gelegt zuvor erforschte API Designrichtlinien zu befolgen, um eine möglichst einfach zu verwendende und erweiterbare Software zu generieren. Weiters wird dargelegt auf welche Art und Weise die resultierende Bibliothek für unterschiedliche Spielgenres und -typen erweitert und angepasst werden kann.

Abstract

Matchmaking algorithms have become increasingly important in the continuously growing world of online multiplayer games. Players want to find the best possible in-game opponent in a fast and straightforward way. For the purpose of this thesis, the best possible opponent is preferably the one who performs as good as the player himself, with other aspects only getting mentioned marginally. In order to estimate the comparative strength of an opponent and the chances of winning for the player, rating systems need to be adopted. The Elo rating system, which is used in chess and various other sports, provides a promising foundation and could also be applied in the world of online gaming. In order to find the best possible rating algorithm, some additional enhancements of the system, in particular the Glicko rating system, are going to be presented. This thesis tries to illustrate the underlying algorithms and mathematics in a comprehensible way and to identify the benefits and possible problems of each one.

Derived from the demands for matchmaking and the findings from the rating domain, a software library is going to be created which provides basic functionality to measure player performance and match equally strong opponents. At the same time, it is emphasized to use API design guidelines, which have been researched and explained beforehand, to generate easy-to-use and upgradeable software. Furthermore, it is stated how the resulting library can be adapted for different game genres and types.

Kapitel 1

Einleitung

Diese Arbeit befasst sich mit Matchmaking Verfahren für Online Multiplayerspiele und zeigt Rating Algorithmen auf durch welche Fähigkeits basiertes Matchmaking möglich gemacht wird.

1.1 Motivation

Im Bereich der Computerspiele wird es immer gängiger die Möglichkeit zur Verfügung zu stellen online mit und/oder gegen andere Leute zu spielen. Auch typische Singleplayerspiele bieten schon oft einen online Modus an, denn es ist viel spannender und lustiger gegen echte Gegner[1] oder mit echten Helfern zu spielen, da diese weitaus anspruchsvoller sind als eine KI [12]. Ein großes Problem bei der Suche nach adäquaten anonymen Mitspielern ist allerdings, dass viele Computerspiele die Gegner willkürlich auswählen und gerade Neueinsteiger oft keine Chance haben zu gewinnen [27]. Das kann sie dazu treiben den Onlinemodus zu vernachlässigen (wenn es auch einen Offlinemodus gibt), oder im schlimmsten Fall sogar das ganze Spiel aufzugeben [12]. Stärkere Spieler, die nach einer Herausforderung suchen, werden sich hingegen auf Dauer langweilen bei zu schwachen Gegnern.

Selbst bei Spielen die Matchmaking Systeme verwenden, gibt es oft Probleme, vor allem wenn diese leicht von den Spielern beeinflusst werden können. Zum Beispiel wenn Spielwerte wie Actions per Minute[2] , Frags[3] oder dergleichen betrachtet werden um auf die Fähigkeiten eines Spielers zu schlie-

[1] Aus Gründen der Leserlichkeit sind Ausdrücke wie Spieler, Gegner oder Benutzer im generischen Maskulinum gehalten und gelten dementsprechend auch für weibliche Personen.

[2] Actions per Minute ist ein Wert dem besonders im Strategiespiel Bereich eine hohe Aussagekraft zugesprochen wird. Er zählt die Aktionen und Befehle die ein Spieler pro Minute an das Spiel gibt.

[3] Ein *Frag* ist der virtuelle Tod einer Spielerfigur, wird vor allem in Shootern so benannt. Üblicherweise folgt auf einen Frag nach einiger Zeit ein Respawn, das heißt die Spielfigur wird quasi wiederbelebt und kann wieder am Spiel teilnehmen

ßen. Dies kann dazu führen, dass Spieler sich einen höheren Rang in der Rangliste sichern als ihnen eigentlich zusteht. Was, wenn das Matchmaking sich danach richten würde, weniger ein Problem wäre, weil diese Spieler sich dann schnell mit Gegnern konfrontiert sehen, die sie nicht besiegen können. Auf der anderen Seite, und das ist ein viel größeres Problem, gibt es leider auch viele Spieler, welche Herausforderung nicht schätzen und ihren eigenen Rang vermindern, um mit ihren besseren Fähigkeiten schwächere Spieler zu drangsalieren, was ihnen leichte Siege einbringt [12]. Dieses Verhalten nennt sich Noob Bashing oder Sandbagging.

Daher ist es sehr wichtig ein Matchmaking System zu verwenden, das wirklich ausgewogene Gewinnchancen für alle Parteien ermöglicht. Im Elo-Rating-System wird nur betrachtet ob ein Spiel gewonnen oder verloren wurde, wodurch es nicht so leicht unrechtmäßig vom Spieler beeinflusst werden kann. Ursprünglich von Arpad Elo für Schach entwickelt [7], wurde es später auch für verschiedene Sportarten adaptiert (Baseball, American Football) und findet mittlerweile, in überarbeiteter Version, auch schon bei vereinzelten Computerspielen (Starcraft, League of Legends, Guild Wars, BattleForge) Verwendung.

1.2 Zielsetzung und Aufbau

Ziel der Arbeit ist es herauszufinden, wie der originale Elo Rating Algorithmus verändern werden muss, um ihn für Computerspiele mit einer großen Spielerbasis verwendbar zu machen. Außerdem soll geklärt werden, für welche Arten von Spielen er verwendbar ist und welche Probleme und Komplikationen auftreten können. Dazu soll das Design einer Software entworfen werden, die Automatchmaking zur Verfügung stellt und auf Elo ähnlichen Algorithmen basiert. Am Anfang der Arbeit, im Kapitel „Matchmaking Verfahren“, steht die Erläuterung von Matchmaking Techniken, wobei diese nur peripher tangiert werden, da das Hauptaugenmerk auf den Rating Algorithmen liegt, die im darauffolgenen Kapitel „Rating Verfahren“ erklärt werden. Danach, in Kapitel „API Design“, wird erläutert welche Richtlinien beim Design von Software APIs zu befolgen sind, bevor diese dann im nächsten Kapitel „Matchmaking und Rating Library“ zur praktischen Anwendung kommen. Es werden einige Game Design Aspekte diskutiert, die für die Entwicklung eines Matchmaking und Rating Algorithmus in den unterschiedlichen Genres von Interesse sind. Zu guter Letzt erfolgt, in Kapitel „Zusammenfassung und Ausblick“ eine Zusammenfassung der Ergebnisse und ein Ausblick auf mögliche Verbesserungen und Weiterentwicklungen des Systems.

Kapitel 2

Matchmaking Verfahren

Matchmaking beschreibt, im Kontext der Online Multiplayerspiele, den Prozess, in dem für ein Match, also eine Spielpartie, geeignete Spieler gesucht und gefunden werden. Dies passiert in manchen Fällen manuell, indem sich Spieler mit Freunden oder Bekannten verabreden, um gemeinsam zu spielen. Für diese Arbeit interessant sind aber jene Fälle, in denen automatisches Matchmaking passiert, also ein Spieler vom Programm mit einer ihm unbekannten Person zusammengebracht wird, um ein Match auszutragen. Die unkomplizierteste Variante hierfür ist per Zufall zwei Spieler aus der Warteliste auszuwählen und gegeneinander antreten zu lassen. Zumeist erhofft sich der Spieler von einem Matchmakingprogramm aber Gegenspieler, die bestimmte Vorraussetzungen erfüllen. Soll ein erfolgreicher Onlinemodus für ein Spiel gestartet werden, so sollten diese Wünsche nicht unbedacht bleiben[1].

Gewinnchancen (Schwierigkeitsgrad, Fähigkeitsgrad): Der wohl bedeutendste Punkt beim Matchmaking, und der Mittelpunkt dieser Arbeit, sind ausgeglichene Gewinnchancen für beide Seiten. Das ist wichtig um den Spass am Spiel und die Spannung zu erhalten. Denn Spass hat der Spieler nur solange er das Gefühl hat den Ausgang des Spieles beeinflussen zu können [27] und Spannung gibt es nur solange einem der Gegner etwas entgegenzusetzen hat. Wenn also ein Spieler von Anfang an das Gefühl hat nicht mehr gewinnen zu können, verliert er schnell den Spass am Spiel, im schlimmsten Fall für immer. Natürlich kann nicht jeder gewinnen und schon gar nicht alle auf einmal, aber durch gelungenes Matchmaking kann größt möglicher Spass für möglichst viele Leute geboten werden. Soll also solch ein Matchmaking zur Verfügung gestellt werden, führt kein Weg daran vorbei auf irgendeine Art und Weise die Fähigkeiten der Spieler zu messen, um entweder gleichstarke Spieler zusammenzubringen oder Stärkeunterschiede aus-

[1] Alle hier angeführten Überlegungen sind natürlich adequat auch für Situationen anwendbar, in denen Spieler miteinander und nicht gegeneinander spielen.

zugleichen [27]. Algorithmen die dies ermöglichen werden in Kapitel 3 vorgestellt.

Latenzzeiten: Wenn Spieler über das Internet miteinander verbunden sind, müssen Latenzzeiten[2] berücksichtigt werden. Wenn diese nicht möglichst gering gehalten werden, so kann es sein, dass vermehrt sogenannte Lags während eines Matches auftreten. Das trübt nicht nur die Spielerfahrung, sondern kann auch die Gewinnchancen beträchtlich beeinflussen.

Spielerspezifische Einstellungen: Optionale Kriterien, die ins Matchmaking einarbeiten werden können, sind spielerspezifische Einstellungen wie zum Beispiel Sprache. In vielen Spielen sind die wichtigen Sprachen zwar von grundauf in verschiedene Spielerbasen getrennt, die nicht miteinander in Berührung kommen, trotzdem könnte dadurch das Matchmaking von minderheitlichen Sprachgruppen auf zum Beispiel vorwiegend englisch-sprachigen Servern ermöglicht werden. Es lassen sich natürlich auch andere, vom Spieler bevorzugte, Einstellungen einarbeiten. Wie zum Beispiel der Spielstil, oder welche optionalen Regeln beachtet werden sollen.

Negative Faktoren: Oftmals gibt es Personen mit denen die übrigen Spieler nicht gerne gematched werden, weil sie zum Beispiel sehr oft Spiele einfach abbrechen oder sich schlechter Umgangsformen bedienen. Solche Zusatzinformationen könnten auch im Matchmaking verarbeitet werden, um solche Spieler nur mit ihresgleichen zusammenzubringen. Um ihnen die Chance zu geben sich zu bessern ist es ratsam, wenn sich diese Faktoren mit der Zeit verringern, damit sie auch wieder am „normalen“ Spielgeschehen teilhaben können.

2.1 Manuelles Matchmaking

Das Ursystem des Matchmakings besteht darin, dass Spieler sich ihre Gegner manuell aussuchen. Die einfachste Möglichkeit hierfür ist sich mit Bekannten oder Freunden für ein Spiel zu verabreden und sich direkt zu verbinden. Der Spieler weiß also genau mit welcher Person er gematched werden möchte, das Programm muss nur noch das Spiel starten.

Wenn nun noch keine Spielpartner vorhanden sind, so gibt es in vielen Spielen eine sogenannte „Lobby“, in die alle Spieler gelangen, die auf der

[2]Latenzzeiten bezeichnen den Zeitraum zwischen einer Aktion eines Spielers und der Antwort des Servers. Wenn also zum Beispiel die Datenübertragung nicht gut funktioniert, weil das Netz überlastet ist oder der Server zu viele Anfragen bearbeiten muss sind die Latenzzeiten sehr hoch. Dies kann dazu führen dass es mitunter sehr lange dauert bis die Aktion eines Spielers bei ihm selbst oder beim anderen Spieler ersichtlich wird. Zum Beispiel wenn ein Spieler in eine Richtung läuft und der Server diese Nachricht erst sehr spät erhält dann kann es sein, dass die Spielfigur sich ruckartig bewegt.

Suche nach einem Match sind. Unter Umständen ist die Lobby noch in unterschiedliche Kategorien oder „Rooms“ unterteilt in denen gleichgesinnte Spieler mit zum Beispiel gleichem Spielstil oder Ähnlichem gefunden werden können. Dann gibt es die Möglichkeit entweder eine neue Spielinstanz[3] zu erstellen oder einer bestehenden Spielinstanz beizutreten. Schließlich muss gewartet werden, bis die Instanz genügend Spieler hat um das Spiel starten zu können [8]. Der Eröffner einer solchen Instanz hat oft auch noch die Möglichkeit bestimmte Spieleinstellungen vorzunehmen, oder Spieler, welche man nicht in seinem Spiel haben will, wieder hinauszuwerfen.

Wie zu sehen ist, ist dies eine recht aufwändige und zeitintensive Art um ein Spiel zu starten. Die meisten Neulinge sind oft von der Komplexität überfordert und geübte Spieler sind frustriert von der Menge an Zeit die dabei verloren geht [17].

2.2 Automatisches Matchmaking

Um die Komplexität des Matchmakingsprozesses vom User fernzuhalten gibt es mittlerweile sehr häufig die Möglichkeit des automatischen Matchmakings. Beim automatischen Matchmaking muss der Spieler im besten Fall nur noch eine einzige Aktion durchführen. Er muss lediglich dem Programm mitteilen, dass er gematcht werden möchte. Das Programm holt sich dann die nötigen Informationen aus spielerspezifischen Einstellungen und Daten und sucht einen Partner der zu diesen Informationen passt. Dies führt zu einem besonders schnellen und einfachen Matchmakingprozess für den Benutzer.

First-Come First-Served

Der einfachste Weg für automatisches Matchmaking ist, die Spielerauswahl komplett dem Zufall zu überlassen, indem User in der Warteliste mit den nächsten ankommenden Spielern gepaart werden. Dieser First-Come First-Served Ansatz führt zu besonders geringen Wartezeiten, beachtet aber keinerlei Auswahlkriterien und ist demnach eher kontraproduktiv.

Behälter

Ein in [17] vorgestellter Ansatz beinhaltet das automatische Bilden von Matchmakingräumen bzw. Behältern, in die Spieler mit gleichen Kriterien eingeordnet werden. Sobald ein Behälter genug Spieler beinhaltet, wird ein Spiel zwischen diesen gestartet. Der Prozess läuft in mehreren Zeitperioden ab. In der ersten Zeitperiode, die ungefähr eine Sekunde dauert, gelten alle Kriterien und es werden relativ viele unterschiedliche Behälter gebildet. Wenn diese Periode verstreicht und noch keine Mitspieler gefunden wurde,

[3] Eine Spielinstanz ist eine abgekappselte Einheit eines Spieles. Es wird für jede Gruppe die miteinander spielt quasi eine eigene Kopie des Spiels gestartet.

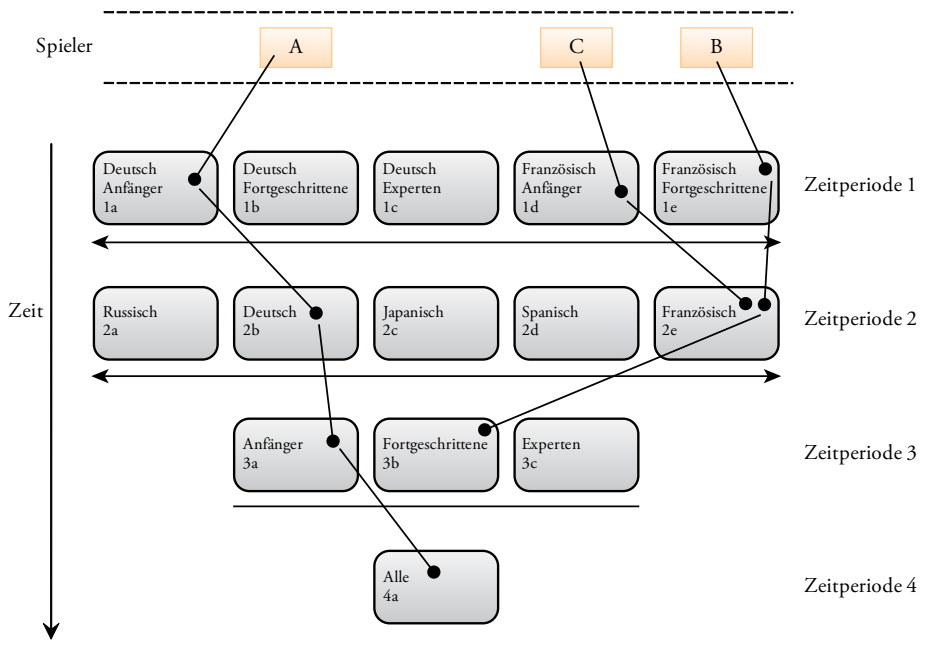


Abbildung 2.1: Eine schematische Abbildung einer Matchmaking Behälter Hierarchie. Spieler A findet keinen Partner und dringt bis in die unterste Stufe der Hierarchie vor. Während Spieler B drei Zeitperioden warten muss bevor ein neuer Spieler C hinzu kommt, mit dem er in Behälter 2e gematched wird. Bildquelle: [17]

so beginnt eine neue Zeitperiode und der Spieler kommt in einen Behälter einer tieferen Stufe. Dort gelten weniger Kriterien oder nur ein einziges Kriterium. Die Dauer der Zeitperioden kann mit jeder Stufe steigen, wenn dies gewünscht ist (auf bis zu 10 oder 20 Sekunden). Das wiederholt sich bis alle Kriterien durchgespielt wurden und eine finale Stufe erreicht wird, in der ein Benutzer bleibt, bis ein oder mehrere beliebige andere Spieler gefunden wurden, die auch noch keinen Spielpartner haben. Während die Matchmaking Hierarchie von oben nach unten durchlaufen wird, bleibt der Benutzer in allen Behältern vorhanden in denen er schon war. Dadurch kann er auch mit neu hinzukommenden Spielern zusammengeführt werden kann. Wenn ein Match zustande kommt, müssen alle daran teilnehmenden Spieler aus allen Behältern der Hierarchie entfernt werden.

Die Autoren von [17] schlagen auch vor, dass Spieler, die eine hohe Zahl an abgebrochenen Spielen aufweisen, bevor sie in die Matchmaking Hierarchie eintreten, vier Zeitperioden lang in einem eigenen Behälter verbringen in dem sie mit ihresgleichen gematcht werden. Diese Methode mag ein wenig hart erscheinen, vor allem, weil diese Personen dann keine anderen Kriterien gemeinsam haben. Es sollte für jedes Spiel neu überlegt und geprüft werden welches Vorgehen sich bei solchen „Vergehen“ am besten eignet.

Ein Beispieldiagramm für diesen Matchmakingprozess findet man in Abbildung 2.1. In der ersten Stufe ist für jedes Kriterienpaar ein eigener Behälter vorhanden. Wenn hier kein Partner gefunden wird so tritt der Spieler nach einer vordefinierten Zeit in Stufe 2 ein, in der Personen nur nach ihrer Sprache zusammengeführt werden. Wird auch hier kein Mitspieler gefunden, gelangt er zu Stufe 3 in der die Spieler nur nach ihrem Fähigkeitenlevel bewertet werden. Sollte er auch hier nach Ablauf der Zeit noch alleine sein, so gelangt er in den untersten Behälter in dem es keine Unterscheidungen mehr gibt. Natürlich kann dieser unterste Behälter weg gelassen werden, damit die Spieler zumindest in einem Kriterium zusammen passen, allerdings verlängert sich dadurch die Wartezeit unter Umständen um ein Vielfaches.

Spieler A befindet sich zuerst in Behälter 1a, weil sein Spielprogramm in deutscher Sprache läuft und er als Anfänger eingestuft ist. Dann wird versucht Gegner für ihn zu finden, die zumindest die gleiche Sprache sprechen wie er. Als das nicht gelingt sucht das Programm für ihn nach Gegnern mit dem gleichen Fertigkeitenlevel. Zu Schluss endet er auf der letzten Stufe, auf der er wartet bis ein anderer Spieler zu einem der Behälter dazustößt, in denen er sich befindet. Spieler B startet in Behälter 1e, weil er Französisch spricht und als fortgeschrittener Spieler gilt. Nach Ablauf der ersten Zeitperiode begiebt er sich auf Stufe 2 wo er, in Behälter 2e, nur nach französischsprachigen Spielern sucht. Währenddessen tritt Spieler C in die erste Zeitperiode ein, in der er in Behälter 1d landet. Nach Abschluss dieser Zeitperiode, landet Spieler B auf Stufe 3, während Spieler C auf Stufe 2 im Behälter für französische Sprache 2e gelangt. Da alle Spieler eine Präsenz in allen Behältern haben, in denen sie schon waren, befindet sich auch Spieler B in Behälter 2e. Dadurch sind nun in diesem Behälter genug Spieler für ein Match zusammengekommen und das Spiel kann gestartet werden.

Wenn die Spielerbasis grundsätzlich englischsprachig ist, so können Spieler, die keine besondern Spracheinstellungen haben, sofort auf Stufe 3 einsteigen, weil davon ausgegangen werden kann, dass alle Spieler, auch jene die zuvor nach anderssprachigen Gegnern suchen, Englisch können. Es werden also keine eigenen Behälter nur für Englisch sprechende Personen benötigt.

Ein Problem, das bei dieser Art des Matchmaking aufkommen kann, ist eine zu große Zahl an Kriterien, weil es dann zu viele Behälter gibt und die Hierarchie zu tief oder zu breit wird. Deshalb sollten die Kriterien möglichst überschaubar gehalten werden. Außerdem müssen die Kriterien in nominale Kategorien aufgeteilt werden. Für Ansätze bei denen Rating[4] verwendet wird, anstelle von Fertigkeitenlevels oder -klassen, ist es daher eigentlich nicht geeignet. Man könnte zwar zum Beispiel alle Spieler mit einem Rating von 1501–1600 in einen Behälter geben und Ratings 1601–1700 in den nächsten, aber auch das ist problematisch. Wenn nun in einem Behälter ein Spieler mit

[4]Ratings sind Punktestände die über eine Reihe von Spielen berechnet werden und die Fähigkeiten eines Spielers beschreiben. Mehr Informationen dazu in Kapitel 3

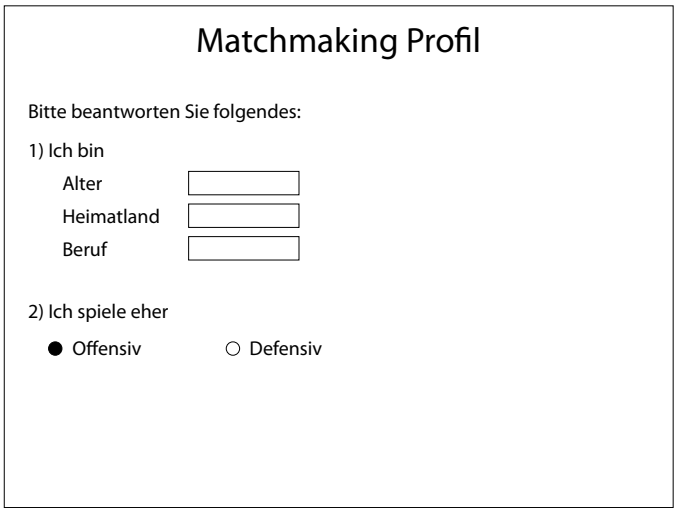


Abbildung 2.2: Ein Beispiel wie ein Profil eines Spielers aussehen kann. Es können persönliche Informationen übergeben werden, anhand derer Spielpartner gesucht und gefunden werden können. Bildquelle: [8]

1600 ist und im nächsten einer mit 1620, dann wäre es besser, wenn diese zwei miteinander spielen könnten. Da sie sich aber in unterschiedlichen Behältern befinden, muss der 1600 Punkte Spieler vielleicht stattdessen gegen einen 1530 Spieler antreten, was eine viel schlechtere Zuteilung darstellt.

Filter

Für jeden Benutzer wird ein Filter erstellt, basierend auf den Matchmaking relevanten Daten des Spielers. Dieser Filter wird dann mit dem Filter jedes anderen Spielers in der Warteliste des Programmes verglichen. Wenn zwei Spieler zueinander passen, also beide den Filter des jeweils anderen passieren, werden sie einander zugeteilt. Wenn genügend Spieler wechselseitig einander zugeteilt sind, also jeder passt zu jedem, können sie ein Spiel starten. Dieser Modus kann noch erweitert werden, indem die Filter durchlässiger werden je länger wartet.

In [8] wird ein Matchmaking Verfahren vorgestellt, das mit Filtern arbeitet. Jeder Spieler besitzt ein detailliertes Profil (wie in Abbildung 2.2), in dem der Spielstil und diverse persönliche Daten gespeichert werden. Wenn ein Benutzer ein Spiel starten will, so kann er sich, ähnlich dem manuellen Matchmaking Prozess, entscheiden, ob er eine Spielinstanz eröffnen oder einer bestehenden Instanz beitreten will. Wenn er eine eigene Instanz eröffnet, so kann er einen Filter anlegen, um zu bestimmen mit welcher Art von Personen er spielen möchte (siehe hierzu Abbildung 2.3). Wenn er hingegen einem bestehenden Spiel beitreten möchte so werden ihm alle Spiele ange-

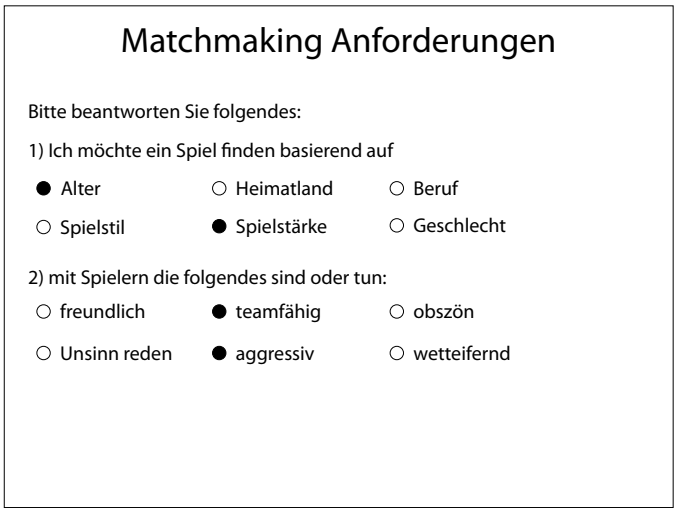


Abbildung 2.3: Ein Beispiel, wie Anforderungen an den Matchmaking Algorithmus aussehen können. In der ersten Kategorie wird anhand Daten gesucht, die der Spieler selbst eingibt und in der Zweiten finden sich Kriterien die durch Spielerbefragungen evaluiert wurden. Bildquelle: [8]

zeigt denen er beitreten kann, also zu deren Voraussetzungen er passt. Es ist außerdem möglich, selbst Kriterien zu wählen, um die verfügbaren Spiele zu filtern. Dann muss nur noch eine Spielinstanz ausgewählt werden, und sobald es genügend Teilnehmer in selbiger gibt, kann das Match beginnen. Nach einem Spiel ist es möglich die einzelnen Teilnehmer bewerten zu lassen. Dadurch bekommt das Programm Feedback zu Daten, die vom Spieler selbst nicht angegeben werden können oder sollen. Wie zum Beispiel ob eine Person recht vulgär ist, oder viel Unsinn redet. Ob es gute Teamspieler sind, oder besonders freundlich. Auch solche Daten können dann in die Filter eingearbeitet werden.

Der Prozess der Matchfindung, der hier beschrieben ist, kann unter Umständen sehr aufwendig sein. Daher sollte der Umfang an Daten die manuell in einen Filter eingegeben werden müssen, möglichst gering gehalten werden, um die Benutzer nicht zu lange aufzuhalten.

Eine Erweiterung dieses Systems wäre durch kollaboratives Filtern möglich. Das beschreibt einen Prozess indem Verhaltensmuster von Benutzern ausgewertet werden, um daraus auf die Interessen von Einzelnen schließen zu können [21]. In diesem Ansatz ist es essentiell, dass Benutzer ihre Mitspieler nach dem Spiel kurz bewerten. Daraus erstellt das Programm dann Netzwerke aus denen sich Wahrscheinlichkeiten ablesen lassen, die bestimmen ob ein Spieler gut zu einem anderen passt oder nicht (siehe Abbildung 2.4). Spielt eine Person A zum Beispiel gerne mit Person B, dann kann er mit anderen Personen zusammengeführt werden, die auch gern mit B spielen

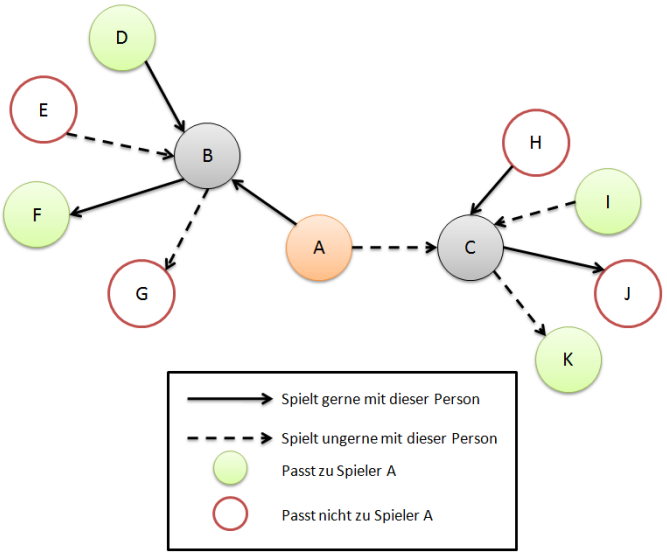


Abbildung 2.4: Beispielhaftes Netzwerk, wie es aus einem kollaborativen Filterprozess gewonnen werden kann. Da Spieler A gerne mit Spieler B spielt, es aber nicht so toll fand mit Spieler C zu spielen, hat er eine hohe Wahrscheinlichkeit mit Spielern D, F, I und K gematched zu werden, wohingegen er eher nicht mit Spielern E, G, H und J zusammengebracht werden wird. Bildquelle: [8]

oder mit denen B gerne spielt. Wenn A hingegen nicht gerne mit Person C spielt, so werden Personen die mit C gerne spielen oder mit denen C gerne spielt eher nicht mit Spieler A gematcht. Daraus folgend werden A auch Personen für ein Match vorgeschlagen, die gleichfalls nicht gerne mit C spielten, wohingegen Personen die nicht gerne mit B gespielt haben eher von ihm fern gehalten werden.

Auch in diesem System ist es wichtig, den Benutzern keine zu langen Fragebögen zum Spielverhalten der anderen Teilnehmer vorzulegen, um sie nicht zu lange aufzuhalten. Am einfachsten ist es, nur zu fragen, ob sie Spass daran hatten mit einer bestimmten Person zu spielen oder nicht. Wenn der Benutzer gewillt ist, kann er noch optionale andere Fragen beantworten, die zur Erstellung genauerer Profile einzelner Personen hilfreich sind.

Kapitel 3

Rating Verfahren

Das Rating ist eigentlich nur einer von verschiedenen Ansätzen mit denen versucht wird, die sogenannte Spielstärke[1] zu messen oder festzuhalten. Arpad Elo schreibt dazu [7]:

> "In a competitive environment such as chess, tournament standings provide tentative rankings, but because individual performances vary from time to time, a ranking list based on a single event is not always reliable. Furthermore, it may be necessary to compare performances of players or teams who never met in direct competition."

Der Bedarf an Spielstärkeermittlungen entsteht aus dem Wunsch der Spieler sich miteinander zu messen und herauszufinden, wer ein Spiel oder einen Sport besser beherrscht. Eine Frage, die möglichst objektiv unter folgenden Grundvoraussetzungen beantwortet werden soll:

Reproduzierbarkeit: Es ist wichtig, dass gleiche erbrachte Leistungen auch gleiche Ergebnisse erzielen und Spieler mit vergleichbaren Fähigkeiten ähnliche Ränge erhalten [27].

Nachvollziehbarkeit: Der Durchschnittsspieler sollte die Punktestände intuitiv verstehen und nachvollziehen können, wie und warum er Punkte verliert oder gewinnt [27].

Integrität: Damit ist gemeint, dass die Bedeutung von Punkteständen sich nicht über die Zeit verändern soll, sondern immer eine bestimmte gleichbleibende Spielstärke repräsentiert [7].

Unbeeinflussbarkeit: Die Bewertung einer Person sollte nicht durch unlautere Methoden beeinflusst werden können, sondern ihre wahre Stärke so gut es geht wiedergeben.

Abgesehen von diesen elementaren Eigenschaften, die das Rating System mitbringen soll, ist es auch wichtig für Game Designer, sich zu vergewissern,

[1] Der Begriff der Spielstärke beschreibt die Leistung, die ein Spieler oder Sportler in seiner Disziplin erbringt.

dass das gewählte Verfahren gut ins Spiel eingebunden ist. Denn ungeachtet der Tatsache, ob es beabsichtigt ist oder nicht, wird damit immer das Spielverhalten der User beeinflusst. Diese sind meist darauf aus, den besten Platz in der Highscoreliste, das höchste Rating oder die meisten Punkte zu haben [12]. Im schlechtesten Fall wird damit unerwünschtes Spielverhalten (zum Beispiel Campen[2] bei Shootern) gefördert. Daher muss es sehr gut überlegt sein, für welches Verhalten Punkte vergeben werden, um das Spiel auf die richtige Art zu unterstützen.

Die wichtigsten Ansätze zur Messung der Spielstärke sind Score, Ranking und Rating, welche sich keinesfalls ausschließen, sondern eher ineinander verwoben sind [27]:

Score *(Punktestand)*: Punkte, die in einem Spiel erreicht werden können. Ein einfacher Zähler, der durch bestimmte Auslöser im Spiel verändert wird (z. B. besiegte Gegner, gesammelte Items[3], gelöste Rätsel, geschossene Tore, usw.) Auch Zeit wird oft zur Ermittlung von Punktetänden verwendet. Da Scores direkt durch Aktionen der Spieler gesteuert werden, sind sie auch sehr einfach zu beeinflussen, was zu Problemen führen kann. Einerseits können sich Spieler unter Umständen einen Highscore erschummeln, der gar nicht ihren wirklichen Fähigkeiten entspricht, andererseits kann eine ungeschickt angelegte Punktemechanik auch zu unerwünschtem Spielverhalten führen. Ganz zu schweigen davon, dass Punkte manchmal auch für Dinge vergeben werden, die über die eigentliche Spielstärke gar nichts aussagen.

Ranking *(Platzierung)*: Durch direkte Gegenüberstellung einzelner Kontrahenten oder anhand einer aussagekräftigen Variable (zum Beispiel einem Punktestand), können Objekte in eine Rangliste sortiert werden. Der Platz in der Rangliste ist das Ranking. Beispiel: Wenn Spieler A Spieler B besiegt, muss A höher platziert werden als B

Rating *(Bewertung)*: Erweitert den Begriff des Rankings, indem eine metrische Skala (mehr dazu in Abschnitt 3.1.1) eingeführt wird (Beispiel: A ist nicht einfach nur besser als B, es kann sogar abschätzen werden, wie viel besser). Im Gegensatz zu einem normalen Punktestand werden Ereignisse nicht nur gezählt sondern auch bewertet. Zum Beispiel bringt einen Boss-Gegner[4] zu besiegen mehr Punkte als einen normalen Gegner. Spieler können anhand ihres Ratings sortiert werden und daraus lässt sich wieder eine Rangliste erstellen bzw. ein Ranking ableiten. Das Rating einer Person kann nicht als absolut gelten, sondern nur relativ zu den Ratings seiner Kontrahenten gesehen werden.

[2]Campen bezeichnet das als unsportlich angesehene Verhalten von Spielern, sich an einem möglichst sicheren Ort zu verschanzen, von wo aus sie andere, vorbeikommende Spieler einfach ausschalten können.

[3]Items werden die sammelbaren Gegenstände in einem Spiel genannt.

[4]Boss-Gegner sind Endgegner oder besonders starke Gegner in einem Computerspiel.

Arpad Elo hat durch seinen Schach Rating Algorithmus den Grundstein für kompetitives Rating gelegt und die meisten anderen, neueren Algorithmen sind aus seinem entstanden, oder dadurch inspiriert. In diesen Algorithmen werden, oberflächlich betrachtet, nur die Gewinne und Verluste einer Person betrachtet und ausgewertet. Dadurch ist die Angriffsfläche für Spielmanipulationen gering, denn wenn ein Spiel gewonnen wird, sagt das in jedem Fall etwas über die Spielstärke aus. Abgesehen von Sonderfällen, in denen ein Spieler einen anderen gewinnen lässt oder das Spiel mit unfairen Mitteln gewonnen wurde. Außerdem können in den meisten Spielen Gewinne und Verluste extrahiert werden, was es zu einem universell einsetzbaren Rating System macht.

3.1 Grundlagen

Um die nachfolgenden, unterschiedlichen Rating Systeme und Algorithmen verständlicher zu machen, sollen hier ein paar Grundlagen dargelegt werden, die allen gemein sind. Arpad Elo drückt den kleinsten gemeinsamen Nenner folgendermaßen aus [7]:

> "[...] rating individual performances in any competitive activity is basically a measurement problem [...]"

Um Leistungen bewerten zu können, müssen sie zu aller erst gemessen werden können. Daraus ergeben sich folgende grundsätzliche Fragen, die beantwortet werden müssen:

Was wird gemessen? Die Ausprägung der Fähigkeiten von Spielern oder Sportlern.

Wie wird gemessen? Durch Wettkämpfe oder Spiele, bei denen immer zwei Spieler oder zwei Teams gegeneinander antreten. Diese Methode nennt sich Paarweiser Vergleich (Mehr dazu in Abschnitt 3.4). Das Ergebnis einer Messung kann ein Gewinn oder ein Verlust, in manchen Fällen auch ein Unentschieden sein.

Wie werden die Messergebnisse abgebildet? Wenn mehrere Messungen durchgeführt wurden, ist es wichtig, diese auch miteinander in Relation zu setzen. Dazu wird vor allem eine Messskala benötigt um zumindest einen Platz in der Rangordnung und, im besten Fall, genaue Bewertungen vergeben zu können. Wie die Messskala und die Ratings funktionieren wird in den folgenden Abschnitten erklärt.

Die wichtigste Frage, nämlich wie man von den einzelnen Messergebnissen zu den fertigen Bewertungen kommt, ist natürlich in jedem Rating System anders beantwortet und wird dann in den jeweiligen Abschnitten erläutert.

3.1.1 Die Messskala

Der erste Schritt um objektive Bewertungen erzeugen zu können, ist die Einführung einer geeigneten Messkala. Bei der Messung von Daten lassen sich folgende vier Skalenniveaus unterscheiden:

Nominalskala: Die Werte der zugrundeliegenden Variable können nur von einander unterschieden werden ($x = y$ oder $x \neq y$), es kann aber keine Aussage darüber getroffen werden, ob ein Wert größer oder kleiner ist als eine anderer. Es kann also nicht nach der Größe sortiert werden [2] (Beispielvariablen: Blutgruppen, Geschlecht).

Ordinalskala: Bei einem ordinalen Skalenniveau lässt sich eine Rangordnung zwischen je zwei unterschiedlichen Variablenwerten herstellen ($x < y$ oder $x > y$). Es lässt sich aber nicht bestimmen, wie groß die Abstände zwischen diesen Werten sind [2] (Beispielvariable: Schulnoten).

Intervallskala: Ist eine sogenannte metrische Skala, da die Werte sich quantitativ mittels Zahlen darstellen lassen. Im Gegensatz zur Ordinalskala lassen sich Aussagen darüber treffen, wie viel größer ein Wert als ein anderer ist ($x + a = y$). Die Abstände zwischen den unterschiedlichen Variablenwerten lassen sich also exakt bestimmen, allerdings ist der Nullpunkt willkürlich festgelegt [2] (Beispielvariable: Datum).

Verhältnisskala: *(=Ratioskala, absolute Skala)* Ist auch eine metrische Skala. Im Unterschied zur Intervallskala gibt es allerdings einen absoluten Nullpunkt. Nur Werte einer Ratioskala können multipliziert und dividiert werden, wodurch sich Verhältnisse zwischen den Werten bilden lassen ($x \cdot a = y$) [2] (Beispielvariablen: Gewicht, Längenmaß, Alter, Preis).

In Rating Systemen für Spiele machen Nominalskalen natürlich wenig Sinn. Ordinalskalen kommen vor allem in Form von Rangskalen[5] des öfteren vor, sind aber eigentlich auch nur mäßig gut geeignet um die Fähigkeiten von Spieleren oder Sportlern darzustellen. Werden Ranglisten betrachtet, so lässt sich nur erfahren welcher Spieler stärker ist, allerdings nicht wie viel stärker er ist. Besser und vor allem aussagekräftiger sind Intervall- oder Verhältnisskalen, weil durch diese abgebildet werden kann, wie unterschiedlich die Fertigkeiten zweier Spieler wirklich sind. In Elo-Rating-Systemen werden Intervallskalen verwendet [7]. Es ist leider nicht wirklich möglich Ratioskalen zu verwenden, da kaum ein absoluter Nullpunkt für etwas so Abstraktes wie Spielerfähigkeiten gefunden werden kann.

Die Intervalle *(=Einheiten)* von Messskalen sind grundsätzlich willkürlich festgelegt. Einige werden nach Standardeinheiten, wie dem Urkilo, aus-

[5] Rangskalen sind eine Sonderform der Ordinalskala in welcher jeder Wert nur einmal vergeben wird. Es ist also nicht wirklich möglich, dass zwei Spieler den gleichen Rang haben (Beispiel: Fussballturnier).

gewählt, andere werden erzeugt aus reproduzierbaren Fixpunkten, wie dem Gefrierpunkt und dem Siedepunkt von Wasser. Manche wissenschaftlichen Skalen haben aber weder Fixpunkte noch Standardeinheiten, wie die Richterskala oder die Dezibelskala. Auch bei der Elo Rating Skala ist dies der Fall, die Einheiten wurden aus statistischen Gründen gewählt [7]. Ratingpunkte werden außerdem als linear angenommen (also nicht wie bei der Dezibelskala wo ein db mehr die doppelte Lautstärke bedeutet). Dadurch kann der Null-Punkt an jede beliebige Stelle gesetzt werden, weil nur die Differenz zweier Ratings als wichtig erachtet wird [7, 27]. Der *Mittelwert*[6] der Ratings sollte am besten so hoch gewählt werden, dass kein Spieler in die höchst demotivierende Situation kommt ein negatives Rating zu erhalten [27]. Die Ratings der *Spielerbasis*[7] werden als annähernd normalverteilt um den Mittelwert angenommen.

3.1.2 Das Rating

Bei Elo Rating basierten Algorithmen wird die Spielstärke in einer Wahrscheinlichkeitsverteilung abgebildet. Diese Verteilung stellt dar, mit welcher Wahrscheinlichkeit die zugrundeliegende Variable (= der Rating Parameter) einen bestimmten Wert annimmt, also mit welcher Wahrscheinlichkeit eine Person eine bestimmte Leistung erbringen kann. Der Wert, der als das Rating verstanden wird, ist eigentlich nur der Erwartungswert dieser Verteilung. Da es aber bei der wahrgenommenen Spielstärke einer Person eine natürliche Varianz gibt, er oder sie also nicht immer gleich gut performt obwohl sich die eigentliche Spielstärke nicht verändert, ist kein absoluter Wert dafür annehmbar.

Erwartungswert $E(X)$, μ: Der Erwartungswert ist nach üblicher Definition jener Wert, der sich nach oftmaligem Wiederholen eines Experiments als Mittelwert der Ergebnisse einstellt [2]. Beispielhaft kann man sagen, dass nur weil Person A einen höheren Wert als Person B hat, dies nicht bedeutet, dass A immer besser performt als B, nur dass die Wahrscheinlichkeit, dass es so ist sehr hoch ist. Wenn sich die Verteilungen, wie in Bild 3.1, überschneiden, kann es trotzdem sein dass Objekt B vorgezogen wird, wenn seine Variable einen sehr hohen Wert annimmt.

Varianz σ^2: Neben dem Erwartungswert ist die Varianz der zweite wichtige Parameter durch den Wahrscheinlichkeitsdistributionen festgelegt werden. Sie gibt an, wie stark die Variable um ihren Mittelwert streut, also wie sehr sie davon abweichen kann. Das heißt je niedriger die Varianz, desto genauer ist die Schätzung des Erwartungswertes. In Abbildung

[6] Ähnlich wie beim IQ gibt es auch bei Ratings einen Mittelwert, der als Durchschnitt aller Ratings in der Spielerbasis gilt.

[7] Die Spielerbasis beschreibt alle Personen die ein Spiel oder einen Sport betreiben und ein Rating besitzen.

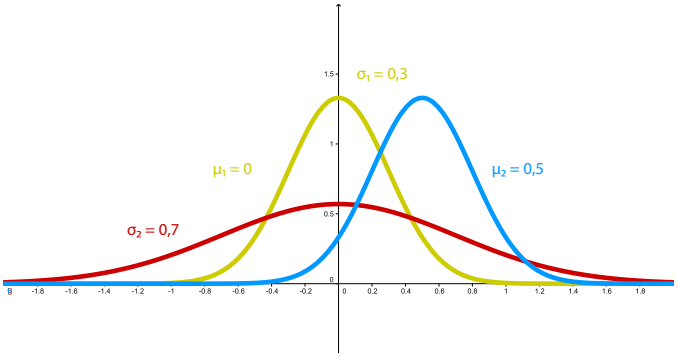


Abbildung 3.1: Verteilungen mit unterschiedlichen Erwartungswerten und Varianzen. Die gelbe und die blaue Kurve haben die gleiche Varianz, aber unterschiedliche Erwartungswerte, während die gelbe und die rote Kurve zwar den gleichen Erwartungswert besitzen, aber die rote eine höhere Varianz hat. Wenn die Varianz höher wird, wird die Kurve flacher, da die Fläche unter der Kurve immer 1 ist.

3.1 sieht man wie Normalverteilungen bei unterschiedlicher Varianz aussehen. Die Wurzel aus der Varianz – also σ – ist die Standardabweichung, die die gleiche Einheit wie der Erwartungswert besitzt. Die Varianz dagegen wird mit der Einheit zum Quadrat notiert [2].

3.2 Elo Rating

Das Elo-Rating-System wurde 1959 von Arpad Elo, im Auftrag der USCF[8], für Schach entwickelt, da das bis dahin verwendete Harkness System[9] einige Probleme mit sich brachte (vor allem, weil die Ratings nur sehr selten aktualisiert wurden) und von vielen Personen als ungenau erachtet wurde [7]. Seit es 1970 von der USCF eingesetzt wurde, ist es schnell populär geworden, weil es fairer war als bis dahin verwendete Systeme. Danach wurde es, gegebenenfalls mit Verbesserungen, auch noch in einigen anderen Sportarten (American Football, Golf, Fußball, Baseball, usw.) und Spiele (Go, Backgammon, usw.) übernommen [22]. Auch in Computerspielen ist es die Grundlage für die meisten Rating Algorithmen (sofern ein solcher vorhanden ist), obwohl jedes Spiel noch seine eigenen Veränderungen am System vornimmt, um es an die speziellen Bedürfnisse anzupassen[10].

[8] United States Chess Federation

[9] Das Harkness System ist benannt nach seinem Entwickler Kenneth Harkness.

[10] http://na.leagueoflegends.com/learn/gameplay/matchmaking

3.2.1 Theorie

Das Grundprinzip des Elo Rating Algorithmus ist recht simpel. Die Leistung eines Spielers wird abgleitet aus den Gewinnen und Verlusten absolvierter Partien. Wenn ein Spieler eine bessere Leistung erbringt als erwartet (er also gegen einen Gegner mit höherem Rating gewinnt) bekommt er Punkte, erzielt er eine schlechtere Leistung verliert er Punkte. Aus einem einzelnen Match kann allerdings keine objektive Aussage über die Spielstärke einer Person gegeben werden. Daher muss eine größere Anzahl an Partien hergenommen werden, um daraus das sogenannte *Performance Rating*[11] zu berechnen. Doch selbst ein Performance Rating wird die wahre Stärke eines Spielers nicht akurat wiedergeben, da es eine natürliche Varianz der Leistung gibt. Das soll heißen eine Person performt nicht immer gleich gut, auch wenn sich ihre Spielstärke in Wirklichkeit nicht ändert. Ein jeder kann gute und schlechte Tage haben, manchmal auch über einen längeren Zeitraum hinweg, was das Performance Rating stark beeinflusst. Daher kombiniert das Elo-Rating-System einzelne Performance Ratings miteinander, um die best mögliche Schätzung der relativen Stärke eines Spielers zu erhalten. Diese Kombination ergibt das Rating oder auch Player Rating. Selbst dieses wird noch zufällige Schwankungen aufweisen, aber nicht mehr in dem Maß wie das Performance Rating [7].

Aus der Differenz der Ratings zweier Kontrahenten (oder einem Spieler und dem durchschnittlichen Rating seiner Kontrahenten dieser *Rating Periode*[12]) lässt sich die Wahrscheinlichkeit für einen Gewinn berechnen. Umgekehrt lässt sich aus Spielergebnissen eine Ratingdifferenz berechnen. Das ist die Differenz, die zwischen den Ratings zweier Spieler bestehen sollte. Es werden also die Ratings so verändert, dass sie dieser Differenz näher kommen.

Der Ablauf des Algorithmus ist wie folgt:

1. Berechnen welcher Prozentsatz an gewonnenen Spielen zu erwarten ist.
2. Abweichung zwischen tatsächlichem Ergebnis und erwartetem Ergebnis feststellen.
3. Rating anpassen.

3.2.2 Berechnung

Im Grunde besteht das Elo-Rating-System aus zwei wichtigen Formeln. Die Erste berechnet die Wahrscheinlichkeit für einen Gewinn ausgehend von der Ratingdifferenz zweier Spieler und die Zweite aktualisiert das aktuelle Rating in Abhängigkeit der Abweichung zwischen erwartetem und tatsächlichem

[11] Performance Rating ist ein Rating über eine bestimmte Anzahl von Spielen, manchmal auch über einen gewissen Zeitraum hinweg, oder das Rating eines einzelnen Turniers.

[12] Eine Rating Periode ist ein bestimmter Zeitraum oder eine bestimmte Anzahl von Spielen über die hinweg ein Performance Rating erstellt wird.

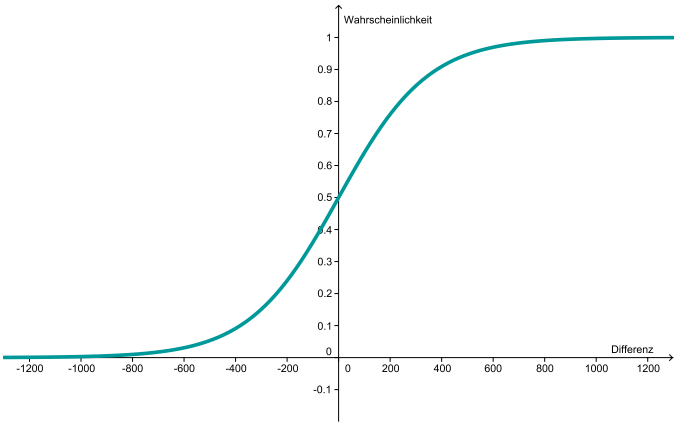
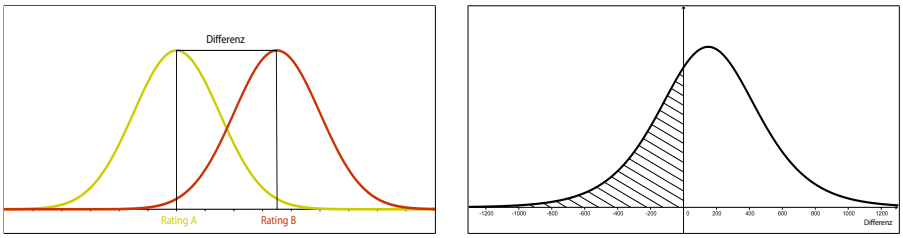


Abbildung 3.2: Die logistische Wahrscheinlichkeitsfunktion über die Ratingdifferenz, die im Elo Rating System verwendet werden kann. Bei einer Ratingdifferenz von 0 liegt die Wahrscheinlichkeit bei 0.5, bei steigender Differenz gleicht sie sich einem Wert von 1 an, bei fallender negativer Differenz wird sie 0.

Ausgang.

Wahrscheinlichkeitsfunktion (Verteilungsfunktion)

Mit der Wahrscheinlichkeitsfunktion werden die zu erwartenden Gewinne bzw. die Wahrscheinlichkeit für einen einzelnen Gewinn gegen einen bestimmten Gegner errechnet. Die einfachste Möglichkeit den möglichen Gewinner zu finden, ist der Gebrauch einer simplen Schwellenwertfunktion. Wer das höhere Rating hat (positive Differenz) gewinnt, wer das niedrigere Rating hat (negative Differenz) verliert. Allerdings ist so ein Ansatz nur passend, wenn es keine Ungewissheit, keinen Zufall in Spielen gibt [27]. Es wurde auch schon mehrfach in dieser Arbeit festgehalten, dass ein Spieler nicht immer die gleiche Leistung erbringt, sein Rating dementsprechend kein absoluter Wert ist. Daher wird eine Funktion benötigt die eine graduelle Unterscheidung ermöglicht, nicht nur das Vorzeichen der Differenz beachtet, sondern auch die Höhe der Differenz. Ab einer gewissen Größe der Differenz soll die Wahrscheinlichkeit an 0 (der schwächere Spieler gewinnt) bzw. an 1 (der stärkere Spieler gewinnt) angenähert werden. Wenn die Spieler gleich stark sind, so soll die Wahrscheinlichkeit für einen Gewinn bei 0.5 liegen – eine 50–50 Chance für beide. Außerdem muss die Funktion streng monoton sein, damit mit steigender Ratingdifferenz auch steigende Gewinnwahrscheinlichkeiten geliefert werden [27]. Unter diesen Voraussetzungen ist eine sogenannte Sigmoid-Funktion, wie sie in Abbildung 3.2 dargestellt ist, am besten dafür geeignet die Wahrscheinlichkeiten zu berechnen. Die Standard Sigmoid



(a) (b)

Abbildung 3.3: (a) zeigt die Ratingverteilungen zweier Spieler. Grundsätzlich wird die Differenz vom Mittelwert aus berechnet, da aber die Ratings wahrscheinlichkeitsverteilt sind, also einen beliebigen Wert innerhalb der Kurve annehmen können, kann auch die Differenz einen anderen Wert annehmen und ist somit auch wahrscheinlichkeitsverteilt wie in (b) zu sehen ist. Der markierte Bereich, der links von der Differenz 0 liegt, ist jener Bereich in dem der schwächere Spieler gewinnt. Der andere Bereich stellt die Gewinnwahrscheinlichkeit des stärkeren Spielers dar.

Funktion

$$sig(x) = \frac{1}{1 + e^{-x}}$$

wird auch Logistische Funktion genannt und ihre Ableitung ist eine logistische Verteilung, die sich gut eignet, um die Verteilungsdichte der Ratingdifferenz darzustellen. Wenn davon ausgegeangen wird, dass die Ratings zweier Spieler wahrscheinlichkeitsverteilt sind (siehe Abbildung 3.3 (a)), so ist auch die Differenz zwischen den Ratings nicht absolut, sondern um den Differenzwert verteilt. Wie in Abbildung 3.3 (b) ersichtlich, fällt ein Teil der Distribution auf die negative Seite des Nullpunktes, was den Bereich darstellt, indem der schwächere Spieler gewinnt. Der andere Bereich stellt die Wahrscheinlichkeit dar, mit der der stärkere Spieler gewinnt. Mit dem Integral der Funktion – die schon dargelegte Wahrscheinlichkeitsfunktion – kann die Fläche unter der Kurve berechnet werden und damit die Wahrscheinlichkeit, mit der die zugrundeliegene Variable – der Differenzwert – in diesen Bereich fällt [19].

Durch Reparametrisierung der Standard Sigmoid Funktion entsteht die üblicherweise im Elo-Rating-System verwendete Gleichung für die Gewinnabschätzung

$$\mathrm{E}(w|D) = \frac{1}{1 + 10^{-D/2C}}, \tag{3.1}$$

wobei $\mathrm{E}(w|D)$ der Erwartungswert für einen Gewinn w in Abhängigkeit von der Differenz D ist. C steht für das Klassenintervall, welches im Elo-Rating-System üblicherweise mit 200 definiert ist und dazu dient, die Differenzwerte für die verwendete Skala ($\sim 0 - 3000$) zu gewichten [7].

Elo stellt in seinem Buch noch verschiedene andere mögliche Gleichungen vor, die zur Wahrscheinlichkeitsberechnung verwendet werden können. Im

Allgemeinen wird aber immer von der logistischen Verteilung ausgegangen (Elo selbst ging von der Normalverteilung aus, die hier nicht dargestellt wurde). Laut Beasley ist es allerdings relativ egal welche Funktion verwendet wird, weil die Fehler, die dadurch entstehen können, nicht so groß sind, wie die Fehler die durch die natürliche Varianz der Leistung entstehen [1].

Mit der Umkehrformel der Erwartungsfunktion

$$D(p) = \lg\left(\frac{1}{p^{-1}-1}\right) \cdot 2C \tag{3.2}$$

kann aus den Prozent der gewonnenen Spiele p die Differenz berechnet werden, die zwischen zwei Spielern stehen soll. Das funktioniert allerdings nur, wenn die Anzahl der Spiele, die herangezogen werden, hinreichend groß ist, um genug unterschiedliche Ergebnisse zu beinhalten [7].

Rating Aktualisierung

Zur kontinuierlichen Berechnung der Ratings werden diese nach jedem Event – das kann eine beliebige Anzahl n an Spielen oder auch ein Turnier sein – mit folgender Gleichung auf den neuesten Stand gebracht:

$$r' = r + K \cdot \sum_{j=1}^{n}(w_j - \mathrm{E}(w|D_j)). \tag{3.3}$$

Das neue Rating r' ergibt sich also aus dem alten Rating r eines Spielers, addiert mit der durch K gewichteten Differenz der tatsächlichen Gewinne w und der erwarteten Gewinne (Gewinnwahrscheinlichkeit) $\mathrm{E}(w|D)$. Die Gleichung rechnet die letzte Leistung so in das alte Rating ein, dass langsam der Effekt der alten Leistungen verringert wird, während der Beitrag der neuesten Leistung vollständig vorhanden bleibt. Die Logik hinter der Gleichung ist offensichtlich – wenn ein Spieler besser ist als erwartet, so gewinnt er Punkte, ist er schlechter als erwartet, so verliert er Punkte [7]. Durch den Ratingkoeffizienten K kann gesteuert werden, wie viele Ratingpunkte pro Event maximal dazugewonnen oder verloren werden können. Wenn der Koeffizient groß ist, so werden die letzterbrachten Leistungen höher gewertet, bei einem geringen Koeffizienten liegt die Gewichtung eher auf den früheren Leistungen [7]. Im Elo-Rating-System liegt der Wert von K zwischen 10 und 32, wobei 32 in Systemen verwendet wird in denen die Ratingperioden kurz und die Fähigkeiten der Spieler sehr unterschiedlich sind. Daher ist dies ein Wert, welcher auch für Onlinespiele geeignet scheint. Der Gewinn w_j gibt den Ausgang eines Matches gegen einen Spieler j an. Wobei 1 für einen Gewinn steht, 0 für einen Verlust und 0,5 für ein Unentschieden. Für die Gewinnwahrscheinlichkeit $\mathrm{E}(w|D_j)$ wird die Differenz $D_j = (r - r_j)$ des Ratings des Spielers zu seinem jeweiligen Gegener benötigt. Berechnet wird die Wahrscheinlichkeit mit der Gleichung 3.1.

Performance Rating Formel

Neben der kontinuierlichen Rating Formel stellt Elo auch noch die sogenannte Performance Rating Formel vor, die eine einfache Möglichkeit darstellt, um die Leistung eines Spielers während einer Rating Periode zu bestimmen. Eine Rating Periode kann ein beliebiges Zeitintervall umfassen, sollte aber zumindest dreißig Spiele beinhalten um das Rating des Spielers sinnvoll ermitteln zu können [7]. In der Gleichung

$$r_p = r_c + D(p) \tag{3.4}$$

wird das Performance Rating r_p eines Spielers berechnet, indem der Mittelwert der Ratings all seiner Konkurrenten r_c mit der Differnz D addiert wird, welche aus dem Prozentsatz p an gewonnenen Spielen mit der Gleichung 3.2 ermittelt wird. Das Ergebnis der Funktion ist jenes Rating, an dem die erwarteten Spielergebnisse gleich den tatsächlichen Ergebnissen sind. Die Gleichung wird vor allem dazu verwendet, vorläufige Ratings für Spieler zu vergeben, die noch kein Rating in einem Spiel besitzen. Sie kann kein Rating erzeugen wenn ein Spieler alle oder kein einziges seiner Spiele gewinnt, daher müssen genug Daten vorhanden sein, um eine gewisse Vielfalt an Ergebnissen beinhalten zu können. Auch sollte die Spanne an Gegner Ratings nicht mehr als drei Klassenintervalle (=600 Punkte) umfassen, damit akzeptable Ergebnisse erzielt werden [7].

3.2.3 Beispiele

Performance Rating Formel

Wenn ein Spieler betrachtet wird, der noch kein Rating besitzt, so wird dieses am besten aus einer Reihe an Spielen mit der Performance Rating Formel berechnet. Wenn die Gegenspieler und ihre Ratings durch folgende Tabelle gegeben sind

Spieler	Rating
Spieler 1	2470
Spieler 2	2485
Spieler 3	2560
Spieler 4	2615
Spieler 5	2510
Spieler 6	2410

und der Spieler von diesen Partien 4,5, also $p = 0{,}75$ Prozent, gewonnen hat, so ergibt sich sein Rating durch eine einfache Berechnung. Zuerst muss das durchschnittliche Rating aller Beteiligten bestimmt werden:

$$r_c = \frac{\sum_{j=1}^{6}(r_j)}{6} = \frac{2470 + 2485 + \ldots + 2410}{6} = 2508.$$

Danach kann mit Gleichung 3.2 die Differenz berechnet werden, die zwischen diesem Rating und dem Rating des Spielers liegen sollte. Das sind

$$D(0{,}75) = \lg\left(\frac{1}{0{,}75^{-1} - 1}\right) \cdot 400 = 191$$

Ratingpunkte. Das Performance Rating des Spielers ist also

$$r_p = 2508 + 191 = 2699.$$

Kontinuierliche Rating Formel

Wenn hingegen schon ein Rating für einen Spieler vorliegt zum Beispiel 2635, so kann die kontinuierliche Rating Formel genommen werden um ein genaueres Rating des Spielers zu erhalten. Dafür gibt es wiederum zwei Möglichkeiten, wenn die Ergebnisse gegen jeden einzelnen Spieler bekannt sind, werden die Gewinne einzeln mit der jeweiligen Gewinnerwartung gegengerechnet. Sollten aber, wie in dieser Angabe, nur die Gesamtzahl der Gewinne bekannt sein, so wird die durchschnittliche Differenz zu den Ratings der Gegner ausgerechnet. In diesem Fall also

$$D_c = r - r_c = 2635 - 2508 = 127$$

und daraus wird mit Gleichung 3.1 die Gewinnwahrscheinlichkeit ermittelt. Diese beträgt

$$\mathrm{E}(w|127) = \frac{1}{1 + 10^{-127/400}} = 0,675$$

Prozent, also eine Wahrscheinlichkeit, dass $0{,}675 \cdot 6 = 4$ Spiele von den 6 gewonnen werden. Da aber 4,5 Spiele gewonnen wurden, ändert sich das Rating bei einem Ratingkoeffizienten von $K = 10$ zu

$$r' = 2635 + 10 \cdot (4,5 - 4) = 2640.$$

3.3 Glicko Rating

Mark E. Glickman entwarf die Grundlagen des Glicko-Rating-Systems im Jahr 1995, weil er im Elo-Rating-System zu viele Probleme erkannt hat und ein besseres Rating System zur Verfügung stellen wollte. Das größte Problem, das er dem Elo-Rating-System vorwarf, und das von seinem System in Angriff genommen wird, ist die Beständigkeit von Spielerratings. Während Elo davon ausging, dass Ratings wenn sie einmal etabliert sind, relativ beständig sind, so soll im Glicko System auch mit einberechnet werden, dass sich Fähigkeiten mit der Zeit verändern können. Glickman ist der Meinung, dass das Rating eines Spielers der häufig spielt vertrauenswürdiger ist, als das Rating eines Spielers der länger ausgesetzt hat [26]. Sollten also zwei Spieler gegeneinander

antreten die ein gleiches Rating haben, von denen einer, Spieler A, häufig spielt und der andere, Spieler B, nach einer längeren Pause wieder anfängt zu spielen. So ist das Rating von Spieler A viel verlässlicher und sollte nur sehr wenig angepasst werden, da das Rating von Spieler B nicht zuverlässig genug ist, wodurch das Spielergebnis nur wenig Information über seine eigene Spielstärke liefert. Wohingegen das Rating von Spieler B stark verändert werden sollte, da die Tatsache, dass er gegen einen etablierten Spieler gewinnt oder verliert, eine sehr hohe Aussagekraft darüber besitzt welches Rating für ihn angemessen wäre.

3.3.1 Theorie

Um die Verlässlichkeit eines Ratings in den Algorithmus einzubringen, geht Glickman davon aus, dass nicht nur der Mittelwert eines Ratings von Bedeutung ist, sondern auch die Varianz beziehungsweise die Standardabweichung, die von ihm als *Rating Deviation (=RD)* benannt wird [26]. Eine hohe RD bedeutet, dass ein Rating relativ unzuverlässig ist, weil der Spieler nur selten für Partien antritt oder schon lange nicht mehr angetreten ist. Wohingegen eine kleine RD bedeutet, dass er häufig spielt, wodurch sein Rating recht etabliert ist. Ein Beispiel für Verteilungen mit unterschiedlichen Varianzen ist in Abbildung 3.1 zu sehen. Das Rating eines Spielers wird, im Gegensatz zum Elo-Rating-System, nicht nur vom Spielergebnis und den Ratings der Spieler beeinflusst, sondern zusätzlich auch noch von den Ratingabweichungen beider Spieler. Die Veränderung der RD wird beeinflusst sowohl von den Spielergebnissen (inklusive Ratings und RD), als auch von der Zeit, in der ein Spieler nicht antritt. Ein gespieltes Spiel verringert die RD immer, wohingegen vergangene Zeit, in der nicht gespielt wird, die RD immer erhöht [26]. Der Grund dafür ist, dass mit jedem gespielten Spiel Information über das Rating eines Spielers gewonnen wird, es also akkurater ist. Wenn hingegen Zeit ohne ein Match vergeht, so wird das Rating immer unsicherer. Der Algorithmus im Glicko-Rating-System sieht wie folgt aus:

1. Spielergebnisse über eine Rating Periode sammeln.
2. Ratingverteilung für jeden Spieler ermitteln.
3. Spielerrating anpassen (ähnlich dem Elo Algorithmus).
4. Ratingverteilung anpassen.

3.3.2 Berechnung

Da die Varianz der Leistungen in den Berechnungen des Glicko-Rating-Systems eine große Rolle spielen, sind die Gleichungen um einiges komplizierter und für die Spieler daher weniger gut nachvollziehbar als jene im Elo-Rating-System. Da der Glicko Algorithmus aber eine Erweiterung des Elo Algorithmus ist, ist trotzdem eine gewisse Ähnlichkeit zwischen den Funktionen erkennbar. Genau genommen ist nämlich der Elo Rating Algorithmus

nur eine besondere Form des Glicko Ratings und kann durch einsetzen bestimmter Parameter erreicht werden [11].

Vorbereitung

Wenn das Ende einer Rating Periode erreicht wurde, so muss zuerst die Ratingabweichung für jeden Spieler ermittelt werden, bevor die Ratings aktualisiert werden können. Wenn ein Spieler bisher unbewertet ist so schlägt Glickman vor sein Rating auf 1500 zu setzen und seine RD auf 350 [26]. Diese Werte sind natürlich von der jeweiligen verwendeten Skala eines Spieles abhängig und in diesem Fall auf die Schach Skala optimiert. Für alle anderen Spieler wird die jeweilige letzte Ratingabweichung verwendet, um daraus mit der Formel

$$RD = \min\left(\sqrt{RD_{old}^2 + v^2 t}, 350\right) \tag{3.5}$$

die neue RD zu berechnen, wobei sichergestellt wird, dass die RD niemals 350 – die RD eines unbewerteten Spielers – übersteigt. Bei der Berechnung der Ratingabweichung wird von der Varianz ausgegangen, die, wie in Abschnitt 3.1.1 festgehalten, das Quadrat der Standardabweichung ist (deswegen muss die Wurzel gezogen werden um RD zu erhalten). Es wird also die alte Varianz RD_{old}^2 addiert mit einer Varianzänderung $v^2 t$, welche für die, seit dem letzten Spiel vor Beginn der Rating Periode, vergangene Zeit angemessen ist. Wobei v^2 die Varianzänderung pro Zeiteinheit ist und t die Anzahl an Zeiteinheiten (= Rating Perioden). Wenn ein Spieler in der letzten Rating Periode teilgenommen hat, so wird $t = 1$, wodurch eine Varianzänderung stattfindet. Dies lässt sich dadurch begründen, dass davon ausgegangen wird, dass alle Spiele einer Rating Periode am Anfang der Periode gleichzeitig stattfinden und dadurch die restliche Periode als Zeit gewertet wird, in der nicht gespielt wurde. Um einen geeigneten Wert für v zu ermitteln kann die Frage gestellt werden, wie viel Zeit (gemessen in Rating Perioden) vergehen muss, bis das Rating eines Spielers so unsicher wird wie das Rating eines bisher unbewerteten Spielers. Wenn dieser Wert, sowie die RD eines durchschnittlichen Spielers eingesetzt wird, kann man die Gleichung nach v lösen und so einen akzeptablen Wert dafür erhalten.

Rating Aktualisierung

Um den Rating Algorithmus anzuwenden, werden alle Spiele, die in einer Rating Periode gespielt werden, so behandelt als ob sie gleichzeitig, am Beginn der Rating Periode stattgefunden hätten. Dadurch ist es nicht relevant in welcher Reihenfolge sie stattfinden, oder wie viel Zeit zwischen einzelnen Spielen verstrichen ist. Es werden alle Ergebnisse auf einmal einberechnet und die verstrichene Zeit wird erst bei der nächsten Rating Periode in die RD einkalkuliert (siehe Formel 3.5). Das Glicko-Rating-System erzielt dann

die besten Ergebnisse, wenn in einer Rating Periode durchschnittlich 5–10 Spiele pro Person stattfinden [26]. Um das neue Rating zu erhalten wird

$$r' = r + \frac{q}{1/RD^2 + 1/d^2} \cdot \sum_{j=1}^{n} g(RD_j) \cdot (w_j - \mathrm{E}(w|D_j, RD_j)) \tag{3.6}$$

als Formel für den Mittelwert benötigt, sowie

$$RD' = \sqrt{\left(\frac{1}{RD^2} + \frac{1}{d^2}\right)^{-1}} \tag{3.7}$$

als Formel für die Standardabweichung, wobei die einzelnen Parameter durch folgende Funktionen gegeben sind:

$$q = \frac{\ln 10}{2C} = \frac{\ln 10}{400} = 0{,}0057565, \tag{3.8}$$

$$g(RD) = \frac{1}{\sqrt{1 + 3q^2 \cdot (RD^2)/\pi^2}}, \tag{3.9}$$

$$\mathrm{E}(w|D_j, RD_j) = \frac{1}{1 + 10^{-g(RD_j)\cdot(D_j)/2C}}, \tag{3.10}$$

$$d^2 = \left(q^2 \cdot \sum_{j=1}^{n} g(RD_j)^2 \cdot \mathrm{E}(w|D_j, RD_j) \cdot (1 - \mathrm{E}(w|D_j, RD_j))\right)^{-1}. \tag{3.11}$$

Es ist erkennbar, dass die Rating Aktualisierungs Gleichung in 3.6 sehr ähnlich zu jener in (3.3) ist. Wobei der Ratingkoeffizient K, welcher im Elo-Rating-System ein konstanter Faktor ist, durch die Gleichung

$$K = \frac{q}{1/RD^2 + 1/d^2}$$

ersetzt wurde. Auch die Wahrscheinlichkeitsfunktion (3.10) ist in der Grundstruktur die selbe wie jene (3.1) im Elo Algorithmus.

Das Glicko-Rating-System nützt die Gewinnwahrscheinlichkeiten, um sowohl die Unsicherheit des Spieler Ratings und des Gegner Ratings als auch die Schwankungen bedingt durch die Spielergebnisse (repräsentiert durch d^2) in Rating und Ratingabweichung einzubinden [11]. Die Spielergebnisse werden in Relation zur Genauigkeit des eigenen Ratings abgewägt, wodurch der Ratingkoeffizient (sowie die Berechnung der neuen Ratingabweichung) davon abhängt wie exakt das eigene Rating ist. Wenn das eigene Rating unsicher ist, so haben die Spielergebnisse einen möglichen wesentlichen Effekt auf das Rating (bzw. die Ratingabweichung) was durch einen hohen Ratingkoeffizienten augedrückt wird. Ist das eigene Rating präzise gemessen, so sollen die Spielergebnisse nur einen geringen Effekt auf das neue Rating haben und

der Koeffizient ist gering. Durch Formel 3.9 werden die unterschiedlichen Unsicherheiten der Gegenspieler wahrgenommen. Wenn die RD eines Gegner hoch ist, so wird $g(RD_j)$ viel kleiner als 1, wodurch der Beitrag dieses Gegners relativ gering wird [11]. Bei der Berechnung der RD' eines Spielers sollte darauf geachtet werden, dass der Wert nicht kleiner wird als 30, damit auch das Rating von Spielern die häufig antreten sich entsprechend verändern kann, falls sie ihre Fähigkeiten verbessern.

3.3.3 Beispiele

Um die Berechnung der Ratings am Ende einer Ratingperiode zu demonstrieren, wird hier angenommen, dass ein Spieler, der ein Rating von 1500 und eine Ratingabweichung von 200 besitzt, gegen drei Gegenspieler antritt, die in folgender Tabelle angegegben sind [26]:

Spieler	Rating	RD	Ergebnis
Spieler 1	1400	30	1
Spieler 2	1550	100	0
Spieler 3	1700	300	0

Für jeden Spieler müssen nun $g(RD)$ und $\mathrm{E}(w|D_j, RD_j)$ bestimmt werden. Die Ergebnisse

Spieler	$g(RD)$	$\mathrm{E}(w|D_j, RD_j)$
Spieler 1	0,9955	0,639
Spieler 2	0,9531	0,432
Spieler 3	0,7242	0,303

können dann in die Berechnung von d^2 einfließen

$$\begin{aligned} d^2 = \Big((0{,}0057565)^2 \cdot [(0{,}9955)^2 \cdot (0{,}639) \cdot (1 - 0{,}639) \\ +(0{,}9531)^2 \cdot (0{,}432) \cdot (1 - 0{,}432) \\ +(0{,}7242)^2 \cdot (0{,}303) \cdot (1 - 0{,}303)]\Big)^{-1} \\ = 53670{,}85 = 231,67^2. \end{aligned}$$

Zu guter Letzt wird das neue Rating aus den nun bekannten Variablen bestimmt und es ergibt sich

$$\begin{aligned} r' = 1500 + \frac{0{,}0057565}{1/200^2 + 1/231,67^2} \cdot [0{,}9955(1 - 0{,}639) \\ +0{,}9531(0 - 0{,}432) \\ +0{,}7242(0 - 0{,}303)] \\ = 1500 + 131.9 \cdot (-0,272) = 1500 - 36 = 1464. \end{aligned}$$

Die neue Ratingabweichung für den Spieler beträgt nun

$$RD' = \sqrt{\left(\frac{1}{200^2} + \frac{1}{231,67^2}\right)^{-1}} = \sqrt{22918.9} = 151{,}4.$$

3.4 Paired Comparison Models

Den Hintergrund der Elo basierten Rating Systeme bilden Paired Comparison Models, die auf der Methode der paarweisen Vergleiche beruhen. Paarweise Vergleiche sind die Basis aller wissenschaftlicher Messungen. Was auch gemessen wird, sei es Gewicht oder Temperatur, es werden quasi zwei Dinge miteinander verglichen, von denen eines als der Standard betrachtet wird und das andere in Abhängigkeit davon einen Wert zugewiesen bekommt.

Paired Comparison Models gehen noch einen Schritt weiter indem sie auch Dinge vergleichen, von denen keines wirklich als Standard gelten kann und die Wertzuweisung daher erschwert ist (wie zum Beispiel kognitive und sportliche Fähigkeiten), wenn es also unmöglich oder umständlich ist wirkliche Messungen durchzuführen. Um trotzdem Werte vergeben zu können werden die Objekte oder Wettkämpfer darauf verglichen, welcher der beiden als besser wahrgenommen wird (durch Modifikationen des Modells ist es auch möglich ein Unentschieden abzubilden) [4]. Durch unzählige Vergleiche lassen sich dann Bewertungen festlegen und die Objekte können im besten Fall auf einer Skala abgebildet werden. Das eigentliche Ziel ist es, Objekte zu reihen und die Ergebnisse zukünftiger Vergleiche – auch zwischen Objekten die noch nicht miteinander gepaart wurden – vorhersagen zu können. Wenn jedes Objekt mit jedem anderen verglichen wird, so wird das ein ausgewogenes paarvergleichs Experiment genannt, bei Spielen oder im Sport entspricht das einem Round Robin Turnier [4], was aber ab einer gewissen Anzahl an Objekten kaum noch durchführbar ist. Verwendet werden Paired Comparison Models zum Beispiel zur Entscheidungsfindung, zur Analyse von Präferenztests und zur Fertigkeitsmessung bei Wettkämpfen(kompetitiven Sportarten) [4].

Manchmal ist es möglich mehrere Objekte zur selben Zeit zu vergleichen, wodurch ein einfaches Ranking von allen Objekten bevorzugbar sein kann. Allerdings, wenn die Unterschiede zwischen Objekten sehr gering sind, ist es wünschenswert zwei Objekte möglichst frei von der Beinflussung anderer vorhandener Objekte vergleichen zu können. Ranking ist nur dann einfach und schnell, wenn die Unterschiede zwischen Objekten sehr auffällig sind, ansonsten ist es nötig viele paarweise Vergleiche zwischen provisorischen Nachbarn auszuführen bevor eine sinnvolle Reihenfolge gebildet ist. Unter diesen Umständen ist Ranking unausführbar, wenn viele Objekte vorhanden sind, außerdem ist es oft nicht möglich ein gänzlich zufriedenstellendes Ranking zu erreichen[4](siehe Intransitivität in Abschnitt 3.5.3).

Das Elo-Rating-System entspricht einem linearen Paired Comparison Model, da es die Spielstärke als weitgehend stabil annimmt sobald sie einmal etabliert ist und der Rating Algorithmus nur dazu dient einen möglichst passenden Wert dafür zu finden. Es wird davon ausgegenangen, dass die Präferenzwahrscheinlichkeiten direkt aus der Differenz der Leistungen abgeleitet werden kann. Wenn es keine zusätzlichen Parameter gibt, kann die Wahrscheinlichkeitsfunktion durch die logistische Verteilungsfunktion (Bradley-Terry Modell) oder durch die Verteilungsfunktion einer Gauss Distribution (Thurstone-Mosteller Modell) dargestellt werden [15]. Welches dieser beiden Modelle zu wählen ist, also welches die relative Wirklichkeit besser beschreibt, ist für jedes Experiment nur durch eine große Menge an Daten ersichtlich.

Das Glicko-Rating-System hingegen entspricht einem dynamischen Paired Comparison Model, welches besser dazu geeignet ist Schwankungen in der Spielstärke abzubilden, wenn zum Beispiel neue Taktiken erlernt werden, oder durch das Verlernen von Taktiken nach längerer Spielpause [11]. Erreicht wird das durch die Einführung von Parametern die zeitabhängig sind, wie es im Glicko Algorithmus die Ratingabweichung ist.

3.5 Probleme

Obwohl die meisten Spieler eine Möglichkeit suchen ihre Spielstärke, ihre eigentliche Fähigkeit, zu messen, so ist das Elo-Rating-System und alle darauf basierenden Algorithmen nur eine Methode um die Leistung zu messen [1]. Der Unterschied mag nicht sofort erkennbar sein, aber er ist wichtig zu verstehen. Eine Person kann unter Umständen eine sehr hohe theoretische Fähigkeit in einem Spiel oder Sport besitzen, in den Partien dann aber nicht die volle Leistung erbringen aufgrund von Nervosität oder Ähnlichem. Es ist einem Rating System nicht möglich die theoretische Stärke eines Spielers zu ermitteln, es kann nur die erbrachte Leistung bewertet werden und auch diese nur in Abhängigkeit zu der Leistung des Gegners. Daraus ergibt sich auch der nächste wichtige Punkt der beachtet werden muss. Ratings sind nie absolut, es sind relative Werte und nur die Differenz zwischen einzelnen Ratings ist wirklich relevant. Die Differenz erlaubt es, die (Gewinn)Erwartungen für jedes Paar an Spielern zu ermitteln, was der Grund ist warum sie für Matchmaking Algorithmen trotzdem gut verwendbar sind.

3.5.1 Unbewertete Spieler

Wenn neue Personen beginnen an einem Spiel teilzunehmen, so sind sie anfangs unbewertet, also wird ihnen ein beliebiger Wert (meist der Mittelwert der Ratingskala) als Rating gegeben, der aber in keinster Weise der tatsächlichen Spielstärke entsprechen muss. Dadurch kann es dazu kommen, dass neue Spieler eigentlich viel besser (oder schlechter) sind als ihr Rating vermuten

lässt, was wieder zur Folge haben kann, dass sie das Rating von etablierten Spielern ungewollt stark beeinflussen. Um dem entgegen zu wirken, sollten die Ratings von Spielern, die weniger als 30 Spiele gegen etablierte Spieler absolviert haben, als vorläufig gekennzeichnet werden [7]. Die in Anschnitt 3.2.2 angesprochene Performance Rating Formel 3.4 kann verwendet werden, um aus einer Reihe von Testspielen ein besseres vorläufiges Rating zu bilden, das der eigentlichen Leistung eher entspricht als ein zufälliges Rating.

Wenn ein etablierter Spieler gegen einen mit vorläufigem Rating spielt, so werden nicht beide Ratings in einem Zug aktualisiert, sondern das unsichere Rating zuerst, und erst mit dem angepassten Rating wird auch das Rating des etablierten Spielers aktualisiert [7]. Außerdem ist es ratsam, wenn der Ratingkoeffizient bei etablierten Spieler in solchen Fällen geringer als normal ist, damit sein Rating sich nicht zu sehr verändert. Der Ratingkoeffizient bei einem unsicheren Spieler sollte hingegen sehr hoch sein, bei 32 oder 50 [7]. Im Glicko System passt sich der Ratingkoeffizient automatisch an, da die Unsicherheit eines Ratings durch die Ratingabweichung gegeben ist und der Koeffizient dadurch beeinflusst wird [11].

Ein anderes Problem tritt auf, wenn ein Spiel eine komplett neue Spielerbasis erhält und die Ratings erst gefunden werden müssen. In so einem Fall ist es wichtig, dass alle Spieler gleichmäßig gegen einander spielen ohne Subgruppen zu isolieren, was allerdings dem Wunsch nach ausgeglichenem Matchmaking entgegenwirkt. Es muss eine Möglichkeit gefunden werden, um Ratings bilden zu können, ohne in Gefahr zu laufen Spieler zu verlieren (durch nicht vorhandens Matchmaking). Wenn Spielergebnisse von zumindest einem Teil der Spieler vorhanden sind (bei Onlinespielen zum Beispiel aus der Beta-Phase), so können daraus Ratings ermittelt und die neuen Spieler in Abhängigkeit davon in die Spielerbasis eingegliedert werden. Eine andere Möglichkeit ist, Ratings in einem speziellen Rating Turnier zu ermitteln (im besten Fall ein Round Robin Turnier – das heißt jeder spielt gegen jeden), was auch weniger (zeit)aufwändig ist als jeden Spieler einem normalen Rating Prozess zu unterziehen [7].

3.5.2 Punkte Inflation und Deflation

Integrität der Ratings war für Elo immer ein wichtiger Punkt, den das Rating System zu gewährleisten hatte. Das heißt, dass eine Punktezahl auch über einen längeren Zeitraum hinweg immer eine bestimmte Leistung repräsentieren sollte [7]. Diese Integrität wird aber bedroht durch das automatische Auftreten von Punkte Inflation und Deflation.

Der Hauptmechanismus des Elo Algorithmus ist die Regel, dass, wenn ein Spieler Punkte gewinnt, so verliert ein anderer Spieler Punkte. Wenn neue Spieler in den Rating-Pool gelangen und ihre Fähigkeiten stark verbessern, so nehmen sie sehr viele Punkte aus dem Rating-Pool. Sofern sie diese Punkte nicht wieder verlieren bevor sie aus dem aktiven Spielgeschehen aus-

scheiden, führt dies zu einer allgemeinen Punktedeflation. Im Gegensatz dazu rührt Punkte Inflation daher, dass Spieler ein zu hohes Einstiegsrating bekommen, was bedeutet, dass sie überbewertet sind und Punkte an andere Spieler verschenken.

Um Deflation und Inflation unter Kontrolle zu halten, sollten mehrere Dinge beachtet werden. Ein wichtiger Punkt ist eine angemessene Behandlung von unbewerteten und vorläufig bewerteten Spielern, wie sie im vorherigen Abschnitt (3.5.1) beschrieben ist. Soll heißen, als erstes müssen die neuen Ratings der unbewerteten Spieler berechnet werden, danach die Ratings der vorläufigen Spieler und dann erst die Ratings aller anderen, um ungerechtfertigten Punkteverlust oder -gewinn zu verhindern [7]. Auch der Ratingkoeffizient K kann verwendet werden, um sowohl Deflation als auch Inflation entgegenzuwirken. Eine Möglichkeit ist einen hohen Wert für neuere Spieler zu verwenden, der dann mit der Zeit geringer wird, wenn sein Rating etabliert ist (im Glicko System passiert dies automatisch). Alternativ kann der Koeffizient an das Rating der Spieler gebunden werden, indem Personen mit geringen Ratings einen hohen K Wert haben und Personen mit hohen Ratings einen geringen [7].

Im Gegensatz zur Deflation kann eine leichte Punkte Inflation unter Umständen gewünscht sein. Mit einer geringen Inflationsrate kann nämlich die Weiterentwicklung der allgemeinen Spielstärke im Laufe der Zeit abgebildet werden. Durch den normalen Rating Prozess ist dies nicht möglich, da nur die Unterschiede der Leistungen wirklich berechenbar sind. Wenn aber alle Spieler um den gleichen Wert besser werden, so sind die Differenzen immer noch gleich und diese allgemeine Leistungssteigerung ist nicht erkennbar.

3.5.3 Intransitivität von Wahrscheinlichkeitsrelationen

Ein Problem mit dem sich ein Entwickler eines Rating Systemes unweigerlich konfrontiert sieht ist, dass Wahrscheinlichkeitsrelationen nicht notwendigerweise transitiv sind. Wenn Spieler A besser gewertet ist als Spieler B und Spieler B besser als Spieler C, so möchte man daraus schlussfolgern, dass Spieler A auch besser ist als Spieler C, was aber nicht zwingend so ist (bestes Beispiel Schere-Stein-Papier) [23]. Da Transitivität aber eine notwendige Vorraussetzung für ein sinnvolles Rating, und vor allem für eine sinnvolle Verwendung von Skalen, ist, muss sie als grundlegende Annahme des Modells gelten. Ansonsten ist es nicht möglich aus Ratingdifferenzen Gewinnwahrscheinlichkeiten abzuleiten.

Kapitel 4

API Design

Einer der größten Vorteile von objektorientierter Programmierung ist die Möglichkeit Software in unterschiedlichen Programmen wiederzuverwenden. Um von einem Programm auf ein solches Softwaresystem zugreifen zu können, werden Programmierschnittstellen, sogenannte APIs[1] benötigt. Obwohl es unzählige Programmierbücher gibt, setzen sich nur einige wenige konkret mit API Design auseinander und sind meist auch auf eine bestimmte Programmiersprache beschränkt. In diesem Kapitel sollen substantielle Programmiersprachen übergreifende Regeln aufgestellt werden um zu veranschaulichen, was gutes API Design ist, was dabei beachtet werden sollte und warum es überhaupt wichtig ist.

Die zuvor erwähnten wiederverwendbaren Codebausteine lassen sich grob in zwei Arten unterteilen nämlich Frameworks und Libraries. Anfangs ist es oft schwer den Unterschied zu verstehen und auch etablierte Programmierer scheinen sich darüber nicht immer einig zu sein (.NET[2] bezeichnet sich zum Beispiel selbst als Framework, obwohl es eigentlich eher eine Library ist). Manchmal wird auch die Meinung vertreten, dass Library der Überbegriff für wiederverwendbare Softwaresysteme ist, worauf aufgebaut es dann noch Frameworks und Toolkits gibt. Aufgrund fehlender wirklich finaler Definitionen kann hier nur wiedergegeben werden, was die gängigste Differenzierung dieser Begriffe ist.

Library: Eine Library ist eine Sammlung von meist vorkompilierten Klassen und Funktionen, die unterschiedliche Funktionalitäten zur Verfügung stellen.

Toolkit: Ein Toolkit ist eine spezielle Form der Libraries, die eine abgekapselte zusammenhängende Funktionalität zur Verfügung stellt. Sie sind keine eigenständig lauffähigen Programme sondern Hilfsmodule die von Programmierern von ihrem eigenen Code aus aufgerufen werden können. Die Kontrolle darüber wann welcher Code aufgerufen wird liegt

[1] API steht für Application Programming Interface

[2] http://www.microsoft.com/germany/net/net-framework.aspx

hier vollständig beim Programmierer [9] Oft wird diese Definition für Libraries an sich verwendet.

Framework: Ein Framework ist ein Programmiergerüst, das die Architektur, das Design eines Programmes (eines Programmteils) vorgibt. Der Programmierer, der ein Framework verwendet, liefert konkrete Implementierungen und bindet diese über die vorhandenen Schnittstellen ans Framework an. Das Framework definiert den Kontrollfluss der Anwendung und bestimmt wann es den Code des Programmierers aufruft [9]. Dieses Prinzip ist die bedeutenste Charakteristik eines Frameworks und nennt sich Inversion of Control (IoC) [16].

Die Vorteile davon, Software so zu gestalten dass sie wiederverwendbar ist, liegen auf der Hand. Die gleiche Funktionalität kann in verschiedenen Programmen verwendet werden, im besten Fall ohne große Änderungen vornehmen zu müssen und vor allem ohne dass sich jeder Programmierer immer wieder über die selbe Thematik den Kopf zerbrechen muss, um die beste Lösung zu finden. Dadurch kann die Implementierungszeit stark verringert werden. Auch kann davon ausgegangen werden, dass wiederverwendbare Codefragmente weniger fehleranfällig sind, weil sie mit hoher Wahrscheinlichkeit schon vielfach getestet und verwendet wurden. Genau an diesen Vorteilen ist aber auch zu sehen, wie wichtig es ist, dass die zur Verfügung gestellte Software auch extra dafür entworfen wurde wiederverwendet zu werden. Ansonsten stellt sich schnell das Problem ein, dass das Einarbeiten in einen fremden Code und das Modifizieren desselben sowie das Korrigieren von Fehlern fast genau so lang, wenn nicht sogar länger, dauert als den Code selbst zu schreiben.

4.1 Motivation

Als API bezeichnet man üblicherweise die Schnittstellen einer Library oder eines Frameworks durch die ein Programmierer auf die Funktionalität oder die Komponenten dieser Software zugreifen kann. Bei genauerer Betrachtung ist festzustellen, dass eigentlich jede Klasse und jedes Modul Schnittstellen zur Außenwelt hat und daher auch hier schon beachtet werden muss wie diese gestaltet werden. Daher sollte jeder Programmierer sich Gedanken über API Desing machen [20]. Solange nur für sich selbst programmiert wird, mag das noch nicht so wichtig scheinen. Sobald aber nur eine Klasse auch von jemand anderem benützt wird, kann eine Methode oder einen Variablennamen nicht mehr ohne weiteres geändert werden, da das dazu führen würde, dass auch die Personen die diese Klasse verwenden ihren Code anpassen müssen. Es gibt etliche Gründe warum gutes Schnittstellen Design wichtig ist und man darauf achten sollte, die wesentlichsten Beweggründe aber sind folgende:

Wiederverwendbarkeit: Wiederverwendbarkeit ist der Grund warum API Design überhaupt benötigt wird. Wenn eine Applikation nur für einen

Endnutzer entworfen wird und dann nie wieder weiterentwickelt sondern in der bestehenden Art mehrere Jahre verwendet und dann für ein neues Programm weggeworfen wird, dann ist keine API Qualität für den Code notwendig [20]. Sobald aber ein Modul von anderen Programmierern benutzt werden soll ist es wichtig eine API zu entwerfen.

Wartbarkeit/Erweiterbarkeit: Wartbarkeit ist ein Kriterium unter welchem festgestellt werden kann, wie viel Energie und Zeitaufwand und mit welchem Erfolg Änderungen an einem System durchgeführt werden können. Dazu ist es wichtig, dass Schnittstellen gut spezifiziert und dokumentiert sind, damit auch applikationsfremde Personen einen möglichst geringen Wartungsaufwand betreiben müssen. Obwohl viele Entwickler es nicht zugeben wollen, so kann man trotzdem immer davon ausgehen, dass Schnittstellen niemals final sind [20]. Die Anforderungen an eine Software ändern sich ständig. Außerdem ist davon auszugehen, dass sich in jedem Programm zumindest ein Bug[3] befindet und obwohl keine API in ihrer ersten Version perfekt ist, muss dafür gesorgt werden, dass sie einfach, schnell und möglichst fehlerfrei erweitert werden kann. Zumindest aber muss vermieden werden Fehler einzubauen, die die Verbesserung der API in Zukunft behindern könnten.

Rückwertskompatibilität: Neue Versionen einer API bringen oftmals Probleme für die Entwickler mit sich. Sie werden eingebaut um Zugang zu neuer Funktionalität zu erhalten und durch die Änderungen kann es passieren, dass der alte Code nicht mehr compiliert, zu Fehlern führt oder andere Ergebnisse hervorbringt. Dies ist nicht nur aufwändig sondern auch frustrierend für die Entwickler, die ihren gesamten Code überarbeiten müssen. Deswegen ist es wichtig in wiederverwendbarer Software Rückwärtskompatibilität zu erhalten, damit der Code einer Applikation, der auf einer alten Version einer Library basiert, auch dann noch verwendbar ist, wenn für eine andere Componente der Applikation eine neuere Version der Library benötigt wird. Rückwärtskompatibilität zu erhalten ist nicht immer möglich, manchmal müssen Kompatibilitätsbrüche hingenommen werden, damit Fehler und Probleme behoben werden können. Wenn man sich aber von vornherein Gedanken über das API Design vor allem unter dem Blickpunkt der Erweiterbarkeit macht, so sollten die Kompatibilitätsbrüche und deren Auswirkungen minimal sein. Wenn eine API von Anfang an darauf ausgelegt ist, erweiterbar zu sein, so können alle anderen Fehler in einer neuen Version behoben werden ohne zu Problemen zu führen [20].

Geringe Fehleranfälligkeit: Um zu vermeiden, dass User mit dem Code den sie schreiben die Applikation zerstören, Bugs auslösen oder die ihnen zur Verfügung gestellten Möglichkeiten einfach nur nicht sinnge-

[3] Bug bezeichnet einen Programmfehler in einer Software.

mäß verwenden, muss darauf geachtet werden, dass die API möglichst wenige Fehler zulässt. Ein einfacher Grundatz lautet: Wenn es eine Möglichkeit gibt etwas falsch zu verwenden, dann werden die Benutzer vermutlich genau das tun [20]. Dies gilt es zu vermeiden.

Wenn eine API gut entworfen ist, dann muss sich der Benutzer keine Gedanken darüber machen, wie die zugrundeliegenden Libraries eigentlich genau funktionieren. Er kann sie einfach verwenden und davon ausgehen, dass sie es tun. Dadurch kann er sich vollständig auf die Implementierung der eigenen Applikation konzentrieren. Wenn der Benutzer allerdings mehr über die Funktionsweise eines Systems erfahren möchte so sollte er von der API nicht dabei behindert oder verwirrt werden. Diesen Umstand nennt Jaroslav Tulach „Selective Cluelessness“ [20].

Werden APIs hingegen nicht vorrausschauend geplant, so kann es sein, dass Rückwärtskompatibilität einbüßt wird, ein unaufhörlicher Strom an Kritik und Fragen hingenommen und viel Zeit und Geld in die Wartung und das Ausbessern von Fehlern gesteckt werden muss, die von vornherein hätten vermieden werden können. So bald eine API öffentlich ist, ist sie kaum noch veränderbar, da einige darauf aufbauende Programme womöglich nicht mehr funktionieren, wenn Schnittstellen entfernt oder verändert werden. Daher bleibt dem Entwickler nur eine einzige Chance um es richtig zu machen. Design Entscheidungen müssen also wohl überlegt sein. Es ist viel Arbeit, eine gute, beständige API zu entwerfen, aber langfristig spart es auch sehr viel Arbeit und Frustration [20].

4.2 Qualitätskriterien

Gutes API Design ermöglicht es den Benutzern unterschiedliche Module schnell und einfach miteinander zu verknüpfen ohne ihre Zeit damit zu verschwenden den Code anderer verstehen zu wollen, zu debuggen und Fehler auszubessern. Um dies zu gewährleisten lassen sich Regeln und Qualitätskriterien bilden, die besagen, dass APIs folgendes sein müssen:

Verständlich: Wer eine API verwenden will muss sie auch verstehen können. Die Verwendung von geläufigen Ausdrücken, Bezeichnungen und Klassen reduzieren die Lernkurve und machen die Schnittstellen verständlicher. Wenn dies nicht oder nur geringfügig möglich ist, so sollten zumindest genügend Beispiele zur Verfügung gestellt werden, die von den Benutzern adaptiert und entsprechend den eigenen Anforderungen verändern werden können. Dadurch wird die Wahrscheinlichkeit größer, dass die Benutzer mit der Zeit verstehen wie eine bestimmte API funktioniert. Je neuer die Konzepte einer API sind, desto größer ist der Nutzen von ausführlichen Beispielen [20].

Konsistent: Wenn bestimmte Konzepte von einer API eingeführt werden, ist es wichtig, dass diese konsistent über die gesamte API verwendet

werden, damit der Benutzer sich möglichst wenige Sonderfälle und Ausnahmen merken muss. Komplexe Systeme sollten ein zusammenhängendes und koordiniertes Design aufweisen [24].

Einfach: Einfache und gebräuchliche Aufgaben sollten einfach zu implementieren sein. Dafür sollten Anwendungsfälle (sogenannte Use-Cases) überlegt werden, um sicher zu gehen, dass die wichtigen Aufgaben einfach zu lösen sind. Im besten Fall können gute APIs, auch ohne die Dokumentation lesen zu müssen, verwendet werden [20].

Lesbar: Der Code, der einer guten API zu Grunde liegt, sollte einfach zu lesen und zu verstehen sein. Der Code einer Applikation wird nur einmal geschrieben aber immer wieder von unterschiedlichen Entwicklern gelesen werden. Daher ist es wichtig, dass schnell ersichtlich ist, was bestimmte Anweisungen genau machen. Lesbarer Code ist einfacher zu dokumentieren und zu warten, außerdem beinhaltet er weniger Fehler, weil diese mehr auffallen [24].

Reduziert: Weniger ist mehr. Nur die wirklich benötigte Funktionalität sollte offen verfügbar sein. Wenn die Benutzer bestimmte Funktionen vermissen, können sie in einer späteren Iteration hinzugefügt werden. Je mehr von der internen Programmierung offen gelegt ist, um so weniger Handlungsfreiraum bleibt, um Änderungen zu machen. Es ist immer einfacher etwas hinzuzufügen als etwas aus einer API zu entfernen. Nur wenn es für eine Klasse oder eine Methode einen Use-Case gibt sollte sie auch implementiert sein [20].

Erweiterbar: Wie schon erwähnt können sich Anforderungen immer ändern. Eine API muss sich weiterentwicklen können ohne sich dabei selbst im Weg zu stehen. Es sollte von Anfang an darauf geachtet werden, dass Binärkompatibilität[4] zu Verfügung gestellt wird [20].

Sollen die Charakteristika von gutem API Design in nur einem Satz zusammenfassen wollte, so wäre es vermutlich jener von Scott Meyers [18]:

> "Make interfaces easy to use correctly and hard to use incorrectly."

Ein Satz der von ihm geprägt, aber auf ähnliche Art auch von einigen anderen Entwicklern publiziert wurde. Dieser Grundsatz ist nicht nur für APIs gültig sondern ist eine wichtige Regel für jegliche Art von Interface Design, ob es sich nun um User Interfaces, Programmierschnittstellen oder Ähnliches handelt. Auch wenn die Definition von gutem API Design grundsätzlich ein eher subjektives Thema ist, so können die meisten wichtigen Regeln in irgendeiner Art darauf zurückgeführt oder heruntergebrochen werden. Gute Schnittstellen müssen es dem User erleichtern richtigen Code zu schreiben

[4] Binärkompatibilität heißt, dass bereits kompilierte Programme auch bei neuen Versionen der zugrundeliegenden Software immer noch ausführbar sind.

und es möglichst verhindern, dass der User unbeabsichtigt falschen Code und Fehler produziert [24].

4.3 Designrichtlinien

Nachdem geklärt wurde warum API Design wichtig ist und welche Eigenschaften das fertige Produkt aufweisen muss, stellt sich nun die Frage wie diese Dinge zu erreichen sind. Es gibt einige Richtlinien die zu befolgen sind und die einem den Design Prozess erleichtern. Dabei kann unterschieden werden zwischen Implementierungs Regeln, also Regeln die helfen den Code besser zu verstehen, und Design Regeln, also Regeln die helfen, den Code besser aufzubauen und nützliche Abstraktionen zu erstellen.

4.3.1 Design Regeln

Vorbereitung

Der erste Schritt bei der Erstellung einer Software ist das Ermitteln von Anforderungen, die an diese Applikation gestellt werden. Dabei ist es sehr wichtig, den kompletten Problemraum aufzubohren. Wobei man sich wegbewegen sollte von festgefahrenem Denken und sich nicht von der erstbesten Lösung ablenken lassen sollte, da oft eine bessere Lösung gefunden werden kann.

Um diesen Prozess zu unterstützen ist es fast unumgänglich Use Case Szenarios zu erstellen, also zu versuchen herauszufinden wer genau welche Aktionen mit der geplanten Applikation ausführen will und wie der Ablauf dieser Aktionen aussehen kann. Dadurch wird sicher gestellt, dass die Software wirklich die Dinge bietet, die der User damit (wahrscheinlich) machen möchte und diese Dinge vor allem einfach umzusetzen sind. Die Anwendungsfälle lassen sich aus den Anforderungen ableiten [20].

Bevor mit der Implementierung begonnen und vielleicht sogar noch bevor bewusst mit dem Designprozess der API angefangen wird, sollten ein paar Code Schnippsel geschrieben werden, die die Verwendung der API in einzelnen Anwendungsfällen zeigt. Dadurch kann sichergestellt werden, dass sie einfach zu benutzen und zu lesen ist. Des weiteren kann dadurch verhindert werden, dass Implementierungen geschrieben werden, die im nachhinein vielleicht komplett überarbeitet werden müssen weil auf einmal unangenehme Überraschungen auftauchen [25]. So entsteht eine erste abstrakte Form der Schnittstelle von der aus der Designprozess gestartet werden kann. Dem folgend kann anfangen werden, die API und ihre Semantik zu definieren. Wohlgemerkt findet auch dieser Prozess statt, bevor mit der Implementierung begonnen wurde.

Design

Die entworfene Schnittstelle muss vor allem eines können, und zwar den Benutzern helfen ihre Aufgaben zu erfüllen. Das Mantra des API Designers ist wie schon erwähnt: „Easy to use, hard to misuse“. Wenn die API dementsprechend entworfen ist, so wird sie automatisch richtig verwendet, weil es der schnellste und naheliegenste Weg ist. Auch muss der Designer versuchen alle möglichen Fehler vorherzusehen und diese so gut es geht verhindern. Ein Beispiel für häufige Fehlerquellen ist wenn bestimmte Methoden in einer festgelegten Reihenfolge aufgerufen werden müssen oder in einem Methodenaufruf mehrere Parameter den selben Typ haben. Dadurch kann es schnell passieren, dass Parameter in der falschen Reihenfolge übergeben werden. Dies kann verhindert werden indem neue eindeutige Typen für den Gebrauch im Interface definiert werden [18]. Wenn zum Beispiel eine Klasse `Date` mit dem Konstruktor `public Date(int day, int month, int year);` vorhanden ist, so ist es für den Benutzer sehr leicht die Parameter in der falschen Reihenfolge zu übergeben, vor allem weil in unterschiedlichen Kulturkreisen das Datum unterschiedlich angeschrieben wird. Werden einfache Wrapperklassen eingeführt, so ist eine fehlerhafte Übergabe kaum noch möglich, weil dann schon beim Compilieren der Fehler erkannt wird und er noch während des Programmiervorgangs ausgebessert werden kann. Noch besser ist es, den Konstruktor zu überladen, also mehrere Konstruktoren mit unterschiedlicher Parameterreihenfolge zur Verfügung stellt, wodurch die Verwendung noch einfacher für den Benutzer wird.

```
1 struct Day{ int d; }
2 struct Month{ int m; }
3 struct Year{ int y; }
4
5 public class Date
6 {
7   public Date(Day d, Month m, Year y);
8   public Date(Month m, Day d, Year y);
9   public Date(Year y, Month m, Day d);
10  ...
11 }
```

Es sollen möglichst viele unterschiedliche Formen des korrekten Benutzens unterstützt werden und gleichzeitig so viele falsche Formen wie möglich verhindert, wobei beide Bestreben gleich wichtig sind. Eines ohne das andere ist nicht genug. Um das zu gewährleisten muss sich der API Designer die Fähigkeit aneignen mögliches Benutzerverhalten vorherzusehen [18]. Dabei ist es genau so hilfreich, Use Cases im Vorhinein durchzuspielen, wie es wichtig ist, die Designergebnisse im nachhinein zu testen und das Interface wenn nötig wieder zu überarbeiten.

Testen

Wenn die API entworfen und implementiert ist, sollten komplette Beispiele geschrieben werden, die die Schnittstelle verwenden. Am besten werden alle Anwendungsfälle durchprogrammiert und auch andere Personen gebeten eigene Anwendungen mit der API zu implementieren. Es muss so viel Feedback und Kritik wie möglich gesammelt werden und zwar bevor das Interface veröffentlicht wird. Man kann auch größer angelegte Usability Studien durchführen. Nur so kann gewährleistet werden, dass die Schnittstellen auch wirklich der Spezifikation entsprechen und die Dinge zur Verfügung stellen, für die sie konzipiert wurden. Es ist quasi unmöglich eine gute API zu schreiben ohne sie ausführlich zu testen [20].

Um Erweiterbarkeit zu gewährleisten sollten für jede Klasse, die als Superklasse verwendet werden kann, drei Subklassen für möglichst unterschiedliche Anwendungsfälle implementiert werden. Dadurch kann sichergestellt werden, dass eine große Bandbreite an Benutzeranforderungen abgedeckt ist und die API gut genug konzipiert ist, um problemlos erweitert zu werden [24].

Wenn es Zweifel an einem bestimmten Teil der API oder einer Funktionalität gibt, so ist es am sichersten, diese erst einmal weg zu lassen oder nur intern zur Verfügung zu stellen. Sie kann dann zu einem späteren Zeitpunkt, falls benötigt, noch eingearbeitet oder veröffentlicht werden. Bevor neue Funktionalität eingebaut wird, muss sichergestellt werden, dass diese auch wirklich benötigt wird. Eine gute Faustregel hierfür ist, dass gewartet werden sollte, bis zumindest drei unabhängige Benutzer diese Funktionalität einfordern [24].

Dokumentation

Jede Software benötigt eine gute Dokumentation, vor allem jene, die für Wiederverwendbarkeit konzipiert sind. Andere Programmierer müssen herausfinden können wozu bestimmte Klassen und Methoden gedacht sind und verwendet werden können. Je besser die Dokumentation desto weniger Aufwand muss der Benutzer später betreiben, um die Software zu verstehen und zu verwenden. Auch muss er einfach herausfinden können welche Fehler wodurch hervorgerufen werden können und wie diese zu beheben sind. Daher ist es unumgänglich Dokumentationen zu schreiben, selbst wenn es ein langweiliger und aufwendiger Prozess ist [20].

Das größte Problem an Dokumentationen ist, dass sie meistens nach der Implementierung geschehen. Zu diesem Zeitpunkt sind die Programmierer einerseits nicht mehr gewillt eine Dokumentation zu schreiben, weil sie das Gefühl haben all die Arbeit, die sie bereits getan haben noch einmal machen zu müssen. Andererseits passiert es dadurch oft, dass zu implementationslastig dokumentiert wird. Das bedeutet, dass zum einen die Dokumentation

unfertig ist weil der Programmierer davon ausgeht, dass manche Dinge offensichtlich sind, obwohl sie es für den Neueinsteiger, der die Dokumentation benötigt, nicht sind. Zum anderen scheinen dadurch oft Implementierungs Details durch, die für den Benutzer nicht relevant sind und ihn verwirren können. Eine gute Dokumentation erklärt nicht nur die Klassen und Methoden, sie stellt auch Use case Szenarien und Beispiele zur Verfügung. Im besten Fall können später auch noch Tutorials hinzugefügt werden, die die Verwendung sogar noch einfacher machen [13, 20].

Wenn eine API fertig dokumentiert und implementiert ist, sollte sie von jemandem erprobt werden, für den sie unbekannt ist. Anfangs sollte diese Person untersuchen wie viel von der API verstanden werden kann ohne die Dokumentation zu Rate zu ziehen. Wenn eine API ohne Dokumentation verwendet werden kann, dann liegt ihr wahrscheinlich ein gutes Design zu Grunde. Eine selbst dokumentierende API ist die beste Art, die erstellt werden kann [25]. Dies enthebt einen aber nicht von der Notwendigkeit Dokumentationen zu schreiben.

4.3.2 Implementierungs Regeln

Benennungsregeln

In verschiedenen Büchern und anderen Quellen werden unterschiedliche Benennungsrichtlinien propagiert. Es ist oft schwer sich für eine zu entscheiden und außerdem oft auch Programmiersprachenabhängig. Aber egal ob man sich nun für `PascalCasing` und `camelCasing` oder doch lieber für Präfixe vor den Membernamen entscheidet, wichtig ist, dass diese Bennenungsrichtlinien über die ganze API konsistent sind und konsequent verfolgt werden.

Die Namen, die für eine Schnittstellen verwendet werden sollten selbsterklärend sein. Folglich sollten keine kryptischen Abkürzungen verwendet werden. Ausnahmen hiervon bieten allgemein verwendete Abkürzungen wie `min` oder `max`. Wenn an einer Stelle Abkürzungen verwendet werden, müssen sie überall zur Anwendung kommen um Konsistenz zu wahren. Denn wenn einmal `max` und ein andermal `maximum` verwendet wird, so kann es leicht sein, dass der Benutzer nicht vorhandene Funktionen aufrufen will. Das wird zwar vom Compiler entdeckt, kann aber trotzdem frustrierend sein. Der Code sollte sich möglichst prosaisch lesen lassen, sodass sofort verstanden wird, was passiert. Dabei ist es auch wieder wichtig auf Konsistenz zu achten, gleiche Namen sollten also auch das gleiche bedeuten. Zwei Methoden mit dem selben Namen in unterschiedlichen Klassen sollten das entsprechend selbe tun [16]. Auch Parameterbenennung darf dabei nicht außer Acht gelassen werden. Die meisten Benutzer orientieren sich an den Tooltips die von den Entwicklungsumgebungen zur Verfügung gestellt werden oder an der Dokumentation. In diesen scheinen die Parameternamen auf und darauf aufbauend entscheiden die Benutzer welche Werte und Objekte sie überge-

ben (müssen). Besonders Parameternamen mit nur einem Buchstaben sollten vermieden werden [24].

Datenkapselung

Eine Regel die in fast jedem Programmierbuch wieder und wieder erklärt wird ist, je weniger exponiert wird, um so besser. Alles was nicht direkt benötigt wird sollte für den späteren Benutzer nicht sichtbar sein um Fehler zu vermeiden und Erweiterbarkeit zu garantieren. Um zu garantieren, dass wirklich nur die nötigsten Funktionen in der API aufscheinen, kann man sich wieder an den schon erwähnten Use Cases orientieren. Nur ein gültiger Anwendungsfall ist Grund eine Klasse oder Methode in die API aufzunehmen. Natürlich können diese Klassen oder Methoden trotzdem in der Implementierung existieren, sie sollten nur für den Benutzer nicht sichtbar sein. Wenn sie später benötigt werden, können sie immer noch veröffentlicht werden. Dinge zu einer API hinzuzufügen ist immer einfacher als Dinge zu entfernen [20].

Datenfelder, also die Membervariablen einer Klasse, sollten niemals public oder protected sein. Um sie zu lesen oder zu setzen müssen Getter und Setter Methoden zur Verfügung gestellt werden. Das ist vor allem auch deswegen vorteilhaft, weil in einer Methode zusätzliche Funktionen ausgeführt werden können, wie zum Beispiel zu überprüfen ob der übergebene Wert richtig ist, was bei direktem Zugriff auf ein Feld nicht möglich ist. Auch können, wenn nötig, Methoden von einer Klasse in eine Superklasse bewegt und trotzdem binäre Kompatibilität gewährleistet werden, bei einem Feld ist dies nicht möglich [20]. Die einzigen Felder welche veröffentlicht werden können, sind Konstanten.

Default Parameter

Der API Designer sollte Default Parameter, also Standardwerte, für alle Schnittstellen liefern. Konfigurationseinstellungen sollten optional sein und durch sinnvolle Standardwerte abgesichert werden [3]. So wird dem Benutzer zum Beispiel die Erstellung neuer Objekte so einfach wie möglich gemacht, was den Einstieg für neue Nutzer erleichtert. Bei Methodenaufrufen müssen nicht unzählige Parameter übergeben werden, die oft nicht sinnvoll sind und den Benutzer nur verwirren. Lange Parameterlisten können, vor allem bei mehreren Parametern mit gleichem Datentyp, dazu führen, dass Parameter vertauscht werden, wodurch der Code zwar noch compiliert sich aber falsch verhält. Besser ist es wenn einzelne Parameter in entsprechenden (kurzen) Methoden konfiguriert werden können, die der erfahrene Benutzer wenn gewünscht aufrufen kann. Dadurch wird auch verhindert, dass ungültige Zustände von Objekten zustande kommen, weil bestimmte Variablen nicht gesetzt wurden [24]. Der Code wird leichter lesbar als wenn bei jedem Ob-

jekt und Methodenaufruf große Codebausteine oder lange Parameterlisten geschrieben werden müssen.

Vererbung

Vererbung sollte nur dort zugelassen werden, wo es auch wirklich sinnvoll ist. Ansonsten wird damit gegen das Prinzip der Datenkapselung verstoßen und der Benutzer bekommt oft mehr Handlungsspielraum als beabsichtigt. Alle konkreten, also nicht abstrakten Klassen, sollten wenn möglich so abgeschlossen werden, dass keine Subklassen von ihnen gebildet werden können. Auch Methoden sollten nur dort für Subklassen überschreibbar gemacht werden, wo Ereiterbarkeit von Benutzerseite explizit gewünscht ist. Wenn Vererbung erlaubt sein soll, so ist es am besten abstrakte Klassen oder Interface Klassen zu verwenden, wobei zu jeder abstrakten Klasse mindestens eine konkrete Implementierung zur Verfügung gestellt werden soll [25, 20].

Exceptions

Der Benutzer muss wissen wie sich eine API verhält, wenn etwas schief geht und brauchen detailierte Fehler Information, die sie im Programm verarbeiten können. Fehlermeldungen die vom Nutzer gelesen werden können sind zwar praktisch, um Fehler zu diagnostiziern und zu debuggen, helfen aber nicht den Programmfluss zu kontrollieren [13]. Statt Error Codes oder Fehlermeldung durch Rückgabewerte, sollten Exceptions verwendet werden, da diese die Konsistenz erhöhen und die Verarbeitung somit einheitlicher und leichter machen. Im Fall von Konstruktoren oder Operatorüberladungen ist es nicht einmal möglich Fehler durch Rückgabewerte bekannt zu geben, und Error Codes können zu einfach ignoriert werden. Als Exception gemeldete Fehler hingegen müssen früher oder später behandelt werden und sorgen so für robusteren Code, sind also offensichtlich die bessere Wahl [3].

Exceptions haben zwei Komponenten, die erste ist die Nachricht, die den Entwickler darüber informiert, was schief gelaufen ist, die zweite ist der Typ der Exception, der von den Handlern[5] für die Entscheidung genutzt wird welche Aktion das Programm jetzt ausführen sollte. Eigene Exceptiontypen sollten nur erstellt werden, wenn Fehlerbedingungen vorhanden sind, die anders behandelt werden müssen als alle schon vorhandenen Ausnahmen. Ansonsten ist es besser aus den vordefinierten Exceptions eine Passende auszuwählen [3].

[5]Handler werden dazu benutzt Exceptions abzufangen und zu bearbeiten, sie können zum Beispiel das Programm sicher beenden, neue Benutzereingaben verlangen oder sie an die darüberliegende aufrufende Funktion übergeben, damit diese darauf reagieren kann.

Kapitel 5

Matchmaking & Rating Library

In folgendem Kapitel soll eine Library entworfen werden, die einerseits verschiedene Rating Algorithmen und andererseits ein Matchmaking Verfahren, das gleichstarke Spieler zusammen bringt, zur Verfügung stellt. Es sollen sowohl Elo Ratings als auch Glicko Ratings berechnet werden, um diese mit einander vergleichen zu können.

5.1 Design

Den Richtlinien aus Kapitel 4 folgend, werden hier zunächst die wichtigsten Anwendungsfälle und Anforderungen zusammengefasst und erklärt. Am Anfang eines jeden Anwendungsfalles muss überlegt werden, welche Akteure eine Software verwenden wollen. In diesem Fall ist der Spieler der einzige Akteur der Anforderungen an das System stellt. Es kann auch die Rolle des Gegenspielers definiert werden, die aber equivalent zur Rolle des Spielers funktioniert. Die benötigte Software kann in zwei Teile aufgeteilt werden, nämlich Matchmaking und Rating. Diese beiden können eigentlich getrennt von einander exisitieren und müssen nichts übereinander wissen. Dementsprechend hat jeder Bereich seine eigenen Anforderungen.

Matchmaking

Im Matchmaking System will der Benutzer vor allem eines, nämlich einen gleichstarken Gegner finden und ein Spiel mit ihm starten. Dazu muss das System folgende Dinge wissen:

- welche Spieler für ein Spiel zur Verfügung stehen, also auch gerade nach einem Partner suchen,
- welche Ratings die Spieler haben,

- wie die Gewinnwahrscheinlichkeit zwischen dem Spieler und einem möglichen Partner ist und
- in welchem Bereich die Gewinnwahrscheinlichkeit liegen muss, damit ein Match gebildet werden darf.

Ein heuristisch einfacherer Weg ist, die Ratingdifferenz der Spieler heranzuziehen, da sich die Gewinnwahrscheinlichkeit ohnehin direkt aus dieser Differenz ableitet. Dazu ist anzugeben wie groß die Differenz maximal sein darf. Um das System erweiterbar zu halten sollten diese Vorraussetzungen abstrahiert werden und nicht direkt auf die Ratings zugreifen. Besser ist es, wenn beliebige Filterkriterien angegeben werden können. Dementsprechend ist der Wert dieser Kriterien und die maximale Differenz zwischen diesen Werten die benötigte Information.

Außerdem ist dem Spieler wichtig, schnell einen Gegner zu finden. Dies kann gewährleistet werden, indem die maximale Differenz mit der Zeit erhöht wird, um eher jemanden zu finden, der in den gültigen Differenzbereich fällt. Dazu braucht das System natürlich mehr Informationen:

- wie lange sucht der Spieler schon nach einem Match,
- um wieviele Punkte erhöht sich der Suchradius pro Zeiteiheit und
- wie groß ist eine Zeiteinheit.

Wobei es besser ist, die Zeiteinheit als fix (als zum Beispiel eine Sekunde) zu definieren und dazu passend die Punkteveränderung pro Zeiteinheit anzupassen.

In Kapitel 2 werden verschiedene Strukturen für ein Matchmakig System vorgestellt. Hier wurden Filter gewählt, um das System zu realisieren. Die Entscheidung erklärt sich dadurch, dass Ratings schlecht in Behälter aufgeteilt werden können. Das einzige was möglich ist, ist andere Kriterien als Behälter zu realisieren. Es könnten zum Beispiel Behälter für verschiedene Sprachen generiert und in dem jeweiligen Behälter dann nach Personen gesucht werden, die zu den Filterkriterien passen.

Das Spiel selbst wird nicht von diesem System aus gestartet, es findet nur passende Spielerpaare und gibt diese nach außen weiter von wo aus die Applikation, die die Bibliothek verwendet, damit verfahren kann wie es ihr beliebt.

Rating

Das Rating System ist um eingies komplexer als das Matchmaking System. Es muss verschiedene Ratings berechnen können, die obwohl von der Grundstruktur ähnlich, sehr unterschiedliche Anforderungen haben. Einfacher zu realisieren ist das Elo System, das zu einem beliebigen Zeitpunkt aufgerufen werden kann, um aus einer bestimmten Anzahl an Spielen ein neues Rating zu berechnen. Dies kann auch problemlos nach jedem Spiel passieren. Dazu müssen folgende Dinge bekannt sein:

- gegen welche Gegner jede Person gespielt hat,
- welchen Wert das Rating des Spielers vor der Berechnung hat,
- welche Ratings die Gegner vor der Berechnung haben,
- das Ergebnis zwischen einem Spieler und dem jeweiligen Gegner,
- auf welcher Skala sich die Ratings befinden und
- welchen Wert der Ratingkoeffizient annimmt.

Um den in Abschnitt 3.5 erklärten Problemen entgegenzuwirken müssen die Ratings bestimmter Spieler zuerst berechnet werden und dann erst die Ratings aller anderen. Wichtig dabei ist, dass die Ratings nicht direkt aktualisiert werden dürfen, sondern erst am Ende des jeweiligen Berechnungsabschnittes, damit nicht manche Spieler mit den Pre-Ratings bewertet werden und manche mit den Post-Ratings.

Das Glicko Rating System zu realisieren ist etwas komplizierter, denn es müssen die Ratings immer nach einer Rating Periode, für alle Spieler die in dieser Rating Periode angetreten sind, aktualisiert werden. Es sind also zusätzlich zu den Informationen des Elo Systemes folgende Informationen von nöten:

- welche Spieler in der letzten Rating Periode gespielt haben und
- welche Spiele jede dieser Personen während dieser Zeit absolviert hat.
- Welchen Wert die Ratingabweichung eines Spielers hat,
- welchen maximalen und minimalen Wert die Ratingabweichung annehmen darf,
- wie viele Ratingperioden vergangen sind, seit das letzte Rating für einen Spieler berechnet wurde und
- wie groß die Varianzänderung des Ratings pro vergangener Rating Periode ist.

Der Wert des Ratingkoeffizienten wird im Glicko System nicht benötigt, weil sich dieser automatisch aus anderen Werten berechnet.

Die Schwierigkeit auf die man hier stößt ist die Frage wer dafür verantwortlich ist die Informationen die über eine Rating Periode gesammelt werden zu speichern und wie bzw. wo sicher gestellt wird, dass am Ende jeder Rating Periode der Berechnungsvorgang gestartet wird.

5.2 Implementierung

Für die Implementierung wurde C# verwendet wobei die Windows Benennungsrichtlinien[3] befolgt wurden. Das heißt `PascalCasing` für alle Member und Typennamen, `camelCasing` für Parameternamen.

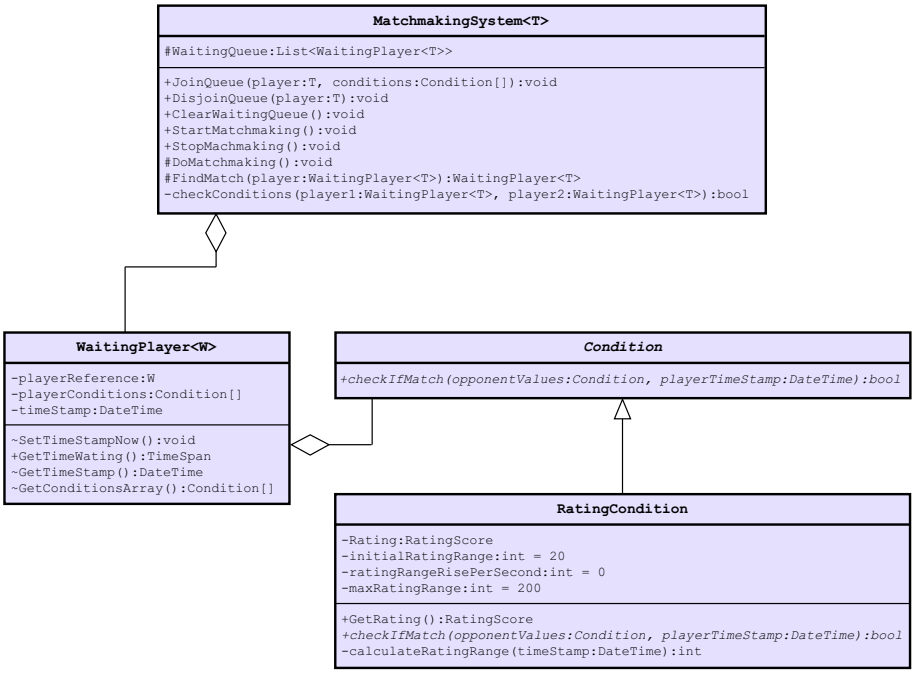


Abbildung 5.1: Klassendigramm des Matchmaking Systems

5.2.1 Architektur

Matchmaking

Das Matchmaking System ist, wie in Abbildung 5.1 veranschaulicht, so aufgebaut, dass es eine zentrale Klasse `MatchmakingSystem` gibt, zu der Spieler hinzugefügt werden können, um einen passenden Gegner zu finden. Dabei können Vorraussetzungen mitübergeben werden, die in unterschiedlichen `Condition` Objekten gespeichert sind. Das System erzeugt dann aus den übergebenen Daten einen Wrapper `WaitingPlayer` und fügt den Benutzer zu einer Warteliste hinzu.

Das Matchmaking System läuft die Warteliste durch und untersucht für jeden Spieler, ob es einen geeigneten Gegenspieler für ihn gibt. Auf diese Art ist auch sichergestellt, dass Benutzer die schon länger warten bevorzugt werden, weil nach jedem gefundenen Match wieder oben in der Warteliste begonnen wird. Um herauszufinden, ob zwei Spieler zu einander passen, werden zuerst die Konditionen des einen Spielers überprüft und wenn all diese Voraussetzungen vom Gegener erfüllt werden, werden noch die Konditionen des Gegenspielers untersucht. Nur wenn beide Spieler den Filter des jeweils anderen passieren (siehe Abschnitt 2.2), wird ein `MatchFoundEvent` ausgelöst und die Spieler werden an das überliegende System übergeben, das dann zum Beispiel ein Spiel starten kann.

`Condition` Es wird eine abstrakte `Condition` Klasse zur Verfügung gestellt, von der eigene, spezifische Conditions abgleitet werden können. Für das hier vorgestellte System wird zum Beispiel eine `RatingCondition` benötigt und bereitgestellt. In der Basisklasse befindet sich eine abstrakte `checkIfMatch()` Methode die von allen Subklassen implementiert werden muss. In dieser Funktion wird überprüft, ob die Werte des jeweiligen Gegners zu der eigenen Vorraussetzung passt. Wenn dem so ist wird in einer `bool` Variable `true` zurück gegeben, im anderen Fall `false`.

`WaitingPlayer` Die `WaitingPlayer` Klasse dient dazu, die übergebenen Daten im Matchmaking System verwendbar zu machen. Es wird eine generische Referenz auf den Spieler, der repräsentiert wird, benötigt, damit im nachhinein, wenn das `MatchFoundEvent` ausgelöst wird, dem umliegenden System mitgeteilt werden kann, welche Spieler es sind, die ein Match austragen sollen. Ausserdem werden alle `Condition` Objekte, die für einen Spieler gelten, in einem Array abgelegt, das vom Matchmaking System Schritt für Schritt im Bezug auf den jeweiligen Gegner überprüft wird. Die dritte wichtige Information, die in dieser Klasse aufbewahrt wird, ist die Zeit zu welcher ein Spieler zum Matchmaking System hinzugefügt wurde. Anhand dieser Zeit kann einerseits die Filterdurchlässigkeit jeder `Condition` gesteuert werden, andererseits hilft sie, um festzustellen wie lange die Spieler durchschnittlich warten müssen, bis sie ein Match gefunden haben.

Rating

Das Rating System sollte weitgehend automatisch laufen, ohne dass sich der Benutzer Gedanken darüber machen muss, wie die Ratings genau zustande kommen (kann bei Bedarf aber erweitert werden). Daher werden einfach nur nach jedem Match die daran beteiligten Spieler und das Ergebnis an das System übergeben. Der Benutzer kann sich dann darauf verlassen, dass die Ratings zu gegebener Zeit angepasst werden. Im Elo Rating System kann das sofort sein, im Glicko Rating System geschieht dies immer nach einer vordefinierten Rating Periode. Der Aufbau ist in Abbildung 5.2 ersichtlich und ist so, dass es `RatingScore` Objekte gibt, die jeweils für das System spezifische Daten speichern. In einem `EloRatingScore` Objekt wird also das Rating gespeichert und ob es ein neues, vorläufiges oder etabliertes Rating ist. In einem `GlickoRatingScore` Objekt wird zusätzlich noch die Ratingabweichung gespeichert. Diese Objekte können dann als Membervariablen für einen Spieler im umgebenden System verwendet werden. Dadurch ist sichergestellt dass ein Spieler seperat mit Glicko und Elo Rating Algorithmen bewertet werden kann, um diese Daten später vergleichen zu können. In einem richtigen Spiel wird zwar vermutlich nur noch eines der Systeme verwendet werden, es kann dadurch aber auch ermöglicht werden, dass für

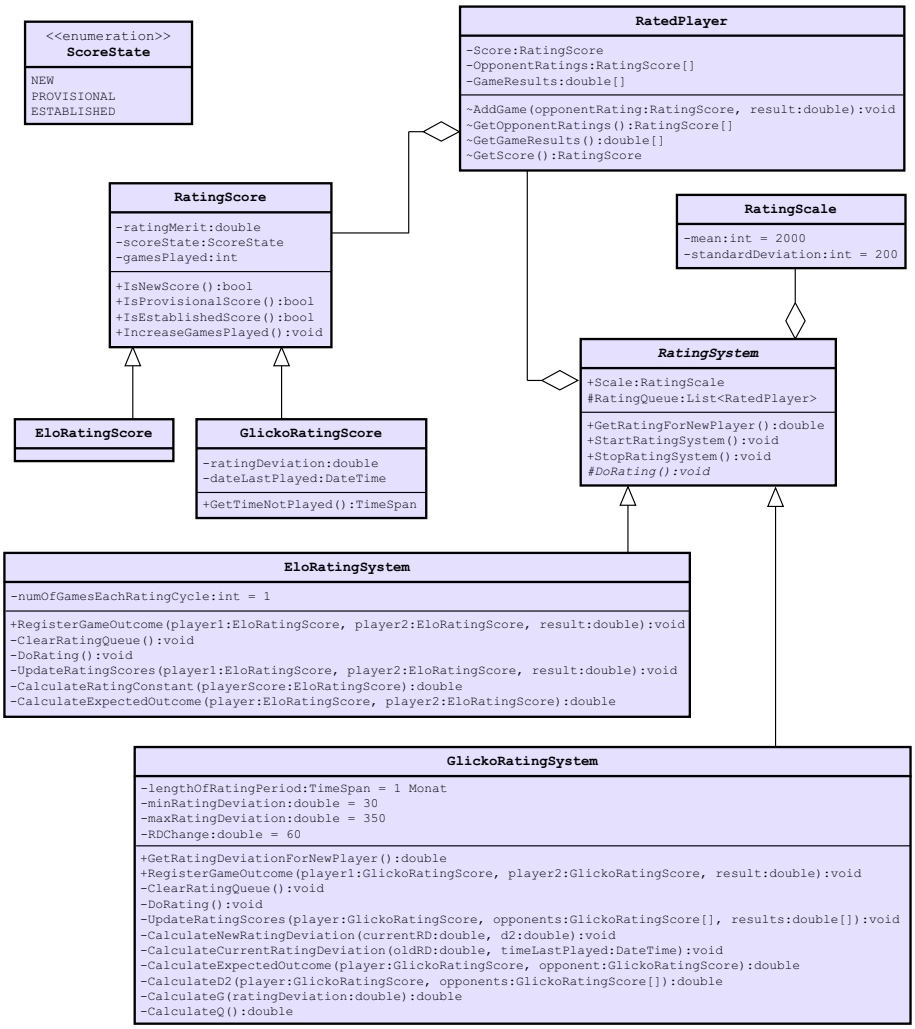


Abbildung 5.2: Klassendigramm des Rating Systems

einen Spieler zum Beispiel unabhängige Ratings für unterschiedliche Spielcharaktere gespeichert werden können. Wenn dem Rating System Spieldaten übergeben werden, so werden diese in eigene `RatedPlayer` Objekte für jeden Spieler gespeichert. Darin wird das Rating Objekt vermerkt, sowie die Ratings aller Gegner und die Spielergebnisse gegen diese Gegner.

Die `RatedPlayer` werden in einer Liste angemeldet, um zu wissen, welche Ratings am Ende der Rating Periode aktualisiert werden müssen. Wenn eine Rating Periode vorüber ist, wird die Berechnung gestartet. Dabei ist es wichtig, dass zuerst die Ratings aller Spieler berechnet werden, die bisher unbewertet waren, also nur ein Standardrating besitzen. Mit den aktualisierten Daten dieser Spieler werden dann die Ratings jener Teilnehmer berechnet,

die ein vorläufiges Rating haben, also nur an einer geringen Anzahl von Spielen teilgenommen haben. Danach erst werden die Ratings aller anderen Spieler berechnet. Wichtig ist, dass in jedem Berechnungsschritt alle Ratings quasi gleichzeitig berechnet werden, soll heißen, dass zur Berechnung der Bewertung einer Person immer die Vorberechnungsratings der Gegner hergenommen werden. Nur am Ende eines Berechnugnsschrittes dürfen die Ratings aktualisiert und dann, falls noch ein Schritt folgen sollte, in die nächsten Ratings einberechnet werden.

`RatingScore` Die RatingScore Objekte dienen dazu das aktuelle Rating eines Spielers zu speichern und ob dieses Rating neu, provisorisch oder etabliert ist. Um zu entscheiden welchen Status ein Rating hat, muss die Anzahl der Spiele im Auge behalten werden, die in ein Rating einfließen. Wenn diese Zahl zum Beispiel größer als 30 wird, so wird der Status von provisorisch zu etabliert geändert. Für das Glicko System wird außerdem noch die Zeit gespeichert zu der das letzte Spiel stattgefunden hat, damit man daraus die aktuelle Ratingabweichung berechnen kann.

`RatedPlayer` Mit der `RatedPlayer` Klasse werden Objekte erzeugt, durch die das Rating System auf die Spieler zugreifen kann, die angemeldet wurden, damit die Berechnung nicht sofort erfolgen muss, sondern zu einem gewünschten Zeitpunkt ausgeführt werden kann. Es wird das `RatingScore` Objekt gespeichert, sowie alle Gegner gegen die ein Spieler angetreten ist sowie das jeweilige Spielergebnis zu diesen Gegnern. Wenn nach einem abgschlossenen Spiel die Teilnehmer und das Ergebnis übergeben wird, so untersucht das Rating System zuerst, ob es die Teilnehmer schon gespeichert hat und wenn ja wird nur der Gegner und das Ergebnis zum jeweiligen `RatedPlayer` hinzugefügt, andernfalls wird ein neues Objekt für diesen Spieler erzeugt.

`RatingScale` Jedes Rating System braucht eine Rating Skala durch die gesteuert wird, welche Einstiegsratings Spieler bekommen und wie die Ratingpunkteveränderung pro Rechenvorgang skaliert ist.

5.2.2 Implementierungsdetails

Man unterscheidet in einer Software zwischen synchronen und asynchronen Aufrufen. Bei einer synchronen Operation wird der aufrufende Thread blockiert und wartet bis die aufgerufene Methode beendet wurde und ihre Ergebnisse zurück gibt. Daher kann ein neuer Schritt erst beginnen, wenn der alte Schritt abgeschlossen wurde. Um dem Abhilfe zu schaffen gibt es asynchrone Methoden, durch die mehrere Arbeitsschritte parallel ablaufen können. Das heißt dass mehere Threads[1] gleichzeitig laufen müssen. Wenn eine

[1]Ein Thread bezeichnet einen sequentiellen Abarbeitungslauf eines Prozesses. Üblicherweise läuft ein einzelner Thread, also ein Ausführungsstrang, in dem bestimmte Befehle

asynchrone Methode aufgerufen wird, so muss der Aufrufer nicht warten bis diese Methode abgeschlossen ist, sondern kann weiter ihre eigenen Aufgaben erledigen.

Im Kontext dieser Library sind sowohl das Matchmaking System als auch das Rating System eigene abgekapselte Anwendungen. Sie müssen also ihre eigenen Threads starten, um separat vom Hauptprogramm laufen zu können. Das Hauptprogramm übergibt nur die Spieler an das jeweilige System, damit sich dieses zu gegebener Zeit damit beschäftigen kann. Die gefundenen Ergebnisse werden entweder über Events an das Hauptprogramm zurück gegeben oder direkt in die übergebenen Parameter geschrieben.

Üblicherweise werden Events von dem Thread behandelt, der sie ausgelöst hat, das kann aber zu Problemen führen, besonders wenn auf GUI[2] Elemente in C# zugegriffen werden soll. Diese sind Steuerungselemente, die nur von dem Thread bearbeitet werden können, der sie auch erstellt hat, was üblicherweise der Hauptthread ist. Wenn nun über ein Event ein anderer Thread auf diese Elemente zugreift, kommt es zu einem Absturz des Programmes, daher müssen die Events vom Hauptthread bearbeitet werden. Um das zu erreichen, muss das Event über ein `AsyncOperation` Objekt abgeschickt werden. Dieses Objekt übergibt die Kontrolle an den Hauptthread der dann die weitere Behandlung durchführt. Dadurch kann auch sicher gegangen werden, dass der Thread, von dem das Event losgeschickt wird, sich nicht mit der Behandlung des Events aufhält, sondern sich wieder seiner eigentlichen Aufgabe zuwendet.

```
1 AsyncOperation asyncOp = AsyncOperationManager.CreateOperation(null);
2 asyncOp.Post(new SendOrPostCallback(delegate(object obj)
3   {
4     OnChanged(EventArgs.Empty);
5   }), null);
```

5.3 Testprogramm

Um die Funktionsfähigkeit der Library zu testen musste ein Beispielprogramm implementiert werden, das Spieler zum Matchmakingsystem hinzufügt und den Verlauf der Ratings speichert und abbildet. Zur Abbildung der Graphen wurde die Open Source Klassenbibliothek ZedGraph[3] verwendet, mit der man unterschiedlichste Diagramme zeichnen kann.

Das Testprogramm besteht aus einem `GameSimulation` Objekt, in dem wieder ein eigenständiger Thread läuft, der verfügbare Spieler der Klasse `CustomPlayer` zum Matchmaking System hinzufügt. Außerdem werden

nach der Reihe ausgeführt werden. Wenn Multithreading möglich gemacht wird, können mehrere Threads nebeneinander laufen und so können in einem Prozess mehrere Aufgaben auf einmal ausgeführt werden.

[2]GUI steht für Graphical User Interface

[3]http://zedgraph.sourceforge.net/index.html

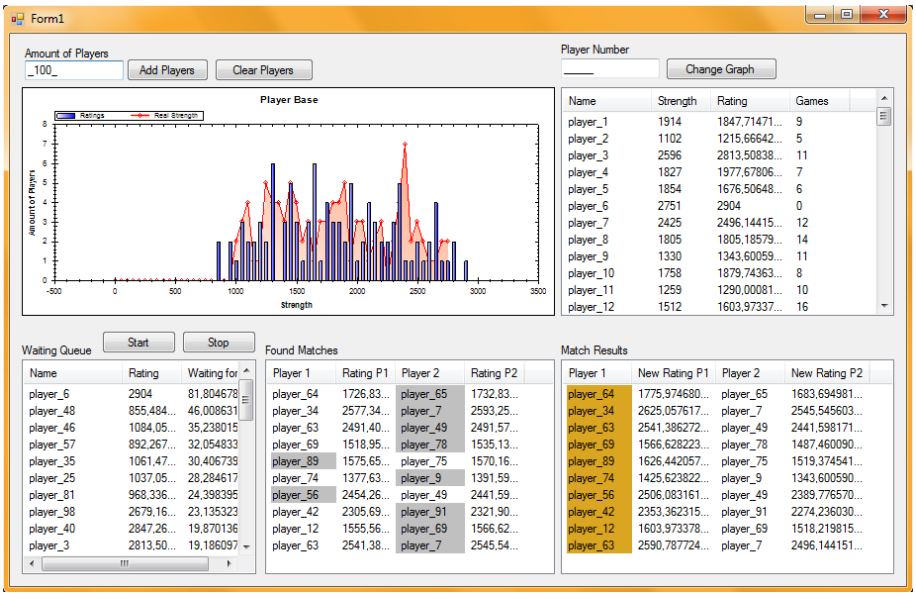


Abbildung 5.3: Die GUI der Testprogrammes. Links oben ist eine grafische Darstellung der Spielerbasis zu sehen und rechts daneben eine Liste alle Spieler mit ihren jeweiligen aktuellen Ratings und ihrer wahren Stärke, sowie die Anzahl an Spielen, an der sie schon teilgenommen haben. Links unten befindet sich eine Liste aller Spieler, die sich zurzeit in der Warteliste befinden und rechts daneben die letzten zehn Matches die gefunden wurden, wobei der Spieler, der den derzeitigen Ratings entsprechend gewinnen sollte, hervorgehoben wurde. Rechts davon wird abgebildet, wie diese Spiele ausgegangen sind und wie sich die Ratings der Teilnehmer nach diesem Ergebnis verändert haben.

Events abgefangen die über ein gefundenes Match informieren, wodurch ein Spielergebnis ausgelöst wird. Das Spielergebnis wird wiederum an die Rating Systeme übergeben wo die neuen Ratings der Spieler automatisch zum richtigen Zeitpunkt aktualisiert werden. In der `CustomPlayer` Klasse wird sowohl ein `EloRatingScore` als auch ein `GlickoRatingScore` Objekt gespeichert, um beide Ratings für einen Spieler berechnen und miteinander vergleichen zu können. Zusätzlich hat jeder Spieler eine Variable `Strength`, die die wahre Stärke eines Spielers darstellt und dazu verwendet wird das Spielergebnis zu berechnen. Alle bisherigen Ratings werden in eigenen Listen gespeichert, damit ihr Verlauf später in der GUI grafisch dargestellt werden kann.

Um das Testprogramm während der Laufzeit beeinflussen und um Ergebnisse darstellen zu können wurde eine C# Form verwendet und an die Bedürfnisse angepasst. Eine Abbildung der GUI ist in 5.3 zu finden. Wobei in dieser Version nur Elo Ratings im Vergleich zur wahren Spielstärke dargestellt werden.

5.4 Ergebnisse

Wie in Abbildung 5.3 zu sehen ist funktioniert das Matchmaking und Rating im Großen und Ganzen, es gibt allerdings ein paar Probleme, für die Lösungen gefunden werden müssen.

5.4.1 Wartezeiten

Zum einen sind in der Warteliste Spieler, die schon sehr lange auf einen brauchbaren Gegner warten. Dies liegt daran, dass sie ungewöhnlich hohe bzw. niedrige Ratings haben und der Suchradius im Matchmaking System so eingestellt ist, dass er maximal 200 Ratingpunkte in jede Richtung beträgt. Das bedingt zwar, dass ein Match von der schwächeren Person immer noch gewonnen werden kann, verhindert in diesem Fall aber, dass ein passender Partner gefunden wird. Hier muss der Game Designer entscheiden, wie dies zu handhaben ist. Sollen Spieler ungewöhnlich lange warten müssen, oder gibt man ihnen lieber ein unpassendes Match? Wenn die Spielerbasis größer ist und somit mehr Personen in der Warteliste sind, steigt natürlich die Wahrscheinlichkeit, dass ein passender Gegner gefunden wird. Daher sollten solche Entscheidungen immer nur nach vielen Testläufen getroffen werden und unter Beachtung dessen, was sich die Spieler wünschen.

5.4.2 Inflation und Deflation

Wie in Abschnitt 3.5.2 schon dargelegt, ist ein großes Problem der Ratingsysteme jenes, dass Punkte Inflation und Deflation auftreten kann. Meist aber in Zusammenhang mit neu hinzukommenden und das Spiel verlassenden Spielern. Da hier die Spielerbasis stabil ist, sollten diese Probleme also nicht auftreten. Trotzdem kann an den Rändern der Spielerbasis beobachtet werden, dass diese sich immer mehr nach außen ziehen. Dies hängt vor allem damit zusammen, dass diese Spieler nicht mehr so viele Gegner haben, gegen die sie gewinnen bzw. verlieren können. Schwächere Spieler verlieren also immer weiter, weil sie keine equivalenten Gegner finden und dadurch sink ihr Rating immer mehr. Wohingegen stärkere Spieler immer wieder gewinnen wodurch ihr Rating immer weiter steigt. Um dem entgegenzuwirken, ist es besonders wirksam, unterschiedliche Ratingkoeffizienten für unterschiedliche Ratings zu verwenden. Wenn also Spieler sich schon eher an den äußeren Rändern der Spielerbasis befinden, so muss der Koeffizient kleiner sein, damit sich ihre Ratings nicht mehr so stark verändern können. Dadurch wird die Inflation und Deflation zwar nicht aufgehalten, aber zumindest stark auf einen akzeptablen Wert vermindert. Anzumerken wäre noch, dass Deflation vermutlich in einer echten Spielumgebung nicht so sehr auffällt, da Spieler, die besonders schlecht sind, das Spiel eher früher verlassen und somit nicht noch mehr Punkte verlieren können.

5.4.3 Elo vs. Glicko

Wenn die Ergebnisse aus dem Elo System mit den Ergebnissen aus dem Glicko System verglichen werden, so wird ersichtlich, dass das Glicko System in einigen Bereichen besser funktioniert. Zum einen wird schneller ein Wert erreicht, der der Spielstärke entspricht, wofür das Elo System oft 50 Spiele und mehr benötigt. Zum anderen bleibt ein etabliertes Rating im Glicko System konstanter und unterliegt weniger Schwankungen als im Elo System, dadurch wird auch das Problem der Punkteinflation automatisch behandelt und vermindert.

5.5 Erweiterbarkeit

Die bestehende Library bietet nur die grundsätzlichsten Funktionen an. In diesem Abschnitt werden Möglichkeiten aufgezeigt, wie sie erweitert werden kann. Die unterschiedliche Anforderungen für verschiedene Genres werden vor allem aus der Game Design und Benutzer Perspektive angesprochen.

Der wichtigste Punkt der erweitert werden muss, ist die Möglichkeit mehr als nur zwei Spieler in einem Match zusammenzuführen und hinterher zu bewerten. Das kann erreicht werden indem sichergestellt wird, dass der Unterschied zwischen den Ratings des schwächsten und des stärksten Spielers einer Gruppe eine bestimmte Ratingdifferenz nicht übersteigt. Wenn dann die Spieler in Teams aufgeteilt werden, so müss sie zuerst anhand ihrer Ratings sortiert werden. Der erste, also stärkste, Spieler wird dann dem ersten Team zugeteilt, der zweite Spieler dem nächsten Team und so weiter. Wenn sich in jedem Team ein Spieler befindet wird der nächste Spieler immer dem Team zugewiesen, das das geringste kumulative Skillrating hat [12].

Unabhängig von den Spielgenres sollte auch die Entscheidung gefällt werden, ob Skill Ratings für die Spieler ersichtlich gemacht werden oder nicht. Vorteilhaft ist das deswegen, weil der Spieler üblicherweise wissen will, wie gut er ist, in Relation zu den anderen Spielern. Dadurch wird Wettkampf gefördert und es wird ein Anreiz geschaffen um weiter zu spielen und zu trainieren. Außerdem wird das Matchmakingsystem dadurch vorhersagbar und transparent für den Spieler. Auf der anderen Seite kann dadurch aber auch der Einsatz von unfairen Mitteln und unsportliches Verhalten gefördert werden, da manche Spieler alles tun würden, um ein hohes Rating zu bekommen mit dem sie angeben können. Zudem kann es sehr demotivierend für den Spieler sein, wenn er das eigene Rating stagnieren oder gar abfallen sieht [12].

Lösbar wäre dieser Konflikt indem zwei unterschiedliche Spielmodi angeboten werden. Einer, in dem kompetitiv gespielt und das Ranking angezeigt wird. Ein anderer, in dem nur für den Spass des Spieles teilgenommen wird, wo keine Rankings angezeigt werden, aber trotzdem noch für Matchmaking verwendet werden können. Ähnlich wird es auch in *League of Legends* gelöst,

wo es Normal Games und Ranked Games gibt[4].

Wenn Rankings angezeigt werden, so kann es oft passieren, dass Spieler, die sich in den oberen Rängen befinden, sich aus dem aktiven Spielgeschehen zurückziehen, um ihre Leaderbordposition nicht durch etwaige Verluste zu gefährden. Solch einem Verhalten kann entgegengewirkt werden, indem Spieler, die über längere Zeit nicht angetreten sind, nicht mehr in der Rangliste angezeigt werden [12]. Dadurch wird die Highscoreliste auch automatisch sauber gehalten, weil Spieler, die aufgehört haben zu spielen, automatisch entfernt werden.

5.5.1 Shooter

Multiplayer Shooter sind meist so aufgebaut, dass zwei Teams gegen einander antreten. Es kann natürlich ganz einfach der Ausgang des Matches bewertet werden, um damit zu akzeptablen Ergebnissen zu kommen. Oft spielen aber auch noch andere Kriterien in die erbrachte Leistung eines Spielers hinein. Zum Beispiel gibt es oft unterschiedliche Maps auf denen sich die einzelnen Spieler unterschiedlich gut auskennen, was sich stark auf die Spielstärke auswirkt. Damit ein Spieler mit Gegnern zusammengebracht wird die auf einer bestimmten Map genau so stark sind wie er selbst, muss das ins Matchmaking eingebracht werden. Ein einfacher Ansatz ist, zu zählen wie oft eine Person eine bestimmte Map schon gespielt hat und bei einem geringen Wert das Rating für den Matchmakingprozess als reduziert zu betrachten. Eine etwas kompliziertere Methode ist, für jede Map ein eigenes Rating zu erstellen, sodass das allgemeine Rating (sofern benötigt) nur ein Mittelwert aus den Einzelratings ist. Ein anderer Faktor mit dem ähnlich verfahren werden kann, sind unterschiedliche Spiel Modi wie „Capture the Flag“ oder „Death Match“.

Um das Rating der Spieler aufzuschlüsseln, können einzelne Frags bewertet werden. Wenn schwächere Gegner besiegt werden, so bekommt der Spieler kaum Punkte, wenn er stärkere niederstreckt, so bekommt er mehr Punkte. Dies ist insofern ein fraglicher Ansatz, weil dadurch das Spielverhalten stark gesteuert wird. Es ist zwar argumentierbar, dass es in Shootern darum geht andere zu erschießen, aber vor allem der Teamgeist bleibt dadurch gern auf der Strecke. Defensives, Team unterstützendes Spielverhalten wird dann für mehr Punkte über Bord geworfen. Wenn dieser Ansatz verwendet wird, so sollte auf jeden Fall getestet werden wie er sich auf das aktive Spielgeschehen auswirkt.

5.5.2 Strategiespiele

Im Grunde kann, ähnlich wie bei Shootern, die Map mit einbezogen werden. Allerdings ist das vielleicht gar nicht nötig, weil Strategiespiele langsamer

[4] http://euw.leagueoflegends.com/

sind und es genug Zeit gibt, die Map zu erkunden. Starcraft 2 wiederlegt diese Theorie zwar, weil es über ein hohes Spieltempo verfügt, vor allem in den höheren Ratings, bildet aber eher eine Ausnahme. Die meisten Spieler sind der Meinung, dass Actions per Minute eine hohe Aussagekraft über die Spielstärke treffen, aber dieser Wert ist so einfach durch sinnloses Klicken mit der Maus zu verfälschen, sodass er nicht guten Gewissens ins Rating einberechnet werden kann. Hingegen ist es sinnvoll Vorteile unterschiedlicher „Klassen“ in das Rating einzuberechnen. Ähnlich wird es durch besondere Modifikationen im Schach gemacht, wo dem Spieler der die weißen Figuren hat ein Vorteil durch den ersten Zug nachgesagt wird. Das schlägt sich dann in seinem Rating nieder. Bei Strategiespielen liegt es nahe ähnlich zu verfahren, da es sehr schwer ist unterschiedliche Klassen wirklich absolut gleichmäßig stark zu gestalten. Wie sich diese Vorteile genau im Rating niederschlagen können, lässt sich in [6] nachlesen, wo ein „Order Effect“ Parameter eingeführt wird.

Interessant wäre noch, mehr Punkte für unpopuläre Gewinnstrategien zu vergeben, damit nicht alle Spieler immer das selbe machen. Dazu müssten aber lernfähige Algorithmen, die die Taktiken der Spieler bewerten, eingebaut werden und es ist fraglich, ob das Resultat den Aufwand rechtfertigt. So ein Vorgehen wäre nur bei offiziellen Turnieren mit der Hilfe von Juroren umsetzbar.

Strategiespiele sind zueinander oft so unterschiedlich, dass es schwer ist eine allgemeine Aussage zu treffen. Hier muss wirklich auf die individuellen Mechaniken des Spieles eingegangen werden, wenn der Ratingalgorithmus verfeinert werden soll. Wobei wiederum die Frage ist, ob überhaupt ins Spielgeschehen eingegriffen werden soll. Wie schon umfangreich dargelegt, ist es am objektivsten, wenn einfach nur Gewinne und Verluste bewertet werden. Nur wenn klar wird, dass etwas schlecht gebalanced ist, sollte eingegriffen werden. Allerdings sollte es gut überlegt sein, ob dieses Ungleichgewicht wirklich durch den Rating Algorithmus korrigiert werden soll, oder ob es nicht besser ist die Spielmechanik zu überarbeiten.

5.5.3 Beat ’Em Ups

Im Bereich der Beat ’Em Up Spiele ist es besonders wichtig die Wahl des Charakters beim Matchmaking und/oder Rating einzuberechnen. Die Wahl des Kämpfers entscheidet ganz maßgeblich darüber welche Leistung erbracht werden kann. Nicht nur das es für jeden Charakter unterschiedliche Tastenkombinationen gibt, die der Spieler von manchen mehr, von manchen weniger beherrschen wird. Sondern auch, dass manche Kämpfer bestimmten anderen gegenüber im Vorteil sind. Darum müssen diese Zusammenhänge unbedingt im Vorhinein umfangreich getestet werden, und wenn sie nicht durch Ändern der Spielmechaniken ausgebessert werden können, so müssen sie ins jeweilige Rating einbrechnet werden.

Wie das Matchmaking gelöst wird hängt davon ab, ob gewollt ist, dass

ein Spieler seinen Charakter auswählen kann bevor er den Matchmaking Prozess startet oder ob er erst im Nachhinein zusammen mit dem anderen Spieler eine Figur wählt. Die Figur erst nach dem Matchmaking zu wählen ist eigentlich nur für Teams relevant, damit sich die Personen untereinander absprechen können. Es ist nicht sinnvoll, zu sehen welche Figur der Gegner wählt, denn dann wird dadurch die eigene Wahl beeinflusst und vielleicht ein Kämpfer ausgesucht, der degen den des Gegenspielers besonders gut ist. Dies führt zu zwei möglichen Problemen. Entweder der Gegenspieler ändert seinen Charakter entsprechend und darauf hin der Spieler und so weiter, bis das Spiel automatisch beginnt, oder eine einmal angegebene Figur kann nicht mehr ausgetauscht werden, was dann dazu führt, dass kein Spieler zuerst seine Figur angeben will. Daher ist es besser, wenn jeder Spieler seine Figur am Beginn des Matchmaking Prozesses auswählt und dann ein Gegenspieler gesucht wird, entsprechend des Ratings, dass dieser Figur zugeordnet ist. Hierbei können auch die Vorteile zwischen den Figuren abgewägt werden, indem man nicht die einfache Möglichkeit der Ratingdifferenz zum Filtern verwendet, sondern auf die Gewinnwahrscheinlichkeitsfunktion des Rating Programmes zugreift, in der die Zusammenhänge automatisch einberechnet werden sollten. Nach dem Spiel wird das Rating auch wieder in Abhängigkeit von diesen Vorteilen berechnet.

5.5.4 Actionspiele

Actionspiele umschreiben so ein umfassendes Feld an unterschiedlichen Spielen, dass schwer allgemeine Konzepte entworfen werden können. Darum werden hier Ideen nur speziell am Beispiel von *League of Legends* präsentiert.

Ähnlich wie bei Beat 'Em Ups stehen hier unzählige verschiedene Charaktere zur Auswahl, die alle ein wenig unterschiedlich zu spielen sind. Es kann zwar davon ausgegangen werden, dass bessere Spieler ein höheres Grundverständnis des Spieles besitzen, allerdings ist es schwer mit unterschiedlichen Charakteren gleich gut zu performen. Vor allem, weil ständig neue Charaktere hinzukommen und es kaum möglich ist mit allen gleich gut sein kann. Die meisten Spieler haben ein Set an Charakteren mit denen sie häufig spielen und demnach besonders gut sind. Manchmal ist es aber unumgänglich auch mit Charakteren zu spielen, die nicht so gut beherrscht werden, wenn zum Beispiel die bevorzugten Champions schon von anderen Spielern gewählt wurden oder eine bestimmte Rolle (z. B. ein Tank[5]), für ein Match benötigt wird. Das größte Problem dabei ist, dass Spielfiguren erst nach dem Matchmaking Prozess ausgewählt werden, womit ein charakterbezogenes Suchen nicht möglich ist. Allerdings kann die Figurenwahl nicht vor dem Matchmaking stattfinden, da es sehr wichtig ist, dass die Spieler ein gut aufeinander abgestimmtes Team aus unterschiedlichen Rollen bilden. Es ist also

[5]Ein Tank ist eine Spielfigur die besonders gut Schaden einstecken kann und die Mitspieler vor den Angriffen der Gegner schützt.

nicht möglich hier in den Matchmaking Prozess einzugreifen. Allerdings ist es unter Umständen für den Spieler selbst interessant, wie gut er mit einer bestimmten Figur spielt, im Vergleich zu anderen Personen. Es könnte also ein Charakter abhängiges Rating angeboten werden, selbst wenn es nicht ins Matchmaking einberechnet wird.

Im Fall von *League of Legends* lässt sich noch ein Punkt aufzeigen, der beachtenswert ist. Wenn die Map nicht symmetrisch genug ist, es also Vorteile für bestimmte Startpunkte gibt, so setzt das Spiel das schwächere Team auf die Seite, die vorteilhafter ist. Eine Technik die equivalent auch für andere Spiele übernommen werden kann, vor allem in Strategiespielen ist dies unter Umständen interessant.

5.5.5 Sonstige Genres

Das letzte Multiplayergenre sind Rollenspiele und für diese eigent sich ein Matchmaking System nicht. Schon allein deswegen, weil es niergendwo eingebaut werden könnte. Kämpfe werden im Spiel vom Spieler ausgesucht und nicht vom Programm. Doch selbst wenn es für Rollenspiele eine Mechanik mit Zufallskämpfen geben sollte, so werden die Gewinnchancen weniger vom Rating eines Spielers, als viel mehr vom Level seines Avatars und von den Items die er besitzt abhängen. Wenn diese Faktoren keine Rolle spielen, ist es eigentlich kein Rollenspiel sondern kann zu den Actionspielen gezählt werden.

Kapitel 6

Zusammenfassung und Ausblick

Im Verlauf dieser Arbeit wurde aufgezeigt, wie Matchmaking für ausgeglichene Gewinnchancen, basierend auf Rating Algorithmen realisiert werden kann. Zuerst wurden unterschiedliche Matchmaking Systeme dargestellt, wobei in der Library ein auf Filter basierendes System umgesetzt wurde. Dann wurden Elo Rating basierte Algorithmen vorgestellt, und im späteren Verlauf getestet. Weiters wurden einige API Designrichtlinien aufgestellt mit deren Hilfe die Library verständlich und einfach verwendbar gestaltet werden konnte.

Die entwickelte Klassenbibliothek bietet grundsätzliche Funktionalität, und kann für den Anfang so verwendet werden wie sie ist. Um die Spielerfahrung allerdings zu perfektionieren, ist es unumgänglich das System nach den Bedürfnissen des Spieles und vor allem der Spieler anzupassen. Es ist nicht möglich ein perfektes Machmaking und Rating System universell für jedes Spiel zur Verfügung zu stellen, weil es unzählige Faktoren gibt, die diese Systeme beeinflussen. Auf der anderen Seite ist es fast unmöglich die Spieler absolut zufrieden zu stellen. Vor allem, weil, selbst wenn der Spieler 50% seiner Matches gewinnt, er üblicherweise eher das Gefühl hat viel zu verlieren. Es ist aber nicht möglich ein Matchmaking System zu gestalten, das jeden Spieler 60% oder mehr seiner Spiele gewinnen lässt, nur um ihn motiviert zu halten. Denn jedes Spiel das ein Spieler gewinnt, muss ein anderer verlieren. Während also versucht wird die einen Spieler motiviert zu halten, werden die anderen entmutigt und hören auf. Das beste was erreicht werden kann, sind möglichst ausgeglichene Chancen für alle. Nach diesem Ziel sollte gestrebt werden.

Um eine möglichst akurate Bewertung der Spielstärken zu erhalten, empfiehlt es sich eher das Glicko System als das Elo System zu verwenden, da es aufgrund der zusätzlichen Faktoren die einberechnet werden, bessere Ergebnisse liefert, aber bewusst konzipiert ist um für eine große Spielerbasis

anwendbar zu sein [26]. Wenn dies noch weiter verbessert werden soll, so kann man sich in [10] über das Glicko-2 System informieren oder in [6] nachlesen wie „Order Effect“ Parameter das Rating beeinflussen können. Ähnlich wie das Glicko-2 System funktioniert auch der TrueSkill Algorithmus der im Xbox Matchmaking System eingesetzt wird [14]. Dieser Algorithmus ermöglicht es außerdem, nicht nur zwei Spieler zu matchen und ihre Ratings zu berechnen, sondern kann auch für Teams verwendet werden. Eine weitere Möglichkeit das System zu verbessern liegt darin Unentschieden als eigenes Ergebnis einzuberechnen. Das Elo System bietet diese Möglichkeit nur bedingt, indem ein Unentschieden als 0,5 gewertet wird, also die Hälfte zwischen einem Gewinn und einem Verlust. Andere Möglichkeiten Unentschieden einzuberechnen wurden in dieser Arbeit nicht diskutiert, da es bei den meisten Onlinespielen diesen Ausgang nicht gibt. Sollte es aber benötigt werden, so kann in [5] nachgelesen werden, wie die Einarbeitung funktionieren kann.

Probleme der Ratingsysteme und der entwickelten Bibliothek sowie besondere Erweiterungsmöglichkeiten wurden in den jeweiligen Kapiteln diskutiert und können dort nachgelesen werden.

Anhang A

Symbole

μ	Mittelwert einer Wahrscheinlichkeitsverteilung
σ	Standardabweichung einer Wahrscheinlichkeitsverteilung
σ^2	Varianz einer Wahrscheinlichkeitsverteilung
C	Klassenintervall (üblicherweise 200 Punkte)
D	Differenz zwischen zwei Ratings, $D = r - r_j$
e	Eulersche Zahl $e = 2,718281...$
$\mathrm{E}(X)$	Erwartungswert einer Variable X
j	Zählvariable für die Gegenspieler
K	Ratingkoeffizient
n	Gesamtanzahl der abolvierten Spiele
p	Prozentsatz der gewonnenen Spiele
r	Rating zu Beginn der Rating Periode
r'	neu berechnetes Rating
r_c	mittleres Rating der Kontrahenten
r_p	Performance Rating
RD	Ratingabweichung zu Beginn einer Rating Periode
RD'	neu berechnete Ratingabweichung
RD_{old}	Ratingabweichung nach der letzten Rating Periode
t	Anzahl der vergangenen Rating Perioden
v^2	Varianzänderung pro Zeiteinheit
w	1 für einen Gewinn, 0 für einen Verlust, 0,5 für ein Unentschieden

Anhang B

Inhalt der CD-ROM

B.1 PDF-Dateien

Pfad: /

MasterArbeit.pdf . . . . die Diplomarbeit

B.2 Bilder

Pfad: /Bilder

*.eps, *.png . . . . . Abbildungen der Diplomarbeit

B.3 Literatur

Pfad: /Literatur

API design matters.pdf Dokument zu [13]
designing reusable classes.pdf Dokument zu [16]
dynamic paired comparison models.pdf Dokument zu [10]
Gamasutra - Rising from the Ranks.pdf Dokument zu [27]
glicko system.pdf . . . . Dokument zu [26]
how to design a good api and why it matters.pdf Dokument zu [25]
matchmaking playstyle information.pdf Dokument zu [8]
most important design guideline.pdf Dokument zu [18]
parameter estimation in large dynamic paired.pdf Dokument zu [11]
rating system based on paired comparison models.pdf Dokument zu [15]
simplified matchmaking.pdf Dokument zu [17]
the little manual of api design.pdf Dokument zu [24]
True Skill - A Bayesian Skill Rating System.pdf Dokument zu [14]

Quellenverzeichnis

Literatur

[1] John D. Beasley. *The Mathematics of Games*. Erstauflage 1989. Mineola, NY: Dover Publications, Inc., 2006.

[2] Günter Clauß und Heinz Ebner. *Grundlagen der Statistik: Band 1*. 5. Aufl. Thun und Frankfurt am Main: Verlag Harri Deutsch, 1985.

[3] Krzysztof Cwalina und Brad Abrams. *Richtlinien für das Framework Design*. München: Addison-Wesley, 2007.

[4] Herbert A. David. *The method of paired comparison*. London: Charles Griffin & Co. Ltd., 1963.

[5] Roger R. Davidson. *On extending the bradley-terry model to accommodate ties in paired comparison experiments*. Techn. Ber. University of Victoria, Canada, 1969.

[6] Roger R. Davidson und Robert J. Beaver. „On Extending the Bradley-Terry Model to Incorporate Within-Pair Order Effects“. In: *Biometrics* 33.4 (Dez. 1977), S. 693–702.

[7] Arpad Elo. *The Rating of Chessplayers: Past & Present*. 2. Aufl. Erstauflage 1978. New York, NY: Ishi Press International, 2008.

[8] Shelly D. Farnham u. a. *Method for online game matchmaking using play style information*. Patent: US 7,614,955 B2. Nov. 2009.

[9] Erich Gamma u. a. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[10] Mark E. Glickman. „Dynamic paired comparison models with stochastic variances“. In: *Journal of Applied Statistics* 28 (2001), S. 673–689.

[11] Mark E. Glickman. „Parameter estimation in large dynamic paired comparison experiments“. In: *Applied Statistics* 48 (1999), S. 377–394.

[12] Thore Graepel und Ralf Herbrich. „Ranking and Matchmaking“. In: *Game Developer Magazine* (Okt. 2006), S. 25–34.

[13] Michi Henning. „API Design Matters“. In: *ACM Queue* 5.4 (Mai 2007), S. 24–36.

[14] Ralf Herbrich, Tom Minka und Thore Graepel. „TrueSkill: A Bayesian Skill Rating System“. In: *Advances in Neural Information Processing Systems*. Bd. 20. MIT Press, 2007, S. 569–576.

[15] Harry Joe. „Rating systems based on paired comparison models“. In: *Statistics & Probability Letters* 11.4 (Apr. 1991), S. 343–347.

[16] Ralph E. Johnson und Brian Foote. „Designing Reuseable Classes“. In: *Journal of Object-Oriented Programming* 1.2 (Juni 1988), S. 22–35.

[17] Jason Mai, Justin D. Brown und Hoon Im. *Simplified Matchmaking*. Patent: US 6,641,481 B1. Nov. 2003.

[18] Scott Meyers. „The Most Important Design Guideline?“ In: *IEEE Software* 21.4 (Juli 2004), S. 14–16.

[19] Lothar Sachs. *Angewandte Statistik: statistische Methoden und ihre Anwendung*. 5. Aufl. Berlin und Heidelberg: Springer Verlag, 1978.

[20] Jaroslav Tulach. *Practical API Design: Confessions of a Java Framework Architect*. Berkeley, CA: Apress, Inc., 2008.

Online-Quellen

[21] URL: http://de.wikipedia.org/wiki/Kollaboratives_Filtern (besucht am 17.11.2011).

[22] URL: http://en.wikipedia.org/wiki/Elo_rating_system (besucht am 12. 09.2011).

[23] URL: http://de.wikipedia.org/wiki/Elo-Zahl (besucht am 12.09.2011).

[24] Jasmin Blanchette. *The Little Manual of API Design*. Juni 2008. URL: http://www4.in.tum.de/~blanchet/api-design.pdf.

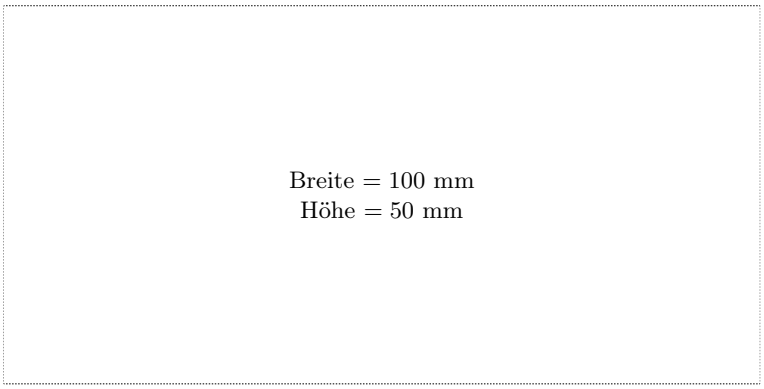
[25] Joshua Bloch. *How to Design a Good API and Why it Matters*. Google Tech Talk. Jan. 2007. URL: http://lcsd05.cs.tamu.edu/slides/keynote.pdf.

[26] Mark E. Glickman. *The Glicko system*. Juni 1999. URL: http://www.glicko.net/glicko/glicko.pdf.

[27] Bernd Kreimeier. *Rising from the Ranks: Rating for Multiplayer Games*. Feb. 2000. URL: http://www.gamasutra.com/view/feature/3428/rising_from_the_ranks_rating_for_.php.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —