Fachbereich Informatik und Mathematik
Institut für Informatik

Masterarbeit

Entwurf und Implementierung einer Simulation eines Strategiespiels unter Verwendung von intelligenten Suchmethoden und Agenten

Sascha Wardel

19. März 2016

eingereicht bei
Prof. Dr. Manfred Schmidt-Schauß
Künstliche Intelligenz / Softwaretechnologie

Erklärung gemäß Master-Ordnung 2008 §24 Abs. 12

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt, den 19. März 2016

Sascha Wardel

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

In unserer heutigen Gesellschaft wird elektronischen Spielen eine immer größer werdende Bedeutung zugeschrieben - sei es für die Unterhaltungsindustrie, als E-Sport oder auch als Simulation für Wirtschaft und Forschung. Dabei spielen intelligente Agenten eine wichtige Rolle, da im seltensten Fall jede Rolle eines Spiels durch menschliche Spieler besetzt werden kann. Aus diesem Grund ist es wichtig, diese kontinuierlich weiterzuentwickeln und zu verbessern. Diese Masterarbeit beschäftigt sich deshalb mit der Erstellung eines Spielprototypen, wie er in komplexerer Form auch Verwendung in den zuvor genannten Bereichen finden könnte und legt den Schwerpunkt dabei auf die Entwicklung von Agenten, die mittels intelligenter Suche den Eindruck eines menschlichen Spielers erwecken.

1.2 Die Idee

In klassischen Einspieler-Rollenspielen gibt es eine große Spielwelt mit vielen NPCs (= None Player Character) sowie der eigenen Spielfigur. Dabei dreht sich die vollständige Welt jedoch nur um die eigene Spielfigur. NPCs besitzen zwar überlicherweise vordefinierte Tagesabläufe (morgens stehen sie an Ort A, abends an Ort B und nachts schlafen sie an Ort C), diese sind jedoch eher kosmetischer Natur. Jegliche Veränderung der Spielwelt geht vom Spieler aus: Dieser triggert bestimmte vordefinierte Ereignisse, wodurch verschiedene Aktionen der NPCs oder Spielwelt ausgelöst werden. Diese Vorgehensweise soll den Eindruck einer lebendigen Welt erwecken. Spätestens bei einem erneuten Durchspielen wird der Schein jedoch offensichtlich, da sich die Spielwelt und die NPCs wieder exakt so wie beim ersten Durchspielen verhalten.
Ausgangspunkt dieser Masterarbeit war die Frage, ob es nicht möglich wäre, eine Spielwelt zu erschaffen, bei der die eigene Spielfigur nicht im Zentrum steht. Eine Spielwelt, in welcher die NPCs intelligent und selbstständig agieren und ihre eigenen Ziele verfolgen. Sollte der Spieler keine Aktivitäten ausführen, so würden die NPCs

nicht auf ihn warten und sich dennoch weiter entwickeln. Eine solche Spielwelt wäre bei jedem Spiel wieder ein neues Erlebnis.
Diese Idee wurde aufgegriffen und weiter ausgereift, bis das Konzept dieser Masterarbeit stand: Entwurf und Implementierung einer Simulation eines Strategiespiels unter Verwendung von intelligenten Suchmethoden und Agenten.

1.3 Struktur dieser Arbeit

Diese Masterarbeit führt den Leser von der Idee bis zur fertig entwickelten Anwendung und bietet darüber hinaus noch eine Auswertung der verschiedenen Agentenstrategien. Um die Thematik verständlich erklären zu können, findet zunächst in Kapitel 2 eine Besprechung der benötigten Grundlagen statt. Anschließend folgt in Kapitel 3 eine Anforderungsanalyse der Anwendung. Mit der gegebenen Anforderungsanalyse wird in Kapitel 4 die Spielwelt modelliert. Auf Grund des Umfangs und der Komplexität der Agenten wird diesen daraufhin das vollständige Kapitel 5 gewidmet. In diesem wird das Spiel als Suchproblem formuliert, um daraufhin die Auswahl eines passenden Suchalgorithmus, sowie einer passenden Bewertungsfunktion zu treffen. Anschließend folgt mit Kapitel 6 die Beschreibung und Erklärung der Anwendungs-Implementierung. Die Bedienung und Funktionalität der fertigen Anwendung wird anschließend in Kapitel 7 erläutert. Als letzter Schritt findet in Kapitel 8 der Vergleich der entwickelten Agenten über einen Benchmark statt, um daraufhin mit einer Zusammenfassung und einem Fazit der Masterarbeit in Kapitel 9 zu enden.

Kapitel 2

Grundlagen

Im folgenden Kapitel werden die in dieser Arbeit benötigten Grundlagen beschrieben. Diese untergliedern sich in zwei Bereiche: Zunächst werden Grundlagen zu Graphen und Suchproblemen beschrieben und zugehörige Algorithmen erläutert. Dieser Bereich basiert großteils auf dem Buch "Artificial Intelligence: A Modern Approach" von Russel und Norvig ([RN95]), welches als Standardwerk für KI-Suche betrachtet werden kann. Im zweiten Teil werden anschließend für die Implementierung relevante Grundlagen vermittelt.

2.1 Suchproblem

Ein Suchproblem beschreibt die Aufgabe, in einem gegebenen Graphen einen Zielknoten zu finden. Dabei ist mindestens ein Startknoten als Ausgangspunkt der Suche gegeben und zudem eine Nachfolgerfunktion und ein Zieltest. Mit Hilfe der Nachfolgerfunktion sind die Kinder jedes Elternknotens berechenbar, wodurch der Aufbau des Graphen, sowie die Suche in selbigem, möglich wird. Der Zieltest wird zur Identifizierung eines Zielknotens verwendet.

2.2 Begrifflichkeiten

Folgend findet sich eine kurze Auflistung und Erklärung der zum Verständnis wichtigen Begrifflichkeiten, welche im Verlauf dieser Arbeit verwendet werden.

- **Vollständigkeit eines Suchalgorithmus:** Ein Suchalgorithmus wird genau dann als vollständig bezeichnet, wenn er dazu in der Lage ist, einen existierenden Zielknoten in endlich vielen Rechenschritten zu finden.
- **Informierte Suche:** Eine Suche ist genau dann informiert, wenn eine Bewertung aller Knoten des Suchraumes angegeben werden kann, welche die Ähnlichkeit des Knotens zu einem Zielknoten abschätzt.

- **Nicht informierte Suche:** Eine Suche ist genau dann nicht informiert, wenn ausschließlich der Suchgraph gegeben ist (auch indirekt), ohne weitere Informationen zu besitzen.
- **Einen Knoten expandieren:** Mit dieser Bezeichnung ist die Auswahl eines Knotens und die weitere Überprüfung dessen Kinder gemeint.
- **Sharing:** Sharing bezeichnet das Verhalten einer Suche, schon besuchte Knoten zu notieren und zu überprüfen, damit Knoten nicht mehrfach expandiert werden.

2.3 Bewertungsfunktionen

Eine Bewertungsfunktion bildet jeden Knoten auf einen Wert ab, welcher den Abstand zum Ziel einschätzt. Informierte Algorithmen nutzen Bewertungsfunktionen zur Verbesserung der Suche. Dabei existieren zwei Ansätze: Die Werte der Bewertungsfunktion minimieren oder maximieren. Es muss vorher bekannt sein, ob eine Bewertungsfunktion und der Algorithmus minimierend oder maximierend arbeiten, da ansonsten die Knoten nicht korrekt miteinander verglichen werden könnten. Eine Bewertungsfunktion ist um so besser, je genauer sie dazu in der Lage ist, den Abstand eines Knotens zum Ziel einzuschätzen.
Folgend ein Beispiel[1] für das 8-Puzzle Problem mit zwei unterschiedlichen minimierenden Bewertungsfunktionen: Das 8-Puzzle Problem ist ein bekanntes Kinderspiel, bei dem acht aufsteigend nummerierte Plättchen in einem 3x3 Spielfeld liegen (d.h. ein Feld ist immer frei). Die acht Plättchen sind zunächst unsortiert und sollen durch geschicktes Verschieben in eine aufsteigende Reihenfolge (d.h. den Zielzustand) gebracht werden. In Abbildung 2.1 ist dies visualisiert. Links befindet sich der Zielzustand und rechts daneben zwei Spielzustände S1 und S2, wobei S2 aus S1 durch Verschieben der 8 entsteht.
Betrachtet werden folgende beide Heuristiken:

- **f1:** Diese Heuristik zählt die Anzahl der Plättchen, die nicht auf der Zielposition liegen. Gibt die Heuristik eine Null zurück, liegen alle Plättchen an der richtigen Stelle und der Zielzustand ist erreicht.[2]
- **f2:** Diese Heurisitk berechnet von jedem Plättchen den "Luftlinienabstand" zum Zielzustand, d.h. die Anzahl der Verschiebungen, die benötigt würden, lägen keine anderen Plättchen im Weg. Anschließend wird über alle einzelnen Werte der Plättchen aufaddiert, wodurch die Bewertung des Zustands entsteht.

Bei der Anwendung der Bewertungsfunktionen auf die Spielzustände entsteht folgendes Ergebnis:

[1] Entnommen aus [SS14]
[2] Die entsprechende maximierende Heuristik arbeitet analog, indem die Anzahl der Plättchen gezählt werden, die auf der richtigen Position liegen.

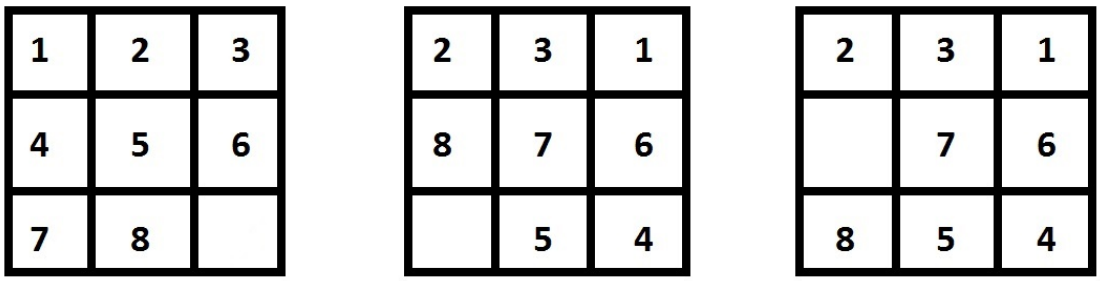

Abbildung 2.1: 8-Puzzle: Zielzustand(l), S1(m), S2(r)

```
1  while (true) {
2      if (L ist leer) {
3          return null;
4      }
5      K:= L[0];
6      R := L \ L[0];
7      if (K ist Zielknoten) {
8          return K;
9      } else: {
10         N(K) := geordnete Liste der Nachfahren von K;
11         L := N(K) ++ R;
12     }
13 }
```

Listing 2.1: Tiefensuche (Pseudocode)

- **f1(S1) = 7**, da sieben Plättchen an der falschen Stelle liegen (nur die 6 ist korrekt)
- **f2(S1) = 2 + 1 + 1 + 3 + 1 + 0 + 2 + 2 = 12**
- **f1(S2) = 7**
- **f2(S2) = 2 + 1 + 1 + 3 + 1 + 0 + 2 + 1 = 11**

Auffällig ist, dass f1 für beide Zustände dieselbe Bewertung liefert, d.h. beide Zustände als gleich gut bewertet. Dagegen erkennt f2 eine Verbesserung von S2 zu S1. Folglich ist f2 genauer als f1 und besitzt somit eine höhere Güte als Bewertungsfunktion.

2.4 Tiefensuche

Die Tiefensuche ist ein einfacher Suchalgorithmus ohne Heuristik oder Bewertungsfunktion (nicht informierte Suche). Sie wird stets den ersten Nachfahren eines Knotens expandieren und folglich zuerst in der Tiefe suchen.

```
while (true) {
    if (L ist leer) {
        return null;
    }
    K:= L[0];
    R := L \ L[0];
    if (K ist Zielknoten) {
        return K;
    } else: {
        N(K) := geordnete Liste der Nachfahren von K;
        L := R ++ N(K);
    }
}
```

Listing 2.2: Breitensuche (Pseudocode)

Datenstrukturen: Liste L von Knoten
Eingabe: L initialisiert mit den Startknoten
Algorithmus: Siehe Listing 2.1

Dieser Suchalgorithmus ist für die später verwendeten Agenten nicht sinnvoll, da er wegen der fehlenden Heuristik oder Bewertungsfunktion zu langsam ist. Auch ist er unvollständig für unendlich tiefe Graphen. Er ist jedoch Grundlage für viele komplexere Suchalgorithmen und wurde deshalb an dieser Stelle erklärt.

2.5 Breitensuche

Die Breitensuche ist ein einfacher Suchalgorithmus ohne Heuristik oder Bewertungsfunktion (nicht informierte Suche). Sie wird stets die Knoten einer Ebene expandieren und erst danach die Kinder der Knoten betrachten. Folglich sucht sie zuerst in der Breite.

Datenstrukturen: Liste L von Knoten
Eingabe: L initialisiert mit den Startknoten
Algorithmus: Siehe Listing 2.2

Die Breitensuche ist nicht vollständig für Graphen mit unendlicher Breite (d.h. Knoten mit unendlicher Anzahl an Kindern), was jedoch ein seltener Spezialfall ist, der in diesem Kontext nicht auftritt. Die Breitensuche wird somit als vollständig eingestuft. Trotzdem ist dieser Suchalgorithmus für die später verwendeten Agenten nicht sinnvoll, da er wegen der fehlenden Heuristik oder Bewertungsfunktion zu langsam ist. Er ist jedoch Grundlage für viele komplexere Suchalgorithmen und wurde deshalb an dieser Stelle erklärt.

```
1  while (true) {
2      if (L ist leer) {
3          return null;
4      }
5      K:= L[0];
6      R := L \ L[0];
7      if (K ist Zielknoten) {
8          return K;
9      } else: {
10         N(K) := absteigend nach h sortierte Liste der Nachfahren von K;
11         Entferne schon besuchte Knoten aus N(K);
12         L := R ++ N(K);
13     }
14 }
```

Listing 2.3: Hillclimbing-Algorithmus (Pseudocode)

2.6 Hillclimbing-Algorithmus

Der Hillcombing-Algorithmus ist eine informierte Suche. Er enspricht einer gesteuerten Tiefensuche mit Sharing.

Datenstrukturen: Liste L von Knoten, Knotenbewertungsfunktion h
Eingabe: L initialisiert mit den Startknoten, absteigend sortiert nach h
Algorithmus: Siehe Listing 2.3

Da der Hillclimbing-Algorithmus einer abgewandelten Tiefensuche entspricht, besitzt er auch ihre Schwächen: Der Algorithmus ist nicht vollständig. Desweiteren findet die Suche zunächst in der Tiefe statt, wodurch lokale Minima/Maxima[3] nur langsam verlassen werden.

2.7 Best-First-Suche

Die von Judea Pearl im Buch [Pea84] beschriebene Best-First-Suche ist eine informierte Suche. Sie entspricht einer gesteuerten Tiefensuche mit Sharing und ist somit nicht vollständig.

Datenstrukturen: Liste L von Knoten, Bewertungsfunktion h
Eingabe: L initialisiert mit den Startknoten, absteigend sortiert nach h
Algorithmus: Siehe Listing 2.4

Die Vorgehensweise ist sehr ähnlich zum Hillclimbing-Algorithmus, mit dem Unterschied, dass nicht ausschließlich die Nachfahren eines Knotens weiter expandiert werden, sondern alle offenen Knoten sortiert werden. Dadurch wird die

[3] Minima, falls der Algorithmus mit einer minimierenden Bewertungsfunktion arbeitet, andernfalls Maxima

```
while (true) {
    if (L ist leer) {
        return null;
    }
    K:= L[0];
    R := L \ L[0];
    if (K ist Zielknoten) {
        return K;
    } else: {
        N(K) := Liste der Nachfahren von K;
        Entferne schon besuchte Knoten aus N(K);
        L := N(K) ++ R;
        Sortiere L absteigend nach h;
    }
}
```

Listing 2.4: Best-First-Suche (Pseudocode)

```
while (Open nicht leer) {
    Wähle Knoten K aus Open mit minimalem f(K) = g(K) + h(K);
    if (K ist Zielknoten) {
        break;
    }
    N(K) := Liste der Nachfahren von K;
    Closed.add(K);
    Open.remove(K);
    for (Knoten L : N(K)) {
        if (!(L in Open oder Closed && g(K) + c(K, L) > g(L))) {
            g(L) = g(K) + c(K, L);
            Open.add(L);
            Closed.remove(L);
        }
    }
}
if (Open ist leer) {
    return null;
} else {
    return N;
}
```

Listing 2.5: A*-Algorithmus (Pseudocode)

Schwäche des Hillcimbings, lokale Minima/Maxima nur langsam zu verlassen, beseitigt.

2.8 A*-Algorithmus

Der A*-Algorithmus wurde erstmals von Hart et al. im Jahre 1968 beschrieben ([HNR68]). Dieser Algorithmus unterscheidet sich bereits im Grundsatz von den bisherigen Algorithmen. Bisher bestand das Problem in der Suche eines Zielknotens. Der A*-Algorithmus löst jedoch das Problem des optimalen Weges zu einem Zielknoten.

Datenstrukturen:

- Menge Open von Knoten
- Menge Closed von Knoten
- Heuristik h
- Kantenkostenfunktion c
- Wert g(K) für jeden Knoten K

Eingabe:

- Open initialisiert mit den Startknoten
- g(S) := 0 für alle Startknoten S

Algorithmus: Siehe Listing 2.5

Der A*-Algorithmus nutzt eine Heuristik, um den Abstand zum Ziel zu schätzen und zählt folglich zu den informierten Suchen. Dabei nutzt er einen Wert g(K) für jeden Knoten K, welcher für den zur Zeit kürzesten Weg nach K steht. Zudem nutzt er zwei Mengen von Knoten: Open und Closed. Die Knoten in der Menge Open müssen noch betrachtet werden, wobei die Knoten in der Menge Closed abgeschlossen sind. Dabei ist es jedoch möglich, einen bereits geschlossenen Knoten wieder zu öffnen.
In jedem Schritt prüft und expandiert der Algorithmus den Knoten K, bei dem der aktuelle Weg plus der geschätzte Weg zum Ziel minimal ist. Für jeden Kindknoten L von K wird überprüft, ob dieser schon bekannt ist und falls ja, ob ein kürzerer Weg zu ihm gefunden wurde. Falls ja, wird für L der Wert g(L) korrigiert und auf den Wert g(K) plus Kantenkosten von K nach L gesetzt. In diesem Fall muss der Knoten L noch einmal expandiert werden und wird somit in die Menge Open geschoben und ggf. aus Closed entfernt. Der Algorithmus besucht die Knoten folglich auf Basis von Schätzungen und bisherigen Erkenntnissen und korrigiert letztere kontinuierlich selbst. Sollte der Algorithmus mit der leeren Menge Open enden, wurde das Ziel nicht gefunden. Andernfalls bricht der Algorithmus mit dem Zielknoten ab.
Die Schwierigkeit des A*-Algorithmus liegt in der Bereitstellung einer guten Heuristik, da dieser Algorithmus nur so gut arbeiten kann, wie die gewählte Heuristik es erlaubt. Bei der Ausführung zweier A*-Algorithmen mit gleichen Parametern, jedoch in der Güte stark unterschiedlichen Heuristiken, entstehen signifikante Unterschiede in der Laufzeit.
Zudem gibt es eine weitere Einschränkung: Die Lieferung korrekter Ergebnisse wird nur dann gewährleistet, wenn die Heuristik h unterschätzend ist. Sollte die Heuristik h zudem monoton sein, ist eine zusätzliche Beschleunigung gewährleistet, da Knoten dann nicht mehrfach expandiert werden müssen (d.h. beim Expandieren eines Knoten N wird immer der optimale g-Wert für N gesetzt). Eine Heuristik h ist genau dann monoton, wenn erstens die Kosten nie überschätzt werden und zweitens, wenn der Schätzwert eines Knotens k nicht größer als der Schätzwert des Nachfolgeknotens k' plus tatsächlicher Kosten von k nach k' ist.

```
t := 0;
while (true) {
    Führe a mit p aus bis zu einer maximalen Tiefe von t;
    if (Zielknoten gefunden) {
        return Zielknoten;
    } else: {
         t++;
    }
}
```

Listing 2.6: Allgemeine Iterative Suche (Pseudocode)

Beispiel: Eine einfache und effiziente unterschätzende Heuristik für das kürzeste Wege Problem liegt in der Luftlinienentfernung vom aktuellen Knoten zum Zielknoten.

2.9 Iterative Suche

Die Iterative Suche ist ein allgemeines Konstrukt, welches auf verschiedene Suchalgorithmen angewendet werden kann. Dabei wird der ausgewählte Suchalgorithmus wiederholt auf einem zunehmend größeren Suchraum angewendet.

Datenstrukturen: Integer t; Suchalgorithmus a; Datenstrukturen von a
Eingabe: a inklusive dessen Parameter p
Algorithmus: Siehe Listing 2.6

Die iterative Suche stellt einen Kompromiss dar: Zum Preis von Laufzeit wird die Vollständigkeit einer Suche gewährleistet. Die Idee ist die Verwendung einer Tiefenschranke t, welche den Suchraum begrenzt. Bei $t = 3$ können beispielsweise maximal drei Generationen von Knoten (Wurzel + Kinder + Kindeskinder) betrachtet werden. Durch die Begrenzung des Suchraumes wird ein Suchalgorithmus vollständig, da es keine unendlichen Pfade gibt. Anschließend wird die Tiefenschranke t erhöht und die Suche erneut gestartet.
Dieses Vorgehen zur Gewährleistung der Vollständigkeit findet unter anderem bei Algorithmen wie Tiefensuche, Hillclimbing oder Best-First-Suche Anwendung.

2.10 Model-View-Controller

"The Model-View-Controller metaphor is a way to design and implement interactive application software that takes advantage of modularity, both to help the conceptual development of the applications, and to allow pieces already developed for one

application to be reused in a new application."[4]

Eine wichtige Grundlage einer Implementierung ist die Trennung von Daten, Steuerung und Benutzeroberfläche. Die Daten stellen die Grundlage jeder Anwendung dar und sollten deshalb unabhängig von der restlichen Anwendung definiert werden. Die Steuerung dient der Verarbeitung der Daten, somit darf (und muss) die Steuerung auf diese zugreifen können. Der Steuerung selbst ist es jedoch nicht erlaubt, auf Konstrukte der Benutzeroberfläche zuzugreifen. Die Benutzeroberfläche selbst ist die hierarchisch höchste Komponente und darf somit auf die Logik und Daten zugreifen. Dabei sind die Aufgaben der Benutzeroberfläche jedoch auf die Visualisierung beschränkt.
Dieses Konzept wurde das erste Mal im Jahr 1979 von Trygve Reenskaug beschrieben. Es bietet eine große Flexibilität und Austauschbarkeit. Auf jede bestehende Logik kann somit eine beliebige Benutzeroberfläche gesetzt werden, ganz ohne Anpassungen an der Logik vornehmen zu müssen. Gleiches gilt für die Daten und die Logik. Grundsätzlich beeinträchtigen Verschränkungen der Komponenten die Veränderbarkeit, da zu viele Anpassungen getroffen werden müssen.
Ist die Funktionsweise einer höheren Komponente abhängig von einem Trigger einer niedrigeren Komponente (Beispiel: Die Benutzeroberfläche wartet auf den Aufruf einer bestimmten Funktion der Steuerung), ist es streng untersagt, von der niedrigeren Komponente die höhere Komponente aus anzusprechen. Stattdessen bietet die niedrigere Komponente ein Interface an, welches andere Komponenten implementieren und dann bei der niedrigeren Komponente registrieren können. Wird ein Event ausgelöst, triggert die niedrigere Komponente für alle registrierten Interfaces die entsprechende Methode.

2.11 Singleton-Klasse

Eine Singleton-Klasse beschreibt in der objektorientierten Programmierung eine Klasse von der nur eine Instanz erzeugt werden kann. Typischerweise erfolgt die Implementierung einer Singleton-Klasse in drei Schritten:

1. Setzen des Konstrukors auf "private"
2. Hinzufügen einer privaten Klassenvariable, welche die einzige Klasseninstanz verwaltet
3. Hinzufügen einer öffentlichen Methode, welche die Klasseninstanz zurückgibt

Singleton-Klassen ähneln somit statischen Klassen, besitzen diesen gegenüber jedoch mehrere Vorteile:

[4]Krasner, G. E. & Pope, S. T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80 JournalOfObjectOrientedProgramming, 1988, Seite 48

- Singletons können Interfaces implementieren und von anderen Klassen erben und somit auch polymorph sein
- Eine Singleton-Referenz kann einer anderen Methode als Parameter übergeben werden, d.h. sie wird als normales Objekt behandelt
- Bei Bedarf ist eine Änderung zu einer Nicht-Singleton-Klasse einfach umsetzbar
- Singletons setzen das Prinzip der Objektorientierung um

Kapitel 3

Anforderungsanalyse

Nachfolgend werden die für die Anwendung geltenden Anforderungen beschrieben und analysiert.

- **Datenmodell:** Das Datenmodell muss flexibel und austauschbar sein. Dies bedeutet, dass die Modellierung eine beliebige Definition der Spielinhalte ermöglichen muss. Ferner muss es möglich sein, ein solches definiertes Datenmodell durch ein anderes zu ersetzen, ohne gleichzeitig die restliche Anwendung anpassen zu müssen. Folglich muss ein Konstrukt geschaffen werden, durch das beliebige Datenmodelle erzeugt werden können. Ziel dieser Anforderung ist, eine möglichst hohe Flexibilität der Anwendung und des Nutzungsszenarios zu bieten.
- **Spielerzahl:** Das Spiel soll mit mehreren Spielern spielbar sein.
- **Spielziel:** Im Datenmodell muss ein klares Ziel definiert werden können. Dieses Ziel soll global sein und somit für alle Spieler gelten. Der erste Spieler, der das Ziel erreicht, gewinnt. Diese Anforderung ist entscheidend für die Vergleichbarkeit der Agenten, da diese ohne einen Bezugspunkt nicht beurteilt werden können.
- **GUI:** Die Anwendung darf nicht rein konsolengesteuert sein, sondern muss eine Benutzeroberfläche (GUI) anbieten. Diese Benutzeroberfläche muss dem Spieler alle zum Spielen relevanten Informationen zur Verfügung stellen. Ziel ist die einfache Steuerung und Nutzung der Anwendung.
- **Agenten:** Es müssen Agenten erstellt werden, welche am Spiel teilnehmen können. Dabei werden mehrere Anforderungen an die Agenten gestellt.
  - Aus Sicht des Spiels dürfen die Agenten nicht anders als menschliche Spieler sein, d.h. diese dürfen nicht anders behandelt werden. Dieses Kriterium ist besonders auf implementierungstechnischer Ebene umzusetzen. Ziel des Kritieriums ist die Anpassungsmöglichkeit der Anwendung an mehrere menschliche und künstliche Spieler, wobei nur die Anpassungsmöglichkeit gefordert wird und nicht die Umsetzung.
  - Agenten müssen rational sein, das heißt ein menschlicher Spieler darf prinzipiell keinen Unterschied zwischen anderen menschlichen Spielern und

Agenten erkennen. Dies ist ein wichtiges Kriterium, um die Güte der Agenten zu gewährleisten.

– Es ist nicht zulässig, dass menschliche Spieler auf die Berechnungen eines Agenten warten. Folglich muss der Agent so entwickelt werden, dass dieser bei Bedarf sofort eine Entscheidung treffen kann und sich mit seiner gegebenen Rechenzeit an die menschliche Spielgeschwindigkeit anpasst.

Kapitel 4

Modellierung

4.1 Spielwelt

Eine Spielwelt sei definiert durch:

- Eine Menge von Spielern
- Eine Menge von Items
- Eine Menge von Aktivitäten
- Einen neutralen Händler
- Eine Siegbedingung (Zielzustand eines Spielers)

Der zeitliche Verlauf des Spiels ist durch Runden modelliert. Innerhalb einer Runde steht die Zeit still, erst beim Übergang einer Runde in die Nächste werden alle Veränderungen der Spielwelt gleichzeitig umgesetzt. Dabei treten alle Änderungen ausschließlich durch die Aktivitäten der Spieler in Kraft, welche jeweils eine Runde lang andauern.
Ziel des Spiels ist es, den Spielercharakter durch eine geschickte Auswahl an Aktivitäten als Erster in den Zielzustand zu führen.

4.2 Items

Ein Item sei definiert durch:

- Einen Namen (String)
- Einen Preis (Zahl)

Der Preis eines Items gilt global für alle Items der gleichen Klasse und ist stückweise zu verstehen. Die Preise richten sich nach dem Bestand des Händlers und werden von diesem, entsprechend dem Prinzip von Angebot und Nachfrage, reguliert.

4.3 Attribute

Ein Attribut ist ein Wert, welcher der Beschreibung eines Spielers dient. Es sei definiert durch:

- Einen Namen (String)
- Einen Wert (Zahl)
- Einen Startwert (Zahl)

Der Startwert dient der Initialisierung des Wertes zu Beginn des Spiels. Ein hoher Wert eines Attributes ist nicht zwangsweise positiv (oder negativ). Erst durch Interpretation des aktuellen Spiels lässt sich ein Rückschluss auf die Güte eines Attributes ziehen.

4.4 Aktivitäten

Eine Aktivität stellt eine atomare Handlung eines Spielers dar. Sie sei definiert durch:

- Einen Namen (String)
- Eine Menge von Voraussetzungen
- Eine Menge von Effekten

Die Voraussetzungen einer Aktivität müssen von einem Spieler erfüllt sein, damit dieser die Aktivität ausführen kann. Dabei beziehen sich die Voraussetzungen direkt und ausschließlich auf den Zustand des Spielers, also den numerischen Werten der Attribute, der Items sowie des Goldes.
Bei Ausführung einer Aktivität werden alle Effekte gleichzeitig umgesetzt. Ein Effekt kann den Zustand des ausführenden Spielers und / oder des Händlers beeinflussen.

4.5 Spieler

Ein Spieler sei definiert durch:

- Einen Namen (String)
- Eine Abbildung: Item $\rightarrow$ Anzahl
- Eine Abbildung: Attribut $\rightarrow$ Anzahl
- Eine Anzahl an Gold

Die beiden Abbildungen dienen als Zähler für das jeweilige Attribut, bzw. Item. Auf diese Weise ist es möglich, das gleiche Attribut oder Item mehr als nur einmal zu besitzen. Zu jedem Zeitpunkt sind in der Item-Abbildung alle existierenden Items enthalten, sowie in der Attribut-Abbildung alle existierenden Attribute enthalten. Die Werte der Abbildungen sind zu Spielbeginn für Items mit 0 und für Attribute mit dem Startwert initialisiert.

4.6 Händler

Ein Händler stellt die einzige reale Interaktionsmöglichkeit eines Spielers dar, um die Spielwelt zu beeinflussen, da ausschließlich über diesen Handel betrieben werden kann. Ein Händler sei definiert durch:

- Eine Map: Item → Anzahl
- Eine Anzahl an Gold

Ein Händler ist ein Spieler ohne Attribute und ohne die Möglichkeit, Aktivitäten auszuführen - er ist folglich vollkommen passiv. Alle in Aktivitäten beschriebenen Effekte, welche in irgendeiner Weise den Handel beschreiben, verändern den Zustand des Händlers.

4.7 Beispielwelt

Folgend soll beispielhaft eine Spielwelt mittlerer Komplexität beschrieben werden.

- Spieler = {Spieler1, Spieler2, Spieler3}
- Items = {Getreide, Mehl, Brot, Fleisch, Wurst}
- Attribute = {Hunger, Skill: BrotBacken, Skill: FleischZubereiten, Skill: GetreideErnten, Skill: Handeln, Skill: Jagen, Skill: MehlMahlen}
- Aktivitäten = {BrotBacken3, BrotBacken6, BrotBacken9, Essen: Brot, Essen: Wurst, FleischZubereiten1, FleischZubereiten2, FleischZubereiten3, GetreideErnten1, GetreideErnten2, GetreideErnten3, Jagen1, Jagen2, Jagen3, MehlMahlen1, MehlMahlen2, MehlMalen3, Kaufen: Brot1, Kaufen: Brot3, Kaufen: Brot9, ... , Kaufen: Wurst9, Verkaufen: Brot1, Verkaufen: Brot3, Verkaufen: Brot9, ... , Verkaufen: Wurst9}
- Händler = {{Getreide → 100, Mehl → 100, Brot → 100, Fleisch → 100, Wurst → 100}, {10000 Gold}}
- Siegbedingung = “Erreiche 1000 Gold”

Die Aktivitäten sind intuitiv und dem Namen entsprechend aufgebaut. Eine Zahl am Ende einer Aktivität drückt die Menge des zu verändernden Items aus. BrotBacken3 erzeugt beispielsweise 3 Brote. Jede Aktivität erhöht das zugehörige Attribut "Skill: [Aktivitätsname]" um 1. Jedes Attribut "Skill" startet mit dem Wert 1 und hat einen maximalen Wert von 10. Jede Aktivität ist in dreifacher Ausführung vorhanden, jeweils mit unterschiedlicher Itemzahl. Um die höherwertigen Aktivitäten zu benutzen, muss der zugehörige "Skill" entsprechend hoch sein: 5 für die zweite Aktivität und 10 für die dritte Aktivität. Alle Aktivitäten, die "Kaufen" oder "Verkaufen" im Namen enthalten, sind dem Attribut "Skill: Handeln" zugeordnet.
MehlMahlenX verbraucht Getreide im Verhältnis 3 Getreide : 1 Mehl, während BrotBackenX für jedes verbrauchte Mehl drei Brote erzeugt. Die Aktivität FleischZubereiten erzeugt eine Wurst für drei Einheiten Fleisch.
Jede Aktivität, die nicht "Kaufen", "Verkaufen" oder "Essen" im Namen hat, erhöht bei Ausführung das Attribut "Hunger" um 1. Sobald "Hunger" den Wert 10 erreicht hat, sind die einzigen ausführbaren Aktivitäten diese mit "Verkaufen" im Namen sowie "Kaufen: BrotX", "Kaufen: WurstX", "Essen: Brot" und "Essen: Wurst". Die Aktivität "Essen: Brot" reduziert das Attribut "Hunger" um 5 und verbraucht ein Brot, "Essen: Wurst" ist analog, reduziert den "Hunger" jedoch um 10.
Die Voraussetzungen der Attribute entsprechen den jeweiligen benötigten Items oder Attributen. Zudem ist Voraussetzung beim Handeln, dass sowohl Spieler als auch Händler ausreichend Gold und Items zur Verfügung haben.

Kapitel 5

Agenten

5.1 Modellierung des Spiels als Suchproblem

Das Spiel ist als gerichteter Graph $G = (V, E)$ beschrieben. Dabei wird für jeden Agenten ein eigenständiger und unabhängiger Suchgraph erzeugt. Dieser Graph beschreibt ausschließlich die Aktivitäten und Entwicklungen eines einzelnen Agenten und berücksichtigt die anderen Agenten nur indirekt. Dies ist möglich, da Agenten nicht in direktem Kontakt zueinander stehen, sondern alle Möglichkeiten des Kaufens und Verkaufens (d.h. der Interaktion) ausschließlich über den neutralen Händler geschehen. Aus diesem Grund reicht es aus, die erwartete Preisentwicklung zu berücksichtigen.
Jeder Knoten V stellt einen möglichen Zustand des entsprechenden Agenten dar. Eine Kante E steht für eine mögliche Aktivität. Die Wurzel des Graphen ist der aktuelle Zustand des Agenten im realen Spiel. Alle von diesem Agenten ausführbaren Aktivitäten werden von der Wurzel als Kanten zu den Kindern geführt, wobei jedes Kind der durch die Aktivität der Kante modifizierte Spielerzustand der Wurzel ist. Der Graph setzt sich auf diese Weise fort, d.h. die Kindknoten der Wurzel besitzen erneut Kanten mit allen möglichen Aktivitäten, die erneut zu den modifizierten Spielerzuständen führen usw. Folglich ist der Suchgraph unendlich. Die Tiefe des Graphen gibt die Runde des Spiels an, bis zu welcher vorausgerechnet wurde.
Zielknoten sind durch das Ziel des Spiels definiert: Jeder Knoten der einen Spielerzustand repräsentiert, in dem der Spieler das Ziel erreicht hat, gilt als Zielknoten.

5.2 Forderungen an den Suchalgorithmus

Der Graph ist unendlich groß, aus diesem Grund ergibt sich die Forderung der Vollständigkeit des Algorithmus. Dies bedeutet, dass jeder Knoten (bzw. Zielknoten) in endlich vielen Rechenschritten erreicht wird. Ein Algorithmus wie die Tiefensuche würde sich auf einen unendlich tiefen Ast des Suchraumes konzentrieren und nie Alternativen besuchen können, d.h. es wäre ähnlich einer Greedy-Suche, bei der nur Verbesserungen zum vorherigen Zustand möglich wären und somit Alternativen, die

einen Rückschritt benötigten, direkt ausgeschlossen würden.
Ebenfalls ergibt sich aus der Größe des Graphen die Forderung, die Suche durch eine Bewertungsfunktion / Heuristik zu optimieren. Dies ist erforderlich, da eine naive Suche über den vollständigen Graphen zu Laufzeitproblemen führen würde.
Eine weitere Forderung besteht in der Ausgabe des Pfades zum Zielknoten. Das bedeutet, dass der Suchalgorithmus nicht nur einen Zielknoten finden, sondern auch den Weg dorthin speichern muss.
Zuletzt muss der Algorithmus dazu in der Lage sein, bei Nichtfinden eines Zielknotens innerhalb eines gegebenen Zeitlimits, zumindest den bis dahin besten gefundenen Knoten (und Pfad) auszugeben. Diese Forderung ist deshalb wichtig, da wegen der Größe des Suchraumes die Wahrscheinlichkeit dafür, einen Zielknoten zu finden, zu Beginn des Spiels sehr gering ist. Erst im Verlauf des Spiels wird das Finden eines Zielknotens wahrscheinlicher. Insofern ist diese Forderung essentiell für die Funktionsweise des Agenten.
Die letzten beiden genannten Forderungen sollten von jedem Suchalgorithmus durch geringe Modifikationen erfüllt werden können.

Anzumerken ist noch, dass vom Suchalgorithmus nicht gefordert (aber auch nicht verboten) wird, den kürzesten Pfad zum Ziel zu finden, wie dies beispielsweise im A*-Algorithmus der Fall ist. Grund dafür ist zum einen die Beschaffenheit des Graphen. Es ist, wie bereits erläutert, unwahrscheinlich, den Zielknoten vor dem unmittelbaren Ende des Spiels zu finden. Insofern liegt der Fokus im effizienten Finden des Zielknotens (bzw. des bestmöglichen Knotens) und nicht im Finden des kürzesten Weges. Zum anderen werden die Agenten jede Runde eine Aktion ausführen und haben somit jede Runde einen veränderten, fortgeschrittenen Ausgangspunkt als Grundlage der Suche. Folglich wird jede Runde über einen Teil des Weges zum Ziel bereits entschieden. Wenn eine Suche einen Zielzustand erreicht, ist der Agent also bereits den Großteil der Gesamtstrecke gelaufen (d.h. die Suche lief nur noch über den Restweg vom Agenten zum Ziel), wodurch die Auswirkungen eines kürzesten Pfades aus Gesamtsicht minimal werden.
Sollten Kantenkosten auf 1 gesetzt werden (und für die Anzahl der Runden stehen), ist zudem gewährleistet, dass die gefunden Wege für iterative Verfahren minimal sind, wenn für jede Iteration genau eine Ebene an Knoten mehr betrachtet wird. Hätte ein kürzerer Weg zum Zielknoten existiert, dann hätte dieser bereits in der vorherigen Iteration gefunden werden müssen, da der vollständige mögliche Suchraum in jeder Iteration betrachtet wird. Insofern kann kein kürzerer Weg existieren.

5.3 Auswahl des Suchalgorithmus

Zur Auswahl stehen nachfolgende, in Kapitel 2 vorgestellte Algorithmen. Dabei ist das Optimierungsproblem für alle das Gleiche: Es ist als erstes der Zielzustand zu

erreichen, bei dem die Spielfigur 1000 Gold besitzt. Anders formuliert: Finde den Weg zu dem Zielknoten "Spielfigur besitzt 1000 Gold", wobei die Kantenkosten immer 1 sind.

- **A*-Algorithmus:** Dieser Algorithmus optimiert die Suche durch Nutzung einer Heuristik zur Berechnung des Abstandes vom aktuellen Knoten zum Zielknoten. Das Problem daran ist, dass der Algorithmus nur dann korrekt arbeitet, wenn es sich um eine unterschätzende Heuristik[1] handelt. Was in Situationen wie der Berechnung der Distanz durch Verwendung der Luftlinie als Heuristik sehr einfach und intuitiv erscheint, erweist sich für dieses Spiel als großes Problem. Es kann in keiner Situation gewährleistet werden, dass eine Heuristik unterschätzend (oder monoton) ist. Würde beispielsweise der Goldwert eines Spielers (bei Spielziel: erreiche X Gold) als Abstand verwendet werden, wäre die resultierende Frage, wie viel Gold ein Spieler jede Runde verdient. Da sich die Preise jedoch stets ändern und ein Spieler zudem manche Runden gar kein Gold verdienen kann, während er in einem anderen Zug viele Items verkauft und mit einem Schlag viel Gold erhält, lässt sich keine vernünftige Antwort auf die gestellte Frage finden.
- **Hillclimbing-Algorithmus:** Dieser Algorithmus bewertet jeden Knoten mit einer Bewertungsfunktion und wählt als nächsten zu untersuchenden Knoten stets das Kind mit dem besten Wert aus. Damit ist der Algorithmus jedoch nicht vollständig und entspricht somit nicht den Anforderungen.
- **Iterativer Hillclimbing-Algorithmus:** Die Funktionsweise ist analog zum Hillclimbing-Algorithmus, mit dem Unterschied, dass die maximale Suchtiefe für eine Ausführung des normalen Hillclimbing-Algorithmus begrenzt wird und erst nach Absuchen dieses Suchraumes die Suchtiefe erhöht und iterativ fortgefahren wird. Auf diese Weise wird die Vollständigkeit sichergestellt. Durch die Iteration wird ein Teil des Suchraumes mehrfach besucht, was jedoch nur einen geringen Teil der Gesamtlaufzeit ausmacht.
- **Best-First-Algorithmus:** Dieser Algorithmus bewertet jeden Knoten mit einer Bewertungsfunktion und untersucht den Knoten mit dem besten Wert zuerst. Das bedeutet, dass anders als beim Hillclimbing global der beste Knoten verwendet wird und nicht nur lokal der beste Nachfolger. Folglich können lokale Maxima schneller verlassen werden. Doch auch hier besteht das Problem, dass der Best-First-Algorithmus nicht vollständig ist und somit den Anforderungen nicht entspricht.
- **Iterativer Best-First-Algorithmus:** Analog zum iterativen Hillclimbing wird auch hier die Tiefe begrenzt und schrittweise erhöht, mit dem Unterschied, dass bei jeder Iteration der Best-First-Algorithmus ausgeführt wird. Auf diese Weise wird erneut die Vollständigkeit der Suche sichergestellt.

[1] Effizienter ist die Verwendung einer monotonen Heuristik.

Die einzigen in Frage kommenden Algorithmen sind somit das iterative Hillclimbing sowie die iterative Best-First-Suche. Beide Algorithmen sind ähnlich, die Auswahl fällt jedoch auf den letzteren, da dort lokale Maxima schneller verlassen werden können und die Suche somit beschleunigt wird. Es existieren weitere Algorithmen, die betrachtet werden könnten, doch diese nutzen entweder eine Heuristik ähnlich dem A*-Algorithmus, so dass sie hier unbrauchbar sind, oder sind lediglich Varianten der Best-First-Suche.

5.4 Auswahl der Bewertungsfunktion

Die iterative Best-First Suche benötigt eine Bewertungsfunktion, um die Güte eines Knotens beurteilen zu können. Dabei gibt es grundsätzlich zwei Arten, eine solche Funktion zu berechnen. Die erste Variante verwendet lediglich den Spielerzustand des Knotens, die zweite Variante bezieht sich auf die Aktivitäten, also die Kanten, die zum aktuellen Knoten geführt haben. Während die erste Variante folglich unabhängig vom Pfad zum Knoten ist, beruht die Nutzung der zweiten Variante auf Kantengewichten. Zunächst sollen Bewertungsfunktionen der ersten Variante betrachtet werden. Dabei wird die Beispielmodellierung aus Abschnitt 4.7 zu Grunde gelegt.

- **Gold:** Da das Ziel der Beispielmodellierung das Erreichen eines definierten Goldwertes ist, stellt die Verwendung des Goldwertes des zu betrachtenden Knotens eine einfach berechenbare Bewertungsfunktion dar.
- **Score:** Der Score soll eine generelle Bewertung über den Spielerzustand geben und dabei alle Attribute, Items und den Goldwert berücksichtigen. Beispielsweise kann ein Spieler sehr wenig Gold besitzen, dem Ziel jedoch durch den Besitz von wertvollen Items trotzdem sehr nahe sein. Der Score berücksichtigt solche Situationen. Er setzt sich zusammen aus dem Gold, dem Goldwert der Items sowie einem definierten Wert für jeden Attributspunkt. Die Attribute werden deshalb berücksichtigt, da sie sich auf die Effektivität der Spieleraktivitäten auswirken. Dabei kann ein Attributspunkt auch negativ sein, den Score also verschlechter (z.B. beim “Hunger”-Attribut).
- **Differenz:** Diese Bewertungsfunktion definiert den Goldunterschied allgemein als Differenz und ist aus diesem Grund auch für andere Ziele zu verwenden. In diesem Fall würde der vollständige Agent mit dem Zielzustand verglichen, statt nur den Goldwert zu überprüfen. Das Ziel ist genau dann erreicht, wenn die Differenz Null ist.
- **Gold / Rounds:** Ein Blick alleine auf das Gold vernachlässigt die Zeitkomponente: ein Knoten mit 100 Gold in der 5. Runde besitzt wahrscheinlich bessere Nachfahren als ein Knoten mit 110 Gold in der 15. Runde. Diese Bewertungs-

funktion berücksichtigt dieses zeitliche Verhalten, indem der momentane Goldwert durch die aktuelle Rundenzahl geteilt wird.

- **Score / Rounds:** Auch diese Bewertungsfunktion berücksichtigt die Zeitkomponente, indem der momentane Score durch die aktuelle Rundenzahl geteilt wird.
- **Gold, Score:** Neben einstelligen Bewertungsfunktionen ist es auch möglich, eine kombinierte zweistellige Bewertungsfunktion zu generieren. Diese erhält zwei Werte, wobei ein Wert primär (Gold) betrachtet wird und erst bei Gleichstand zwischen zwei Knoten der zweite Wert (Score) hinzugezogen wird. Durch die Kombination der Bewertungen Gold und Score lassen sich Schwächen besser kompensieren.
- **Gold / Rounds, Score / Rounds:** Erneut eine zweistellige Bewertungsfunktion, welche zunächst nach Zeit / Runde optimiert und bei Gleichstand auf Score / Runde zurückgreift.

Es folgt die zweite Variante. In dieser wird eine Bewertungsfunktion mit Kantengewichten[2] betrachtet. Zunächst eine kurze Erklärung: Die Bewertungsfunktion berechnet den Wert eines Knotens durch die Summe der Kantengewichte des Pfades zum betrachteten Knoten. Die Bewertung eines Knotens beruht also auf dem Pfad mit maximalem (oder ggf. minimalem - je nach Bewertungsfunktion) Kantengewicht. Es ist folglich möglich, dass ein kürzerer Pfad (Anzahl der Kanten) schlechter bewertet wird als ein längerer Pfad, der dafür aber ein größeres Kantengewicht besitzt.

Die betrachtete Bewertungsfunktion[3] bewertet die Aktivitäten in Abhängigkeit des Spielerzustandes. Das heißt, dass die gleiche Aktivität für unterschiedliche Spielerzustände auch unterschiedliche Bewertungen erhält. Die Suche soll durch Wissen über das Spiel verbessert werden.
Die Idee dahinter ist folgende: Das vollständige Spiel wird vor Spielbeginn analysiert. Dabei wird jedes Item betrachtet und alle existierenden Möglichkeiten, wie dieses Item produziert werden kann, notiert. Dies ist rekursiv zu betrachten: benötigt ein Item zur Produktion das Vorkommen eines anderen Items, wird dessen Produktionsweg ebenfalls dem Produktionsweg des ersten Items hinzugerechnet. Da jedes Item mehrere Möglichkeiten der Produktion besitzt, entsteht so eine Multiplikation der Möglichkeiten. Dies ist zunächst rechenintensiv, findet jedoch nur einmal zu Beginn eines Spieles statt und ist deshalb zu vernachlässigen. Zu den erhaltenen Produktionspfaden werden zusätzlich die Voraussetzungen der Ausführung betrachtet (benötigte Attribute, Items oder auch Gold). Anschließend

[2] Die Bewertungsfunktionen der ersten Variante haben ebenfalls eine "Default"-Kantenbewertung, nämlich diese, bei der jede Kante genau 1 kostet. Die Kantenkosten werden jedoch nicht verwendet: Lediglich der Knoten selbst wird bewertet.
[3] Später durch den **GameAnalyzer** beschrieben

wird der Gewinn für jeden Produktionsweg berechnet. Der Gewinn setzt sich aus "(Verkaufserlös - Investitionen) / benötigte Zeit" zusammen. Es entsteht eine lange Liste mit den Produktionspfaden, Gewinn und Voraussetzungen. Diese Liste wird vor jeder Runde des Spiels mit den aktuellen Itempreisen neu sortiert, so dass an erster Stelle der Produktionsweg mit dem größten Gewinn steht.
Bei der Berechnung einer Bewertung wird anschließend der für den Spieler beste Produktionsweg der Liste ausgewählt, mit der Zusatzbedingung, dass dieser auch vom Spieler generell ausführbar sein muss (d.h. benötigte Items werden in den Produktionsschritten erzeugt). Anschließend wird der Produktionsweg näher betrachtet: Es erhält ausschließlich die Aktivität eine positive Bewertung, die möglichst weit hinten im Produktsionsweg und noch vom Spieler ausführbar ist. Auf diese Weise wird sicher gestellt, dass der Spieler sich an der korrekten Stelle des Produktionspfades befindet.

Ein Beispiel: Es seien die beiden Produktionspfade [Jagen3, FleischZubereiten1] und [GetreideErnten3, MehlMahlen1] gegeben, wobei der erste Pfad mehr Gewinn erzielt, da Fleisch zur Zeit wertvoller ist als Mehl, also in der Liste vorher steht. Der Agent besitzt bereits 3 Getreide, hat aber noch zu niedrige Attribute, um "Jagen3" auszuführen.
Zunächst wird festgestellt, dass der erste Listeneintrag nicht ausführbar ist. Anschließend führt eine Betrachung des zweiten Listeneintrages zu dem Ergebnis, dass der Agent die prinzipiellen Voraussetzungen erfüllt. Also wird der Produktionspfad [GetreideErnten3, MehlMahlen1] ausgewählt. Da der Agent bereits 3 Getreide besitzt, ist "MehlMahlen1" schon ausführbar (obwohl "GetreideErnten3" im Produktionspfad zuerst käme) und erhält somit eine positive Bewertung. Das heißt, wenn die aktuell betrachtete Aktivität (im Prozess der Generierung der Kantensumme) "MehlMahlen1" ist, dann bekommt diese einen positiven Wert, eine Aktivität wie "GetreideErnten3" dagegen einen neutralen Wert.

Ziel ist es, mit dieser Bewertungsfunktion den stets individuell für den Agenten wirtschaftlich am sinnvollsten (d.h. den Gewinn maximierenden) Pfad durch den Graphen zu wählen.
Mathematischer formuliert: Die Bewertungsfunktion wählt den Knoten k aus, welcher die Summe der Kantenwerte des Weges zu k maximiert. Der Wert einer Kante entspricht dabei dem Erlös in Gold der zugeordneten Aktivität, d.h. Wert der produzierten Güter minus dem Wert der aufgebrachten Kosten / Güter.

Anders als bei den Algorithmen kann bei den Bewertungsfunktionen im Vorfeld keine klare Aussage über die Güte der verschiedenen Bewertungen getroffen werden. Folglich kann keiner Bewertungsfunktion der Vorrang gegenüber einer anderen gegeben werden. An dieser Stelle findet somit die eigentliche Gegenüberstellung der Agenten statt: Es ist zu prüfen, welche der vorgestellten Bewertungsfunktionen im Schnitt die besten Ergebnisse erzielt.

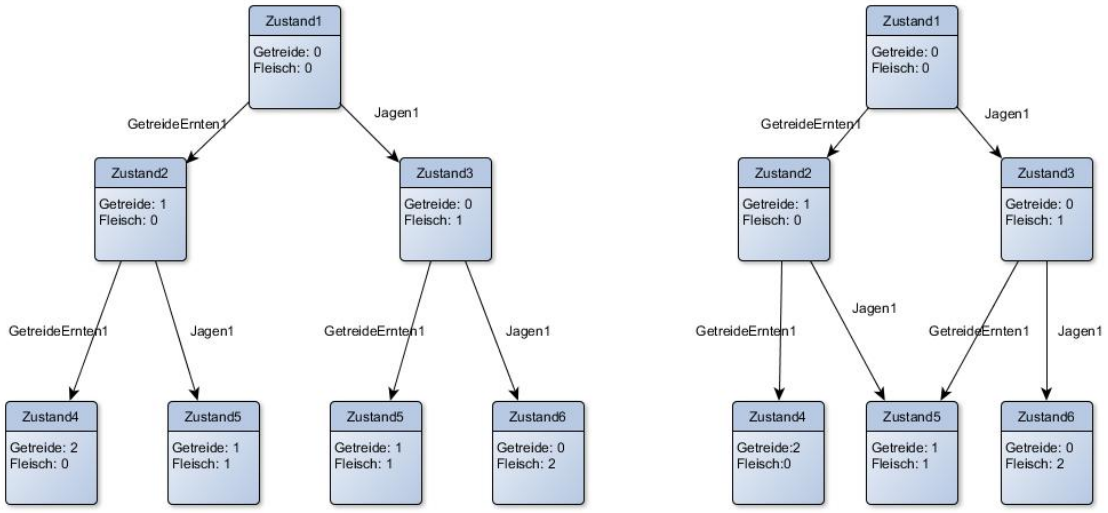

Abbildung 5.1: Suchgraph: als Baum vs als allgemeiner gerichteter Graph

5.5 Verbesserungen

Folgend sollen weitere Verbesserungen der Agenten diskutiert werden. Um Problemsituationen besser zu veranschaulichen, wird die Beispielmodellierung aus Abschnitt 4.7 zu Grunde gelegt.

5.5.1 Algorithmus

Eine Verbesserung ist mit Sharing zu erreichen. Dieses ist einfach anwendbar und bietet einen großen Laufzeitgewinn. Durch das Merken bereits besuchter Knoten müssen diese nicht mehrfach expandiert werden. Das Sharing besitzt zwar auch Nachteile: Knoten müssen gespeichert werden und es existiert ein Zeitverlust durch die Verwaltung der dazu notwendigen Strukturen. Der Speicherplatzverbrauch spielt hier jedoch keine entscheidende Rolle und aus zeitlicher Sicht wiegt das Verhindern doppelter Expansionen die Verwaltung der Strukturen mehr als auf.
Die Umsetzung erfolgt über ein Flag für jeden Knoten, welches vom Algorithmus beim ersten Besuch des Knotens gesetzt wird. Bei jedem weiteren Besuch wird anhand des Flags erkannt, dass dieser Knoten bereits betrachtet und deshalb nicht erneut expandiert werden muss.

5.5.2 Graph

Je größer der Suchgraph ist, desto schlechter wird die Laufzeit der Suche. Folglich besteht das Bestreben, den Suchraum möglichst klein zu halten. In Abschnitt 5.1 wird der Aufbau des Suchgraphen beschrieben. Nach dieser Beschreibung lässt sich

der Graph als Baum interpretieren. Da es sehr viele Aktivitäten und somit ausgehende Kanten für jeden Knoten gibt, ist ein Baum jedoch wegen seinem Wachstum in jeder Ebene denkbar ungeeignet. Folgend wird deshalb eine Anpassung beschrieben, mit der aus dem Baum ein allgemeiner gerichteter Graph konstruiert wird.
In der Beispielmodellierung existieren die Aktivitätfolgen [GetreideErnten, Jagen] und [Jagen, GetreideErnten]. Nachdem die Aktivitäten jeweils durchgeführt werden, befindet sich der Spieler für beide Folgen in demselben Endzustand. Wird der zugehörige Suchgraph als Baum beschrieben (siehe Abbildung 5.1, linkes Bild), existieren für denselben Zustand in unterster Ebene zwei Knoten. Jede weitere Expansion muss demnach für zwei Knoten durchgeführt werden, d.h. die Laufzeit hat sich bereits in diesem simplen Beispiel für diesen Ast verdoppelt. Stattdessen können die beiden identischen Knoten zusammengefügt werden (siehe Abbildung 5.1, rechtes Bild), wodurch nachfolgend doppelte Berechnungen vermieden werden. Das in diesem Beispiel beschriebene Problem lässt sich verallgemeinern und gilt für mehrere Situationen im Suchgraphen.
Die Vorgehensweise zur Vermeidung der Baumstruktur ist dabei Folgende: Jeder Spielerzustand wird durch eine eindeutige ID beschrieben. Bei Erzeugen der Kinder eines Knotens wird für jede zu erzeugende ID des Kindes überprüft, ob ein Knoten mit dieser ID bereits existiert. Falls ja, wird der Knoten nicht neu erzeugt, sondern lediglich eine Kante zu diesem gezogen.

5.5.3 Bewertungsfunktionen

In der Beispielmodellierung ist das negative Attribut “Hunger” beschrieben, welches bei Ausführung aller regulärer Aktivitäten automatisch bis zu einem Maximum erhöht wird. Ist der Hunger auf einem Maximum, sind bis auf wenige Ausnahmen alle Aktivitäten deaktiviert, so dass der Spieler indirekt gezwungen wird, ein Brot oder eine Wurst zu essen.
Die Schwierigkeit liegt an der temporären Verschlechterung durch das Essen, da dadurch der Wert des gegessenen Objektes verloren geht. Der Best-First Algorithmus expandiert stets den besten Knoten, wird den Knoten nach dem Essen also unter Umständen schlecht bewerten, da der Wert der Nahrung verloren gegangen ist. Folglich werden die anderen Knoten zuerst betrachtet. Die Wahrscheinlichkeit, dass die nachfolgenden Knoten nach dem Essen jedoch deutlich besser sind, ist sehr groß, da durch das Essen deutlich mehr Möglichkeiten zur Verfügung stehen. Um das Problem allgemeiner zu formulieren: Es ist möglich, dass nach einem zunächst schlechten Knoten mehrere sehr gute Knoten folgen, die jedoch durch den schlechten Knoten erst spät betrachtet werden.
Umgangen wird dieses Problem durch eine Anpassungen der Bewertungsfunktionen:

1. **Hunger ist maximal:** Knoten, welche den Hunger reduzieren, erhalten die beste Bewertung abzüglich (maximierend) / zuzüglich (minimierend) der Anzahl an Zügen vom Startknoten zum betrachteten Knoten; andere Knoten erhalten die schlechteste Bewertung

2. **Sonst:** Auswertung mit bisheriger Bewertungsfunktion

Je nach Bewertungsfunktion kann es geschehen, dass der Abstand zum Zielknoten zwar reduziert, dieser aber dennoch nicht erreicht wird. Zur Veranschaulichung soll die Bewertungsfunktion "Score" verwendet werden, das Problem ist jedoch nicht auf diese begrenzt.
Der Score verläuft über alle Attribute und Items eines Spielers und versucht, ein komplettes Bild über den Spieler zu geben. Intuitiv gilt: je wertvoller die Items eines Spielers sind, desto näher gelangt dieser zum Ziel, da er den Goldwert der Items durch Handel erhalten kann. Ein mögliches Problem bei dieser Bewertungsfunktion ist jedoch, dass immer weiter über den Score optimiert wird und das Ziel des Spiels dabei ignoriert wird, obwohl es theoretisch schon durch den Verkauf der Items möglich wäre, das Ziel in wenigen Runden zu erreichen. Der Grund hierfür liegt an der Komplexität des Graphen. Der Verkauf eines Items bringt für den Score keinen Zugewinn, die Produktion eines weiteren Items dagegen schon. Aus diesem Grund sind Itemverkäufe schlecht bewertet und werden entsprechend spät betrachtet. Sind nun zum Erreichen des Ziels gleich mehrere Verkäufe hintereinander nötig, besteht die große Gefahr, dass wegen der Komplexität des Graphen und der begrenzten Zeit dieser Pfad niemals überprüft wird, da nicht so weit in der Tiefe des Graphen gesucht werden kann.
Diese Problematik lässt sich jedoch durch eine Erweiterung der Bewertungsfunktionen vermeiden, so dass der Verkauf der Items sichergestellt wird:

1. **Itemwert + Spielergold $\geq$ Zielgoldwert:** Knoten, die mehr Gold als der Ursprungsknoten besitzen, erhalten den Ursprungswert verbessert um die Golddifferenz als Bewertung; andere Knoten erhalten die schlechteste Bewertung.
2. **Hunger ist maximal:** Knoten, welche den Hunger reduzieren, erhalten die beste Bewertung abzüglich (maximierend) / zuzüglich (minimierend) der Anzahl an Zügen vom Startknoten zum betrachteten Knoten; andere Knoten erhalten die schlechteste Bewertung
3. **Sonst:** Auswertung mit bisheriger Bewertungsfunktion

Kapitel 6

Implementierung

Im nachfolgenden Kapitel ist die Implementierung der Anwendung beschrieben. Zur Implementierung wurde Java als Programmiersprache gewählt. Nachfolgend wird deshalb unter anderem die Java Syntax für Erklärungen verwendet.
Die Anwendung ist in vier Kategorien unterteilt: Daten, Strukturen, Agenten, GUI. Da die Agenten aus Sicht des Spiels wie gewöhnliche Spieler agieren und folglich eine abgeschlossene innere Logik besitzen, ist die Steuerung der Anwendung in Strukturen und Agenten aufgeteilt. Jede Kategorie besitzt ein eigenständiges Unterkapitel, in dem zunächst eine grafische Übersicht gegeben und anschließend auf die einzelnen Klassen eingegangen wird.

Eine generelle Übersicht der Implementierungsstruktur findet sich in Abbildung 6.1. Diese Abbildung visualisiert ausschließlich das wichtigste Zusammenspiel der Komponenten und verzichtet deshalb zur besseren Übersicht auf die Vollständigkeit aller Klassen und Verbindungen. Zu erkennen ist zunächst die Unterteilung in Daten, Logik und GUI.
Bei den Daten existiert eine Unterscheidung zwischen Stammdaten und normalen Spieldaten. Die Stammdaten dienen der Definition des Spiels und erfahren nach der Initialisierung keine Änderung mehr. ItemManager und AttributeManager beschreiben jeweils einen Teilzustand eines Spielers und werden während des Spiels geändert. Dabei ist zu erwähnen, dass sowohl Agent als auch Spieler Attribute und Items besitzen, der Händler jedoch ausschließlich den ItemManager nutzt.
Die Änderungen der Daten erfolgt im Spiel: Spieler (Agenten und menschliche Spieler) führen Aktivitäten aus. Diese manipulieren ItemManager und AttributeManager (und den Goldwert der Spieler), wodurch der Zustand der Spieler sowie des Händlers geändert wird. Dabei ist es für einen Spieler jedoch nur möglich, den eigenen Zustand sowie ggf. den Zustand des Händlers zu ändern. Zustandsänderungen anderer Spieler sind nur indirekt durch die Preisentwicklung des Händlers möglich.
Die Verwaltung des Zusammenspiels übernimmt die Game-Klasse. Ein Spiel wird folglich durch eine Game-Instanz beschrieben.
Die GameWindow-Klasse greift auf eine gegebene Game-Instanz zu und visualisiert das enthaltene Spiel, bzw. dessen Spielzustand. Dabei besteht das GameWindow aus mehreren Unterkomponenten. Jede Komponente wiederum ist durch drei Klassen beschrieben: Model, Controller und View.

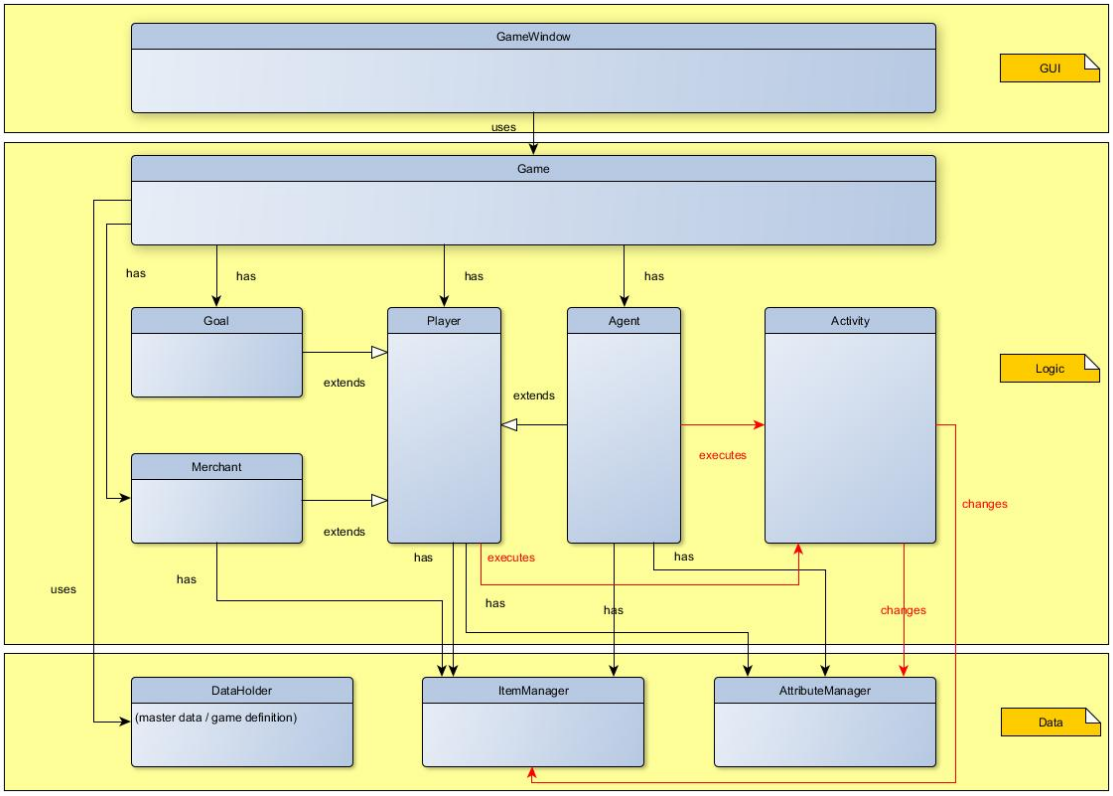

Abbildung 6.1: Generelle Übersicht der Implementierung

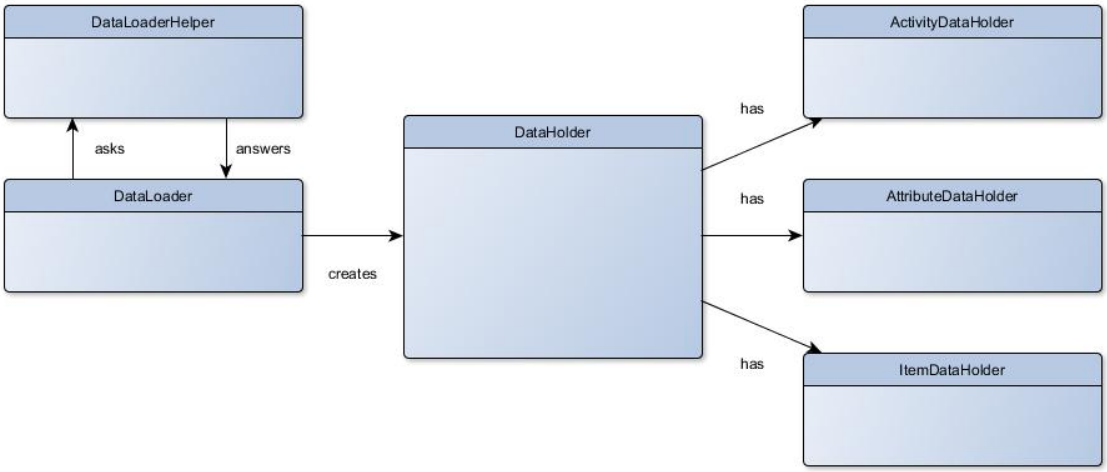

Abbildung 6.2: Übersicht der datenbezogenen Implementierung

6.1 Daten

Eine Übersicht der zur Bereitstellung der Daten implementierten Strukturen befindet sich in Abbildung 6.2. Die wichtigste Klasse ist dabei die DataHolder-Klasse, da diese für die Verwaltung aller spielbeschreibender Daten zuständig ist. Die in der DataHolder-Klasse definierten Daten werden somit nur einmalig zu Beginn des Spiels erzeugt und sind danach für dessen Dauer unveränderlich. Logik-Komponenten, welche auf diesen Daten basieren, erhalten eine DataHolder-Instanz[1] und greifen über diese Instanz auf alle benötigten Informationen zu. Dabei besteht die DataHolder-Klasse aus mehreren Unterkomponenten[2]: dem ActivityDataHolder zur Verwaltung der Aktivitäten, dem AttributeDataHolder zur Verwaltung der Attribute sowie dem ItemDataHolder zur Verwaltung der Items. Zudem besitzt die DataHolder-Klasse eine Instanz der Goal-Klasse, um das Ziel des Spiels zu beschreiben.
Die DataLoader-Klasse dient ausschließlich der Erzeugung der DataHolder-Instanz. Dabei lässt sich die Implementierung des DataLoaders beliebig anpassen und erweitern: In dieser Anwendung geschieht die Beschreibung der Daten direkt in einer Methode des DataLoaders, prinzipiell könnte dieser jedoch auch eine Datei einlesen oder sich benötigte Informationen aus einer Datenbank beschaffen. Außerhalb der Klasse entsteht dadurch kein Unterschied. Die DataLoaderHelper-Klasse dagegen ist nur eine Utility-Klasse für den DataLoader, um komfortabel Daten erzeugen zu können.

[1] Die DataHolder-Instanz ist für alle Komponenten des gleichen Spiels identisch.
[2] Diese Unterkomponenten dienen ausschließlich der Definition der jeweiligen Daten und nicht der Beschreibung von konkreten Spielerinstanzen.

6.1.1 DataHolder-Klasse

Die DataHolder-Klasse dient der Aufbewahrung aller spieldefinierender Daten. Folglich greifen andere Komponenten zur Abfrage solcher Daten auf eine hinterlegte DataHolder-Instanz zu. Eine Instanz beschreibt die Regeln und Möglichkeiten für ein konkretes Spiel. Es besteht also durch Austausch der DataHolder-Instanz die Option, ein vollständig anderes Spiel zu spielen[3]. Ein Spiel ist defniert durch die existierenden Aktivitäten, Attribute, Items sowie einem Spielziel.
Diese Daten des DataHolders sind auf folgende Komponenten (Klassen-Attribute) verteilt:

- **ActivityDataHolder:** Alle im Spiel existierenden Aktivitäten mit ihren Effekten und Vorbedingungen werden über eine ActivityDataHolder-Instanz verwaltet.
- **AttributeDataHolder:** Die möglichen Spieler-Attribute, sowie ihre Beschreibung, sind in einer AttributeDataHolder-Instanz beschrieben.
- **ItemDataHolder:** Alle existierenden Items des Spiels inklusive deren Informationen (z.B. Preis) befinden sich in einer ItemDataHolder-Instanz.
- **Goal:** Das Ziel des Spiels ist durch die Goal-Klasse beschrieben. Folglich verwendet der DataHolder eine Instanz dieser Klasse.

Die DataHolder-Klasse besitzt keine eigene Logik zur Konstruktion dieser Parameter. Diese werden ausschließlich von außen gesetzt und können daraufhin zur Nutzung abgefragt werden.

6.1.2 ActivityDataHolder-Klasse

Die ActivityDataHolder-Klasse beinhaltet alle existierenden Aktivitäten eines Spiels. Das heißt im Umkehrschluss, dass ausschließlich Aktivitäten im Spiel nutzbar sind, die in der ActivityDataHolder-Klasse beschrieben sind. Dabei werden die Aktivitäten nicht in der Klasse selbst definiert, sondern nur über die addActivity(Activity activity)-Methode hinzugefügt. Eine über diese Methode hinzugefügte Aktivität ist daraufhin im Spiel enthalten.
Zudem besitzt diese Klasse noch eine Methode, welche für einen übergebenen Spieler alle ausführbaren Aktivitäten zurückgibt. Diese Methode wird von der GUI genutzt, um das Fenster der ausführbaren Aktivitäten zu befüllen.

[3] Beispielsweise eine Sci-Fi Spielwelt, statt der hier verwendeten, mittelalterlich angehauchten Spieldefinition

6.1.3 AttributeDataHolder-Klasse

Diese Klasse besitzt eine große Ähnlichkeit zur ActivityDataHolder-Klasse. Sie definiert alle im Spiel existierenden Attribute. Auch hier findet die Definition außerhalb der Klasse statt, so dass Attribute lediglich über eine addAttribute(Attribute attribute)-Methode hinzugefügt werden und somit im Spiel zur Verfügung stehen. Zusätzlich ist es möglich, nach der fertigen Definition, die calculateAttributesOrder()-Methode aufzurufen, woraufhin alle vorhandenen Attribute alphabetisch sortiert werden. Da die Definition nur einmalig stattfindet, ist die Reihenfolge im gesamten Spiel fest. Eine Sortierung an dieser Stelle spart somit im späteren Verlauf Zeit und Mühe. Genutzt wird die Sortierung von der Benutzeroberfläche, um dem Nutzer alle Informationen stets in gleicher Reihenfolge zu visualisieren.

6.1.4 ItemDataHolder-Klasse

Auch die ItemDataHolder-Klasse dient, wie die in den beiden vorherigen Abschnitten, ausschließlich zur Aufbewahrung von Daten. Alle im Spiel existierenden Items müssen dem ItemDataHolder hinzugefügt werden und können daraufhin von den Logik- und GUI-Komponenten über den DataHolder genutzt werden. Wie beim AttributeDataHolder besteht die Möglichkeit, die Items nach der vollständigen Definition einmalig zu sortieren.

6.1.5 DataLoader-Klasse

Die DataLoader-Klasse ist die Hauptklasse der Datendefinitionen. Alle für ein Spiel relevanten, statischen Daten werden bei der Initialisierung des Spiels über den DataLoader erzeugt. Dafür existiert nur eine Methode: loadData(). Diese Methode erzeugt eine DataHolder-Instanz und gibt sie anschließend zurück. Folglich wird die DataLoader-Klasse nur dazu genutzt, die spieldefinierenden Daten bei der Initialisierung eines Spieles zu erzeugen und in eine DataHolder-Instanz zu packen, welche dann während des Spiels verwendet wird. Auf Grund der einfachen Struktur der DataLoader-Klasse wurde diese als Singleton implementiert (siehe Abschnitt 2.11).
Die Implementierung der loadData()-Methode ist theoretisch beliebig. Für das Spiel ist nur entscheidend, dass eine DataHolder-Instanz erzeugt wird. Auf welchem Weg dies geschieht, ist irrelevant. Denkbar wäre zum Beispiel eine Datenbank, auf die der DataLoader zugreift und zur Erzeugung des DataHolders nutzt. Auch wäre eine Struktur möglich, in welcher der DataLoader sich mit einem Server verbindet und die Aufgabe an diesen weiter delegiert, um anschließend die entsprechenden Daten zu erhalten. In einer größeren und verteilten Anwendung wären solche

Ansätze vorzuziehen. Auf Grund des Einzelspiel-Charakters des Spiels[4] sowie der übersichtlichen Datenmenge, wurde hier jedoch darauf verzichtet. Hervorzuheben ist lediglich die Tatsache, dass eine Erweiterung, bzw. ein Umstieg auf solche Strukturen problemlos durchführbar wäre, ohne Anpassungen an der restlichen Anwendung vornehmen zu müssen.

Die Initialisierung der Daten folgt einem einfachen Muster: Erzeuge eine Instanz (Item, Attribut, Aktivität, Ziel) und füge sie dem DataHolder hinzu. Während die Initialisierung der Items sehr einfach abläuft, da lediglich der Itemname und der Startkaufpreis spezifiziert werden müssen, nimmt eine Attribute-Instanz bei der Initialisierung deutlich mehr Parameter entgegen: neben dem Namen folgt der Startwert, der minimale und der maximale Wert, ein boolscher Wert zur Festlegung, ob es sich um ein positives oder negatives Attribut handelt, sowie zuletzt ein Scorewert[5]. Die Beschreibung des Ziels ist im konkreten Fall schnell umgesetzt, da das Ziel nur aus einem festen Goldwert besteht. Somit genügt die Initialisierung einer Goal-Instanz[6] und das Setzen des Goldwertes.
Aufwendiger ist die Definition der Aktivitäten. Zum einen existieren für jedes Item eine Vielzahl von Aktivitäten[7], zum anderen definiert sich eine Aktivität auch durch die Summe seiner Effekte und Vorbedingungen. Insofern muss für jede Aktivität ein Array Effect[ ] und ein Array PreCondition[ ] erzeugt werden. Zur Vereinfachung der Aktivitätenerzeugung wird die DataLoaderHelper-Klasse verwendet, welche Methoden zur gleichzeitigen Erzeugung mehrerer ähnlicher Aktivitäten besitzt. Die Funktionsweise dieser Helper-Klasse findet sich im nächsten Abschnitt.

6.1.6 DataLoaderHelper-Klasse

Diese Klasse dient ausschließlich der vereinfachten Initialisierung von ähnlichen Aktivitäten für den DataLoader. Sie ist somit eine reine Hilfsklasse und wird nur vom DataLoader verwendet. Es existieren zwei Ansätze zur vereinfachten Aktivitätenerzeugung:

- **Erzeuge gleiche Aktivitäten mit mehrfachen Qualitätsstufen:** Von jeder Produktionsaktivität sind drei Qualitätsstufen modelliert[8]. Diese sind in den Grundzügen gleich, unterscheiden sich lediglich durch die numerischen Werte der erhöhten und reduzierten Items, sowie in den Voraussetzungen. Diese Gleichheit wird ausgenutzt, um mittels einer Liste der Stufen, welche genau

[4] Im Bezug auf menschliche Spieler
[5] Dieser Scorewert beschreibt, wie viel Score eine Stufe des Attributes bringt. Bei negativen Attributen ist der Wert negativ.
[6] Die Goal-Instanz erbt von der Player-Klasse und besitzt somit dessen Klassenattribute, inklusive dem Goldwert.
[7] In der Regel existieren drei Produktionsaktivitäten, drei Aktivitäten zum Kauf des Items, sowie drei Aktivitäten zum Verkauf des Items.
[8] Erkennbar an der Nummerierung am Ende des Aktivitätennamens.

die unterschiedlichen numerischen Werte beinhaltet, sowie der festen Werte, jeweils drei Aktivitäten zu erzeugen.

- **Erzeuge die Handelsaktivitäten eines Items:** Ähnlich wie bei den Qualitätsstufen, existieren für jedes Item genau sechs Handelsaktivitäten: drei für den Verkauf und drei für den Kauf.[9] Die Aktivitäten unterscheiden sich folglich erneut ausschließlich durch ihre numerischen Werte (inklusive der Voraussetzungen, welche den "Skill" berücksichtigen). Insofern wird eine Liste mit den unterschiedlichen numerischen Werte erzeugt, um so mit den statischen Werte gleich alle Handelsaktivitäten für ein gegebenes Item zu erzeugen.

6.2 Strukturen

Eine Übersicht über das Zusammenspiel der Strukturen ist in Abbildung 6.3 visualisiert. Hauptkomponente ist die Game-Klasse. Diese steuert und verwaltet das Zusammenspiel aller Komponenten. Das Voranschreiten in die nächste Runde wird ebenfalls von der Game-Klasse übernommen. Zu diesem Zweck fragt das Game für alle Agenten die auszuführende Aktivität ab und berechnet die entstehenden Änderungen für Spieler und Spielwelt.
Die Player-Klasse stellt eine weitere Hauptkomponente dar, da andere Strukturen von dieser Klasse erben. Diese Klasse beschreibt den aktuellen Zustand des Spielers und nutzt dazu unter anderem die Hilfsklassen ItemManager und AttributeManager. Desweiteren lässt sich die nächste auszuführende Aktivität setzen.
Der AttributeManager dient zur Verwaltung der Attribute des Spielers, sowie der ItemManager zur Verwaltung der Items des Spielers dient. Diese beiden Komponten stellen somit eine Art Inventar dar. Die Klassen Item und Attribute dienen lediglich als Rahmen zur Beschreibung von Items bzw. Attributen (siehe Kapitel 4).
Das Ziel des Spiels wird in der Goal-Klasse beschrieben, welche von der Game-Klasse verwaltet wird. Aus diesem Grund erbt die Goal-Klasse lediglich von der Player-Klasse, da somit der Zielzustand des Spielers (Wert des Goldes, der Items und der Attribute) einfach beschrieben werden kann. Das Game prüft nach jeder Runde, ob ein Spieler den Zielzustand erreicht hat. Obwohl die Goal-Klasse vom Player erbt, nimmt sie nicht aktiv am Spiel teil.
Ähnlich verhält es sich mit der Merchant-Klasse: Auch diese erbt vom Player. Grund ist die benötigte Verwendung des ItemManagers, um den Handel mit Spieler darzustellen. Dabei ist der Merchant jedoch passiv.
Die Aktivity-Klasse stellt den letzten großen strukturellen Block dar. Eine Aktivität ist das zentrale Element, um am Spiel teilzunehmen, benötigt also eine gewisse Vielfalt, um alle möglichen Geschehnisse abzubilden. Aus diesem Grund beschreibt sich die Activity-Klasse vor allem durch Effekte und Vorbedingungen, die sich mehrfach kombinieren und / oder verwenden lassen. Die Effekte sind in der abstrakten Effect-

[9] Die Schritte bei den Handelsaktivitäten sind stets 1, 3 und 9 gehandelte Items.

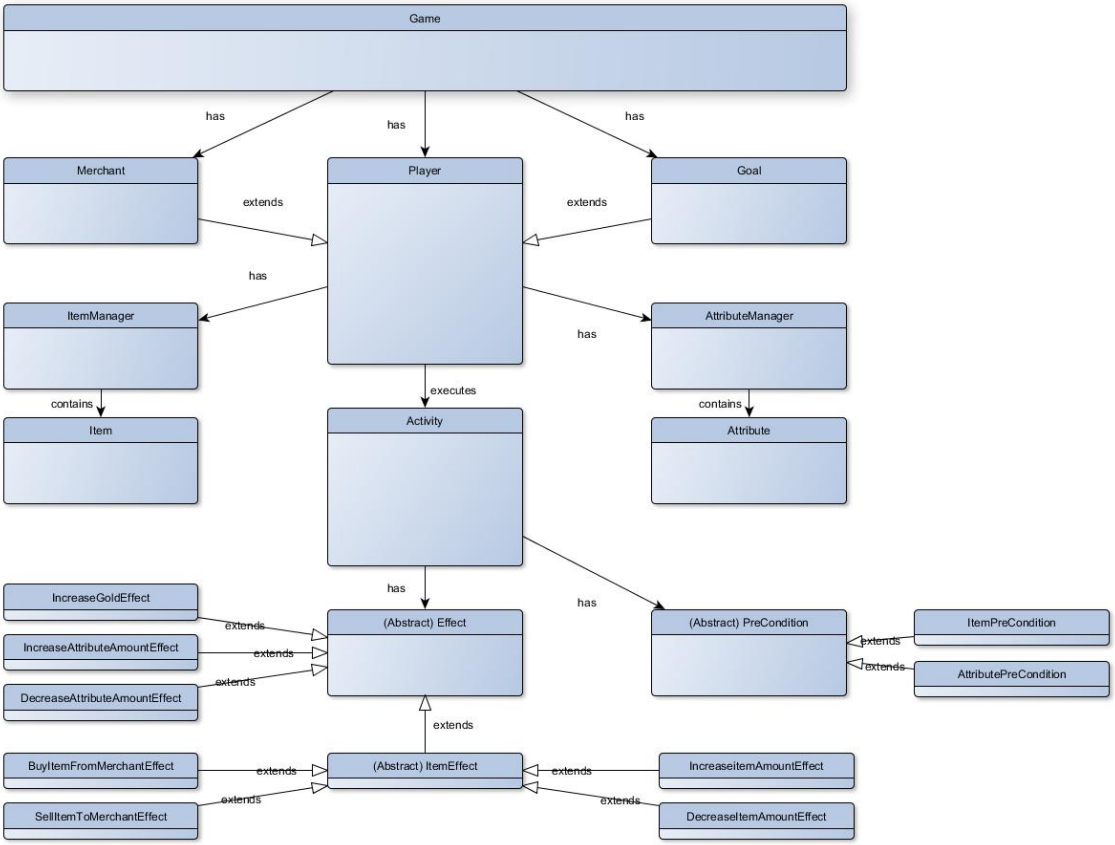

Abbildung 6.3: Übersicht der implementierten Strukturen

Klasse als Schnittstelle für die zahlreichen implementierten Subklassen beschrieben. Ein Effekt verändert Spieler und / oder Händler. Die Vorbedingungen dagegen stellen nur eine Bedingung dar, welche zur Ausführung einer Aktivität erfüllt sein muss. Die Implementierung erfolgt analog über die Verwendung der abstrakten PreCondition-Klasse.

6.2.1 Attribute-Klasse

Die Attribute-Klasse dient der Beschreibung eines Attributes im Spiel. Dabei ist die Definition allgemeiner Natur: eine Attribut-Instanz beschreibt nicht, welche Attribute ein Spieler in welcher Ausprägung besitzt, sondern ausschließlich wie ein konkretes Attribut aussieht. Für den Spieler nutzbar werden Attribute erst durch die AttributeManager-Klasse. Folglich ist die in Abschnitt 4 beschriebene Modellierung eines Attributes in der Implementierung auf zwei Klassen aufgeteilt: Die Attribute-Klasse als definierendes Element und die AttributeManager-Klasse zur Verwaltung der Spieler-Attributwerte.
Folgende Klassenattribute existieren:

- **name:** Der Name des Attributes; entspricht dem Namen aus der Modellierung.
- **startValue:** Der Startwert des Attributes ist der Wert, den ein Spieler zu Spielbeginn erhält.
- **maxValue:** Dies beschreibt den Maximalwert, den ein Attribut annehmen kann.
- **minValue:** Das entsprechende Gegenteil vom Maximalwert: minValue beschreibt den niedrigsten Wert, den ein Attribut annehmen kann.
- **scoreFactor:** Dieser Wert entspricht dem Score, der eine Erhöhung des Attributes um 1 entspricht. Folglich steigt der Score für jeden Attributwert linear an.[10]
- **positiveAttribute:** Gibt an, ob ein Attribut positiv (z.B. "Skills") oder negativ (z.B. "Hunger") interpretiert werden muss. Dies wird beispielsweise zur Berechnung des Scores verwendet. Der scoreFactor wird entsprechend positiv oder negativ zum Gesamtscore hinzugerechnet.

Die Klasse besitzt keine logischen Methoden und dient lediglich der Verwaltung oben beschriebener Informationen. Zusätzlich findet eine Überschreibung der toString()-Methode mit dem Namen des Attributes statt, um die Nutzung von Attribute-Instanzen in der GUI zu vereinfachen.

6.2.2 AttributeManager

Der AttributeManager beschreibt, wie hoch oder niedrig Attribute eines Spielers ausgeprägt sind. Während die Attribute-Klasse der Definition der Attribute dient, wird die AttributeManager-Klasse zur Beschreibung der Attribute-Ausprägungen eines Spielers verwendet.
Zur Initialisierung verwendet der AttributeManager den AttributeDataHolder des Spiels, um alle existierenden Attribute mit ihrem Startwert zu initialisieren. Ergebnis der Initialisierung ist eine Abbildung "Attribut $\rightarrow$ Anzahl". Diese wird im Verlauf des Spiels mittels AttributeManager gesetzt, erhöht und reduziert. Alle Änderungen von Spielerattributen erfolgen somit ausschließlich über den AttributeManager. Änderungen eines Attributes werden zudem auf den Wertungsbereich (d.h. minValue und maxValue) überprüft.

6.2.3 Item-Klasse

Die Item-Klasse beschreibt die aus Abschnitt 4 modellierten Items. Dabei ist die Umsetzung dieser Klasse analog zur Umsetzung der Attribute-Klasse: Die Item-Klasse

[10] Der Score berechnet sich durch die Ausprägung des Attributes im AttributeManager multipliziert mit dem scoreFactor.

definiert nur das Item an sich und trifft keine Aussagen über die Menge an Items, die ein Spieler besitzt. Dieses Verhalten wird von der ItemManager-Klasse übernommen. Eine Item-Instanz wird durch einen Namen und einen Preis initialisiert. Desweiteren ist die toString()-Methode mit dem Itemnamen überschrieben, um eine komfortablere Verwendung in der GUI zu ermöglichen. Da die Item-Klasse ausschließlich der Beschreibung eines Items dient, besitzt sie keine eigene Logik.

6.2.4 ItemManager-Klasse

Der ItemManager beschreibt den Itembesitz für Spieler. Während die Item-Klasse der Definition einer Itemart dient, listet der ItemManager alle existierenden Items auf und gibt für jedes Item die entsprechend vorhandene Anzahl an. Folglich besitzt jeder Spieler genau einen ItemManager. Änderungen des Itembesitzes eines Spielers sind ausschließlich über den ItemManager möglich. Der ItemManager ist zudem vollständig, d.h. auch nicht vorhandene Items werden mit einem 0-Zähler gespeichert.
Der ItemManager nutzt zur Initialisierung die ItemDataHolder-Instanz des Spiels, um eine Abbildung "Item → Anzahl" initial mit 0-Werten zu befüllen. Anschließend werden Methoden genutzt, um ein ausgewähltes Item zu setzen, zu erhöhen oder zu reduzieren.

6.2.5 PreCondition-Klasse

Die PreCondition-Klasse ist eine der Komponenten, mit welchen eine Aktivität beschrieben wird. Eine PreCondition-Instanz definiert eine Voraussetzung, die ein Spieler erfüllen muss, bevor eine Aktivität ausführbar ist. Diese Voraussetzungen sind dabei so aufgebaut, dass stets zwei beliebige Werte miteinander verglichen werden. Dies ist zunächst allgemein gehalten, d.h. die PreCondition-Klasse ist als abstrakte Klasse definiert. In den implementierten Unterklassen werden die verglichenen Werte jedoch konkretisiert, so dass Attribute und Items einen bestimmten Wert überschreiten oder unterschreiten müssen.
Die abstrakte Basisklasse besitzt nur zwei Methoden:

- **abstract boolean isExecutable(Player player):** Diese Methode ist die von der Logik verwendete Methode zur Verifizierung, ob eine Aktivität ausführbar ist oder nicht. Da diese Prüfung immer genau einen Spieler betrifft, wird diese Spieler-Instanz als Parameter übergeben. Die Methode ist abstrakt und wird folglich erst von Unterklassen implementiert.
- **boolean isValue1OperatorValue2(int value1, int value2, String operator):** Diese Methode ist nicht öffentlich und dient als Hilfskonstrukt für erbende Klassen. Die Schwierigkeit lag darin, eine allgemeine Methode zu finden, um die logischen Operationen EQUALS, GREATER, GREATEREQUALS, LESSER

und LESSEREQUALS abbilden zu können, statt für jeden Operator eine eigene Methode zu verwenden. Dies wurde gelöst, indem die PreCondition-Klasse fünf statische und finale Strings, welche jeweils für einen der Operatoren stehen, verwendet. Diese Strings werden der Methode mit der Variable operator übergeben und in der Methode selbst zu dem jeweils resultierenden logischen Operator aufgelöst. Dies nutzen die Unterklassen, um einen Wert w1 zu definieren und einen Operator o1, wobei das Bezugsobjekt o1 w1 erfüllen muss, damit die Vorbedingung erfüllt ist[11].

Konkret existieren zwei implementierte Unterklassen: AttributePreCondition und ItemPreCondition. Diese sind analog zueinander aufgebaut. Die AttributePreCondition-Klasse nimmt im Konstruktor ein Attribut entgegen, sowie eine Anzahl amount und einen der vordefinierten Operatoren operator. Die ItemPreCondition-Klasse ebenso, nur statt eines Attributes wird ein Item übergeben. Die isExecutable(Player player)-Methoden verwendet einen Aufruf der isValue1OperatorValue2-Methode, wobei das Bezugsobjekt entweder das ausgewählte Item (z.B. "Getreide") oder das ausgewählte Attribut ist. Dieses Bezugsobjekt wird entsprechend durch die Player-Instanz abgefragt. Die Anzahl ist der bei der Initialisierung übergebene Parameter amount und als Operator dient der Parameter operator.

6.2.6 Effect-Klasse

Die Effect-Klasse ist eine Hilfsklasse zur Beschreibung einer Aktivität. Eine Effect-Instanz besitzt genau eine Auswirkung auf den Spieler, bzw. das Spiel, wobei die Klasse selbst abstrakt ist und somit erst von Unterklassen implementiert werden muss. Dabei ist es möglich, dass eine Aktivität mehrere Effekte besitzen kann.
Folgende abstrakte Methoden werden durch die Klasse bereit gestellt:

- **abstract boolean performEffect(Player player, Game game):** Mit dieser Methode wird der Effekt ausgelöst, wobei die Unterklasse die Art des Effektes zu beschreiben hat. Der Effekt benötigt sowohl die Player-Instanz, als auch eine Instanz des aktuellen Spiels, da Auswirkungen nicht nur für den Spieler alleine gelten müssen.
- **abstract boolean simulateEffect(Player player, Game game, int timePassedBy):** Diese Methode wurde zur Implementierung der Agenten hinzugefügt. Der Effekt wird dabei nicht wirklich ausgeführt, sondern nur simuliert, ohne das vollständige Spiel zu beeinflussen.[12] Auf diese Weise können für den

[11] Beispiel: Operator: LESSER, Wert: 10, Bezugsobjekt: Hungerattribut. Es folgt: Die Vorbedingung ist nur ausführbar, wenn der Hunger niedriger als 10 ist.

[12] Nur die Spieler selbst werden beeinflusst: Dieses Verhalten ist gewünscht, da der Suchgraph der Agenten für jeden Knoten eine Agentenkopie anlegt. Für diese Agentenkopie wurde im Gegensatz zum Original eine weitere Aktion simuliert.

Spielgraphen der Agenten diverse Aktivitäten ausprobiert werden. Der Parameter timePassedBy dient zudem als Zeitangabe der Runden, wodurch Ereignisse besser kalkuliert werden können.[13]

- **abstract boolean isExecutable(Player player, Game game):** Ein Effect besitzt eine Methode zur Überprüfung der Ausführbarkeit. Dies ist auf den ersten Blick ähnlich zu den PreConditions und stellt das Konzept jener in Frage. Die Antwort liegt jedoch in der Art der Ausführbarkeit: Während eine PreCondition prinzipielle Auführbarkeit abfragt ("Der Skill ist hoch genug für diese Aktivität"), fragt diese Methode natürlich wirkende Voraussetzungen ab, die für einen menschlichen Spieler ohne Erklärung selbstverständlich sind ("Um eine Einheit Getreide zu verkaufen, muss eine Einheit Getreide vorhanden sein."). Diese Trennung dient vor allem der Vereinfachung und der Übersicht, aber ist logisch auch für die Berechnungen des GameAnalyzers notwendig.

Nach Vorstellung der abstrakten Klasse werden nun die implementierten Unterklassen erläutert:

- **IncreaseGoldEffect:** Dieser Effekt verlangt bei der Initialisierung einen Integer, welcher dem zu erhöhenden Goldwert entspricht. Bei ausgeführter Aktivität wird dem Spieler der Wert gutgeschrieben, die Simulation erfolgt identisch. Dieser Effekt ist immer ausführbar.
- **IncreaseAttributeAmountEffect:** Im Konstruktor wird ein Attribut, sowie ein Wert übergeben. Dieses Attribut wird von der Player-Instanz abgefragt und um den definierten Wert erhöht, analog bei der Simulation. Der Effekt ist genau dann ausführbar, wenn die Erhöhung des Attributes dessen Wertebereich nicht verlässt.
- **DecreaseAttributeAmountEffect:** Identisch zum IncreaseAttributeAmountEffect, nur dass das angegebene Attribut reduziert wird.
- **abstract ItemEffect:** Die ItemEffect-Klasse ist wie die Hauptklasse abstrakt und beschreibt eine Gruppe von Effekten, welche alle ein Item zum Gegenstand haben. Aus diesem Grund ist bereits ein Konstruktor mit dem Itemnamen und einem Wert definiert. Nachfolgende Effekte sind alle Unterklassen der ItemEffect-Klasse.
- **IncreaseItemAmountEffect:** Das angegebene Item des Spielers wird um den definierten Wert erhöht. Die Simulation ist identisch und der Effekt ist immer ausführbar.
- **DecreaseItemAmountEffect:** Analog zum IncreaseItemAmountEffect, nur dass das Item reduziert wird. Voraussetzung ist, dass die Anzahl Items nach der Reduzierung nicht negativ ist.
- **BuyItemFromMerchantEffect:** Dieser Effekt dient dem Verkauf eines Items an den Händler. Bei der Ausführung wird das ausgewählte Item des Spielers

[13] Konkret: Die Händlerpreise mit ihrem erwarteten Verlauf

um den angegebenen Wert erhöht und gleichzeitig der Goldwert des Items vom Spieler abgezogen. Gleichzeitig findet der gegenteilige Operator beim Händler statt (welcher über die Game-Instanz abgefragt wird): Bei diesem wird das Item reduziert und das Gold erhöht. Der Effekt ist nur dann ausführbar, wenn sowohl Spieler, als auch Händler ausreichend Items, bzw. Gold zur Verfügung haben. Die Simulation erfolgt modifiziert: Zum einen bleibt der Händler-Zustand unverändert. Der berechnete Kaufpreis wird zum anderen nicht einfach vom Händler abgefragt, sondern zusätzlich mit der Anzahl der Runden (Parameter timePassedBy) und den vom Händler berechneten voraussichtlichen Preisänderungen multipliziert.

- **SellItemToMerchantEffect:** Analog zum BuyItemFromMerchantEffect, nur mit dem Unterschied, dass die Rollen des Käufers und Verkäufers vertauscht sind. In der Simulation wird statt dem Kaufpreis der Erlös berechnet, wobei ebenfalls die voraussichtlichen Preisänderungen berücksichtigt werden.

6.2.7 Activity-Klasse

Die Activity-Klasse beschreibt eine Aktivität, wie diese in Abschnitt 4 modelliert ist. Dazu finden die beiden zuvor erläuterten Hilfsklassen PreCondition und Effect Anwendung: Eine Aktivität ist definiert über einen Namen, sowie je einer Liste von Effekten und Vorbedingungen. Diese werden direkt im Konstruktor der Klasse übergeben. Somit können durch Kombination verschiedener Effekte und Vorbedingungen beliebig viele Aktivitäten erschaffen werden.
In dieser Klasse finden sich dieselben Methoden wie in den Hilfsklassen wieder:

- **void execute(Player player, Game game):** Die Aktivität wird ausgeführt. Dies geschieht über den Aufruf der execute-Methode für alle zugehörigen Effect-Instanzen.
- **boolean isExecutable(Player player, Game game):** Für alle vorhandenen PreCondition- und Effect-Instanzen wird die isExecutable-Methode aufgerufen. Geben alle “true” zurück, gibt auch diese Methode “true” zurück.
- **void simulate(Player player, Game game, int timePassedBy):** Simuliert die Aktivität durch Aufruf der simulate-Methode aller zugeordneter Effekte.

6.2.8 Player-Klasse

Die Player-Klasse ist eine der wichtigsten Klassen der Anwendung, da sie einen Spieler beschreibt und Grundlage für viele Unterklassen[14] ist.
Die wichtigsten Attribute der Klasse:

[14]Erbende Klassen: Agent, Merchant, Goal

- **String name:** Diese Variable repräsentiert den Namen des Spielers.
- **AttributeManager attributeManager:** Der Zustand der Attribute des Spielers wird über einen AttributeManager verwaltet.
- **ItemManager itemManager:** Analog zu den Attributen werden die Items des Spielers über einen ItemManager verwaltet.
- **Activity nextActivity:** Diese Variable repräsentiert die nächste auszuführende Aktivität des Spielers. Bei Überführung des Spiels in die nächste Runde wird die Game-Instanz die nextActivity des Spielers abfragen und ausführen. Ist diese bis dahin nicht gesetzt, wird für den Spieler keine Aktivität ausgeführt. Folglich setzt er aus. Für den Spieler muss diese Variable über die GUI gesetzt werden.
- **int gold:** Die finanziellen Mittel des Spielers werden über die gold-Variable gespeichert.
- **int round:** Diese Zahl steht für die aktuelle Spielrunde, in welcher sich der Spieler befindet. Diese wird absichtlich für jeden Spieler einzeln und nicht von der Game-Klasse verwaltet, da die Möglichkeit existiert, dass ein Spieler freiwillig eine Runde aussetzt. Insofern wird in einem solchen Fall die Anzahl der Runden nicht erhöht, während das Spiel global weiter läuft.

Die Methoden der Player-Klasse dienen ausschließlich der Verwaltung oben genannter Klassenattribute und besitzen somit wenig eigene Logik. Erwähnenswert sind jedoch folgende Methoden:

- **void nextRound():** Diese Methode wird am Anfang einer neuen Runde über die Game-Instanz für alle Spieler aufgerufen. Für die Player-Implementierung erfolgt darin ausschließlich die Inkrementierung der round-Variable. Die später vorgestellten Klassen Agent und Merchant überschreiben diese Methode jedoch, um so weitere wichtige Initialisierungen zum Rundenanfang auszuführen.
- **int getScore():** Diese Methode berechnet den aktuellen Score der Player-Instanz. Dieser setzt sich aus folgenden drei Komponenten zusammen:
  - Gold des Spielers (direkt als Variable vorhanden)
  - Itemwert des Spielers (Iteration über alle Items mit Multiplikation des Goldwertes)
  - Attributescore des Spielers (Iteration über alle Attribute mit Multiplikation mit dem Attributescore-Wert des zugehörigen Attributes)

  Diese drei Werte werden aufsummiert und als Score zurückgegeben.

6.2.9 Merchant-Klasse

Die Merchant-Klasse implementiert den Händler des Spiels. Dieser stellt die Interaktions-Schnittstelle aller Spieler dar. Zu diesem Zweck erbt er von der Player-Klasse um dessen Strukturen zur Verwaltung von Gold und Items zu nutzen. Abgesehen davon nimmt der Händler jedoch nicht am Spielgeschehen teil, führt also selbst keine Aktivitäten aus. Die Klasse besitzt Logik zum Kauf und Verkauf von Items (inklusive Bedarf und Nachfragekalkulation), sowie zur Kalkulation der Itempreise. Eine besondere Wichtigkeit der Implementierung liegt im Zeitpunkt aller Handelsaktivitäten: Diese müssen in jeder Runde gleichzeitig stattfinden, da ansonsten Spieler benachteiligt würden[15].
Folgende Klassenattribute wurden hinzugefügt:

- **Map<String, List<Integer>> itemPriceDevelopment:** Diese Variable speichert für jedes Item die Entwicklung der letzten Runden. D.h. jeder Eintrag der inneren Liste beschreibt die Differenz des Itemwerts zur Vorrunde.
- **Map<String, Integer> newItemAmount:** Da Items nicht sofort gehandelt werden dürfen, sondern erst am Ende, nachdem die Aktivitäten aller Spieler berechnet und ausgeführt wurden, dient diese Variable als Speicher der Zwischenergebnisse, welche aus der Differenz des Itembestandes dieser Runde zur vorherigen Runde bestehen. In den zugehörigen Handelsmethoden wird diese Abbildung stets gepflegt.
- **Map<String, Double> newItemPricePrediction:** In dieser Variable werden die Annahmen über die Preisentwicklung gespeichert. Für die Prognose wird die durchschnittliche Veränderung in den letzten Runden zugrunde gelegt.
- **int itemPricesSaved:** Diese Zahl gibt an, für wie viele Runden die Preisentwicklung gespeichert werden soll. Im implementierten Test ist sie auf 10 gesetzt.
- **boolean itemPricesChange:** Dieses Flag dient dazu festzulegen, ob die Itempreise veränderlich sind oder nicht. Bei unveränderlichen Itempreisen fallen die meisten Berechnungen weg.

Anschließend folgt die Betrachtung der wichtigsten Methoden:

- **@override void init():** Die init()-Methode der Player-Klasse ist überschrieben, um den Händler bereits mit einem Vorrat an Items auszustatten. Jedes Item wird mit einem Vorkommen von 100 initialisiert. Zudem erhält der Händler einen praktisch unerschöpflichen Goldvorrat von 1000000.

[15] Für jeden abgeschlossenen Handel verändern sich die Spielerpreise, d.h. wenn zwei Spieler in einer Runde nacheinander handeln würden, hätten diese unterschiedliche Preise. Dies widerspricht jedoch der Definition einer Runde.

- **int sellItem(String itemName, int amount):** Diese Methode berechnet die Kosten für den Verkauf der angegebenen Itemmenge und gibt diese zurück. Zudem wird der Goldwert des Händlers bereits hier aktualisiert[16]. Die gekauften Items werden jedoch noch nicht mit dem Bestand verrechnet, sondern stattdessen mit dem Wert des newItemAmount.
- **int buyItem(String itemName, int amount):** Diese Methode ist (innen und außen) analog der sellItem-Methode, nur dass stattdessen ein Kauf abgewickelt wird.
- **@override void nextRound():** Diese Methode überschreibt die nextRound()-Methode der Player-Klasse. Sollte itemPricesChange auf "False" gesetzt sein, erfolgt ein direktes return ohne Ausführung der Logik, da die Preise dann immer konstant den Ausgangswerten entsprechen. Im anderen Fall werden folgende drei Aktionen für jedes Item abgearbeitet:
  - Die Itemmenge wird aktualisiert. Dazu wird die im newItemAmount gespeicherte Veränderung mit dem Wert des itemManagers verrechnet und die newItemAmount-Variable anschließend wieder zurückgesetzt.
  - Die Itempreisentwicklung wird aktualisiert. Hierfür wird über alle Einträge der itemPriceDevelopment-Liste für das ausgewählte Item iteriert und aufsummiert. Anschließend wird die Summe durch die Anzahl der Einträge geteilt und als Voraussage verwendet.
  - Der Itempreis wird aktualisiert. Dies erfolgt über die updateItemPrice-Methode, welche nachfolgend vorgestellt wird.
- **void updateItemPrice(String itemName, int newAmount, int oldAmount):** Diese Methode aktualisiert den Itempreis[17]. Die Itempreise verändern sich genau um die Differenz von der alten zur neuen Itemanzahl. Zudem erfolgt eine Aktualisierung der itemPriceDevelopment-Variable: der neue Itempreis wird hinzugefügt und bei einer Überschreitung der definierten Länge der Liste (d.h. größer als itemPricesSaved) wird der älteste Wert verworfen.

6.2.10 Goal-Klasse

Die Goal-Klasse beschreibt das Ziel des Spiels. Dazu erbt sie von der Player-Klasse. Die Idee hinter dieser Vererbung ist die Beschreibung eines Spielerzustandes, welcher mindestens erreicht werden muss. Mindestens bedeutet dabei, dass Werte über- oder unterschritten werden dürfen, entsprechend ihrer Ausprägung: Itemzähler, Goldzähler und positive Attribute dürfen überschritten werden, negative Attribute jedoch nur unterschritten werden. Somit ist die Goal-Klasse sehr viel flexibler als das

[16] Da der Goldvorrat des Händlers praktisch unerschöpflich ist, muss die Verrechnung des Händlergoldes nicht am Rundenende stattfinden.
[17] D.h. der Preis der in jeder Item-Instanz definiert ist.

festgelegte Ziel "Erreiche 1000 Gold", da sie die Festlegung beliebiger Ziele zulässt.[18] Somit ist eine Anpassung und Erweiterung des Spiels in dieser Hinsicht problemlos möglich.
Zur Festlegung eines Ziels genügt die Initialisierung einer Goal-Instanz und das anschließende Setzen aller Klassenattribute nach den Zielvorgaben[19]. Die Klasse besitzt zudem eine Methode zur Klärung der Frage, ob das Ziel erreicht ist: **boolean isGoalReached(Player player)**. Diese Methode vergleicht die übergebene Spieler-Instanz mit der eigenen und nur, wenn alle Werte des Spielers besser sind, wird ein "true" zurück gegeben.

6.2.11 Game-Klasse

Die Game-Klasse beschreibt die zentrale Stelle, an der alle Komponenten der logischen Anwendung zusammenlaufen. Ein Spiel wird durch eine Game-Instanz beschrieben, sie beinhaltet alles vom DataHolder bis zu den Player-Instanzen. Zudem wird in der Klasse das Interface GameListener definiert. Dieses besitzt die abstrakte Methode handlePlayerReachedGoal(Player player), welche von der Game-Instanz für alle registrierten GameListener ausgelöst wird, sobald ein Spieler das Ziel erreicht.[20]. Folgend sind die wichtigsten Klassenattribute aufgelistet:

- **DataHolder dataHolder:** Die dataHolder-Instanz, welche die Stammdaten des Spiels beinhaltet, wird von der Game-Klasse verwaltet und an die Unterkomponenten weitergereicht.
- **GameAnalyzer gameAnalyzer:** Der GameAnalyzer basiert auf der Analyse des Spiels: einmal vor Beginn des eigentlichen Spiels und jeweils nach einer Runde. Insofern verwaltet die Game-Instanz eine GameAnalyzer-Instanz, um die entsprechenden init und update Methoden auszuführen.
- **GameListener[ ] gameListeners:** Diese Liste führt alle GameListeners, die sich bei der Game-Instanz registriert haben. Für diese wird folglich die vormals erwähnte handlePlayerReachedGoal(Player player)-Methode aufgerufen, wenn ein Spieler das Ziel erreicht.
- **Merchant merchant:** Ein Spiel besitzt genau einen Händler, dieser wird folglich auch von der Game-Instanz verwaltet.
- **List<Player> players:** Diese Liste beinhaltet alle am Spiel teilnehmenden Spieler, irrelevant ob diese menschliche Spieler oder Agenten sind.
- **Player user:** Einer der Spieler aus der players-Liste ist als user definiert. Diese Variable ist ausschließlich ein Hilfsmittel für die GUI, um damit den Spieler abzufragen, für den die GUI angezeigt werden soll und dessen Aktivitäten folglich auch ausgewählt werden müssen.

[18] Z.B. "Erreiche 100 Fleisch und Hunger 0"
[19] Hier: das Setzen des Goldwertes auf 1000
[20] Die GUI verwendet diesen Listener zur Erzeugung eines Popup-Fensters bei Erreichen des Ziels.

Die Methoden der Game-Klasse sind bis auf wenige Ausnahmen struktureller Natur. Anschließend folgen die wichtigsten Methoden:

- **void addPlayer(Player player):** Auch wenn diese Methode nur dem Spiel (d.h. der players-Liste) eine Player-Instanz hinzufügt, ist sie dennoch so essentiell, dass sie Erwähnung findet.
- **void init():** Die Initialisierung erfolgt zu Beginn. Ein neuer Händler wird definiert und initialisiert, für den GameAnalyzer wird analyzeGame() aufgerufen und über eine Iteration folgt der Aufruf der init()-Methode für jeden Spieler.
- **void nextRound():** Ein Aufruf dieser Methode führt das Spiel in die nächste Runde über. Dabei werden folgende Schritte durchgeführt:

  1. Iteriere über alle Spieler und rufe die player.getNextActivity()-Methode auf. Falls die erhaltene Aktivität ausführbar und nicht null ist, führe diese aus.
  2. Rufe für jeden Spieler die player.nextRound()-Methode auf, um diesem den Start der nächsten Runde anzuzeigen.
  3. Rufe merchant.nextRound() auf, um den Händler zu aktualisieren.
  4. Rufe gameAnalyzer.update() auf, um das Ranking des GameAnalyzers zu aktualisieren.
  5. Überprüfe für jeden Spieler: Hat dieser das Ziel erreicht, setze das finished-Flag des Spielers und erzeuge das playerReachedGoal-Event für alle registrierten Gamelistener.

6.3 Agenten

Im nachfolgenden Kapitel wird die Implementierung der Agenten beschrieben. Die generelle Struktur ist dabei in Abbildung 6.4 zu erkennen. Aus Sicht der Gesamtanwendung wird jedoch nur die Agent-Klasse genutzt. Die restlichen Klassen dieses Kapitels dienen ausschließlich den Agenten zur Berechnung ihrer Aktivitäten und werden folglich nur von diesen verwendet.
Die Implementierung ist dabei so flexibel wie möglich gehalten. Statt einen Algorithmus hart zu kodieren, existiert ein Konstrukt (AbstractGraphSearch-Klasse) zur Implementierung beliebiger Algorithmen. Dadurch ist es zu jedem Zeitpunkt möglich, die Agenten ohne zusätzliches Refactoring um einen Suchalgorithmus zu erweitern. Auch konnte die Abhängigkeit von einzelnen Bewertungsfunktionen durch die Verwendung einer allgemeinen Schnittstelle (Rating-Klasse) verhindert werden.
Die Funktionsweise ist folgende: Der Agent erbt von der Player-Klasse und besitzt somit alle Eigenschaften, die auch ein normaler menschlicher Spieler besitzt. Zudem besitzt er die Funktionalität, eine Suche “AbstractGraphSearch” auszuführen

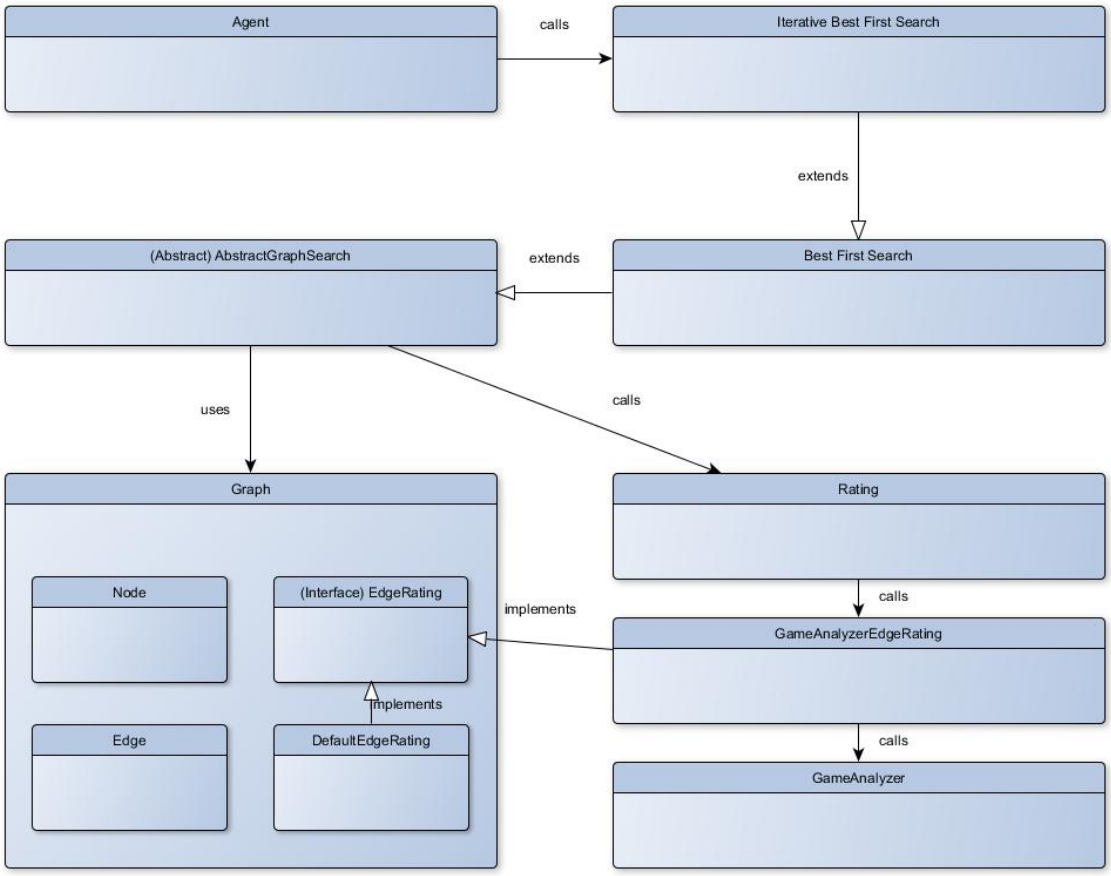

Abbildung 6.4: Übersicht der KI-Implementierung

und so die nächste Aktivität zu berechnen (Vergleich: beim menschlichen Spieler geschieht dies durch GUI-Eingabe). Die AbstractGraphSearch-Klasse wiederum nutzt die Graph-Klasse zur Modellierung des Spiels für den aktuellen Agentenzustand. Dabei entsprechen die Kantenkosten denen im implementierten Interface EdgeRating definierten Kosten. Die AbstractGraphSearch-Klasse navigiert und expandiert den Graphen entsprechend der in der Rating-Klasse definierten Bewertungsfunktion sowie ihrer Implementierung (IterativeBestFirst Search, BestFirstSearch).
Im Default entspricht das EdgeRating stets dem Wert 1 für jede Kante. Das GameAnalyzer-Rating definiert dieses jedoch entsprechend um und nutzt dafür die GameAnalyzer-Klasse. Diese Bewertungsfunktion ist die einzige auf Kanten basierende Bewertungsfunktion und besitzt eine deutlich höhere Komplexität als die anderen Funktionen. Aus diesem Grund wird diese gesondert in eigenen Klassen beschrieben und in späteren Unterkapiteln erläutert.

6.3.1 Graph-Klasse

Die Graph-Klasse beschreibt den Suchgraphen der Agenten. Die Logik ist noch weiter auf die inneren Klassen Edge und Node verteilt. Desweiteren besitzt die Graph-Klasse ein Interface EdgeRating, welches zur Beschreibung der Kantenkosten genutzt wird, sowie die Default-Interfaceimplementierung DefaultEdgeRating, bei der die Kantenkosten alle auf 1 gesetzt sind und somit den Spielrunden entsprechen.

Die Hauptklasse dient der Verwaltung von globalen Einstellungen und Parametern, die für alle Knoten und Kanten gelten und die von einer aufrufenden Klasse angepasst werden können. So wird unter anderem die maximale Tiefe des Graphen gespeichert, das ausgewählte EdgeRating, der Startknoten sowie eine Liste aller existierender Knoten. Der Startknoten wird im Konstruktur des Graphen erstellt. Zudem können für alle existierenden Knoten die visited-Flags gelöscht werden.

Die Kernlogik besitzt die innere Node-Klasse. Dabei führt jede Instanz mehrere Informationen mit sich:

- **agent:** Der aktuelle Spielerzustand des Knotens. Dieser Zustand spiegelt alle bisher simulierten Aktivitäten wider.
- **children:** Eine Liste von Knoten, die zu Beginn leer ist. Erst bei Aufruf der calculateChildren()-Methode werden diese berechnet. Auf diese Art und Weise wird sichergestellt, dass der Suchgraph nur so weit aufgebaut wird, wie er gerade benötigt wird.
- **finalNode:** Dieses Flag dient dazu, einen Knoten als Zielknoten zu markieren.
- **outgoingEdges:** Eine Liste von ausgehenden Kanten. Wie die Kinderknoten auch, ist diese Liste zu Beginn leer und wird erst bei einem Aufruf der calculateChildren()-Methode berechnet.
- **path:** Diese Liste von Kanten dient der Speicherung des Pfades von der Wurzel zum aktuellen Knoten. Sie ist relevant, um bei einem gefunden Zielknoten den Weg zu diesem nachvollziehen zu können. Im Konstruktor des Knotens wird diese automatisch aus dem Pfad des Vorgängerknotens und der zuletzt erzeugten Kante generiert. Zudem gibt es die Möglichkeit, diese Liste manuell zu setzen: Dies ist für den Suchalgorithmus später wichtig, falls ein besserer Pfad gefunden wurde und dieser für den Knoten aktualisiert werden muss.
- **visited:** Dieser boolsche Wert kann gesetzt werden um anzuzeigen, dass der Knoten bereits betrachtet wurde. Das auf diese Weise betriebene Sharing dient der Reduzierung der Laufzeit, da Knoten nicht mehrfach expandiert werden müssen.

Die wichtigsten beiden Methoden sind getChildren() und calculateChildren(). Letztere berechnet die Kinderknoten des aktuellen Knotens. Das Vorgehen ist dabei

folgendes: Der Spielerzustand des aktuellen Knotens K wird als Ausgangspunkt genutzt. Für diesen Zustand K wird für jede existierende Aktivität X überprüft, ob diese unter den gegebenen Kriterien ausführbar ist. Der resultierende Knoten sei K'. Falls X ausführbar ist, muss dem aktuellen Knoten K der Knoten K' als Kind hinzugefügt werden. Dabei wird zunächst überprüft, ob K' bereits existiert oder neu erstellt werden muss. Dazu findet die ID des Spielerzustandes von K' Anwendung: nach dieser ID wird in den der existierenden Knoten gesucht. Wenn die ID gefunden wird, dann existiert K' bereits und es genügt eine Kante von K nach K' zu erstellen. Bei nicht auffindbarer ID wird K' als neuer Knoten erzeugt. Die K' übergebene Spielerinstanz ist dabei eine Kopie der vorherigen Spielerinstanz K, nur mit bereits ausgeführter Aktivität X. Anzumerken ist, dass bei der Erzeugung der Kinder die aktuelle Tiefe des Graphen Berücksichtigung findet: Kinder, die eine Überschreitung der vom Graphen vorgegebenen maximalen Tiefe bedeuten würden, werden nicht erzeugt.
Die Methode getChildren() dagegeben überprüft, ob bereits Kinderknoten berechnet wurden. Falls ja, werden diese direkt zurückgegeben. Falls nein, wird zunächst calculateChildren() aufgerufen. Diese Methode ist wichtig, damit keine doppelten Knoten erzeugt werden. Eine einmal berechnete Menge an Kinderknoten ändert sich nicht mehr.

Die Existenz der inneren Edge-Klasse begründet sich durch die Kantenkosten: Zur Ermittlung der Kantenkosten müssen die Aktivitäten vermerkt werden. Diese gehören logisch jedoch nicht zu den Knoten. Eine reine Erweiterung der Knoten-Klasse um die Aktivitäten wäre folglich nicht intuitiv und zudem problematisch, da weitere Berechnungen ebenfalls den Start- sowie Endknoten einer Kante benötigen. Aus diesem Grund ist die Verwendung der Edge-Klasse sinnvoll. Die Hauptaufgabe besteht in der Speicherung der zugeordneten Aktivität, während anderen Informationen (child, parent) nur der Beschreibung der Kante im Graphen dienen.

Das Interface EdgeRating besteht aus zwei Methoden, die jedoch jeweils einmal überlagert sind:

- **getEdgeRating(Edge edge):** Die Kernfunktion eines EdgeRatings. Hier findet die eigentliche Definition der Kantenkosten statt.
- **getEdgeRating(List<Edge> edges):** Die überlagerte Variante für mehrere Kanten.
- **compare(int rating1, int rating2):** Vergleicht beide Ratings und gibt entsprechend des Ergebnisses eine negative Zahl (rating2 ist besser), Null (gleich) oder eine positive Zahl (rating1 ist besser) aus. Dies wird benötigt, um zwischen zu maximierenden und zu minimierenden Kantenkosten zu unterscheiden.
- **compare(Edge edge1, Edge edge2):** Die überlagerte Variante des Ratingvergleichs.

Zur Navigation durch den Graphen existieren zwei Möglichkeiten: Iterativ über die Liste der Kinder des Startknotens (top down) oder über den Pfad eines Knotens (bottom up).

6.3.2 Rating-Klasse

Die Rating-Klasse dient dem Suchalgorithmus als Bewertungsfunktion. Mit ihr ist es möglich, für jeden Knoten eine Punktzahl zu ermitteln, nach der ein Suchalgorithmus anschließend optimiert. Dabei ist die Rating-Klasse als universelle Schnittstelle für jede Bewertungsfunktion zu verstehen: Die zu verwendenden Methoden bleiben aus Sicht des Nutzers unverändert. Ausschließlich bei der Initialisierung der Rating-Klasse ist die zu verwendende Bewertungsfunktion einmalig auszuwählen. In der Implementierung selbst wird dann entsprechend der ausgewählten Funktion die Ausgabe aller weiteren Methoden bestimmt. Die in Abschnitt 5.4 besprochenen Bewertungsfunktionen stehen bei der Initialisierung zur Auswahl.

Die Verwendung der Knotenbewertungen ist wegen der Vielfalt der Bewertungsfunktionen besonders hervorzuheben. Da zwei verschiedene Ansätze zur Optimierung existieren (Minimieren und Maximieren), müssen verschiedene Bewertungsfunktionen unterschiedlich behandelt werden. Aus diesem Grund sind die Methoden getBestValue(), getWorstValue() und compare(...) implementiert, wodurch von außen eine vom Ansatz unabhängige Verwendung erfolgt, da die innere Logik alle Unterscheidungen übernimmt.
Desweiteren existieren Bewertungsfunktionen, die über mehrere Werte optimieren (bspw. Gold, Score). Aus diesem Grund kann eine Bewertung nicht aus einer Zahl bestehen, sondern muss als eine Datenstruktur wie Liste, Vektor oder Tupel implementiert werden.

Das generelle Vorgehen der Implementierung ist in zwei Schritte gegliedert: Bei der Initialisierung erfolgt die Auswahl der Bewertungsfunktion. Bei jeder weiteren die Bewertung betreffenden Methode wird daraufhin ein Switch-Case Konstrukt genutzt, um zwischen der Auswahl der Bewertungsfunktion zu unterscheiden. Entsprechend der Komplexität der Methode wird dann im Switch-Case direkt ein Wert zurückgegeben, oder eine entsprechende Submethode aufgerufen.

- **Konstruktor:** Der Konstruktor nimmt lediglich einen Integer entgegen. In der Klasse stehen mehrere public static final Integer zur Verfügung, welche jeweils einer Bewertungsfunktion entsprechen. Einer dieser Integer ist bei der Initialisierung zu wählen.
- **int compare(int[ ] value1, int[ ] value2):** Dies ist die wichtigste Methode zum Vergleich zweier Bewertungen. Verglichen wird nach der lexikographischen Ordnung. Sind value1 und value2 identisch, wird eine Null zurückgeliefert. Ist value1 besser als value2, ist die Rückgabe eine positive Zahl. Eine negative Zahl

Tabelle 6.1: Übersicht der schlechtesten und besten Werte der Bewertungsfunktionen

Bewertungsfunktion	**schlechtester Wert**	**bester Wert**
Gold	[0]	[Integer.MAX]
Score	[0]	[Integer.MAX]
Differenz	[Integer.Max]	[0]
Gold/Rounds	[0]	[Integer.MAX]
Score/Rounds	[0]	[Integer.MAX]
Gold, Score	[0, 0]	[Integer.MAX, Integer.MAX]
Gold/Rounds, Score/Rounds	[0, 0]	[Integer.MAX, Integer.MAX]
EdgeRating (GameAnalyzer)	[0]	[Integer.MAX]

wird zurückgeliefert, sollte value2 besser als value1 sein. Bei zu maximierenden Bewertungen ist der größere Wert besser, bei zu minimierenden Bewertungen der kleinere Wert. Sollte eine Bewertungsfunktion mehrwertig sein, wird der Reihe nach jeder Wert verglichen, wobei ausschließlich bei Identität zum nächsten Wert gewechselt wird.

- **int[ ] getBestValue():** Bei zu maximierenden Bewertungen entspricht dies dem Maximalwert, bei zu minimierenden Bewertungen dem Minimalwert. Bei mehrstelligen Bewertungen wird eine entsprechende Liste mit Maximal-, bzw. Minimalwerten zurückgegeben. Die Rückgabewerte sind in Tabelle 6.1 visualisiert.
- **int[ ] getWorstValue()[21]:** Analog zu getBestValue(), nur entsprechend umgedreht. Die Darstellung der Rückgabewerte befindet sich in Tabelle 6.1.
- **int[ ] getRating(Graph.Node node):** Die Hauptmethode liefert die Bewertung für einen Knoten. Der Rückgabewert ist genau der in Abschnitt 5.4 beschriebene Wert. Zwei Spezialfälle sollen jedoch noch erwähnt werden. Der erste Fall: EdgeRating. Dieses Rating summiert die Kantengewichte des Weges zum betrachteten Knoten auf und gibt diese zurück. Konkret existieren jedoch nur zwei Kantengewichte: alle Kanten kosten 1 oder das Kantengewicht des GameAnalyzers (letzteres siehe Abschnitt 6.3.3). Folglich ist die Nutzung des EdgeRatings nur dann sinnvoll, wenn der GameAnalyzer verwendet wird. Da diesem ein eigenständiges Kapitel gewidmet wird, findet hier keine Besprechung dieser Implementierung statt. Hervorzuheben ist jedoch, dass benötigte zu den Knoten führende Kanten direkt über das Pfad-Attribut des Knotens abgefragt werden können (siehe Implementierung Abschnitt 6.3.1) und deshalb nicht eigenständig als Eingabe erforderlich sind. Der zweite zu erwähnende Fall ist das Differenz-Rating. Diese Berechnung ist für ein Goldziel

[21] Auf Grund negativer Attribute ist es theoretisch möglich, dass der Score negative Werte annimmt. Dieser Sonderfall tritt im implementierten Beispiel jedoch nicht auf und wird deshalb ignoriert.

jedoch trivial, da ausschließlich die Differenz der Goldwerte von Spieler und Zielzustand ermittelt werden muss. Sollte die Differenz der beiden Null sein, ist das Ziel erreicht.

6.3.3 GameAnalyzer-Klasse

Die GameAnalyzer-Klasse implementiert die im zweiten Teil von Abschnitt 5.4 beschriebene Bewertung der Kantengewichte. Dabei ist das Ziel der Implementierung, den Suchalgorithmus durch Wissen über das Spiel und der konkreten Spielsituation des Spielers zu führen. Der GameAnalyzer empfindet dabei das menschliche Vorgehen nach: Wenn ein Mensch ein Spiel beginnt, wird er dieses in der Regel vorher grob analysieren und seine Möglichkeiten abschätzen. Anschließend plant er seine ungefähre Strategie. Der GameAnalyzer setzt genau diese beiden Schritte (im Hinblick der größtmöglichen Ausbeute) um: Vor Spielbeginn findet eine Analyse der vollständigen Spielbedingungen statt, d.h. alle existierenden Produktionsmöglichkeiten für jedes Item werden ermittelt. Dabei entsteht für jedes Item ein Produktionsteilbaum, wobei die Tiefe abhängig von den Produktionsteilbäumen der zur Produktion benötigten Items ist. Für jede Produktionsmöglichkeit wird die benötigte Zeit sowie evtl. benötigte Investitionen (z.B. Kauf von Rohstoffen statt Eigenproduktion) ermittelt. Anschließend kann dadurch für jede Runde mit den aktuellen Händlerpreisen der erzeugten Items eine Liste der ertragsreichsten Produktionsmöglichkeiten erstellt werden. Die Entscheidung, einen Produktionsweg zu wählen, entspricht dabei der Anfangs erwähnten ungefähren Strategie eines menschlichen Spielers[22].

Die Implementierung des GameAnalyzers kann in drei Logikblöcke untergliedert werden:

1. Zu Beginn des Spiels: Analyse des Spiels
2. Jede Runde: Aktualisierung des Rankings mit neuen Händlerpreisen
3. Beliebg oft: Bewertung einer Kante

Der Umfang für den ersten Block ist nach wie vor so groß, dass die Logik auf mehrere innere Hilfsklasse aufgeteilt ist.

6.3.3.1 Übersicht der Analyse des Spiels

Die Analyse des Spiels findet in fünf Schritten statt:

[22] Ungefähre Strategie deshalb, da ausschließlich die Produktionswege berechnet werden und diese sich von der Realität auf Grund von Dingen wie negativer Attribute, z.B. “Hunger”, unterscheiden können. Diese Gegebenheiten werden erst im Suchgraphen vom Suchalgorithmus berücksichtigt.

1. Analyse der Aktivitäten: Benötigt zur Ermittlung der im Spiel bestehenden Möglichkeiten. Jede Aktivität wird auf ihre Vorbedingungen hin untersucht, ebenso, welche Items sie produziert und verbraucht. Die Hilfsklasse ActivityData wurde dazu erzeugt. Sie verwaltet benötigte Informationen einer Aktivität und stellt Methoden zur Abfrage der ausgewerteten Daten bereit.
2. Analyse der Item-Abhängigkeiten: Die Produktionswege müssen bekannt sein, d.h. wenn ein Item X zur Produktion ein Item Y verwendet, besteht eine Abhängigkeit von X zu Y. Folglich muss Y im Produktionsweg von X berücksichtigt werden. Die Abhängigkeiten werden als Abhängigkeitsgraph dargestellt, was durch die Hilfsklasse ItemNode erfolgt. Jedes im Spiel existierende Item wird dabei durch eine ItemNode-Instanz repräsentiert. In den nächsten Schritten werden jedem ItemNode konkrete Produktionswege (ItemData) zugeordnet. Insofern stellt ein ItemNode des Abhängigkeitsgraphen die zentrale Verwaltung aller dieses Item betreffenden Daten und Produktionswege dar.
3. Analyse der Item-Basis: Berechnung der einfachsten Produktionswege, d.h. Wege der Länge Eins (z.B. GetreideErnten1 zum Erhalten von einer Einheit Getreide oder FleischZubereiten1 zum Erhalt von einer Einheit Wurst zu Ungunsten von drei Einheiten Fleisch). Dazu wird die Hilfsklasse ItemData verwendet, welche für einen konkreten Produktionsweg alle Daten sammelt und verwaltet. Diese Produktionswege werden dem entsprechenden ItemNode des Abhängigkeitsgraphen zugeordnet.
4. Rekursive Analyse der Items: Auf Basis der einfachsten Produktionswege werden rekursiv neue Produktionswege erzeugt, wobei die zur Herstellung benötigten Items mit den Informationen aus den bereits abgeschlossenen Produktionswegen ersetzt werden. Dazu wird die bei der Abhängigkeitsanalyse erzeugte Graphstruktur der ItemNodes genutzt: nur über die Verwandtschaftsverhältnisse der Knoten wird die korrekte Reihenfolge der Rekursion gewährleistet.
5. Erzeugen des Rankings: Alle existierenden Produktionswege (ItemData-Instanzen) werden in einer Liste gesammelt, welche anschließend nach dem Erlös (genormt auf die Produktionszeit) sortiert wird.

6.3.3.2 Analyse der Aktivitäten

Die Analyse der Aktivitäten wird fast vollständig von der ActivityData-Hilfsklasse übernommen. Die Hauptklasse konstruiert lediglich eine Map activityDataMap, mit dem Aktivitätennamen als Schlüssel und der ActivityData als Wert. Diese wird per Iteration über alle bestehenden Aktivitäten angelegt, wobei es ausreicht, für jede Aktivität eine neue Instanz der ActivityData zu erzeugen. Jede weitere Logik wird von der Hilfsklasse ausgeführt.
Nachfolgend wird die ActivityData-Hilfsklasse beschrieben. Zunächst wichtige Attribute, welche für die Funktionalität entscheidend sind:

- **Activity:** Die Aktivität wird im Konstruktor übergeben und ist Bezugspunkt der Hilfsklasse.
- **int yield:** Diese Zahl stellt den Erlös der Aktivität dar, also den Gewinn abzüglich der Kosten. Dabei bezieht sich der Erlös auf genau die eine Aktivität, d.h. auf einen einzigen Schritt.
- **Set<PreCondition> preConditions:** Die Voraussetzungen zur Ausführung der Aktivität werden mitgeführt. Dies ist später entscheidend, damit ein Agent weiß, ob ein Produktionspfad ausführbar ist.
- **Map<Item, Integer> increasedItems:** Eine Map der Items, die von dieser Aktivität erhöht werden, sowie der Wert dieser Erhöhung. Ausgenommen sind vom Händler erworbene Items.
- **Map<Item, Integer> decreasedItems:** Eine Map der Items, die von dieser Aktivität reduziert werden, sowie der Wert dieser Reduzierung. Ausgenommen sind an den Händler verkaufte Items.
- **Map<Item, Integer> boughtItems:** Eine Map der Items, die mit dieser Aktivität vom Händler gekauft werden, sowie die gekaufte Anzahl.
- **Map<Item, Integer> soldItems:** Eine Map der Items, die mit dieser Aktivität an den Händler verkauft werden, sowie die verkaufte Anzahl.

Anschließend wird die Logik der Hilfsklasse besprochen. Dafür findet eine Betrachtung der wichtigsten Methoden statt.

- **ActivityData(Activity activity):** Dies ist der Konstruktor der Hilfsklasse. Die eigentliche Analyse der Aktivität findet hier statt, d.h. alle vorher besprochenen Attribute werden gesetzt. Zur Befüllung der vier Maps findet eine Iteration über alle Effekte der übergebenen Aktivität statt. Dabei wird für jeden Effekt die Art des Effektes[23] bestimmt und die dem Effekt entsprechende Itemmap befüllt.
- **void calculateYield():** Die Anzahl der erhöhten Items wird mit deren Wert multipliziert. Anschließend wird die Anzahl der reduzierten Items mit deren Wert multipliziert und vom ersten Wert abgezogen. Die so erhaltene Zahl stellt den Erlös dar. Diese Methode muss jede Runde aufgerufen werden, damit der Erlös aktualisiert wird. Die gekauften und verkauften Items spielen für den Erlös keine Rolle, da sie diesen nicht beeinflussen: Der Verlust des Itemwertes durch dessen Verkauf entspricht logischerweise immer dem Ertrag durch den Verkauf.
- **void update():** Diese Methode wird verwendet um anzuzeigen, dass alle veränderlichen Werte neu berechnet werden müssen. Dies betrifft im konkreten Fall jedoch nur den Erlös.

[23] Relevante Effekte: IncreaseItemAmountEffect, DecreaseItemAmountEffect, BuyItemFromMerchantEffect, SellItemToMerchantEffect

- **boolean isSellingActivityForItem(String itemName):** Für die spätere Berechnung des Ratings ist es wichtig zu wissen, ob eine Aktivität eine Verkaufsaktivität für ein ausgewähltes Item ist. Deshalb überprüft diese Methode, ob das Item in der Menge der soldItems enthalten ist.
- **boolean isIncreasingItem(Item item):** Diese Funktion gibt true aus genau dann, wenn das gegebene Item von der Aktivität entweder erzeugt, oder gekauft wird[24].
- **int getIncreasedItemAmount(Item item):** Gibt den erhöhten Wert des Items zurück (keine Unterscheidung zwischen Produktion und Handel). Dabei ist vorher über die isIncreasingItem(Item item)-Methode sicherzustellen, dass das Item tatsächlich erhöht.

6.3.3.3 Analyse der Item-Abhängigkeiten

Die Analyse der Item-Abhängigkeiten ist wichtig zur korrekten Erzeugung der Produktionswege. Sie ist als simpler gerichteter Graph ohne Kantengewichtung implementiert. Zur vereinfachten Darstellung wurde die Hilfsklasse ItemNode (Repräsentant eines Knotens) verwendet, welche intern die Beziehungen zu anderen ItemNodes (d.h. Kanten) verwaltet. Das Vorgehen ist folgendermaßen: Zunächst erfolgt für jedes existierende Item die Erzeugung eines ItemNodes. Anschließend folgt eine Iteration über alle Einträge der activityDataMap, um mit der vormals definierten Analyse der Aktivitäten die Kanten zu erzeugen. Dabei werden die Listen der von der Aktivität verringerten und erhöhten Items abgefragt, um so für jeden ItemNode aus der Liste der verringerten Items eine Kante zu jedem ItemNode aus der Liste der erhöhten Items zu ziehen.

Beispiel: Die Aktivität "MehlMahlen1" reduziert Getreide um 3 und erhöht Mehl um 1, d.h. in der Liste der erhöhten Items ist Mehl enthalten und in der Liste der reduzierten Items ist Getreide enthalten. Folglich wird für den Eintrag "Getreide" der verringerten Items der passende ItemNode ausgewählt und eine Kante zu den ItemNodes der erhöhten Items, hier "Mehl", gezogen. Die Interpretation der Kante von Getreide nach Mehl: Zur Produktion von Mehl wird Getreide benötigt.

Zusätzlich wird überprüft, ob eine Aktivität ein Item verkauft, und dann ggf. im ItemNode als spezielle Verkaufsaktivität abgelegt. Diese Logik wird jedoch erst später bei der Kantenbewertung benötigt. Um von der Hauptklasse aus eine einfache Verwendung des Abhängigkeitsgraphen zu ermöglich, wurde eine Map mit dem Namen dependencyGraph erzeugt, welche einen Eintrag "Itemname nach ItemNode" für jeden ItemNode besitzt. Dabei muss die Namensgebung noch kurz erläutert werden: der Abhängigkeitsgraph selbst ist bereits durch die Verknüpfung der ItemNodes gegeben. Die Map liefert jedoch für jedes Item den passenden

[24]Spätere Berechnungen unterscheiden nicht mehr zwischen gekauften und produzierten Items.

Einstiegspunkt im Graphen. Andernfalls müsste im Graphen zuerst nach dem passenden ItemNode gesucht werden.

Nachdem die Konstruktion des Abhängigkeitsgraphen beschrieben wurde, folgt die Betrachtung der Hilfsklasse ItemNode. Dabei ist anzumerken, dass die ItemNode-Klasse bereits Funktionalität besitzt, welche über dieses Unterkapitel hinaus geht.

Attribute:

- **Set<ItemNode> parents:** Dieses Attribut repräsentiert die Elternknoten.
- **List<ItemData> closedItemData:** Bei der Analyse der Produktionswege werden die verschiedenen Produktionsmöglichkeiten ihrem entsprechenden ItemNode zugeordnet. Aus diesem Grund wird die Liste closedItemData mitgeführt und verwaltet. Dabei findet eine Unterscheidung zwischen geschlossenen und offenen ItemData-Instanzen statt. Dies dient ausschließlich der Konstruktion. Eine ItemData-Instanz zählt genau dann als geschlossen, wenn alle Abhängigkeiten aufgelöst wurden. Die Logik dahinter ist jedoch an anderer Stelle implementiert.
- **List<ItemData> openItemData:** Das Gegenstück zur closedItemData. Eine ItemData-Instanz zählt genau dann als offen, wenn noch unaufgelöste Abhängigkeiten (im Sinne des Abhängigkeitsgraphen) existieren.
- **List<ActivityData> sellItemActivities:** Die vormals erwähnte Auflistung aller Aktivitäten, welche das Item des betrachteten ItemNodes verkaufen. Diese wird erst später bei der Bewertung der Kanten benötigt.

Die wichtigsten Methoden sind nachfolgend aufgelistet. Dabei sind zur Schaffung eines strukturellen Überblicks auch triviale Methoden enthalten:

- **void addParent(ItemNode itemNode):** Ein Elternknoten wird gesetzt. Diese Methode dient der Kantensetzung.
- **Set<ItemNode> getParents():** Bei der Analyse der Produktionswege werden die Abhängigkeiten durch die vorhandenen Elternknoten dargestellt, deshalb müssen diese abgefragt werden können.
- **void addSellItemActivity(ActivityData activityData):** Fügt eine ActivityData zu der Liste der Verkaufsaktivitäten hinzu.
- **List<ActivityData> getSellActivities():** Fragt alle ActivityData-Instanzen ab, welche das im aktuellen ItemNode beschriebene Item verkaufen.
- **void addItemData(ItemData itemData):** Verwendet die itemData.isOpen()-Methode, um die gegebene itemData-Instanz passend zur Liste der offenen oder geschlossenen ItemData-Instanzen hinzuzufügen.
- **void removeOpenItemData(ItemData itemData):** Entfernt die angegebene offene ItemData-Instanz.

- **void updateItemData(ItemData itemData):** Wird bei der Berechnung der Produktionswege zur Aktualisierung der offenen und geschlossenen ItemData-Instanzen benötigt. Dazu wird die gegebene Instanz aus beiden Listen entfernt und über die addItemData(ItemData itemData)-Methode wieder passend eingeordnet.
- **boolean hasOpenItemData():** Liefert genau dann "true" zurück, wenn openItemData nicht leer ist.

6.3.3.4 Basis Analyse der Produktionswege

Bevor die eigentliche Berechnung aller Produktionswege beginnen kann, muss zunächst die Basis dieser Berechnungen erzeugt werden. Diese Basis stellen all jede Produktionswege dar, welche genau die Länge Eins besitzen. Dies bedeutet, dass in der Basis für jede Aktivität, die ein Item erzeugt, ein Produktionsweg erzeugt und dem entsprechenden ItemNode zugeordnet wird. Dabei ist es irrelevant, ob die betrachtete Aktivität noch weitere Abhängigkeiten besitzt. Diese werden für den Basisschritt ignoriert und erst in der Rekursion aufgelöst.
Ein Produktionsweg wird durch die Hilfsklasse ItemData beschrieben, welche die Daten für die Produktion dieses Items verwaltet. Die Implementierung der Basis erfolgt daraufhin mittels einer verschachtelten Schleife: Für jedes Item item wird jede ActivityData-Instanz activityData betrachtet. Liefert die activityData.isIncreasingItem(item)-Methode dabei true zurück, wird eine neue ItemData-Instanz itemData für das Item item erzeugt, über die Methode itemData.addActivityData(activityData, 1) die Daten der Aktivität hinzugefügt und die itemData-Instanz anschließend dem entsprechenden ItemNode zugeordnet.
Es folgt eine Betrachtung der ItemData-Klasse. Diese besitzt zudem bereits Logik zur Ermittlung des Erlöses, zur rekursiven Analyse und zur Kantenbewertung, da diese logisch der Hilfsklasse zugeordnet werden können.

Zunächst eine Erläuterung der Klassen-Attribute. Es ist hervorzuheben, dass einige Klassenattribute denen der ActivityData-Klasse ähneln. Dabei ist der Betrachtungspunkt jedoch ein anderer: Während sich die einen Attribute auf eine konkrete Aktivität beziehen, beziehen sich die anderen auf einen Pfad von Aktivitäten und haben somit einen globaleren Anspruch.

- **int yield:** Der Erlös bezieht sich hierbei auf den vollständigen Produktionsweg. Dies bedeutet, dass Gewinn und Kosten verrechnet und durch die benötigte Zeit (d.h. Anzahl der Runden) geteilt werden.
- **Set<PreCondition> preConditions:** Für die Entscheidung eines Agenten, einen Produktionsweg auszuwählen, benötigt dieser das Wissen darüber, ob die Aktivitäten generell ausführbar sind. Zu diesem Zweck werden die Voraussetzungen aller Aktivitäten in einer Menge gesammelt. Der Produktionsweg kann

folglich nur ausgeführt werden, wenn jede einzelne Aktivität (d.h. die Menge der PreConditions) ausführbar ist.

- **Map<Item, Integer> costs:** Die Kosten beziehen sich hierbei nicht auf den Erlös, sondern werden für die rekursive Berechnung benötigt. Dabei werden für jede hinzugefügte Aktivität die Kosten notiert und anschließend Schritt für Schritt aufgelöst. Das bedeutet, dass ein Produktionspfad erst dann "geschlossen" ist, wenn die Map der Kosten leer ist.
- **List<ActivityData> activityData:** Die Liste aller dem Produktionsweg hinzugefügten Aktivitäten. Dabei spielt die Reihenfolge der Liste eine entscheidende Rolle, da diese die Reihenfolge der Ausführungen (Aktivität an Position Null zuletzt) festlegt.[25]
- **List<Integer> activityDataCounter:** Diese Liste korreliert direkt mit der activityData-Liste, d.h. ein Eintrag an Position i dieser Liste bezieht sich auf einen Eintrag an der gleichen Position i der anderen Liste. Folglich sind beide Listen immer identischer Länge. Diese Liste gibt an, wie oft eine zugeordnete Aktivität ausgeführt werden muss.[26]
- **int increasedItemAmount:** Aktivitäten produzieren nicht ausschließlich eine Einheit eines Items, sondern beliebig viele. Aus diesem Grund wird die Anzahl der vom Zielitem produzierten Menge mitgeführt. Diese wird in der Rekursion der Analyse dazu verwendet, die passende Anzahl an auszuführenden Aktivitäten zu ermitteln.

Die wichtigsten Methoden:

- **void addActivityData(ActivityData activityData, int counter):** Diese Methode fügt eine neue activityData-Instanz hinzu. Es wird eine Logik zur Aufbearbeitung der Kosten costs ausgeführt, welche entscheidend für die Funktionsweise der rekursiven Analyse ist. Die Logik verwendet die von der übergebenen activityData geführten Maps der erhöhten, reduzierten und gekauften Items. Für jede dieser Maps wird dabei die Kostenmap angepasst: Erhöht die Aktivität ein Item der Kosten, werden die Kosten um diese Erhöhung reduziert (bis minimal Null). Das Gleiche gilt für durch die Aktivität gekaufte Items. Reduziert die Aktivität ein Item, wird dieses Item in die Kosten aufgenommen oder, falls schon vorhanden, die Kosten erhöht. Anzumerken ist, dass bei der Aktualisierung erzeugte Werte den in der Methode übergebenen Zähler counter berücksichtigen. Dieser gibt an, wie oft die activityData ausgeführt werden muss, folglich müssen auch die Kosten mit demselben Faktor

[25] Die Reihenfolge muss auf Grund der Abhängigkeiten berücksichtigt werden. So kann z.B. nicht MehlMahlen vor GetreideErnten ausgeführt werden, da das benötigte Getreide erst produziert werden muss.

[26] Die Aktivität MehlMahlen1 benötigt beispielsweise drei Einheiten Getreide, während GetreideErnten1 nur eine Einheit Getreide erzeugt. Folglich muss für eine Ausführung von MehlMahlen1 dreimalig die Aktivität GetreideErnten1 ausgeführt werden.

multipliziert werden. Auf diese Weise sind für jede aufgenommene Aktivität die Kosten immer korrekt, bis sie am Ende der Rekursion letztlich Null (d.h. leer) sind.
Desweiteren werden die Vorbedingungen der Aktivität zum preConditions-Set hinzugefügt, damit immer die Ausführbarkeit des Produktionsweges berechnet werden kann. Eine letzte Besonderheit existiert, sollte die übergebene activityData-Instanz der erste Eintrag der activityData-Liste sein. In diesem Fall stellt die zugehörige Aktivität die letzte auszuführende Aktivität des Produktionsweges dar und bestimmt somit automatisch das zu produzierende Item. Insofern wird die Variable increasedItemAmount genau mit der Anzahl des von dieser Aktivität produzierten Items gesetzt.

- **void addActivityData(List<activityData> activityData, List<Integer> counter):** Diese Methode fügt gleich mehrere activityData-Instanzen hinzu. Dazu werden die Listen iterativ durchlaufen und für jeden Eintrag die Methode addActivityData(activityData.get(i), counter.get(i)) aufgerufen. Diese Methode wird bei der rekursiven Analyse dazu benötigt, die vollständige ActivityData eines Teilpfades zu einem übergeordneten Pfad hinzuzufügen.
- **int getActivityPosition(Activity activity):** Fragt die aktuelle Position der gegebenen Aktivität in der activityData-Liste ab.Gibt "-1" zurück, sollte die Aktivität nicht enthalten sein. Diese Methode wird zur Berechnung der Kantenkosten benötigt.
- **boolean isOpen():** Ein Produktionsweg ist genau dann offen, wenn nicht alle Kosten vollständig aufgelöst (d.h. Null, bzw. leer) sind. Insofern ist der Rückgabewert "!costs.isEmpty()".
- **void calculateYield():** Der Erlös des Produktionsweges wird in dieser Methode berechnet. Dazu wird die getYield()-Methode jeder zugeordneter activityData-Instanz aufgerufen und mit der Anzahl der benötigten Activity-Ausführungen (activityDataCounter) multipliziert. Über diese Werte wird summiert und anschließend durch die benötigten Runden geteilt. Die benötigte Anzahl der Runden ergibt sich durch die Aufsummierung aller Werte des activityDataCounters.
- **void update():** Diese Methode stellt die Aufforderung dar, alle variablen Werte neu zu berechnen, was in diesem Fall dem Aufruf der calculateYield()-Methode entspricht.

6.3.3.5 Rekursive Analyse der Produktionswege

Die rekursive Analyse der Produktionswege basiert auf der zu diesem Zeitpunkt bereits durchgeführten Basisanalyse aus Abschnitt 6.3.3.4. Ziel ist es, die in den ItemData-Instanzen bestehenden Kosten bis auf Null zu verringern und stattdessen

```
List<ItemNode> closed = new ArrayList<ItemNode>();
List<ItemNode> open = new ArrayList<ItemNode>(dependencyGraph.values());
int itemNodeIterator = 0;
int itemDataIterator = 0;

while (!open.isEmpty()) {

  ItemNode currItemNode = open.get(itemNodeIterator);
  itemNodeIterator++;

  if (!closed.containsAll(currItemNode.getParents())) {
    continue;
  }

  while (currItemNode.hasOpenItemData()) {

    ItemData currItemData = currItemNode.getOpenItemData().get(itemDataIterator);
    itemDataIterator++;

    if (!currItemData.isOpen()) {
      continue;
    }

    Item neededItem = new ArrayList<Item>(currItemData.getCosts().keySet()).get(0);
    int neededItemAmount = currItemData.getCosts(neededItem);
    ItemNode costsItemNode = dependencyGraph.get(neededItem.toString());

    for (ItemData costsItemData : costsItemNode.getClosedItemData()) {

      ItemData newItemData = createNewItemDataWithUpdatedCosts(currItemData,
          costsItemData, neededItemAmount);
      currItemNode.addItemData(newItemData);
    }
    currItemNode.removeOpenItemData(currItemData);
    itemDataIterator = 0;
  }

  closed.add(currItemNode);
  open.remove(currItemNode);
  itemNodeIterator = 0;
}
```

Listing 6.1: Algorithmus zur rekursiven Analyse der Produktionspfade (Java)

```
1  private ItemData createNewItemDataWithUpdatedCosts(ItemData mainItemData, ItemData
       costsItemData, int neededItemAmount) {
2
3    ItemData newItemData = new ItemData(mainItemData.getItemName());
4
5    HashSet<PreCondition> preConditions = new HashSet<PreCondition>();
6    preConditions.addAll(mainItemData.getPreConditions());
7    preConditions.addAll(costsItemData.getPreConditions());
8    newItemData.setPreConditions(preConditions);
9
10   int offeredItemAmount = costsItemData.getIncreasedItemAmount();
11   int neededTimes = (int) Math.ceil((double) neededItemAmount / (double)
         offeredItemAmount);
12   newItemData.addActivityData(mainItemData.getActivityData(), mainItemData.
         getActivityDataCounter());
13   for (int i = 0; i < costsItemData.getActivityData().size(); i++) {
14     newItemData.addActivityData(costsItemData.getActivityData().get(i),
           costsItemData.getActivityDataCounter().get(i) * neededTimes);
15   }
16
17   newItemData.update();
18
19   return newItemData;
20 }
```

Listing 6.2: createNewItemDataWithUpdatedCosts-Methode (Java)

die entsprechenden Aktivitäten hinzuzufügen, welche die in den Kosten notierten Items erzeugen. Die Kosten entsprechen dabei den Abhängigkeiten und dienen nicht der Erlösberechnung. Bestehende Produktionswege werden im Algorithmus nicht verändert, sondern ausschließlich neue Instanzen angelegt. Die ItemNode-Klasse besitzt zu diesem Zweck zwei Listen mit offenen und geschlossenen ItemData-Instanzen. Geschlossen ist eine ItemData-Instanz genau dann, wenn sie keine Kosten mehr besitzt, d.h. alle Abhängigkeiten aufgelöst wurden. Auf Grund der Komplexität des Algorithmus ist dieser in Listing 6.1 visualisiert und wird folgend Schritt für Schritt besprochen.

Zunächst initialisiert der Algorithmus zwei Listen mit offenen und geschlossenen ItemNodes, wobei erstere Liste mit allen ItemNodes des Abhängigkeitsgraphen gefüllt wird und die zweite Liste zu Beginn leer ist. Dabei sind diese Listen der offenen und geschlossenen ItemNodes nicht mit den Listen innerhalb der ItemNodes für offene und geschlossene ItemData zu verwechseln. Die hier erzeugten Listen beziehen sich auf alle im ItemNode enthaltenen Produktionswege. Ein ItemNode ist genau dann geschlossen, wenn alle in ihm enthaltenen offenen ItemData-Instanzen betrachtet und entsprechend der Abhängigkeiten aufgelöste neue ItemData-Instanzen erzeugt sind.[27] Die offenen ItemData-Instanzen bleiben in ihrem Zustand bestehen, werden folglich also auch nicht gelöscht.

[27] Die alternative Formulierung "Ein ItemNode ist genau dann geschlossen, wenn alle in ihm enthaltenen ItemData-Instanzen geschlossen sind." hätte die Kernidee zwar erfasst, wäre aber dennoch falsch, da die offenen ItemData-Instanzen nicht verändert oder gelöscht werden (also noch vorhanden sind).

Der Hauptalgorithmus besteht aus einer while-Schleife (Zeile 6), die sich auf die Größe der offenen ItemNodes bezieht, da die genaue Anzahl der Schleifen vorher nicht bekannt ist. Ein ItemNode currItemNode kann jedoch nur dann betrachtet werden, wenn alle ItemNodes, von denen er abhängig ist, bereits geschlossen sind. Aus diesem Grund wird überprüft, ob die Elternknoten von currItemNode vollständig in der closed-Liste enthalten sind (Zeile 11) (d.h. die Abhängigkeiten bereits aufgelöst wurden). Falls nicht, erfolgt ein continue mit dem nächsten ItemNode.
Sind alle Elternknoten des currItemNodes bereits geschlossen, wird die innere while-Schleife ausgeführt (Zeile 15). Diese wird so lange iteriert, bis der currItemNode geschlossen ist. Dazu wird in jedem Iterationsschritt eine offene ItemData-Instanz currItemData abgearbeitet und eine oder mehrere entsprechende neue ItemData-Instanzen newItemData werden erzeugt. Dabei muss newItemData nicht zwangsweise geschlossen sein, sondern kann ebenfalls offen sein und somit in einem weiteren Iterationsschritt betrachtet werden.[28] Zunächst wird ein benötigtes Item neededItem zur Kostenreduzierung ausgewählt[29], sowie die benötigte Anzahl neededItemAmount (Zeile 24+25).
Für das neededItem kann anschließend aus dem Abhängigkeitsgraphen der entsprechende ItemNode costsItemNode erfragt werden (Zeile 26). Dadurch ist es möglich, über eine For-Schleife aller ItemData-Instanzen costsItemData des costsItemNodes, die neuen ItemData-Instanzen newItemData zu erzeugen (Zeile 30). An dieser Stelle findet die mehrfach erwähnte Multiplikation der Produktionswege statt, da für jede Möglichkeit neue ItemData-Instanzen erzeugt werden. Dies setzt sich entsprechend fort für weitere ItemNodes. Die genaue Vorgehensweise zur Erzeugung der newItemData ist in der Methode createNewItemDataWithUpdatedCosts(currItemData, costsItemData, neededItemAmount) beschrieben, welche in Listing 6.2 zu finden ist und im Anschluss noch erläutert wird. Die erzeugten newItemData werden dem currItemNode hinzugefügt.
Nach der For-Schleife ist currItemData fertig betrachtet und zählt folglich für den currItemNode nicht mehr als Kritierium für den Zustand “offen”. Aus diesem Grund wird currItemData anschließend aus den offenen ItemData-Instanzen des currItemNode entfernt (Zeile 33).
Ist die innere While-Schleife abgeschlossen, wurden alle ItemData-Instanzen abgearbeitet und der currItemNode ist geschlossen. Folglich kann er zu closed hinzugefügt werden (Zeile 37) und muss zudem aus open entfernt werden (Zeile 34).

[28] Die zu betrachtenden ItemData-Instanzen können somit in der Anzahl sowohl steigen als auch fallen. Die Terminimierung ist jedoch sicher gestellt, da in jeder neuen Instanz die Kosten reduziert werden.

[29] Im Code wird immer das erste Element des KeySets der Kosten ausgewählt (Zeile 24). Dabei darf nicht der falsche Eindruck entstehen, es würden nicht alle Kosten berücksichtigt werden oder es würden immer nur Kosten mit einem Item existieren. Es wird zwar ausschließlich ein Kosteneintrag pro Iteration betrachtet, die verbleibenden Kosten werden jedoch in der neu erzeugten ItemData-Instanz berücksichtigt und folglich in einer späteren Iteration weiter abgearbeitet.

Nach Abschluss des Algorithmus wurden alle ItemNodes geschlossen und alle Produktionswege folglich fertig berechnet.

Abschließend wird die Hilfsmethode createNewItemDataWithUpdated-Costs(currItemData, costsItemData, neededItemAmount) beschrieben (siehe Listing 6.2), welche im vormals erläuterten Algorithmus zur Erzeugung einer neuen ItemData-Instanz verwendet wird. Diese Methode erhält zwei ItemData-Instanzen (d.h. zwei Produktionspfade): einmal mainItemData, welches die gerade betrachtete offene ItemData-Instanz darstellt, sowie die geschlossenen costsItemData, welche den Produktionspfad des benötigten Items beinhaltet. Außerdem wird ein Integer neededItemAmount übergeben, welcher die Anzahl der benötigten, von costsItem-Data produzierten, Items beschreibt.
Ziel ist die Erstellung einer neuen ItemData-Instanz, welche auf der currItemData basiert, jedoch um die Produktionspfade von costsItemData erweitert wurde (und somit die Abhängigkeiten reduziert). Dazu wird zunächst eine neue Instanz newItemData angelegt, welche auf demselben Item wie mainItemData basiert (Zeile 3). Anschließend folgt eine Aufsummierung der Vorbedingungen der mainItemData und costsItemData als Menge, welche daraufhin der newItemdata zugeordnet wird (Zeile 5 bis 8).
Es ist zwar bekannt, wie oft das Item der costsItemData gebraucht wird, dies entspricht jedoch nicht automatisch der Anzahl der benötigten Ausführungen der costsItemData, da diese nicht immer genau ein Item produziert. Aus diesem Grund muss die tatsächlich benötigte Anzahl der Ausführungen von costsItemData berechnet werden. Dazu wird zunächst ermittelt, wie viele Items costsItemData produziert (Zeile 10). Anschließend wird die kleinste Ganzzahl berechnet, welche größer als das Ergebnis der Rechnung "neededItemAmount / offeredItemAmount" ist (Zeile 11). Diese entspricht den benötigten Ausführungen neededTimes.
Danach können der newItemData alle benötigten ActivityData hinzugefügt werden.[30] Zunächst wird in einem Schritt die ActivityData (sowie deren Counter) der mainItemData übernommen (Zeile 12). Anschließend findet eine Iteration über die ActivityData der costsItemData statt. In dieser Schleife wird jede Aktivität hinzugefügt, der Counter jedoch noch mit dem berechneten Wert neededTimes multipliziert. Somit wird die korrekte Anzahl der Ausführungen jeder Aktivität sichergestellt.
Am Ende findet noch ein Aufruf der update()-Methode des newItemData statt, wodurch der Erlös neu kalkuliert wird, bevor die erzeugte Instanz letztlich zurück gegeben wird.

[30] Die Reihenfolge, in welcher die ActivityData-Instanzen hinzugefügt werden, spielt eine entscheidende Rolle. Neue Teilproduktionspfade müssen immer am hinteren Ende der ActivityData-Liste anzugefügt werden. Andernfalls kann die korrekte Ausführungsreihenfolge nicht gewährleistet werden, da Situationen entstehen könnten, in denen Aktivitäten ausgeführt werden sollen, deren Abhängigkeiten noch nicht ausgeführt wurden.

6.3.3.6 Erzeugen des Rankings

Durch die bereits geleistete Vorarbeit erfolgt die Erzeugung des Rankings schnell und einfach. Die ItemData-Klasse wurde um das Interface Comparable<ItemData> erweitert, wodurch Instanzen dieser Klasse miteinander verglichen werden können. Dazu reicht die Implementierung der Methode "int compareTo(ItemData o)" aus. Diese verwendet den Erlös als Vergleichswert. Anschließend ist nur noch die Erstellung einer Liste<ItemData> ranking durchzuführen und alle geschlossenen ItemData-Instanzen aller ItemNodes hinzuzufügen. Durch das zuvor implementierte Comparable-Interface kann die Liste ranking anschließend über einen Aufruf von Collections.sort(ranking) nach dem Erlös sortiert werden.
Mit diesem Schritt ist die einmalige Analyse des Spiels vor Beginn desselben abgeschlossen.

6.3.3.7 Aktualisierung des Rankings

Die Aktualisierung des Rankings muss zu Beginn einer neuen Runde geschehen, damit die Berechnungsgrundlage der Agenten immer an die aktuelle Spielsituation angepasst ist. Zu diesem Zweck sind die ActivityData-Klasse und die ItemData-Klasse bereits mit der Methode update() ausgestattet. Diese aktualisiert für jede ActivityData-Instanz den Erlös der Aktivität auf Grundlage der aktuellen Händlerpreise. Die update()-Methode der ItemData-Klasse wiederum basiert auf dem Erlös der enthaltenen ActivityData-Instanzen, insofern berechnet diese anschließend den Erlös mit der zuvor aktualisierten ActivityData neu.
Nachdem alle Erlösberechnungen aktualisiert wurden, erfolgt nur noch eine Neusortierung der Liste ranking, wodurch die Aktualisierung vollständig abgeschlossen ist.

6.3.3.8 Bewertung einer Kante

Das eigentliche Ziel des GameAnalyzers ist die Bewertung von Kanten, um so eine alternative Bewertungsfunktion für die Graphsuche der Agenten zu liefern.[31] Dabei werden Kanten innerhalb ihres Kontextes betrachtet: die Bewertung einer Kante hängt vom Startknoten (d.h. dem Spielerzustand) derselben ab. Die implementierte Methode nimmt deshalb eine Spielerinstanz[32] currState (entspricht einem Knoten)

[31] Durch die Lieferung von Kantenbewertungen kann mittels der Rating-Klasse über die Kantengewichte summiert werden, damit so jedem Knoten der Wert des aktuell besten Pfades zugeordnet werden kann. Dies bedeutet, dass die Suchalgorithmen nicht angepasst werden müssen, sondern diese weiterhin auf Knotenbewertungen basieren können. Dies geschieht jedoch nicht in der GameAnalyzer-Klasse - diese bietet ausschließlich die Funktionalität zur Kantenbewertung.

[32] Für den GameAnalyzer ist es unerheblich, ob die Spielerinstanz von der Unterklasse Agent ist oder nicht. Insofern werden "Spieler" und "Agent" in diesem Abschnitt der Implementierung gleichbedeutend verwendet.

```
1  ItemData bestItemData = getBestItemDataSet(currState);
2  String finalItem = bestItemData.getItemName();
3  ActivityData currActivityData = activityDataMap.get(activity.toString());
4  boolean currActivityIsBestSellActivity = isThereNoBetterTradeActivity(currState,
       currActivityData);
5
6
7  if (isGoalReachableWhenSellingItems(currState, dataHolder.getGoal()) &&
       currActivityIsBestSellActivity) {
8
9    return 100;
10
11 } else if (currActivityData.isSellingActivityForItem(finalItem)
12     && currState.getItemManager().getItemAmount(finalItem) >= bestItemData.
           getIncreasedItemAmount()
13     && currActivityIsBestSellActivity) {
14
15   return 100;
16
17 } else if (bestItemData.getActivityPosition(activity) >= 0) {
18
19   ActivityData nextActivityData = null;
20
21   for (ActivityData activityData : bestItemData.getActivityData()) {
22     if (activityData.getActivity().isExecutable(currState, game)) {
23       nextActivityData = activityData;
24       break;
25     }
26   }
27   if (nextActivityData.getActivityName().equals(currActivityData.getActivityName()))
         {
28     return 60;
29   }
30 }
31
32 return 0;
```

Listing 6.3: GameAnalyzer: Bewertung einer Kante (Java)

sowie eine Aktivität activity (entspricht einer Kante) entgegen und gibt das Rating zurück.
Im Listing 6.3 findet sich die genaue Umsetzung der Kantenbewertung, welche folgend betrachtet wird. Zunächst wird aus der vormals berechneten Liste ranking der für den Spieler beste Produktionsweg gewählt (Zeile 1). Dies erfolgt über die getBestItemDataSet(currState)-Methode, welche das Ranking iterativ durchgeht und den ersten Eintrag (d.h. ItemData) zurückliefert, für welchen der Spieler alle Vorbedingungen erfüllt. Diese ItemData-Instanz entspricht genau dem besten vom Spieler ausführbaren Produktionspfad.
Die darauf folgende Auswertung basiert auf einer Fallunterscheidung, wobei die Fälle der Reihe nach abgearbeitet werden. Das heißt, dass bei Eintreffen eines Falls die darauffolgenden Fälle ignoriert werden.

1. **Der Itemwert des Spielers in Gold entspricht mindestens dem Goldwert des Zielzustandes (Zeile 6):** Befindet sich der Spieler in diesem Zustand, erreicht er das Ziel am schnellsten, wenn er seinen Itembestand verkauft. Dies bedeutet, dass die bestmöglichen Verkauf-Aktivitäten die beste Wertung (hier: 100) erhalten müssen, andere Aktivitäten dagegen die Wertung 0. Dabei spielt es keine Rolle, welches Item verkauft wird, sofern nur die größtmögliche Menge dieses Items in einem Schritt gehandelt wird. Die Methode isThereNoBetterTradeActivity(currState, currActivityData) prüft nach der besten ausführbaren Handelsaktivität, indem über alle existierenden Handelsaktivitäten iteriert und jene mit der größtmöglichen Verkaufsmenge ausgewählt wird. Das heißt, wenn die betrachtete Aktivität der berechneten besten Handelsaktivität entspricht, ist der Rückgabewert “true”. Dabei bezieht sich die Berechnung ausschließlich auf das Item, welches mittels der aktuellen Methode verkauft wird (d.h. nur Verkaufsaktivitäten dieses Items finden Betrachtung).
2. **Die finalen Items des Produktionspfades wurden bereits produziert (Zeile 11):** Verkaufsaktivitäten sind nicht in den Produktionswegen enthalten[33]. Aus diesem Grund muss in jeder Bewertung gesondert geprüft werden, ob die finalen Items bereits produziert worden sind und verkauft werden können. Eine betrachtete Aktivität erhält deshalb nur dann die beste Wertung, wenn folgende Kriterien erfüllt sind:
   - Die Aktivität muss die produzierten Items verkaufen
   - Der Spieler muss bereits die am Ende der Produktionskette produzierten Items im Inventar haben
   - Die Aktivität muss die beste verfügbare Verkaufsaktivität dieses Items sein, d.h. sie muss die größtmögliche Menge verkaufen. Die Überprüfung

[33] Technisch wäre es möglich, die Verkaufsaktivitäten in die Produktionspfade zu integrieren. Dies würde jedoch zu einer weiteren Multiplikation der Möglichkeiten führen, da produzierte Items sich nicht immer in einem Schritt verkaufen lassen. Zur Reduzierung der Anzahl der Produktionswege wurde deshalb darauf verzichtet.

nach einer besseren Handelsaktivität wird wie zuvor durch die isThereNoBetterTradeActivity(currState, currActivityData)-Methode durchgeführt.

3. **Die betrachtete Aktivität ist Teil des Produktionsweges (Zeile 17):** Falls die vorherigen beiden Fälle nicht eintreffen, wird der Regelfall zur Bewertung der Aktivitäten betrachtet: Die Aktivität ist im Produktionsweg enthalten und es ist zu prüfen, ob diese auch auszuführen ist, da es andere Aktivitäten der Produktionskette geben könnte, welche eine höhere Priorität besitzen[34]. Zu diesem Zweck wird die Liste der Aktivitäten durchlaufen und für jeden Eintrag geprüft, ob dieser für den Spieler ausführbar ist. Die erste gefundene, ausführbare Aktivität entspricht jener, welche als nächstes im Produktionsweg ausgeführt werden sollte. Das heißt, sollte die gefundene Aktivität gleich der gerade betrachteten Aktivität sein, kann ein guter Wert (hier: 60)[35] zurückgegeben werden.
4. **Keine der vorherigen Fälle tritt ein (Default-Fall, Zeile 32):** In diesem Fall hat die betrachtete Aktivität keine Relevanz für den besten Produktionsweg. Eine solche Aktivität erhält ein entsprechend schlechtes 0-Rating.

6.3.4 GameAnalyzerEdgeRating-Klasse

Die GameAnalyzerEdgeRating-Klasse implementiert das in Abschnitt 6.3.1 beschriebene EdgeRating Interface zur Definition einer Kantenkostenfunktion. Dazu wird die GameAnalyzer-Klasse verwendet, mit deren Hilfe die Kosten jeder Kante ermittelt werden. Die eigentliche Logik findet folglich im GameAnalyzer statt. Zur Verwendung des GameAnalyzerEdgeRatings muss für einen Suchalgorithmus das Rating "EdgeRating" ausgewählt werden, andernfalls werden die Kantenkosten ignoriert.

Die Implementierung ist kurz und verwendet lediglich die getRating(Edge edge)-Methode des GameAnalyzers:

- **int getEdgeRating(Edge edge):** Über die Kante wird die zugeordnete Aktivität und der Spieler des Elternknotens ermittelt, damit so die Kantenkosten über einen Aufruf der gameAnalyzer.getRating(Player player, Edge edge)-Methode berechnet und zurückgegeben werden können.
- **getEdgeRating(List<Edge> edges):** Die Implementierung erfolgt über eine Iteration aller Kanten und die Aufsummierung der Kantenkosten, welche mit der getEdgeRating(Edge edge)-Methode berechnet werden.

[34] Die Priorität entspricht der Reihenfolge der ActivityData-Liste, d.h. Eintrag 0 besitzt die höchste Priorität.

[35] Produzierende Schritte besitzen einen etwas schlechteren Wert als verkaufende Schritte. Diese Priorisierung ist deshalb sinnvoll, damit Items nicht unnötig im Inventar des Spielers liegen, während dieser weiter produzieren und die Itempreise fallen.

- **compare(int rating1, int rating2):** return rating1 - rating2;
- **compare(Edge edge1, Edge edge2):** Die Kantenkosten von edge1 und edge2 werden ermittelt und anschließend der Methode compare(rating1, rating2) übergeben.

6.3.5 AbstractGraphSearch-Klasse

Die AbstractGraphSearch-Klasse stellt ein allgemeines Konstrukt zur Implementierung eines beliebigen Suchalgorithmus dar. Aus Sicht des Agenten ist jede Suche eine AbstractGraphSearch, d.h. die dahinter liegende Implementierung ist für diesen unbekannt. Dadurch ist eine Erweiterung, bzw. ein Austausch des Suchalgorithmus ohne Probleme möglich. Die AbstractGraphSearch-Klasse bietet dabei jedoch schon implementierte Funktionalität an, wodurch die Implementierung einer Suche weiter vereinfacht wird. Konkret handelt es sich um Methoden zur Beurteilung von Knoten sowie zur Ausführung der Suche. Zur Beurteilung wird dabei eine Instanz der Rating-Klasse verwendet, welche einmalig zur Initialisierung gesetzt werden muss.
Die AbstractGraphSearch-Klasse implementiert das Runnable-Interface und ist somit in einem eigenen Thread ausführbar. Dies ist erforderlich, damit die Suche parallel und unabhängig der eigentlichen Spiellogik ausgeführt werden kann. Auch ist dies Grundbedingung zur Ausführung mehrerer Agenten, da diese sonst nicht parallel suchen könnten. Ein weiteres wichtiges Kriterium für die Spielbarkeit ist zudem, dass ein menschlicher Spieler nicht auf die Entscheidungen der Agenten warten muss. Zur Erfüllung dieses Kriteriums wurden bei der Implementierung InterruptExceptions verwendet, damit der Thread jeder Zeit korrekt abgebrochen werden kann. Desweiteren speichert der Algorithmus Zwischenergebnisse ab. Bei Versetzung des Spiels in die nächste Runde ist lediglich das Zwischenergebnis zurückzugeben. Es muss nicht auf das Ende der Suche gewartet werden.

Nachfolgend findet eine genauere Beschreibung der Attribute der Klasse statt.

- **long maxTime:** Dem Suchalgorithmus kann eine maximale Laufzeit übergeben werden. Eine Subklasse muss bei der Implementierung berücksichtigen, dass die Suche bei Überschreitung dieses Wertes stoppt. Dies dient der Ressourcenschonung, da nach einer gewissen Rechenzeit keine signifikanten Verbesserungen für die Auswahl der nächsten Aktivität zu erwarten sind.
- **List<Graph.Node> bestNodes:** Die besten Zwischenergebnisse sind in dieser Liste gespeichert. Dabei ist hervorzuheben, dass die Knoten alle die gleiche Bewertung besitzen. Bei Auffinden eines Knotens mit besserer Wertung werden alle alten Knoten gelöscht und dieser gespeichert. Bei einer Anfrage des besten Knotens wird ein zufälliger Knoten dieser Liste ausgegeben (zufällig, da die

Werte identisch sind). Dabei ist diese Liste für jeden überprüfenden Knoten vom implementierten Suchalgorithmus aktuell zu halten.

- **int[ ] bestScore:** Dieser Wert gibt den aktuell besten Score für die in bestNodes gespeicherten Knoten wieder. Dies geschieht aus Performance technischen Gründen: dadurch kann beim Abgleich mit neuen Knoten die Berechnung des Scores gespart werden (Tausch von Speicher gegen Laufzeit).

Die Funktionalität ist auf die folgenden Methoden aufgeteilt:

- **abstract void findFinalNode(Graph graph):** Diese abstrakte Methode ist die einzige Methode, die von einer Subklasse implementiert werden muss. Mit ihr wird der Suchalgorithmus beschrieben. Dabei wird der Suchgraph als Argument übergeben, ein Rückgabewert existiert jedoch nicht, da Ergebnisse kontinuierlich während des Algorithmus mit der bestNodes-Liste abgeglichen werden müssen. Diese Methode wirft eine InterruptedException, sollte der ausgeführte Thread von außen gestoppt werden. Folglich wird die Suche durch stoppen des laufenden Threads beendet.
- **Graph.Node getBestNode():** Diese Methode gibt den bisher besten Knoten zurück, d.h. einen zufälligen Knoten aus bestNodes. Dies geschieht unabhängig vom aktuellen Fortschritt der Suche. Dadurch ist es möglich, auf beliebige zeitliche Vorgaben zu reagieren.
- **void checkForNewBestNode(Graph.Node currentNode):** Eine der Hilfsmethoden für die Implementierung eines Suchalgorithmus. Der aktuell betrachtete Knoten kann dadurch mit den bestNodes verglichen werden: Ist die Bewertung identisch, wird er hinzugefügt. Ist die Bewertung besser, wird eine neue, ausschließlich aus diesem Knoten enthaltende Liste bestNodes erzeugt. Ist die Bewertung schlechter, geschieht nichts. Zudem wird die Variable bestScore entsprechend angepasst.
- **Graph.Node getBestNode(List<Graph.Node> nodes):** Eine weitere Hilfsmethode. Auf diese kann zurückgegriffen werden, um den besten Knoten aus einer Liste von Knoten auszuwählen. Die Implementierung erfolgt über das simple Durchsuchen der Liste nach dem besten Rating.
- **Object getRandomListEntry(List<?> nodes):** Diese Methode findet bei der Auswahl eines Knotens in der bestNodes-Liste Anwendung. Es wird lediglich ein Random-Generator erzeugt und ein zufälliger Integer entsprechend der Länge der Liste erzeugt. Das Objekt am entsprechenden Index ist der Rückgabewert.
- **run():** Die run()-Methode existiert auf Grund des Interfaces Runnable und beschreibt das Verhalten bei Ausführung der Klasse in einem Thread. Die Implementierung ruft hierzu lediglich die findFinalNode(Graph graph)-Methode

auf. Der Graph muss dabei vorher von außen im Rahmen des Initialisierungsprozesses gesetzt werden. Zudem wird eine InterruptedException abgefangen, welche von der findFinalNode(Graph graph)-Methode geworfen wird.

6.3.6 BestFirstSearch-Klasse

Diese Klasse erbt von der abstrakten AbstractGraphSearch-Klasse, um die im Abschnitt 2.7 beschriebene Best-First-Suche zu implementieren. Die Implementierung findet dabei ausschließlich durch die findFinalNode(Graph graph)-Methode statt. Neben der normalen Such-Logik existieren weitere Anpassungen. So wird zu Beginn der Methode die aktuelle Systemzeit erfasst, damit nach jedem Iterationsschritt die Differenz zur neuen Systemzeit berechnet werden und diese danach mit dem maxTime-Attribut der AbstractGraphSearch-Klasse abgeglichen werden kann. Bei einem Überschreiten der Zeit bricht die Suche ab. Zudem findet in jedem Iterationsschritt eine Überprüfung statt, ob der aktuelle Thread interrupted wurde. Falls ja, wird eine InterruptedException geworfen, wodurch die Suche ebenfalls beendet wird. Desweiteren wurde der Algorithmus angepasst, damit nicht ausschließlich eine Überprüfung des Zielknotens stattfindet, sondern der bisher beste Knoten stets mitgeführt wird. Dies ist erforderlich, da der Suchraum sehr groß und ein Finden des Zielknotens in den ersten Runden des Spiels sehr unwahrscheinlich ist. Zu diesem Zweck wird in jedem Iterationsschritt die checkForNewBestNode(Node currentNode)-Methode der Superklasse aufgerufen.
Auch die Umsetzung des Sharings erfolgt, indem das visited-Flag jedes besuchten Knotens auf “true” gesetzt wird. Fällt in einem Iterationsschritt die Auswahl auf einen solchen Knoten mit gesetztem visitied-Flag, wird dieser zwar wie üblich aus der Knotenliste entfernt, die übrigens Aktionen werden jedoch nicht ausgeführt. Es erfolgt somit ein “continue” zum nächsten Iterationsschritt.
Zuletzt existiert noch die Implementierung eines Zufallsschrittes. Dieser ist durch einen Zähler umgesetzt, welcher pro Iteration um eins erhöht wird. Erreicht der Zähler einen vorher definierten Wert, wird nicht der beste Knoten aus der Knotenliste expandiert, sondern ein zufälliger Knoten. Anschließend wird der Zähler wieder auf Null gesetzt. Die Auswahl eines zufälligen Knotens erfolgt über die getRandomListEntry(List<Graph.Node> nodes)-Methode der Superklasse.

6.3.7 IterativeBestFirstSearch-Klasse

Die IterativeBestFirstSearch-Klasse implementiert die in den Abschnitten 2.7 und 2.9 beschriebene iterative Best-First-Suche. Dazu erbt die Klasse von der BestFirstSearch-Klasse. Die findFinalNode(Graph graph)-Methode wird dabei modifiziert, so dass in mehreren Iterationsschritten die super.findFinalNode(Graph graph)-Methode ausgeführt wird, mit entsprechend erhöhter maximaler Graphtiefe.

Wie bei der nicht iterative BestFirst-Suche (Abschnitt 6.3.6) wird die Dauer der Suche überprüft und bei einem Überschreiten der maximalen Zeit gestoppt. Zudem wird die in der super.findFinalNode(Graph graph)-Methode geworfene InterruptedException nicht abgefangen, wodurch auch hier ein Stop entsteht.
Zur Zeitersparnis findet ein Recycling des in jedem Iterationsschritt verwendeten Suchgraphen statt. Die Flags aller besuchten Knoten werden dazu entfernt. Auf diese Weise müssen einmal erzeugte Knoten und berechnete Kinder nicht erneut kalkuliert werden.

6.3.8 Agent-Klasse

Die Agenten-Klasse erbt von der Player-Klasse und erweitert deren Strukturen um die selbstständige Entscheidungsfindung bei der Auswahl der nächsten Aktivität. Dazu werden die in diesem Unterkapitel beschriebenen KI-Strukturen in der Agenten-Klasse zusammengeführt und verwaltet. Für die restliche Anwendung ist es unerheblich, ob ein menschlicher oder künstlicher Spieler eine Aktion ausführt: Beide verwenden dieselben Methoden und sind somit gleich, auch wenn sich die Implementierung unterscheidet (Mensch: setzen der Aktivität über die GUI, Agent: Berechnung).
Zunächst eine Übersicht über die neu hinzugefügten Attribute:

- **Graph graph:** Der Suchgraph wird in der Agenten-Klasse initialisiert und anschließend dem Suchalgorithmus übergeben.
- **Graph.Node finalNode:** Der finalNode ist nicht zwangsweise identisch mit dem Zielknoten, sondern stellt das bisher beste Ergebnis der Suche dar.
- **EdgeRating edgeRating:** Dieses Attribut ist optional. Es stellt die Kantenkostenfunktion dar, mit welcher der Graph initialisiert wird. Bei einem Null-Wert verwendet der Graph das DefaultEdgeRating, welches jeder Kante die Kosten 1 zuordnet.
- **Rating rating:** Die Bewertungsfunktion der Knoten wird zur Initialisierung der abstractGraphSearch benötigt.
- **AbstractGraphSearch: abstractGraphSearch:** Der verwendete Suchalgorithmus wird von der Agenten-Klasse jede Runde neu initialisiert und anschließend als Thread ausgeführt.
- **Thread abstractGraphSearchThread:** Der aktuelle Thread der Suche. Die Laufzeit dieses Threads beträgt maximal eine Spielrunde. Zu Beginn der Runde wird er gestartet und am Ende interrupted. In der darauffolgenden Runde findet die Erzeugung eines neuen Threads statt.

Die Agenten-Klasse muss einmalig zu Beginn des Spiels mit den benötigten KI-Strukturen initialisiert werden, d.h. eine Auswahl des Suchalgorithmus, der Kantenkostenfunktion sowie der Knotenbewertungsfunktion muss getroffen werden. Die

zu jeder Runde neu anstehenden Initialisierungen geschehen daraufhin automatisiert. Um für den Agenten zu definieren, wann Beginn und Ende einer Runde ist, wurden die aus der Player-Klasse vorhandenen Methoden getNextActivity() und nextRound() überschrieben.

- **@override void nextRound():** Diese Methode stellt für den Agenten den Anfang einer neuen Runde dar. Dies bedeutet, dass ein neuer Suchgraph über die initGraph()-Methode, sowie ein neuer Suchalgorithmus über die initAbstractGraphSearch()-Methode, erzeugt wird. Der Suchalgorithmus, welcher ein Runnable ist, wird anschließend einem Thread übergeben und gestartet, wodurch die Berechnungen der nächsten Aktivität starten.
- **void initGraph():** Bei der Initialisierung des Suchgraphen wird überprüft, ob die Spielsituation einen aus Sicht des Agenten unerwarteten Verlauf genommen hat. Das bedeutet, dass sich die bei der letzten Suche verwendeten Prämissen (z.B. Händlerpreise) geändert haben. Falls zutreffend, wird ein neuer Graph mit angepassten Parametern erzeugt. Falls nicht, findet eine Laufzeitsparung durch Wiederverwendung des alten Suchgraphen statt. Dabei wird ausschließlich der Startknoten der Suche angepasst.
- **void initAbstractGraphSearch():** Die Initialisierung erzeugt lediglich eine neue IterativeBestFirstSearch[36] und übergibt benötigte Parameter.
- **@override Activity getNextActivity():** Diese Methode symbolisiert das Ende der aktuellen Runde. Zu Beginn wird für den abstractGraphSearch-Thread ein Interrupt ausgelöst und über ein Thread.join() auf dessen abschließenden Berechnungen gewartet (maximale Wartezeit entspricht einer Iteration der BestFirstSearch, d.h. die Betrachtung eines Knoten). Anschließend wird über die getBestNode()-Methode des Suchalgorithmus der finalNode gesetzt. Da jeder Knoten ebenfalls den Pfad zu sich mitführt, kann folgend die nächste Aktivität auf dem Weg vom aktuellen Zustand zum finalNode zurück gegeben werden.

6.4 GUI

Im nachfolgenden Kapitel wird die Implementierung der GUI erläutert, wobei eine Übersicht der GUI-Implementierung in Abbildung 6.5 zu finden ist. Die Abbildung beschreibt zum einen die Anordnung der Komponenten in der realen GUI, zum anderen beschreibt sie die Verwendung anderer Komponenten: Ein Fenster innerhalb eines anderen zeigt eine Verwendung dieser Komponente an. Dabei ist hervorzuheben, dass jede GUI-Komponente aus drei Klassen besteht: Model, Controller, View. Zur Verwendung dieses Prinzips existieren drei entsprechende abstrakte Klassen. Eine Erklärung dieses Prinzips, sowie der verwendeten abstrakten Klassen, finden sich

[36] Die IterativeBestFirstSearch ist der einzige verwendete Suchalgorithmus. Die Wahl des Suchalgorithmus ist jedoch problemlos anpass- und erweiterbar.

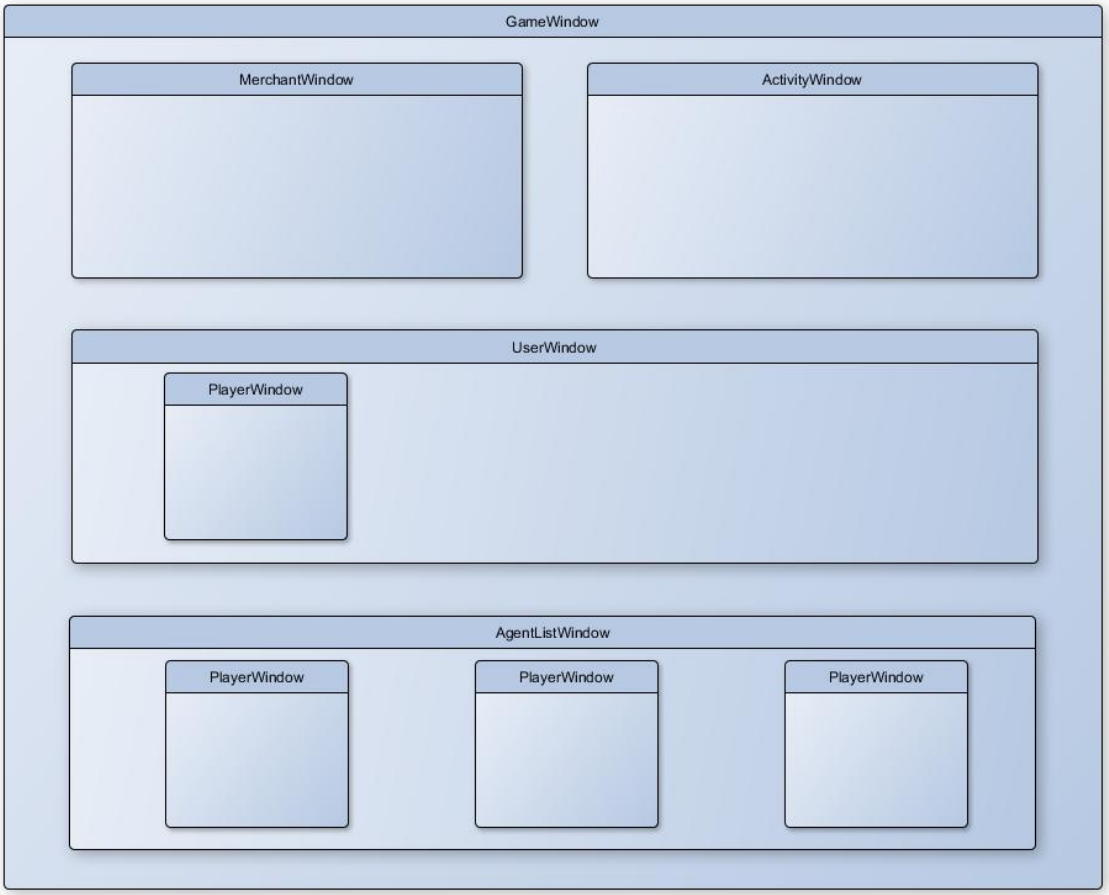

Abbildung 6.5: Übersicht der GUI-Komponenten

in Abschnitt 6.4.1.
Zentrale Verwaltungsstelle der GUI ist das Game-Fenster. Dieses setzt alle anderen Komponenten zusammen und bietet für den Nutzer eine Komplett-Übersicht des gesamten Spielgeschehens. Die Komponenten des Game-Fensters sind folgende: Ein Activity-Fenster (listet alle existierenden Aktivitäten auf), ein Merchant-Fenster (zeigt den Zustand des Händlers an), ein User-Fenster (beschreibt den Zustand des Spielers und gibt ihm die Auswahl der nächsten auszuführenden Aktivität) sowie ein AgentList-Fenster (Auflistung aller mitspielender Agenten sowie deren Zustand). Die Zustandsanzeige für den User und die Agenten bestehen dabei immer aus der Player-Fensterkomponente. An dieser Stelle findet somit ein Recycling der Komponenten statt.

6.4.1 Model-View-Controller

Jedes Fenster der Anwendung besteht aus drei Klassen: Model, View und Controller. Diese Aufteilung dient der Trennung von Daten, Logik und Visualisierung. Zur

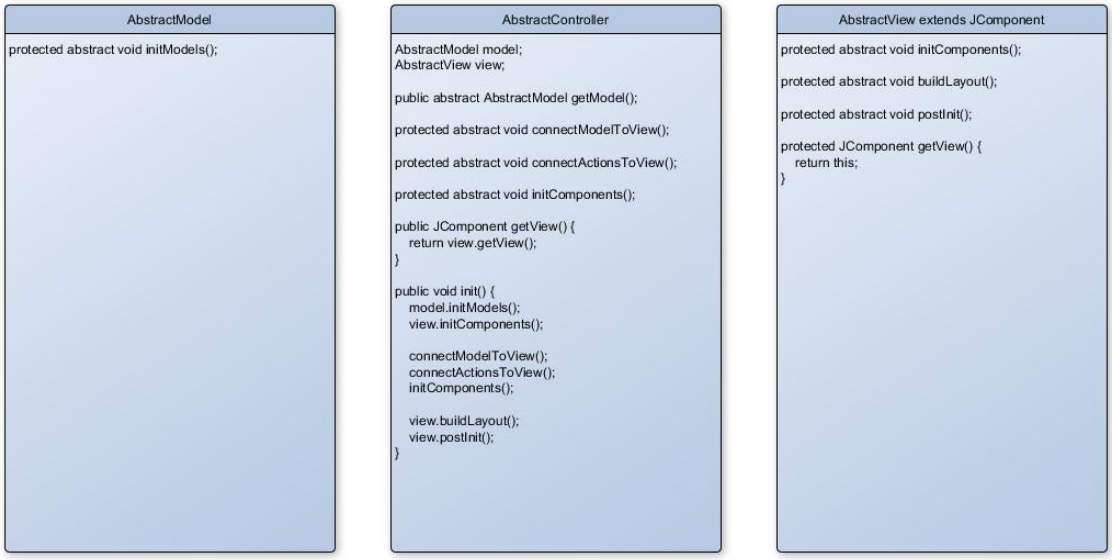

Abbildung 6.6: Model-Controller-View: Abstrakte Klassen

Vereinheitlichung wurden deshalb die abstrakten Klassen AbstractModel, Abstract-Controller und AbstractView erstellt, welche zur Erstellung einer GUI-Komponente stets implementiert werden müssen (siehe Abbildung 6.6).

Der Zuständigkeitsbereich der View-Klasse besteht einzig und alleine im Bauen und Verwalten des Fensters. Zu diesem Zweck erbt die Klasse von JComponent. Sollte eine andere Komponente das Fenster anpassen müssen, wird der entsprechende Aufruf an die View weitergeleitet und dort bearbeitet. Die View verwendet die selbsterklärenden Methoden initComponents(), buildLayout() und postInit(). Eine Implementierung dieser Methoden reicht zum Bauen des Fensters aus. Alle weiteren Verknüpfungen werden vom Controller durchgeführt.

Die Model-Klasse dient der Verwaltung und Bereitstellung der Daten. Hierzu ist die Methode initModels() zu implementieren. In Java besitzen grafische Objekte (z.B. Textfelder) keine Daten, sondern ein Model oder Document, welches die Daten für das grafische Objekt verwaltet. Die Model-Klasse verwaltet diese Models und Documents. Hier ist zu erwähnen, dass die in Java bereits genutzte Idee der Trennung von grafischem Objekt und Daten im verwendeten View-Controller-Model-Prinzip aufgegriffen und erweitert wurde.
Alle eventuell benötigten Hilfskonstrukte zur Verwaltung der Daten werden ebenfalls in der Model-Klasse untergebracht, wie etwa Konvertierungen der Daten.

Die Controller-Klasse beinhaltet die Logik und ist zudem als Brücke zwischen Model und View zu verstehen. Eine der wichtigsten Aufgaben des Controllers liegt in der Initialisierung aller Komponenten. In der abstrakten Controller-Klasse ist

die init()-Methode bereits implementiert und logisch in abstrakte Submethoden untergliedert. Folgend wird die init-Methode genauer erläutert: Zunächst werden in dieser die Models (model.initModels()) und dann die View (view.initComponents()) initialisiert. Anschließend findet die Herstellung der Verbindung zwischen Model und View statt. Dazu werden in der Methode connectModelToView() alle grafischen Objekte mit dem zugehörigen Model der Model-Klasse verbunden. Über die connectActionsToView()-Methode werden anschließend alle Buttons mit ihrer Funktionalität verbunden. Diese Funktionalität der Buttons ist dabei als AbstractAction Subklassen im Controller implementiert. Als letzter Schritt findet das Bauen der GUI-Komponente statt (view.buildLayout()) und ein Aufruf von view.postInit() sorgt für die abschließende Initialisierung (z.B. müssen Buttongroups hier definiert werden, da in der ersten Initialisierung noch keine Models verbunden sind).
Für andere Komponenten ist nur der Controller sichtbar. Zur Erstellung der Komponente reicht folglich die Erstellung einer Controller-Instanz sowie deren Initialisierung. Das Model und die View können von anderen Komponenten über den Controller verwendet werden.

Das hier vorgestellte View-Controller-Model-Prinzip ermöglicht eine dynamische Verwendung der erstellten Komponente. Eine so erstellte Komponente kann beispielsweise alleine in einen Frame gelegt werden, jedoch ebenfalls als Komponente eines anderen Fensters Verwendung finden. Dadurch ist es insbesondere möglich, die so implementierten Komponenten an mehreren Stellen in unterschiedlichen Kontexten gleichzeitig zu verwenden. Das Subfenster verhält sich dabei wie eine normale Komponente aus Sicht des Hauptfensters: Das Hauptmodel erhält eine Referenz auf das Submodel, die View-Klasse auf die untergeordnete View und der Controller erstellt und verwaltet den Subcontroller.

6.4.2 Verwendung von Tabellen

Die erzeugte grafische Oberfläche nutzt eine Vielzahl von Tabellen zur Aufbereitung und Präsentation der Daten. Dabei ist das Vorgehen immer das Gleiche, weshalb es hier kurz Erläuterung finden soll. In Java bestehen Tabellen aus zwei Komponenten: Einer visualisierenden Komponente (JTable), welche ausschließlich die Darstellung von Daten übernimmt, sowie einer Komponente zur Verwaltung der Daten (TableModel). Das JTable kann wie eine normale Swing-Komponente verbaut werden, während alle Informationen der Tabelle[37] im TableModel definiert werden. Zu diesem Zweck existiert bereits ein abstraktes TableModel (AbstractTabelModel). Zur Nutzung dieses Models müssen folgende Methoden implementiert werden:

- **int getColumnCount():** Gibt die Anzahl der Spalten zurück.
- **int getRowCount():** Gibt die Anzahl der Zeilen zurück.

[37]Spaltenzahl, Zeilenzahl, Editierbarkeit, Wert einer Zelle

- **boolean isCellEditiable(int arg0, int arg1):** Definiert, ob eine Zelle an Position (arg0, arg1) editierbar ist. Für nicht editierbare Tabellen[38] erfolgt die Implementierung über ein einfaches "return false".
- **Object getValueAt(int arg0, int arg1):** Definiert, welcher Wert an Position (arg0, arg1) stehen soll. Das Laden der Daten folgt somit über diese Methode für jede Zelle einzeln.

Entstehen Änderungen der Daten, werden diese nicht direkt in der Tabelle übernommen. Das TableModel muss erst über die bereits zur Verfügung stehende fireTableDataChanged()-Methode darüber informiert werden. Anschließend werden alle Daten erneut über die getValueAt(int arg0, int arg1)-Methode geladen. In dieser Implementierung existiert deshalb im Model eine gleichnamige Methode fireTableDataChanged(), welche für alle im Model existierenden TableModels die zugehörige fireTableDataChanged()-Methode aufruft. Im Controller wurde eine öffentliche Methode update() hinzugefügt, welche wiederum die fireTableDataChanged()-Methode des Models aufruft. Die update()-Methoden aller Komponenten werden bei einem Klick auf den "Next Round"-Button[39] aufgerufen, so nach der Berechnung der nächsten Runde alle visualisierten Daten aktualisiert sind.

6.4.3 ActivityWindow

Das ActivityWindow dient der Visualisierung aller existierender Aktivitäten des Spiels, damit sich Spieler einen Überblick ihrer prinzipiellen Möglichkeiten machen können und daraus folgend Strategien entwickeln können. Der Hauptbestandteil dieser Komponente besteht deshalb aus einer großen JTable, für welche im Model die Unterklasse ActivityTableModel definiert ist. Dieses Model definiert die Tabelle auf drei nicht editierbare Spalten, wobei es genau so viele Einträge gibt, wie im ActivityDataHolder vorhanden sind. Die erste Spalte visualisiert den Namen der Aktivität, die zweite Spalte die Effekte und die letzte Spalte die Vorbedingungen. Diese werden direkt aus dem ActivityDataHolder abgefragt (Zeile der Tabelle entspricht Eintrag der Liste des ActivityDataHolders) und nutzen die dort definierten toString()-Methoden. Die Daten dieser Tabelle sind unveränderlich, weshalb keine fireDataChanged()-Aufrufe stattfinden.

6.4.4 PlayerWindow

Das PlayerWindow visualisiert den Zustand eines Spielers, wobei keinerlei Unterscheidung zwischen Agenten und menschlichen Spielern gemacht wird. Grafischer

[38] In dieser Implementierung sind alle Tabellen nicht editierbar.

[39] Dieser Button liegt im GameWindow, welches den höchsten Einstiegspunkt darstellt und deshalb Zugriff auf alle existierenden Unterkomponenten besitzt.

Hauptbestandteil sind zwei Tabellen: Eine Tabelle zur Darstellung der Items im Besitz des Spielers, sowie eine Tabelle zur Darstellung der Spielerattribute. Die letzte Tabelle wurde zudem um die Darstellung des Gold- und Scorewertes erweitert. Für diese Tabellen wurden im Model die Unterklassen ItemTableModel und AttributeTableModel erzeugt, welche die Daten des ItemManagers, bzw. AttributeManagers des Spielers auslesen. Das AttributeTableModel ist zudem so definiert, dass die ersten Zeilen statisch sind und immer den Score- und Goldwert darstellen.

6.4.5 AgentListWindow

Das AgentListWindow bietet eine Übersicht über alle am Spiel teilnehmenden Agenten.[40] Dazu existiert im Controller eine Methode zum Hinzufügen von Agenten. Für jeden dieser hinzugefügten Spieler wird daraufhin bei der Initialisierung ein neues PlayerWindow erzeugt, welche in der View iterativ nebeneinander angelegt werden.

6.4.6 UserWindow

Das UserWindow bietet alle Informationen für den spielenden Benutzer der Anwendung. Dies beinhaltet ein PlayerWindow, sowie eine List über alle für den Benutzer ausführbaren Aktivitäten. Letztere wird erneut über ein JTable erzeugt, wobei das Model ein ExecutableActivitiesTableModel definiert. Die ausführbaren Aktivitäten werden auf Grund der Übersicht nur mit dem Namen visualisiert und nicht mehr mit Effekten und Vorbedingungen[41]. Genutzt wird dazu der ActivityDataHolder, welcher mit der Methode getExecutableActivities(Player player, Game game) bereits die gewünschten Daten zurückliefert. Zudem ist eine Methode getSelectedActivity() implementiert, welche die in der Aktivitäten-Tabelle ausgewählte Aktivität zurück gibt. Diese Auswahl wird beim Klick des “Next Round”-Buttons dazu genutzt, die nächste Aktivität des menschlichen Spielers zu setzen. Das bedeutet, dass genau die Aktivität ausgeführt wird, die der Spieler in der Liste selektiert. Dabei wird ein ListSelectionModel verwendet, um die Selektion der Liste auf “single” einzustellen, was bedeutet, dass ausschließlich eine Zeile markiert werden kann.

6.4.7 MerchantWindow

Im MerchantWindow sind alle Informationen des Händlers visualisiert. Dazu zählt für jedes existierende Item der Name, die verfügbare Anzahl, der Preis sowie die Schätzung über die Preisentwicklung. Genutzt wird dazu erneut ein JTable. Das

[40] In dieser Implementierung werden neben dem menschlichen Spieler nur Agenten verwendet, weshalb dieses Fenster den Namen AgentListWindow trägt. Dabei existiert jedoch keine Beschränkung auf Agenten. Sollte die Anwendung um einen Mehrspielermodus erweitert werden, könnte dieses Fenster auch dazu genutzt werden, andere menschliche Spieler anzuzeigen.

[41] Diese können vom Spieler bei Bedarf im ActivityWindow überprüft werden.

MerchantTableModel greift auf die Merchant-Instanz des Spiels zu, welche bereits Methoden besitzt, um jede der benötigten Informationen zu setzen.

6.4.8 GameWindow

Das GameWindow ist das Fenster, indem alle vorher erstellten Komponenten zusammenlaufen und somit die vollständige grafische Oberfläche bilden. Folglich besitzt das GameWindow die Komponenten AgentListWindow, MerchantWindow, ActivityWindow und UserWindow, wobei die Controller, Models und Views entsprechend dem Controller, Model und View des GameWindows zugeordnet sind. Zusätzlich ist der "Next Round"-Button angelegt und mit der im Controller definierten NextRoundButtonAction verbunden. In dieser Unterklasse befindet sich die Implementierung der Logik zur Überführung des Spiels in die nächste Runde. Folgende Schritte werden dazu ausgeführt:

1. Frage vom Model des UserWindows die ausgewählte Aktivität ab.
2. Sollte die ausgewählte Aktivität nicht "null" sein, setze diese in der Player-Instanz.
3. Führe das logische Spiel in die nächste Runde, indem über die Game-Instanz die Methode nextRound() aufgerufen wird.
4. Führe die grafische Oberfläche in die nächste Runde, indem für alle zugehörigen Komponenten die update()-Methode aufgerufen wird.

Kapitel 7

Bedienung und Funktionalität

In Abbildung 7.1 ist die GUI der Anwendung visualisiert. Diese erscheint beim Starten der Anwendung und ermöglicht ein direktes Losspielen. Die GUI ist in mehrere logische Komponenten untergliedert, welche nachfolgend in ihrer Bedienung und Funktionalität erläutert werden.

- **Player-Fenster:** Das Fenster des Spielers befindet sich im linken mittleren Bereich der GUI. In diesem sind alle Informationen zum Status des Spielers über Tabellen visualisiert. Eine Tabelle stellt die Items des Spielers dar: In der linken Spalte steht der Name des Items und in der rechten Spalte eine positive Ganzzahl (inklusive Null), welche für die Anzahl der im Besitz befindlichen Items steht. Die Tabelle darüber visualisiert die Attribute des Spielers und ist analog der Items aufgebaut. Hinzu kommt, dass diese Tabelle die vier Einträge "Gold", "Attribute Score", "Item Score" und "Total Score" besitzt. Der Item Score setzt sich aus dem Gold-Gegenwert der Items zusammen, während der Attribute Score für jedes Attribut fest definierte Werte zusammenrechnet (Hinweis: "Hunger"gibt negativen Score). Total Score ist als die Summe von Attribute Score, Item Score und Goldwert definiert.
- **"Executable Activities"-Fenster:** Dieses Fenster befindet sich in der Mitte der GUI. Es listet alle vom Spieler zur Zeit ausführbaren Aktivitäten auf. Zusätzlich dient es dem Spieler zur Auswahl der als nächstes auszuführenden Aktivität. Zu diesem Zweck ist lediglich die gewünschte Aktivität in der Auflistung auszuwählen. Bei einem Klick auf den "Next Round"-Button wird genau die Aktivität ausgeführt, die zuvor in der Tabelle selektiert wurde. Wird keine Selektion getroffen, bleibt der Spielerzustand unverändert, das Spiel schreitet jedoch voran ("Spieler setzt aus").
- **Activities-Fenster:** Dieses Fenster ist oben rechts in der GUI angeordnet. Es stellt eine Auflistung aller im Spiel existierender Aktivitäten dar. Zudem werden Voraussetzungen und Effekte der Aktivitäten angegeben. Dieses Fenster dient ausschließlich der generellen Übersicht des Spielers und bleibt während des kompletten Spiels unverändert. Während die Voraussetzungen der Aktivitäten genereller Natur sind, befinden sich in den Effekten der Aktivitäten

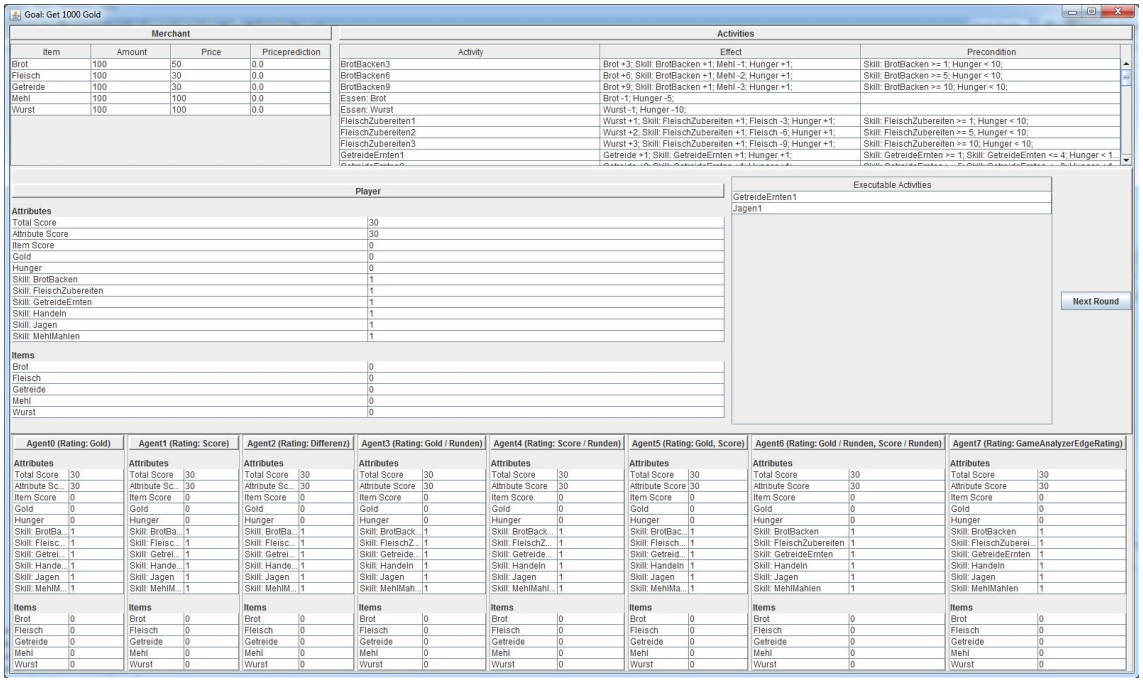

Abbildung 7.1: GUI

zusätzliche indirekte Voraussetzungen bezüglich der Existenz, die jedoch intuitiver Natur sind: Wenn beispielsweise ein Effekt ein Item A um eins erhöht und dafür ein anderes Item B um eins reduziert, dann muss zur Ausführung folglich auch Item B vorhanden sein (auch wenn dies nicht explizit ausformuliert wird).

- **Agenten-Fenster:** Im unteren Bereich der GUI sind alle Agenten des Spiels nebeneinander aufgelistet. Dabei ist das Fenster eines einzelnen Agenten äquivalent zum Aufbau des Spieler-Fensters. Die Auflistung der Agentenfenster passt sich der Anzahl der im Spiel befindlichen Agenten an.
- **Merchant-Fenster:** Dieses Fenster befindet sich oben links in der GUI. Es besitzt eine Auflistung aller existierender Items mit weiteren Informationen: Menge, Preis und Preisentwicklung. Die Menge der Items gibt an, wie viele Items über den Händler gehandelt werden können. Dabei ist die Menge direkt abhängig von der Anzahl der gehandelten Items: Werden X Items verkauft, steigt die Menge um X, beim Kauf entsprechend umgekehrt. Ist die Menge erschöpft, ist ein Kaufen des Items nicht mehr möglich. Der Preis der Items entspricht genau dem Erlös an Gold beim Verkauf, bzw. dem Verlust an Gold beim Kauf eines Items. Dabei reagiert der Preis auf den Markt: pro verkauftem Item sinkt er um 1 und pro gekauftem Item steigt er um 1. Die Preisentwicklung stellt letztendlich eine Zusammenfassung der Preisänderungen der letzten 10 Runden dar und kann als Prognose der Zukunft interpretiert werden. Diese Prognose ist unter anderem Grundlage für die Berechnungen der Agenten.
- **“Next Round”-Button:** Dieser Button befindet sich im mittleren rechten

Bereich der GUI. Durch einen Klick schreitet das Spiel eine Runde voran. Dies bedeutet, dass die vom Spieler selektierte Aktivität ausgeführt wird. Ebenfalls führen die Agenten ihre gewählte Aktivität aus. Diese erhalten zur Berechnung ihrer Auswahl genau so viel Rechenzeit, wie der Spieler benötigt, um den "Next Round"-Button zu betätigen. Denkt der Spieler in einer Runde länger nach, haben folglich auch die Agenten mehr Zeit, ihre Entscheidung zu optimieren. Durch die ausgeführten Aktivitäten aktualisieren sich die Fenster der Spieler und Agenten. Desweiteren werden die Preise und Kalkulationen der Items im Merchant-Fenster aktualisiert.

- **Ziel und Ende des Spiels:** Das Ziel des Spiels ist auf "Erreiche 1000 Gold" festgelegt und befindet sich oben links in der Kopfleiste der Anwendung. Bei Erreichen des Ziels wird, unabhängig vom Spieler, eine Popup-Nachricht ausgegeben, welche neben der Meldung über das erreichte Ziel zusätzlich die Anzahl der benötigten Runden ausgibt. Nachdem ein Spieler das Ziel erreicht hat, können die anderen Spieler noch weiter spielen, bis diese ebenfalls das Ziel erreicht haben. Ein Agent wird nach Erreichem des Ziels jedoch keine Aktivitäten mehr ausführen.

Kapitel 8

Vergleich der Agenten

Im folgenden Kapitel sind die Tests der Agenten beschrieben. Dabei existieren zwei grundlegende Ansätze: Agent vs Agent und Mensch vs Agent. Im ersten Fall spielen Agenten ausschließlich gegeneinander. Die Rundendauer ist festgelegt und es erfolgt eine automatisierte Ausführung einer Vielzahl von Spielen, so dass möglichst präzise Aussagen getroffen werden können. Das primäre Ziel ist der Erkenntnisgewinn über die Güte von Bewertungsfunktionen.
Der zweite Fall besteht aus einer Reihe von Spielen menschlicher Spieler gegen Agenten. Dabei liegt der Fokus auf den Gewinner des Spiels: Dominiert der menschliche Spieler oder ein Agent? Dieser Test soll Rückschlüsse auf die Vergleichbarkeit von Mensch und Agent zulassen.
Vor der Betrachtung der einzelnen Testfälle erfolgt zunächst eine Beschreibung der Testrahmenbedingungen wie Agenten und Testsystem. Anschließend folgt eine Aufgliederung der beiden zuvor beschriebenen Testsituationen. Beide Tests sind dabei gleich aufgebaut: Zunächst erfolgt die Beschreibung der Testdurchführung. Anschließend erfolgt eine Präsentation der Testdaten, um daraufhin mit der Analyse der Testdaten zu schließen.

8.1 Verwendete Agenten

Die implementierten Agenten unterscheiden sich durch die in Abschnitt 5.4 beschriebenen Bewertungsfunktionen. Jede Bewertungsfunktion wird deshalb einem eigenen Agenten zugeordnet. Folglich nehmen acht Agenten an den Tests teil, wobei die Auflistung den Namen des Agenten sowie die zugeordnete Bewertungsfunktion definiert:

1. **Agent0:** Gold
2. **Agent1:** Score
3. **Agent2:** Differenz
4. **Agent3:** Gold / Runden
5. **Agent4:** Score / Runden
6. **Agent5:** Gold, Score

7. **Agent6:** Gold / Runden, Score / Runden
8. **Agent7:** Kantenkosten (GameAnalyzer)

8.2 Testsystem

Das verwendete Testsystem ist durch folgende Eckdaten spezifiziert:

- **Betriebssystem:** Windows 7 Professional, Service Pack 1
- **Systemtyp:** x64-basierter PC
- **Prozessor:** AMD FX(tm)-8350 Eight-Core Processor, 4000 MHz
- **Arbeitsspeicher:** 16 GB

Hervorzuheben ist, dass der verwendete Prozessor aus acht Kernen besteht, was genau der Anzahl der verwendeten Agenten entspricht. Im Idealfall verwaltet folglich jeder Kern genau einen Agenten.

8.3 Agent vs Agent

Im ersten Testverfahren stehen die Agenten im direkten Vergleich zueinander, um Aussagen über die Güte der verwendeten Bewertungsfunktionen treffen zu können. Die Tests erfolgen dabei automatisiert[1]: Programmatisch wird eine definierte Anzahl an Spielen gespielt, wobei nach einer konstant definierten Wartezeit die nächste Runde ausgeführt wird. Für jeden Agenten wird die benötigte Zeit zum Erreichen des Ziels vermerkt. Am Ende werden die Rohdaten ausgewertet, wobei Informationen, wie die durchschnittliche Anzahl benötigter Runden zum Ziel, berechnet werden.

8.3.1 Durchführung

Die Güte der Agenten hängt von mehreren Faktoren[2] ab. Dabei ist es möglich, dass Agenten unterschiedlich gut auf verschiedene Situationen reagieren[3]. Aus diesem Grund erfolgt die Durchführung des Tests mehrfach mit verschiedenen Parametern, um ein größtmögliches Spektrum abzubilden. Folgende Parameter werden betrachtet:

[1] Auf Grund der Automatisierung und der Testidee "Agent vs Agent" nimmt an diesen Spielen kein menschlicher Spieler teil.

[2] z.B. Rechenzeit

[3] Beispielsweise wäre es denkbar, dass ein Agent, der mit ausreichend Rechenzeit sehr gute Ergebnisse erzielt, bei geringer Rechenzeit nur mäßige Ergebnisse liefert.

- **Rundendauer:** Die Dauer einer Runde hängt direkt mit der Rechenzeit der Agenten zusammen. Je länger die Runde, desto länger können Agenten folglich den besten Weg im Suchgraphen suchen. Aus diesem Grund wurde die Dauer der Runden einmal auf 2 Sekunden und einmal auf 20 Sekunden gesetzt. Im normalen Spiel hängt die Rundendauer vom menschlichen Spieler ab: Dieser benötigt Zeit zur Auswahl einer Aktivität[4]. Insofern stellen 2 Sekunden eine sehr kurze Runde dar, während mit 20 Sekunden Rundendauer auch etwas längere Überlegungen eines (theoretischen) menschlichen Spielers simuliert werden.
- **Entfernung des Ziels:** Je weiter das Spielziel entfernt ist, desto mehr Spielraum existiert für Veränderungen der Spielwelt und desto größer wiegt die Summe schlechter Entscheidungen. Agenten, die in kurzen Spielen gut abschneiden, könnten in langen Spielen sehr viel schlechter sein. Aus diesem Grund wurden zwei Spielziele definiert, mit denen anschließend getestet wird: Das nahe Ziel ist "Erreiche 500 Gold", während das ferne Ziel mit "Erreiche 1000 Gold" doppelt so weit entfernt ist.
- **Veränderbarkeit der Händlerpreise:** Im normalen Mehrspieler-Spiel werden die Händlerpreise durch Mitspieler verändert, wodurch Preisschwankungen entstehen können, welche die vorher kalkulierten Pläne der Agenten durcheinander bringen. Aus diesem Grund berücksichtigt ein Test das normale Spielverhalten, während in einem weiteren Test die Veränderbarkeit der Händlerpreise ausgeschaltet ist. Dieser Test entspricht einem vereinfachtem Einzelspieler-Spiel. Ziel des Tests ist der indirekte Vergleich der Agenten in einer Umgebung, in welcher die Spieltwelt immer den eigenen Kalkulationen entspricht.
- **Anzahl der Spiele:** Dieser Parameter ist statisch. Er dient ausschließlich der Vergleichbarkeit der Testergebnisse durch eine möglichst große Anzahl an Testdurchläufen. Die Anzahl der pro Testlauf durchzuführenden Spiele ist auf 100 festgelegt.

Aus der Kombination der obrigen Parameter entstehen acht Testfälle, welche in Tabelle 8.1 beschrieben sind.

8.3.2 Testergebnisse

Die Testergebnisse sind in Tabelle 8.2 dargestellt. Die Tabelle ist dabei sehr groß und umfasst alle "Agent vs Agent"-Testergebnisse. Eine Zeile entspricht einem der in Tabelle 8.1 beschriebenen Testfälle, wobei die erste Spalte den Testfall spezifiziert. In der zweiten Spalte sind die Testzeiten vermerkt, d.h. wie lange der vollständige Testlauf dauerte, sowie die Dauer eines durchschnittlichen Spiels. Die Spalte "Ø-Runden"

[4]Zeit für den Gedankengang zur Auswahl einer Aktivität sowie Zeit die Akivität in der GUI zu selektieren und den Button zur nächsten Runde zu betätigen.

Tabelle 8.1: Agent vs Agent Testfälle

Testfall	Ziel (Gold)	Rundendauer (Sekunden)	Variable Händlerpreise
0	500	2	nein
1	500	2	ja
2	500	20	nein
3	500	20	ja
4	1000	2	nein
5	1000	2	ja
6	1000	20	nein
7	1000	20	ja

gibt zudem die durchschnittliche Zeit (in Runden) an, die ein Agent benötigte, um das Ziel zu erreichen. Anschließend folgt eine kleinere Tabelle mit den konkreten Ergebnissen des entsprechenden Testfalls.
In einer solchen kleineren integrierten Tabelle erfolgt eine Unterscheidung auf Agenten-Ebene. Für jeden Agententyp wurde die Anzahl der Siege[5] angegeben, sowie Informationen über die benötigten Runden: Zunächst die durchschnittlich benötigten Runden[6], anschließend das beste Ergebnis (d.h. das Spiel mit den niedrigsten benötigten Runden), sowie das schlechteste Ergebnis (d.h. das Spiel, für das am meisten Runden zum Erreichen des Ziels benötigt wurden).

Tabelle 8.2: Agent vs Agent: Testergebnisse

Test	Zeit	Ø-Runden	Ergebnis
500g 2s nein	Total: 118m Ø: 70,88s	30,34	

Agent	Siege	Runden		
		Ø	Min	Max
Gold	0	36	26	36
Score	26	26	20	33
Differenz	0	36	26	36
Gold / R.	0	36	26	37
Score / R.	27	24	20	30
Gold & Score	0	31	26	34
Gold/R. & Score/R.	0	31	26	36
GameAnalyzer	73	22	22	22

Fortsetzung auf nächster Seite

[5]Bei Gleichstand mehrerer Agenten wurde allen der Sieg zugerechnet.
[6]Zur besseren Lesbarkeit auf Ganzzahlen gerundet.

Tabelle 8.2 – *Fortsetzung von vorheriger Seite*

Test	Zeit	Ø-Runden	Ergebnis
500g 2s ja	Total: 5,25h Ø: 189s	54,35	**Agent** / **Siege** / **Runden** (Ø, Min, Max) Gold: 0 / 82 / 31 / 121 Score: 2 / 40 / 29 / 58 Differenz: 0 / 78 / 28 / 145 Gold / R.: 1 / 55 / 28 / 78 Score / R.: 10 / 34 / 25 / 48 Gold & Score: 0 / 72 / 35 / 127 Gold/R. & Score/R.: 1 / 45 / 28 / 68 GameAnalyzer: 91 / 28 / 24 / 32
500g 20s nein	Total: 19,4h Ø: 11,7m	30,17	**Agent** / **Siege** / **Runden** (Ø, Min, Max) Gold: 0 / 36 / 30 / 36 Score: 18 / 26 / 20 / 31 Differenz: 0 / 36 / 32 / 36 Gold / R.: 0 / 36 / 32 / 37 Score / R.: 15 / 24 / 20 / 30 Gold & Score: 0 / 31 / 26 / 34 Gold/R. & Score/R.: 0 / 31 / 26 / 34 GameAnalyzer: 82 / 22 / 22 / 22
500g 20s ja	Total: 51,19h Ø: 30,71m	54,08	**Agent** / **Siege** / **Runden** (Ø, Min, Max) Gold: 0 / 80 / 25 / 141 Score: 5 / 38 / 29 / 60 Differenz: 0 / 79 / 34 / 140 Gold / R.: 0 / 54 / 33 / 78 Score / R.: 16 / 34 / 25 / 46 Gold & Score: 1 / 73 / 25 / 145 Gold/R. & Score/R.: 2 / 47 / 21 / 64 GameAnalyzer: 86 / 28 / 24 / 34

Fortsetzung auf nächster Seite

Tabelle 8.2 – *Fortsetzung von vorheriger Seite*

Test	Zeit	Ø-Runden	Ergebnis
1000g 2s nein	Total: 3,33h Ø: 2m	47,43	**Agent** / **Siege** / **Runden** (Ø, Min, Max) Gold: 0 / 60 / 52 / 61 Score: 45 / 36 / 33 / 41 Differenz: 0 / 60 / 50 / 61 Gold / R.: 0 / 58 / 46 / 68 Score / R.: 69 / 35 / 31 / 40 Gold & Score: 0 / 46 / 44 / 48 Gold/R. & Score/R.: 0 / 46 / 44 / 48 GameAnalyzer: 2 / 38 / 37 / 47
1000g 2s ja	Total: 8,61h Ø: 310s	89,37	**Agent** / **Siege** / **Runden** (Ø, Min, Max) Gold: 0 / 138 / 56 / 218 Score: 2 / 61 / 43 / 81 Differenz: 0 / 135 / 48 / 195 Gold / R.: 0 / 86 / 55 / 107 Score / R.: 1 / 56 / 43 / 87 Gold & Score: 0 / 124 / 58 / 183 Gold/R. & Score/R.: 0 / 72 / 56 / 93 GameAnalyzer: 97 / 42 / 33 / 54
1000g 20s nein	Total: 32,7h Ø: 19,63m	46,96	**Agent** / **Siege** / **Runden** (Ø, Min, Max) Gold: 0 / 59 / 46 / 61 Score: 63 / 35 / 32 / 40 Differenz: 0 / 59 / 46 / 61 Gold / R.: 0 / 57 / 48 / 67 Score / R.: 65 / 35 / 31 / 39 Gold & Score: 0 / 46 / 44 / 48 Gold/R. & Score/R.: 0 / 46 / 44 / 48 GameAnalyzer: 1 / 39 / 37 / 49

Fortsetzung auf nächster Seite

Tabelle 8.2 – *Fortsetzung von vorheriger Seite*

Test	Zeit	Ø-Runden	Ergebnis				
			Agent	**Siege**	**Runden**		
					Ø	Min	Max
1000g 20s ja	Total: 92,8h Ø: 55,66m	94,35	Gold	0	148	90	243
			Score	2	60	42	89
			Differenz	0	151	82	228
			Gold / R.	0	88	46	124
			Score / R.	2	55	41	71
			Gold & Score	0	134	76	214
			Gold/R. & Score/R.	0	75	56	91
			GameAnalyzer	97	43	30	55

8.3.3 Auswertung

Ziel des Tests war die Bewertung der Agenten. Es sollten Aussagen über deren Güte und Leistung getroffen werden. In den 800 Spielen, die mit verschiedenen Parametern ausgeführt wurden, gab es zahlreiche Details zu beobachten und Erkenntnisse zu gewinnen. Folgend eine Auflistung der wichtigsten Beobachtungen und Schlussfolgerungen.

- **Das globale Ranking:**
  1. GameAnalyzer: 529 Siege
  2. Score / Runde: 205 Siege
  3. Score: 163 Siege
  4. Alle übrigen Agenten zusammen: 5 Siege
- **Der GameAnalyzer dominiert die anderen Agenten.** Der GameAnalyzer gewinnt in 529 von 800 Spielen und ist somit 2,58 mal besser als der zweitplatzierte Agent mit 205 Siegen. Bei einer ausschließlichen Betrachtung von Spielen mit variablen Itempreisen, gewinnt der GameAnalyzer sogar mit einer relativen Häufigkeit von 92,75%, wobei die Häufigkeit bei einem 1000-Gold-Ziel auf 97% steigt. Für selbiges 1000-Gold-Ziel gewinnt der Agent zudem im Schnitt mit einem Vorsprung von 13 Runden vor dem Zweitplatzierten. Da variable Händlerpreise dem realen Standardspiel entsprechen, steht der GameAnalyzer-Agent somit auf einem unangefochtenem ersten Platz.
- **Score / Runde sowie Score führen das übrige Feld an.** Diese beiden Agenten liegen zwar global deutlich hinter dem GameAnalyzer, dafür jedoch

ebenso deutlich vor den anderen auf Gold basierten Agenten. Bei einem 500-Gold-Ziel betrug der Abstand des "Score / Runde"-Agenten zum GameAnalyzer zudem im Schnitt nur 4 Runden. Die Platzierung des "Score / Runde" Agenten vor dem Score-Agent spricht für die Entscheidung, die Zeitkomponente in die Bewertungsfunktion zu integrieren. Im besten Testfall wurde so ein durchschnittlicher Zeitgewinn von 6 Zügen erreicht.

- **Auf Gold basierende Agenten schneiden schlecht ab.** In 800 Spielen schaffen es zusammen genommen gerade einmal fünf Agenten, einen Sieg zu erringen. Die Konkurrenzfähigkeit dieser Agenten gegenüber den Übrigen ist somit so gering, dass diese Agenten in einem realen Einsatz des Spiels nicht zu gebrauchen sind. Nicht nur in globaler Betrachtung zeigen sich die Schwächen dieser Agenten, sondern auch im Detail: Die Differenz zwischen besten und schlechtestem Ergebnis ist um ein Vielfaches höher als bei den anderen Agenten. Im extremsten Fall (Test: 500g, 20s, ja) betrug der Maximalwert das 5,8-fache des Minimalwertes, während zum Vergleich beim GameAnalyzer der Maximalwert lediglich das 1,42-fache des Minimalwertes betrug. Dies war auch ein wichtiger Aspekt, weshalb die Testfälle so viel Zeit in Anspruch nahmen. Bei einer Beobachtung des Verhaltens der auf Gold basierenden Agenten fällt auf, dass diese sich im überwiegenden Fall für die Produktion und den Verkauf von Getreide und Fleisch entschieden. Diese Strategie wird jedoch fatal, sobald die Preise dieser Produkte auf Grund des Überangebotes sinken. Die Agenten verkauften jedoch trotz einem Erlös von einem Gold pro Einheit weiter. Die Ursache liegt darin, dass andere Aktivitäten keine Betrachtung finden, da diese auf kurzer Sicht kein Gold produzieren, was wahrscheinlich mit der Größe des Suchraumes zusammenhängt. Mit theoretisch unendlich viel zur Verfügung stehender Rechenzeit, müssten die auf Gold basierenden Agenten jedoch ausgezeichnete Resultate erzielen. Anzumerken ist hierbei jedoch, dass die Verzehnfachung der Rechenzeit von 2 Sekunden auf 20 Sekunden noch keine Verbesserung brachte.
- **Die Steigerung der Rechenzeit von 2 auf 20 Sekunden bringt keine Verbesserung.** Auch wenn es in manchen Testfällen eine minimale Verbesserung der durchschnittlichen Rundenzahl aller Agenten gab (Beispiel: 500g, 2s, ja $\rightarrow$ 54,35 Runden; 500g, 20s, ja $\rightarrow$ 54,08 Runden), gab es gleichzeitig ein Beispiel (100g, 2s/20s, ja), in dem sich die durchschnittliche Rundenzahl durch mehr Rechenzeit sogar um ca. 5 Runden erhöhte. Somit kann global keine Verbesserung festgestellt werden, vermutlich handelt es sich bei den Unterschieden lediglich um statistische Schwankungen. Diese Erkenntnis ist insofern überraschend, als dass zu Beginn davon ausgegangen wurde, dass die Agenten durch eine Erhöhung der Rechenzeit bessere Ergebnisse zeigen würden. Wahrscheinlich ist jedoch, dass die Basishypothese dennoch korrekt ist, aber eine Rechenzeit von 2 Sekunden bereits ausreicht, um die wichtigsten "nahen" Zwischenergebnisse zu sichern und eine Verzehnfachung der Rechenzeit nicht genügt, um signifikante Verbesserungen zu erzielen. Begründen lässt sich dies

mit dem exponentiellen Wachstum des Suchraums: die Rechenzeit müsste folglich genau so stark anwachsen wie der Graph. Dieser wächst im Worstcase jedoch in jeder Ebene um das 47-fache an[7]. Somit reicht eine Verzehnfachung der Rechenzeit nicht aus, um im Graphen eine Ebene tiefer zu gelangen.

- **Der GameAnalyzer ist anpassungsfähiger als andere Agenten.** Diese Aussage bezieht sich auf die variablen Händlerpreise. In einem Spiel, in welchem sich die Umstände und somit auch die besten Produktionswege schnell ändern, sticht der GameAnalyzer besonders heraus. Während beispielsweise im Testfall [500g, 20s, nein] der GameAnalyzer im Schnitt zwei Runden vor dem Zweitplatzierten das Ziel erreichte, verdreifachte sich dieser Abstand im Testfall [500g, 20s, ja] auf 6 Runden. Diese Beobachtung bestätigt die Grundidee der Implementierung des GameAnalyzers, welche darauf basierte, abhängig der gegebenen Spielsituation die beste Entscheidung zu treffen, statt den Spielerzustand alleine zu bewerten.
- **Bei einem Ziel von 1000 Gold und unveränderlicher Händlerpreise schlagen auf Score basierte Agenten den GameAnalyzer.** Obwohl letzterer sich in allen anderen Testfällen deutlich durchsetzt, verliert er dort mit einem durchschnittlichen Abstand von 3,5 Runden zum Erstplatzierten. Hinzu kommt, dass das beste Ergebnis des GameAnalyzers mit 37 Runden ganze sechs Runden schlechter ist als das Ergebnis des "Score / Runden"-Agenten. Dies bedeutet, dass auch der GameAnalyzer noch Möglichkeiten zur Optimierung besitzt. Das deutlich bessere Abschneiden der beiden auf dem Score basierenden Agenten gegenüber den Spielen mit veränderlichen Händlerpreisen lässt sich dadurch erklären, dass der Itemscore hier nicht über Zeit schlechter wird. In den anderen Spielen sinken oder schwanken die Preise, wodurch es häufig vorkommt, dass Items im Besitz der Agenten an Wert verlieren.[8] In einer Spielwelt, in der die Items jedoch nie an Wert verlieren, können diese beliebig lange im Inventar des Spielers verharren und diesem so eine größtmögliche Optionsvielfalt geben. Folglich ist in einer solchen Situation eine auf dem Score basierende Bewertungsfunktion hoch angepasst und effizient.

8.4 Mensch vs Agent

In diesem Test treten die Agenten gegen menschliche Spieler an, um einen direkten Vergleich zwischen dem Abschneiden von Menschen und Agenten ziehen zu können. Zu diesem Zweck soll eine möglichst große und vielseitige Menge an Testpersonen gewählt werden, die jeweils einzeln gegen die Agenten antreten.

[7] Insgesamt existieren 47 Aktivitäten, was im schlimmsten Fall 47 Kindknoten für einen aktuellen Zustand bedeutet.

[8] Erinnerung: Die Bewertungsfunktion dieser Agenten ist so definiert, dass der Score entscheidend ist, wodurch Items nur in seltenen Fällen verkauft werden, da dadurch kein Scoregewinn erfolgt. Erst wenn der Verkauf aller Items zum Zielzustand führt, werden diese gegen Gold eingetauscht.

8.4.1 Durchführung

Da dieser Test nicht automatisiert ablaufen kann, ist die Anzahl der durchführbaren Spiele um einiges geringer als beim automatisierten Test. Aus diesem Grund werden die Test-Parameter nicht verändert, sondern einmalig festgelegt:

- **Rundendauer:** Die Rundendauer ist in diesem Test nicht konstant, sondern hängt vom menschlichen Spieler ab. Dieser wird nach eigenem Ermessen, in der Regel genau dann, wenn er sich für seine eigenen Handlungen entschieden hat, das Spiel in die nächste Runde überführen. Insofern werden die Agenten stets eine unterschiedliche Rechenzeit zur Verfügung haben, die wenige Sekunden bis eventuell auch Minuten betragen kann.
- **Entfernung des Ziels:** Als Ziel wurde "Erreiche 1000 Gold" festgelegt, was dem ferneren Ziel des vorherigen Tests entspricht.
- **Veränderbarkeit der Händlerpreise:** Dieser Test findet unter normalen Mehrspieler-Bedingungen statt. Deshalb sind die Händlerpreise immer veränderlich.
- **Anzahl der Spiele:** Menschliche Spieler haben - gerade zu Beginn - eine steile Lernkurve. Aus diesem Grund finden pro Testperson zwei Spiele statt. Im ersten Spiel ist die Testperson mit der Anwendung völlig unvertraut, genau so wie mit den Spielregeln. Im zweiten Spiel besitzt die Testperson bereits Erfahrungen und kann ihre eigenen Aktionen und Strategien optimieren. Von menschlichen Spielern ist im zweiten Spiel daher eine deutliche Leistungssteigerung zu erwarten.

Für die Tests wurden insgesamt zehn verschiedene Testpersonen ausgewählt[9]. Folgend eine kurze Auflistung der wichtigsten Eckdaten zu den Versuchspersonen:

- **Testperson0:** 23 Jahre, Architektur Studentin
- **Testperson1:** 24 Jahre, Lehramt Deutsch und Ethik Studentin
- **Testperson2:** 25 Jahre, Elektrotechnik Student
- **Testperson3:** 25 Jahre, Informatik Student
- **Testperson4:** 26 Jahre, Maschinenbau Student
- **Testperson5:** 29 Jahre, Projektmanager
- **Testperson6:** 39 Jahre, Angestellter im Vertrieb für Schweißtechnik
- **Testperson7:** 53 Jahre, kaufmännische Angestellte
- **Testperson8:** 55 Jahre, Dipl. Elektroingenieurin
- **Testperson9:** 55 Jahre, Laborantin

[9]D.h. es sind insgesamt 20 Testspiele, zwei für jede Person

Tabelle 8.3: Mensch vs Agent Testfälle

Spieler	Spiel	#Züge	Platzierung	Bester Agent	#Züge (Agent)
0	1	49	2	GameAnalyzer	34
0	2	44	1	Score / Round	46
1	1	58	3	GameAnalyzer	43
1	2	55	3	GameAnalyzer	40
2	1	48	2	GameAnalyzer	39
2	2	44	2	GameAnalyzer	42
3	1	45	2	GameAnalyzer	39
3	2	40	2	GameAnalyzer	37
4	1	80	4	GameAnalyzer	41
4	2	41	1	GameAnalyzer	44
5	1	44	2	GameAnalyzer	35
5	2	46	2	GameAnalyzer	42
6	1	61	3	GameAnalyzer	39
6	2	65	4	GameAnalyzer	37
7	1	57	2	GameAnalyzer	43
7	2	57	3	GameAnalyzer	41
8	1	73	4	GameAnalyzer	40
8	2	43	2	GameAnalyzer	41
9	1	70	4	GameAnalyzer	43
9	2	68	5	GameAnalyzer	43
Durchschnitt:		**54,4**	**2,65**		**40,45**

8.4.2 Testergebnisse

Die Testergebnisse sind in Tabelle 8.3 dargestellt. Alle 20 durchgeführten Spiele wurden mit den wichtigsten Daten notiert. Dazu zählst zunächst die Beschreibung des Spielers, welche der Nummerierung aus dem vorherigen Kapitel entspricht, sowie die Angabe, ob es sich um das erste oder zweite durchgeführte Spiel handelt. Anschließend folgen in der Spalte "# Züge" die bis zum Erreichen des Ziels benötigten Züge des menschlichen Spielers. Die Gesamtplatzierung des Spielers befindet sich in der darauffolgenden Spalte[10]. Im Anschluss erfolgt die Betrachtung des besten Agenten des jeweiligen Spiels sowie dessen benötigte Spielzüge bis zum Ziel.

8.4.3 Auswertung

Der Test Mensch vs Agent diente dem Vergleich des Abschneidens zwischen Agenten und Menschen. Ziel war unter anderem die Analyse der Konkurrenzfähigkeit der Agenten. Folgende Erkenntnisse lassen sich gewinnen:

[10]Bei Platz X waren genau X-1 Agenten besser als der menschliche Spieler.

1. **Der durchschnittliche Spieler benötigt deutlich länger als der beste Agent.** Während ein Spieler im Schnitt 54,4 Züge zum Ziel benötigte, benötigte der beste Agent im Schnitt 40,45 Züge. Das heißt, dass ein Spieler ungefähr 34,48% mehr Zeit benötigt.
2. **Die besten Spieler sind fast auf dem gleichen Niveau wie der beste Agent.** Die besten Spiele waren aus Sicht der menschlichen Spieler 40, 41 und 43 Züge. Dies entspricht in etwa dem durchschnittlichem Niveau des besten Agenten (40,45 Züge). Das beste Ergebnis eines Agenten (34 Züge) ist jedoch deutlich besser als das beste Ergebnis eines Menschen (40 Züge), welches in etwa 17,64% mehr Runden entspricht.
3. **In den meisten Fällen gewinnt ein Agent.** Dies ist eine direkte Konsequenz aus den vorherigen beiden Punkten: Die Agenten konnten 18 von 20 Spielen für sich entscheiden, während in nur zwei Spielen ein menschlicher Spieler gewann.
4. **Der GameAnalyzer dominiert die anderen Agenten.** Während der GameAnalyzer in 19 Spielen der beste Agent war, setzte sich nur in einem einzigen Spiel ein anderer Agent durch (Sore / Runde). Dies spricht erneut für die hohe Leistungsfähigkeit des GameAnalyzer-Agenten.
5. **Es gab nicht immer eine Verbesserung zum 2. Spiel.** Die ursprüngliche Idee des Versuchsaufbaus war, dass menschliche Spieler im zweiten Spiel eine Leistungssteigerung zeigen. Tatsächlich gab es jedoch Spieler, die sich nicht verbesserten (Spieler 7), oder sich sogar verschlechtert haben (Spieler 6). Die Verschlechterung lässt sich damit erklären, dass die Testperson eine andere Strategie wählte, welche sich im Nachhinein als weniger effizient zeigte. Andere Spieler zeigten jedoch die erwartete Steigerung. Im extremsten Fall erfolgte einer Verbesserung von 80 auf 41 Runden (Spieler 4).
6. **Nicht alle Agenten sind spieltechnisch relevant.** Menschliche Spieler landeten im Schnitt auf Platz 2,65 (von 9). Im schlechtesten Spiel landete ein Mensch auf Platz 5. Wie im Test Agent vs Agent auch, gab es Agenten (z.B. Gold-Bewertungsfunktion), welche stets schlecht abschnitten und somit den menschlichen Spielern nicht ansatzweise nahe kamen. Diese Agenten beeinflussten zwar durch ihre Aktivitäten das Spiel, besaßen aber im Rennen auf das Ziel keinerlei Bedeutung. Für diese speziellen Agenten ist die Konkurrenzfähigkeit zu Menschen folglich nicht gewährleistet und sie bestehen den Test nicht.
7. **Test erfolgreich: Es existieren konkurrenzfähige Agenten.** Der Test im Allgemeinen ist erfolgreich abgeschlossen, da menschliche Spieler eine große Herausforderung darin sahen, die Agenten zu besiegen. Somit wurden Agenten geschaffen, insbesondere der GameAnalyzer-Agent, welche es mit Menschen aufnehmen und diese sogar im Großteil der Fälle schlagen können.

Kapitel 9

Zusammenfassung und Fazit

9.1 Zusammenfassung

In dieser Masterarbeit wurde der Entwurf und die Implementierung eines Strategiespiels sowie intelligenter Agenten beschrieben. Dabei gab es zunächst eine Einleitung mit der Erläuterung der Idee dieser Arbeit, sowie ein einführendes Kapitel über die Grundlagen. Anschließend fand eine Anforderungsanalyse statt, wobei ein Schwerpunkt auf der flexiblen Definition der Spielwelt lag. Auf dieser aufbauend wurde die Modellierung des Spiels und der Spielwelt beschrieben. Als letzter Schritt im Entwurf folgte die Konzeption der Agenten auf Grundlage der Spielmodellierung. Dabei fiel die Wahl des Suchalgorithmus auf eine iterative Best-First-Suche, wobei die beste Bewertungsfunktion vorher nicht eindeutig zu bestimmen war, weshalb die Auswahl auf eine Gruppe von Bewertungsfunktionen fiel, welche im letzten Schritt gegeneinander getestet wurden. Zwei Ansätze der Bewertungsfunktionen kristallisierten sich heraus: Während die einen lediglich auf dem Zustand des Knotens basierten, gab es einen Ansatz, welcher Kantengewichte verwendete, um somit, mit Informationen über das gesamte Spiel, bessere Bewertungen liefern zu können. Anschließend folgte die ausführliche Beschreibung der Implementierung, wobei besonders viel Wert auf die Agenten gelegt wurde. Im letzten Schritt erfolgten ausführliche Tests der Agenten, untergliedert in zwei Vorgehensweisen: Im ersten Test traten Agenten gegen Agenten an und in einem zweiten Test anschließend Menschen gegen Agenten. Auf Grundlage dieser Tests konnte die Güte der Bewertungsfunktionen festgestellt werden. Dabei stach besonders der GameAnalyzer-Agent heraus, welcher den zweiten Ansatz der Bewertungsfunktionen mit Kantengewichten verwendete, und dabei nicht nur in den überwiegenden Fällen in der Lage war, seine menschlichen und künstlichen Kontrahenten zu besiegen, sondern zudem auch für das mit Abstand beste Spielergebnis verantwortlich war.

9.2 Auswertung der umgesetzten Anforderungen

In diesem Abschnitt werden erneut die in Kapitel 3 beschriebenen Anforderungen betrachtet. Dabei erfolgt eine Auswertung der gestellten Anforderungen im Hinblick

auf deren Umsetzung.

- **Datenmodell:** Die Anforderung bestand in einem flexiblen und austauschbaren Datenmodell. Diese Anforderung wurde erfüllt: Die Modellierung wurde so gestaltet, dass beliebige Items, Attribute und Aktivitäten erstellt werden können. Die Implementierung basiert auf dieser Modellierung und ermöglicht über den DataLoader die Erzeugung eines DataHolders, welcher alle relevanten Informationen besitzt. Diese Daten können beliebig vielfältig sein[1]. Die aufbauende Logik verwendet anschließend die Daten des DataHolders in dynamischer Weise. Durch einen Austausch der DataHolder-Instanz ist zudem das Spielen eines neuen, anderen Spiels möglich. Insofern ist das Datenmodell flexibel und austauschbar.
- **Spielerzahl:** Die Forderung bestand aus der Erstellung eines Spiels mit mehreren teilnehmenden Spielern. Die Spielerzahl der Anwendung ist nach oben hin offen, einzige Begrenzung sind Systemressourcen. In den Tests wurden bereits Spiele mit neun Spielern durchgeführt. Insofern ist dieser Punkt erfüllt.
- **Spielziel:** Das Spiel sollte ein klares und globales Ziel besitzen. Auch diese Anforderung wurde erfüllt: Das Ziel des Spiels "Erreiche 1000 Gold" gilt global für alle Spieler und ist eindeutig, d.h. es ist zu jeder Zeit klar, ob ein Spieler das Ziel erreicht hat oder nicht.
- **GUI:** Die Anforderung war die Erzeugung einer Benutzeroberfläche, welche Informationen zum Spieler, sowie den relevanten Spielinformationen zur Verfügung stellt und einfach zu bedienen ist. Auch diese Anforderung wurde erfüllt. Die erzeugte Benutzeroberfläche listet alle allgemeinen Spielinformationen (Itempreise, Aktivitäten), sowie spielerspezifische Informationen (Zustand der Spieler, ausführbare Aktivitäten) auf. Die Bedienung ist denkbar einfach: Wähle eine Aktivität in der Liste aus und klicke auf "Next Round".
- **Agenten:** Die Basisanforderung, dass Agenten erstellt werden, die am Spiel teilnehmen, wurde erfüllt. Dies belegen die durchgeführten Tests. Darüber hinaus wurden weitere Anforderungen an die Agenten gestellt:
  - Agenten dürfen vom Spiel nicht anders als menschliche Spieler behandelt werden. Insbesondere sollte ein mögliches Hochskalieren der menschlichen und künstlichen Teilnehmer möglich sein, wobei die Anforderung auf der Möglichkeit und nicht der Umsetzung lag. Diese Forderung wurde ebenfalls erfüllt. Möglich wurde dies vor allem dadurch, dass die Agenten-Klasse von der Player-Klasse erbte und somit aus Sicht des Spiels und aller Komponenten lediglich einen weiteren Spieler darstellte. Aus Sicht eines menschlichen Spielers unterscheiden sich Agenten in ihrer Darstellung auch nicht von menschlichen Spielern: In der GUI sind alle anderen Spieler im unteren Bereich aufgelistet. Diese Auflistung visualisiert zwar zur Zeit nur Agenten, da kein Mehrspielermodus implementiert wurde.

[1] Nach den Regeln und Möglichkeiten, die durch die Modellierung gegeben sind.

Die Möglichkeit besteht jedoch, bei einer Erweiterung der Anwendung auch menschliche Spieler dort darzustellen.

- Die Agenten müssen rational sein. Auch diese Anforderung wurde erfüllt. Insbesondere bei den Tests Mensch vs Agent kristalisierte sich diese Eigenschaft heraus, da menschliche Spieler große Schwierigkeiten hatten, die Agenten zu besiegen. Allerdings gab es Unterschiede zwischen Agenten: Während der GameAnalyzer als sehr rational empfunden wurde, haben die Agenten mit einer auf Gold basierenden Bewertungsfunktion des öfteren nicht rationale (d.h. vom Menschen als schlecht oder unsinnig empfundene) Züge ausgeführt.
- Eine Anforderung lag im Verbot des Wartens menschlicher Spieler auf Agenten. D.h. die maximale Rechenzeit der Agenten sollte entspechend der dynamischen Entscheidungszeit des Menschen sein. Umgesetzt und erfüllt wurde diese Anforderung durch eine Anpassung des Suchalgorithmus: Dieser wurde modifiziert, um gute Zwischenergebnisse abzuspeichern und bei einer Unterbrechung durch die Überführung des Spiels in die nächste Runde, die Suche sofort abzubrechen und das beste Zwischenergebnis zurück zu geben. Auf diese Weise hat der Benutzer jede Entscheidungsgewalt über die Länge einer Runde.

Alle Anforderungen an die Anwendung wurden somit erfüllt.

9.3 Ausblick

Die in dieser Arbeit entwickelte Anwendung, sowie die zusammengetragenen Ergebnisse, sind erweiterbar und bieten Raum für viele neue Fragestellungen. Folgend ein kurzer Ausblick über bestehende Möglichkeiten.

- **Erweiterung der Spielwelt:** Die bisherige Spielwelt ist auf zwei Hauptproduktionspfade (Brot und Wurst) beschränkt. Mit den implementierten Konstrukten ist es bereits jetzt möglich, eine deutlich größere Spielwelt mit einer Vielzahl an Produktionspfaden zu erzeugen. Zudem ist es möglich, weitere negative Attribute wie “Hunger” einzubauen oder ganz neue Attribute zu entwerfen, die entweder als Voraussetzung dienen oder auch Boni zu Aktivitäten geben könnten. Es gilt, das Verhalten von Menschen und Agenten in dieser “Super”-Spielwelt zu erforschen.
- **Hochskalierung der Anzahl guter Agenten:** In den bisherigen Tests gab es von jedem Agenten ausschließlich eine Version. Ein Spiel mit nicht nur acht, sondern deutlich mehr Agenten, bei dem die schwachen Gold basierten Agenten zudem noch entfernt sind, würde zusätzliche Konkurrenz untereinander bedeuten. Wie schneidet der Mensch gegen mehrere GameAnalyzer ab? Stören diese sich gegenseitig?

- **Menschlicher Mehrspielermodus:** Die Implementierung erlaubt eine Erweiterung um einen Mehrspielermodus[2]. Wenn nicht nur ein menschlicher Spieler, sondern gleich mehrere antreten, wäre neben dem Verlauf der Spiele vor allem auch ein Turing-Test[3] interessant. Können Mitspieler unterscheiden, ob der gerade betrachtete Gegenspieler ein Agent oder ein Mensch ist?

9.4 Fazit

Der Titel dieser Masterarbeit lautete "Entwurf und Implementierung einer Simulation eines Strategiespiels unter Verwendung von intelligenten Suchmethoden und Agenten". Diese Aufgabe wurde erfüllt: Das Strategiespiel selbst wurde so implementiert, dass es alle gestellten Anforderungen erfüllt. Zudem wurden mehrere Agenten implementiert, welche alle unterschiedliche Bewertungsfunktionen nutzten, um eine möglichst intelligente Suche zu gewährleisten. Dabei zeigte sich vor allem für die auf Gold basierenden Bewertungsfunktionen, dass diese nicht "intelligent" genug waren, um mit den gegebenen Umständen Schritt zu halten. Die anderen Agenten dagegen bewiesen rationales, intelligentes Verhalten. In Tests gegen menschliche Spieler dominierte vor allem der GameAnalyzer: Dieser setzte sich mit einer relativen Häufigkeit von 90% gegen seine menschlichen Kontrahenten durch. Insofern wurden nicht nur Agenten geschaffen, die für menschliche Spieler rational wirkten, sondern die sogar als ernsthafte Kontrahenten angesehen werden konnten. In den Tests Agent vs Agent konnte jedoch festgestellt werden, dass auch der GameAnalyzer in manchen Situationen noch bessere Entscheidungen treffen könnte.
Insgesamt wurde mit dieser Masterarbeit ein Rahmen gegeben, um beliebige Simulationen von Strategiespielen mit menschlichen und künstlichen Spielern durchzuführen und dabei Agenten und Menschen mindestens ebenbürtig zu betrachten.

[2]Mehrspieler im Bezug auf mehrere menschliche Spieler.
[3]In einem Turing-Test muss eine Testperson entscheiden, ob sein Gegenüber ein Mensch oder eine Maschine ist.

Literaturverzeichnis

[CLRS01] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Robert L. ; Stein, Clifford: *Introduction to Algorithms*. 2. MIT Press, 2001

[HNR68] Hart, P.E. ; Nilsson, N.J. ; Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In: *IEEE Transactions on Systems Science and Cybernetics SSC4*, 1968, S. 100–107

[KP88] Krasner, Glenn E. ; Pope, Stephen T.: A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. In: *Journal Of Object Oriented Programming* (1988), August / September, S. 26–49

[Ora16] Oracle. *Java8 Dokumentation*. https://docs.oracle.com/javase/8/docs/api/. 2016

[Pea84] Pearl, Judea: *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., 1984

[RN95] Russell, Stuart J. ; Norvig, Peter: *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995

[SS14] Schmidt-Schauss, Manfred ; Sabel, David. *Skript zur Vorlesung "Einführung in die Methoden der Künstlichen Intelligenz" im Sommersemester 2014*. http://www.ki.informatik.uni-frankfurt.de. 2014