Berner Fachhochschule
Technik und Informatik

Fachbereich Technik und Informatik

Bachelor Thesis - AI Bot für Computerspiele

Ants AI Challenge

Studierende: Lukas Kuster
Stefan Käser

Betreuung: Dr. Jürgen Eckerle

Experte: Dr. Federico Flückiger

Datum: 10. Januar 2013
Version: V01.00

Management Summary

Ants AI Challenge ist ein Programmierwettbewerb, bei welchem ein Bot programmiert wird, der ein Ameisenvolk steuert. Das Ameisenvolk soll auf einer Karte Futter suchen sowie gegnerische Völker angreifen und vernichten. Dabei müssen Problem wie die Pfadsuche, das Verteilen von Aufgaben sowie das Schwarmverhalten gelöst werden. Als Einstieg in die AI-Programmierung haben wir vor der Bachelorarbeit, ein Bot geschrieben der bereits die Basisfunktionen beherscht. Dieser wurde nun während der Bachelorarbeit mit strategischen und taktischen Modulen erweitert. Dazu gehören unter anderem die Verwendung einer Influence Map, Ressourcenverteilung, die Parametrisierbarkeit des Bots. Im Weiteren wurden die Fähigkeiten des Bots verfeinert. Es wurde ein Suchframework erstellt, dass für ähnliche Computerspiele wiederverwendet werden kann. Dieses bietet die Pfadsuchalgorithmen A* und HPA* sowie eine Breitensuche an.

Datum	10. Januar 2013
Name Vorname	Lukas Kuster
Unterschrift	.....
Name Vorname	Stefan Käser
Unterschrift	.....

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingsverzeichnis

1 Einleitung

Nachdem wir uns im Rahmen des Moduls "Projekt 2" (7302) mit der Implementierung eines Bots für den Online-Wettbewerb AI-Challenge (Ants) beschäftigt hatten, haben wir uns für die Bachelorarbeit eine Verbesserung dieses Bots vorgenommen. Die AI-Challenge ist ein Wettbewerb, der im Herbst 2011 zum 3. Mal stattfand und jedes Jahr mit einem anderen Spiel durchgeführt wird. Ziel ist es jeweils, einen Bot zu programmieren, der durch geschickten Einsatz von KI-Technologien das Spiel möglichst erfolgreich bestreiten kann. In dieser Durchführung ging es darum, ein Ameisenvolk durch Sammeln von Ressourcen und Erobern von gegnerischen Hügeln zum Sieg über die gegnerischen Ameisen zu führen.

Im "Projekt 2" hatten wir zwar einen Bot implementiert, der alle Aspekte des Spiels einigermassen beherrscht, also Nahrung sammeln, die Gegend entdecken, Hügel erobern und verteidigen, sowie gegen feindliche Ameisen kämpfen. Einige dieser Fähigkeiten waren aber eher rudimentär ausgebaut, da wir uns vor allem auf die Pfadsuche konzentriert hatten.

In der Bachelorarbeit ging es nun darum, die taktischen und strategischen Fertigkeiten des Bots auszubauen. Der Schwerpunkt lag auch bei der Bachelorarbeit nicht auf der Optimierung einer Teilaufgabe, sondern auf der Implementierung eines ausgewogenen Bots, der alle Aspekte des Spiels gleichermassen gut beherrscht.

Besonderes Augenmerk legten wir dabei auf einen modularen Aufbau des Codes. Nebst einem sauberen objektorientierten Programmdesign spiegelt sich das vor allem in den separaten Modulen "AITools-Api", "Search" und "Strategy", die so generisch implementiert wurden, dass sie mit geringem Aufwand auch in anderen Projekten einsetzbar sind.

1.1 Spielbeschrieb

1.1.1 Der Wettbewerb

Die AI Challenge[1] ist ein internationaler Wettbwerb des University of Waterloo Computer Science Club der im Zeitraum Herbst 2011 bis Januar 2012 zum 3. Mal stattgefunden hat. Das Spiel in dieser 3. Ausführung war ein zugbasiertes Multiplayerspiel in welchem sich Ameisenvölker gegenseitig bekämpfen. Ziel in der AI-Challenge ist es, einen Bot zu schreiben, der die gegebenen Aufgaben mit möglichst intelligenten Algorithmen löst. Die zu lösenden Aufgaben der Ants AI Challenge sind die Futtersuche, das Explorieren der Karten, das Angreifen von gegnerischen Völkern und deren Ameisenhaufen sowie dem Schützen des eigenen Ameisenhaufen.

1.1.2 Spielregeln

Nachfolgend sind die wichtigsten Regeln, die während dem Spiel berücksichtigt werden müssen, aufgelistet.

- Pro Zug können alle Ameisen um ein Feld (vertikal oder horizontal) verschoben werden.
- Pro Zug steht insgesamt eine Rechenzeit von einer Sekunde zur Verfügung. Es dürfen keine Threads erstellt werden.
- Bewegt sich eine Ameise in die 4er Nachbarschaft eines Futterpixels, wird dieses eingesammelt. Beim nächsten Zug entsteht bei einem Ameisenhügel eine neue Ameise.
- Die Landkarte besteht aus passierbaren Landpixeln sowie unpassierbaren Wasserstellen.
- Ein Gegner wird geschlagen, wenn im Kampfradius der eigenen Ameise mehr eigene Ameisen stehen als gegnerische Ameisen im Kampfradius der Ameise, die angegriffen wird.

[1] `http://www.aichallenge.org`

- Ein Gegner ist ausgeschieden wenn alle seine eigenen Ameisenhügel vom Gegner vernichtet wurden. Pro verlorenem Hügel gib es einen Punkteabzug. Pro feindlichen Hügel, der zerstört wird gibt es zwei Bonuspunkte.
- Steht nach einer definierbaren Zeit (Anzahl Züge) kein Sieger fest, wird der Sieger anhand der Punkte ermittelt.

Die ausführlichen Regeln können auf der Webseite nachgelesen werden: `http://aichallenge.org/specification.php`

1.1.3 Schnittstelle

Die Spielschnittstelle ist simpel gehalten. Nach jeder Spielrunde erhält der Bot das neue Spielfeld mittels String-InputStream, die Spielzüge gibt der Bot dem Spielcontroller mittels String-OutputStream bekannt. Unser MyBot leitet von der Basis-Klasse Bot[2] ab. Ein Spielzug wird im folgendem Format in den Output-Stream gelegt:

o <Zeile> <Spalte> <Richtung>

Beispiel:

```
o 4 7 W
```

Die Ameise wird von der Position Zeile 4 und Spalte 7 nach Westen bewegt.
Der Spielcontroller ist in Python realisiert, der Bot kann aber in allen gängigen Programmiersprachen wie Java, Python, C#, C++ etc. geschrieben werden.

1.2 Abgrenzungen

Da unsere Arbeit auf dem vorgängigen Modul 'Projekt 2' aufbaut wurden nicht alle Module während der Bachelorarbeit erstellt. Da wir im Modul 'Projekt 2' auf einen sauberen Aufbau geachtet haben, war es uns möglich die meisten Komponenten zu übernehmen. Es folgt eine Auflistung was bereits bestand bzw. was wir noch erweitert haben.

Erstellt in Modul "Projekt 2"	**Erweiterung während der Bachelorarbeit**
Grundfunktionalitäten des Bots	–
Pfadsuche Simple, A*, HPA*	Auslagerung in ein eigenes Framework, Performanceverbesserungen, Erweiterung des HPA* Clustering, Pfadsuche unter Berücksichtigung der Influence Map
Logging	Loggen in verschiedene Logfiles, Lazy Initialization
Tasks	Die Funktionsweise jedes Tasks wurden nochmals überdacht und angepasst
Missionen	Die Missionen wurden verfeinert und mit strategischen und taktischen Entscheidungen erweitert
MinMax-Algorithmus	Bei den Kampfsituationen wurde versucht, den MinMax-Algorithmus einzubauen, welcher wir bereits im Modul Spieltheorie erarbeitet haben. (Mehr dazu siehe Kapitel 5)

Tabelle 1.1: Abgrenzungen

[2] Die Klasse ist im Code unter ants.bot.Bot.Java auffindbar

1.3 Projektverlauf

Die Projektarbeit richtete sich nach folgendem Zeitplan:

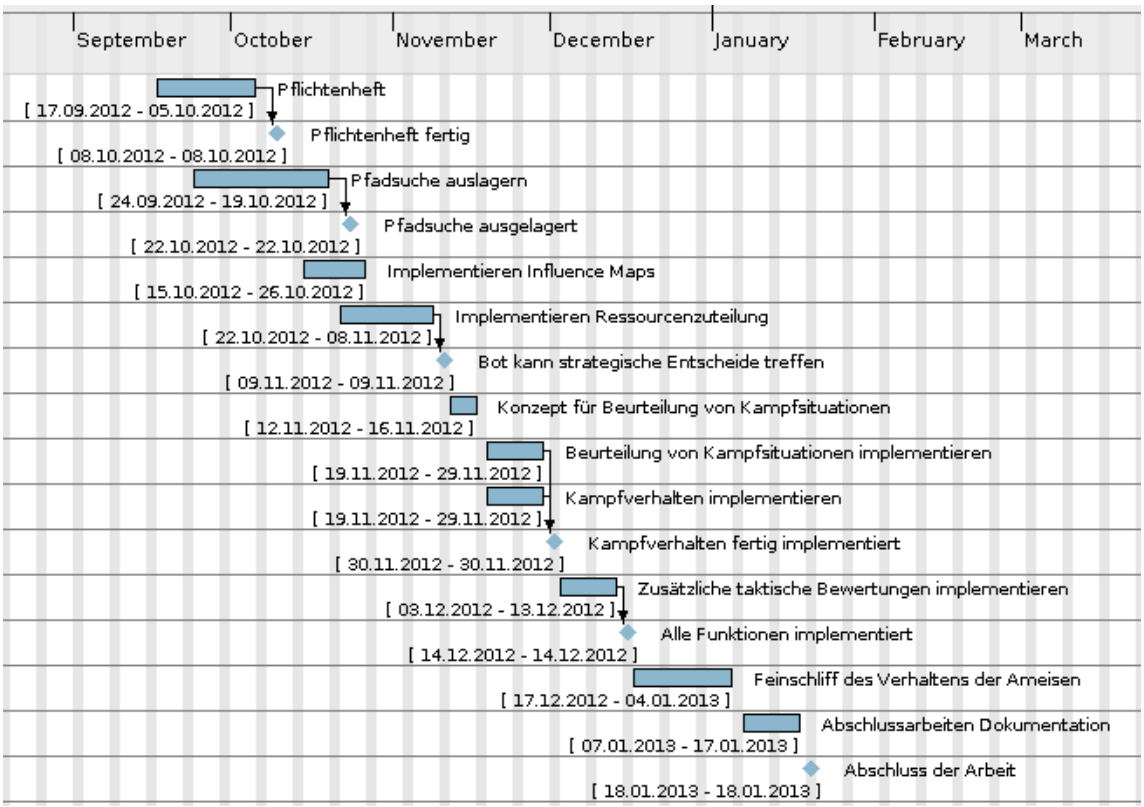

Abbildung 1.1: Projektablauf

Dieser Zeitplan konnte mit lediglich geringen Abweichungen eingehalten werden, weshalb ein detaillierter Soll/Ist Vergleich hier wenig Sinn macht. Folgende Punkte sollen der Vollständigkeit halber dennoch festgehalten werden:

- Obwohl die meiste Arbeit für bestimmte Aufgaben (z.B. Implementieren Influence Maps) im geplanten Zeitraum stattfand, nahmen wir später an den betreffenden Modulen noch Erweiterungen und Verbesserungen vor, die sich aus der Anwendung der Module im Bot ergaben.
- Die Beurteilung der Kampfsituationen bereitete uns einige Mühe (s. Kapitel 1.8). Diese Arbeiten verzögerten sich daher ein wenig, und das taktische Verhalten des Bots ist daher nicht ganz so ausgeklügelt, wie wir es erhofft hatten.

1.4 Projektorganisation

1.4.1 Beteiligte Personen

Studierende:
Lukas Kuster *kustl1@bfh.ch*
Stefan Käser *kases1@bfh.ch*

Betreuung:
Dr. Jürgen Eckerle *juergen.eckerle@bfh.ch*

Experte:
Dr. Federico Flückiger *federico.flueckiger@bluewin.ch*

1.4.2 Projektmeetings

- Es fand jeweils ein Treffen mit dem Betreuer alle 1-2 Wochen statt.
- Ein Treffen mit dem Experten fand am Anfang der Arbeit statt. Ein zweites Meeting wurde von beiden Seiten nicht für notwendig erachtet.

1.4.3 Arbeitsweise

Für die gemeinsame Arbeit am Projekt hatten wir uns jeweils den Freitag reserviert. An diesen Tagen arbeiteten wir meist mit Pair-Programming und erzielten dabei gute Fortschritte. Die verbleibenden Aufgaben teilten wir uns auf und arbeiteten einzeln daran; dabei war die Versionskontrolle mit Git ein sehr hilfreiches Tool, um die Arbeiten zu koordinieren.

Wir arbeiteten mit einem zentralen Repository auf GitHub (`https://github.com/koschder/ants/`). Nebst dem Vorteil einer vereinfachten Zusammenarbeit bietet GitHub auch einige graphische Auswertungen der Aktivität in einem Projekt, die Aufschluss über den Verlauf unserer Arbeit geben.

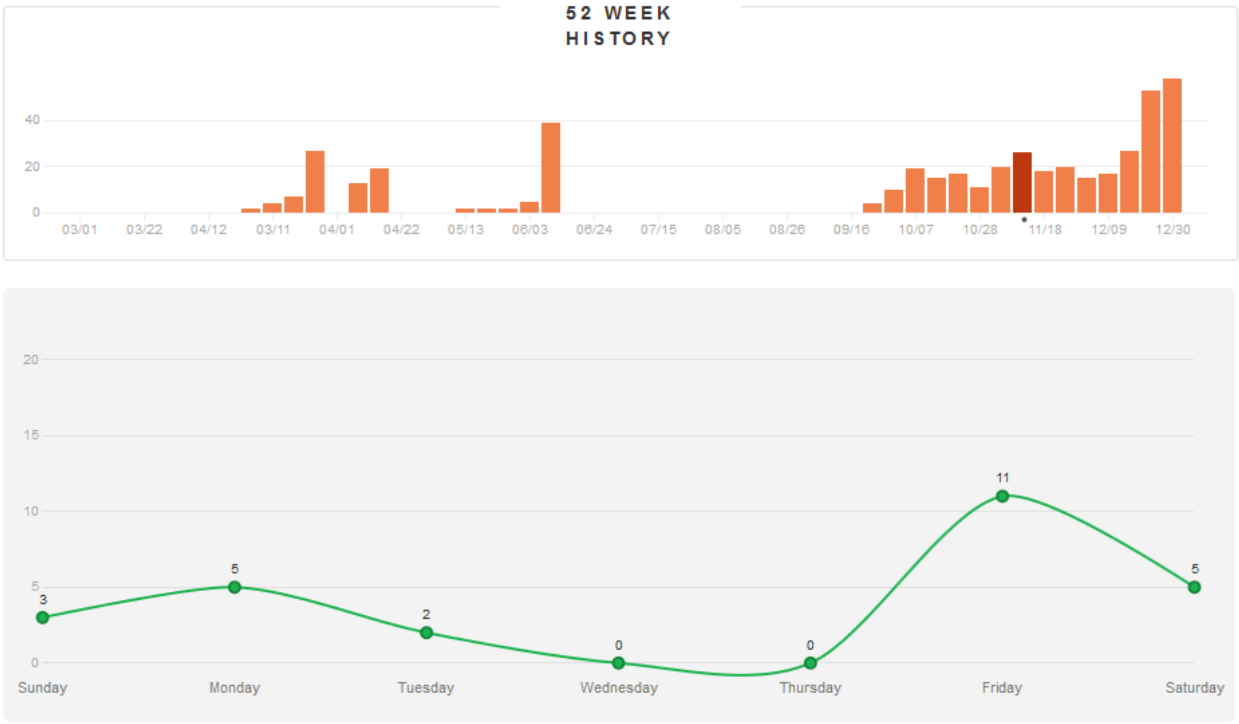

Abbildung 1.2: Commit History

Die obere Hälfte der Abbildung 1.2 zeigt den Aktivitätsverlauf des Projekts. Links ist der Verlauf des Projekts 2 zu sehen, nach der Sommerpause der Verlauf der Bachelorarbeit. Die Aktivitätssteigerung gegen Ende des Projekts ist vor allem auf Dokumentations- und Aufräumarbeiten zurückzuführen.

Die untere Hälfte zeigt die Anzahl Commits über eine Woche verteilt. Man sieht schön, dass der Freitag mit Abstand der produktivste Tag war, mit reduzierter Aktivität über den Rest der Woche verstreut.

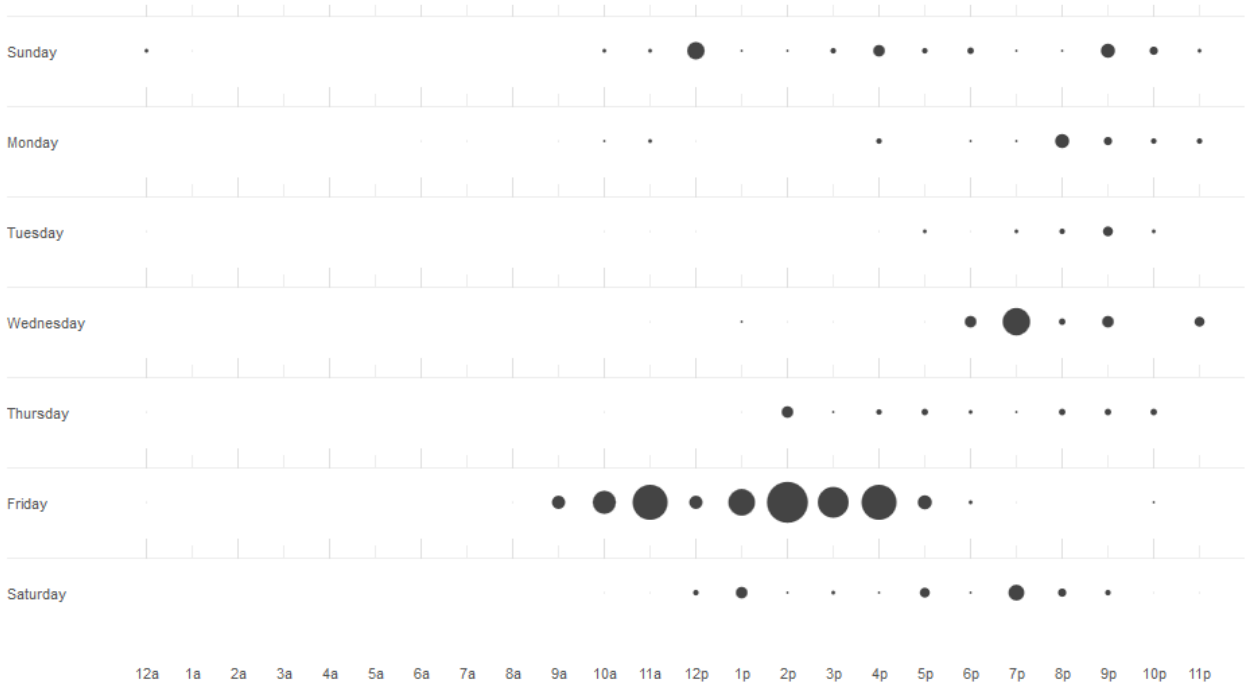

Abbildung 1.3: Aktivitäten

Abbildung 1.3 zeigt für den gesamten Projektverlauf die Zeiten mit der grössten Aktivität. Auch hier kann man deutlich erkennen, dass der grösste Teil der Arbeit an den Freitagen erstellt wurde.

1.5 Tools

Für diese Arbeit verwendeten wir folgende Tools:

- Eclipse für die Java-Entwicklung (`http://www.eclipse.org`)
- ANT für die Build-Automatisierung (`http://ant.apache.org`)
- Git (`http://git-scm.com`) für die Versionskontrolle, mit einem zentralen Repository für die einfachere Zusammenarbeit auf GitHub (`http://www.github.com`)
- TeXnicCenter (`http://www.texniccenter.org`) und MiKTeX (`http://miktex.org`) für die Dokumentation in LaTeX
- GanttProject (`http://www.ganttproject.biz`) für den Zeitplan
- Visual Paradigm for UML (`http://www.visual-paradigm.com/product/vpuml/`) für die UML-Diagramme

1.6 Artefakte

Folgende Artefakte werden zusammen mit dieser Dokumentation in elektronischer Form abgegeben:

- Der komplette Source-Code inklusive Git-History
- Ein Archiv mit der generierten Code-Dokumentation (Javadoc)
- Ein Archiv mit den Testprotokollen
- Das Pflichtenheft

1.7 Zielerreichung

An dieser Stelle soll anhand des Anforderungskatalogs aus dem Pflichtenheft die Zielerreichung differenziert beurteilt werden.

1.7.1 Funktionale Anforderungen

1.7.1.1 Musskriterien

Ziel: Der Bot unterscheidet zwischen diversen Aufgaben:

- Nahrungsbeschaffung
- Angriff
- Verteidigung
- Erkundung

Implementiert durch: Die Kategorisierung der Aufgaben erfolgt duch die Enumeration Task.Type. Sie unterscheidet bspw. zwischen GATHER_FOOD, ATTACK_HILLS, EXPLORE, DEFEND_HILL und weiteren. Alle Tasks (Kapitel 6.4) und Missionen (Kapitel 6.5) werden jeweils einem Aufgabentyp zugeordnet; auch die Ressourcenverwaltung funktioniert über diese Aufgabentypen.

Bewertung: Dieses Ziel wurde klar erfüllt; die Aufteilung auf Aufgaben ist ein zentraler Bestandteil des Bots.

Ziel: Der Bot identifiziert zur Erfüllung dieser Aufgaben konkrete Ziele, wie z.B.:

- Gegnerische Hügel angreifen, was bei Erfolg den Score erhöht und das eigentliche Ziel des Spiels ist.
- Isolierte gegnerische Ameisen angreifen.
- Schwachstellen in der gegnerischen Verteidigung ausnutzen.
- Engpässe im Terrain sichern bzw. versperren.
- Konfliktzonen, d.h. viele Ameisen auf einem engen Raum, erkennen und entsprechend reagieren.

Implementiert durch: Da die Logik zum Identifizieren von Zielen je nach Aufgabe unterschiedlich ausfällt, sind Methoden zur Zielidentifikation an verschiedenen Stellen im Code implementiert. Bspw. wird im AttackHillsTask eine Liste der gegnerischen Hügel geführt, damit diese angegriffen werden können; oder es wird in der DefendHillMission berechnet, wie bedroht unsere eigenen Hügel sind und entsprechend werden zusätzliche Ressourcen angefordert.

Bewertung: Dieses Ziel wurde erfüllt; Ein "Ziel" ist zwar nicht als explizites Konzept in den Code eingeflossen (es gibt keine Klasse `Goal`), die identifizierten Ziele werden aber klar gekennzeichnet und können über unsere Visualizer-Erweiterungen (s. Kapitel 7.3) auch grafisch dargestellt werden.

Ziel: Die Auswahl von Taktik und Strategie basiert auf der Bewertung der Situation auf dem Spielfeld, z.B. anhand folgender Kriterien:

- Dominante/unterlegene Position
- Sicherheit verschiedener Gebiete des Spielfelds (eigener/gegnerischer Einfluss)
- Konfliktpotenzial in verschiedenen Gebieten des Spielfelds

Implementiert durch: Die Bewertung der Situation auf dem Spielfeld wurde hauptsächlich mit Influence Mapping realisiert (s. Kapitel 5.1). Die Influence Map kann mit geringem Rechenaufwand zu Beginn jedes Zugs aktualisiert werden und liefert eine Vielzahl von Informationen, die aus den unterschiedlichen Einflüssen der Spieler auf das Spielfeld gewonnen werden können.

Bewertung: Dieses Ziel wurde erreicht; die Entscheidung, Influence Mapping einzusetzen, wurde aufgrund der Recherchen zum Thema von Lukas Kuster im Rahmen des Informatik-Seminars an der BFH getroffen und die Technik hat sich bewährt.

Ziel: Anhand der Situationsberteilung werden die unterschiedlichen Aufgaben entsprechend gewichtet. Stark gewichtete Aufgaben erhalten mehr Ressourcen (Ameisen) zur Durchführung.

Implementiert durch: Basierend auf den verschiedenen Aufgabentypen und den Informationen aus der Influence Map haben wir ein Ressourcen-Managementsystem implementiert, das auf Regeln basiert (s. Kapitel 6.6). Verschiedene Regeln werten die Situation aus und erhöhen resp. verringern anhand der gewonnenen Informationen die Zuweisung von Ressourcen auf die einzelnen Aufgabentypen. Tasks und Missionen können jeweils nur so viel Ameisen beschäftigen, wie der zugehörigen Aufgabe zugewiesen sind.

Bewertung: Das Ziel wurde erreicht; Das Regelsystem zur Situationsbeurteilung ist modular aufgebaut, so dass mit minimalem Aufwand weitere Regeln erstellt werden können, um das Gesamtverhalten noch raffinierter zu gestalten.

Ziel: Die Situationsbeurteilung fliesst auch in die taktische Logik ein, wie folgende Beispiele illustrieren:

- Bei der Pfadsuche wird die Sicherheit der zu durchquerenden Gebiete berücksichtigt
- In Kampfsituationen kann der Bot die Ameisen in Formationen gliedern, die geeignet sind, eine lokale Überzahl eigener gegenüber gegnerischen Ameisen zu erzeugen
- Beim Aufeinandertreffen mit gegnerischen Ameisen wird entschieden, ob angegriffen, die Stellung gehalten oder geflüchtet wird.

Implementiert durch: Unser Pathfinder kann für die Pfadsuche auf die Influence Map zugreifen und die Sicherheit der Pfade in die Kostenberechnung einfliessen lassen. Für Kampfsituationen haben wir das CombatPositioning entwickelt, das für verschiedene Kampfsituationen (Angriff auf Hügel, Verteidigung eines eigenen Hügels, Kampf im offenen Feld) jeweils eine sinnvolle Formation der Ameisen errechnet.

Bewertung: Dieses Ziel wurde grösstenteils erreicht, weist aber noch Verbesserungspotenzial auf. Nachdem unser erster spieltheoretischer Ansatz mit einem MinMax Algorithmus nicht geklappt hatte, mussten wir uns mit einer relativ groben Näherung an eine optimale Positionierung der Ameisen begnügen. In vielen Fällen reicht diese völlig aus, im Duell mit kampftaktisch versierteren Gegnern zeigen sich allerdings Schwächen.

1.7.1.2 Kannkriterien

Ziel: Das Verhalten des Bots ist konfigurierbar, so dass zum Beispiel ein "agressiver" Bot gegen einen defensiven Bot antreten kann.

Implementiert durch: Das Verhalten des Bots kann durch Profile beeinflusst werden (s. Kapitel 6.8).

Bewertung: Das Ziel wurde erreicht. Da wir die Profile erst gegen Schluss der Arbeit eingebaut haben, beschränken sie sich aktuell auf 11 Parameter, die meisten davon für die Ressourcenverteilung. Weitere Parameter können aber mit geringem Aufwand eingebaut werden.

1.7.2 Nicht funktionale Anforderungen

1.7.2.1 Musskriterien

Ziel: Modularer Aufbau für eine gute Testbarkeit der Komponenten.

Bewertung: Das Ziel wurde erreicht. Das Herauslösen von separaten Modulen für Suche, Strategie, Logging usw. führte zu einer verbesserten Testbarkeit der einzelnen Komponenten und einem sauberen Design für den gesamten Bot

---

Ziel: Wichtige Funktionen wie die Pfadsuche und die Berechnung von Influence Maps sollen in separaten Modulen implementiert werden, damit sie auch von anderen Projekten verwendet werden könnten.

Bewertung: Das Ziel wurde erreicht; Alle Module bis auf das "Ants“ Modul sind so generisch implementiert, dass sie in jedem Spiel, das pixelbasierte Karten verwendet, eingesetzt werden können.

Ziel: Die Codedokumentation ist vollständig und dient der Verständlichkeit.

Bewertung: Dieses Ziel wurde erreicht. Der Code ist wo sinnvoll mit Kommentaren versehen; die HTML-Ausgabe der Javadoc-Kommentare sind in einem separaten Archiv dieser Arbeit beigelegt. Ausserdem achteten wir während der Realisierung auf das Schreiben von les- und wartbarem Code.

1.7.2.2 Kannkriterien

Ziel: Für die wiederverwendbaren Module wird jeweils ein kleines Tutorial geschrieben, wie die Module verwendbar sind.

Bewertung: Die einzelnen Module sind im Rahmen dieser Dokumentation beschrieben; die entsprechenden Kapitel beinhalten auch Verwendungsbeispiele.

1.8 Herausforderungen

An dieser Stelle sollen einige Herausforderungen, die wir während der Arbeit angetroffen haben, besonders hervorgehoben werden.

1.8.1 Module testen

Ein neuer Algorithmus oder eine neue Idee ist schnell mal in den Bot integriert, doch bringen die geschriebenen Zeilen den gewünschten Erfolg? Was wenn der neue Codeabschnitt äusserst selten durchlaufen wird und dann noch fehlschlägt? Wie wissen wir welche Ameise genau diesen nächsten Schritt macht?

Um diese Probleme zu bewältigen haben wir ein ausgeklügeltes Logging auf die Beine gestellt, in welchem wir schnell an die gewünschten Informationen gelangen. (siehe Kapitel 7) Zudem können wir dank der Erweiterung des HTML-Viewer sofort sehen, welches die akutelle Aufgabe jeder einzelen Ameise ist. (siehe 7.3) Weitergeholfen haben uns auch etliche Unittests und visuelle Tests, mit welchen wir neu geschriebenen Code testen und auf dessen Richtigkeit prüfen konnten. (siehe Kapitel 8.1 und 8.2)

1.8.2 Kampfsituationen beurteilen

Die Beurteilung von Kampfsituationen stellte sich wie erwartet als Knacknuss heraus. In einem ersten Anlauf versuchten wir eine gegebene Kampfsituation mittels MinMax Algorithmus zu beurteilen. Wir hatten bereits aus einem früheren Projekt im Rahmen des Moduls Spieltheorie (7501) eine Implementation des Algorithmus erstellt und hatten eigentlich auch keine Mühe, diese für unsere Ameisen anzupassen. Aufgrund des hohen Branching-Faktors einer Kampfsituation (eine Ameise hat 5 mögliche Züge, und an einer Kampfsituation sind i.A. 5-10 Ameisen beteiligt) konnten wir die nötigen Berechnungen aber nicht in der zur Verfügung stehenden Zeit durchführen. Die aktuelle Implementierung der Kampfsituationen verfolgt jetzt einen weniger rechenintensiven Ansatz; im Gegensatz zum MinMax Algorithmus kann sie aber keine optimale Lösung finden.

1.8.3 Vergleich mit Bots aus dem Wettbewerb

Nach Ablauf des Wettbewerbs im Januar 2012, haben einige der Teilnehmer ihren Bot zugänglich gemacht. Dadurch war es uns möglich unseren Bot gegen Bots antreten zu lassen die tatsächlich am Wettbewerb teilgenommen haben. So konnten wir auch eine vage Einschätzung machen wir stark unser Bot ist. Mehr dazu unter siehe 8.5.

1.9 Weiterführende Arbeiten

Obwohl wir mit dieser Arbeit grundsätzlich unsere Ziele erreicht haben, gibt es nach wie vor Verbesserungspotenzial in unserem Bot. Die wichtigsten davon sind folgende:

CombatPositioning In unserem CombatPositioning haben wir zwar eine ganz gute Näherung an eine optimale Formation gefunden, aber im Kampf gegen die besten Bots zeigte sich doch deutlich, dass diese noch nicht gut genug ist. Hier könnten wir das grundsätzliche Code-Design zwar gut beibehalten, aber die Logik für die Positionierung der Einheiten müsste noch verbessert werden.

Profile Die Profile haben wir erst relativ spät eingeführt und uns deshalb etwas eingeschränkt im Bezug darauf, was alles konfigurierbar ist. Mit etwas Fleissarbeit könnte man diese Profile noch stark erweitern; dies könnte man sogar so weit treiben, dass aus dem Verhalten zweier unterschiedlich konfigurierter Bots kaum mehr erkennbar wäre, dass sie die gleiche Codebasis haben.

Regeln für Ressourcenverwaltung Ebenfalls eine Komponente, mit der sich die "Persönlichkeit" unseres Bots noch flexibler gestalten liesse: Mit der Implementierung von weiteren Regeln für die Ressourcenverwaltung könnte die Umsetzung von strategischen Entscheidungen noch dynamischer gemacht werden.

Weitere Challenges Und falls die Organisatoren der AI-Challenge doch noch einmal ein neues Spiel lancieren, könnten wir vermutlich dank des modularen Aufbaus unseres Codes einige Komponenten für dieses neue Spiel wiederverwenden; welche genau, hängt natürlich vom Spiel ab.

1.10 Fazit

Alles in allem sind wir mit dem Verlauf der Arbeit zufrieden. Wir konnten unsere Ziele erreichen und konnten dank einer strukturierten Arbeitsweise auch den Zeitplan grösstenteils einhalten. In Bezug auf die Wettbewerbsfähigkeit des Bots hatten wir uns bewusst keine konkreten Ziele gesetzt, da wir wussten, dass es in der beschränkten Zeit sehr schwierig werden würde, mit den Top 100 des Wettbewerbs mitzuhalten. Insofern sind wir auch mit dem Ergebnis zufrieden: Gegen den Bot auf Platz 93 konnten wir rund 60% der Spiele gewinnen, und gegen den Sieger des Wettbewerbs immerhin um die 20% der Spiele. (Details s. Kapitel 8.5)

Besonders positiv hervorheben möchten wir die Arbeitsprozesse und Tools, die uns diesen guten Projektverlauf ermöglicht haben. Die wöchentlichen Treffen mit Pair Programming waren dabei besonders produktiv; unter der Woche konnten wir uns dank einer guten Arbeitsaufteilung und der Unterstützung durch die Versionsverwaltung jeweils auf das Wesentliche konzentrieren, ohne durch administrative oder technische Probleme abgelenkt zu werden.

Ausserdem war natürlich die Arbeit am Bot sehr spannend, da wir hier die Gelegenheit hatten, den in den letzten 8 (und insbesondere den letzten 3) Semestern behandelten Stoff in die Praxis umzusetzen. Besonders interessant war dabei die Herausforderung, die einzelnen, isolierten Techniken, die wir gelernt hatten, zu einem strukturierten Ganzen zusammenzufügen.

Ein kleiner Wermutstropfen bleibt natürlich, dass wir unseren Bot nicht unter Wettbewerbsbedingungen testen konnten; die Testläufe gegen die verschiedenen Gegner erlaubten uns zwar eine ungefähre Standortbestimmung, aber das ist natürliche nicht das gleiche. Auch sind wir, wie bereits erwähnt, nicht ganz zufrieden mit der Implementierung des Kampfverhaltens unserer Ameisen. Mit etwas mehr Zeit hätten wir da gerne noch eine genauere und dynamischere Bewertung von Kampfsituationen eingebaut.

1.11 Anmerkungen zur Dokumentation

Hier noch einige Anmerkungen zur Dokumentation:

Diagramme Sämtliche UML-Diagramme stellen jeweils eine vereinfachte Sicht dar und erheben keinen Anspruch auf Vollständigkeit. Bei den Diagrammen, die einzelne Packages darstellen, sind jeweils Klassen oder Interfaces, die eigentlich aus einem anderen Package stammen, farblich hervorgehoben.

Listings Die meisten Listings sind zwar direkt aus dem Code unseres Bots kopiert. Der Lesbarkeit halber wurden sie jedoch mehr oder weniger stark gekürzt und einzelne Abschnitte wurden durch Pseudocode ersetzt.

2 Architektur

2.1 Module

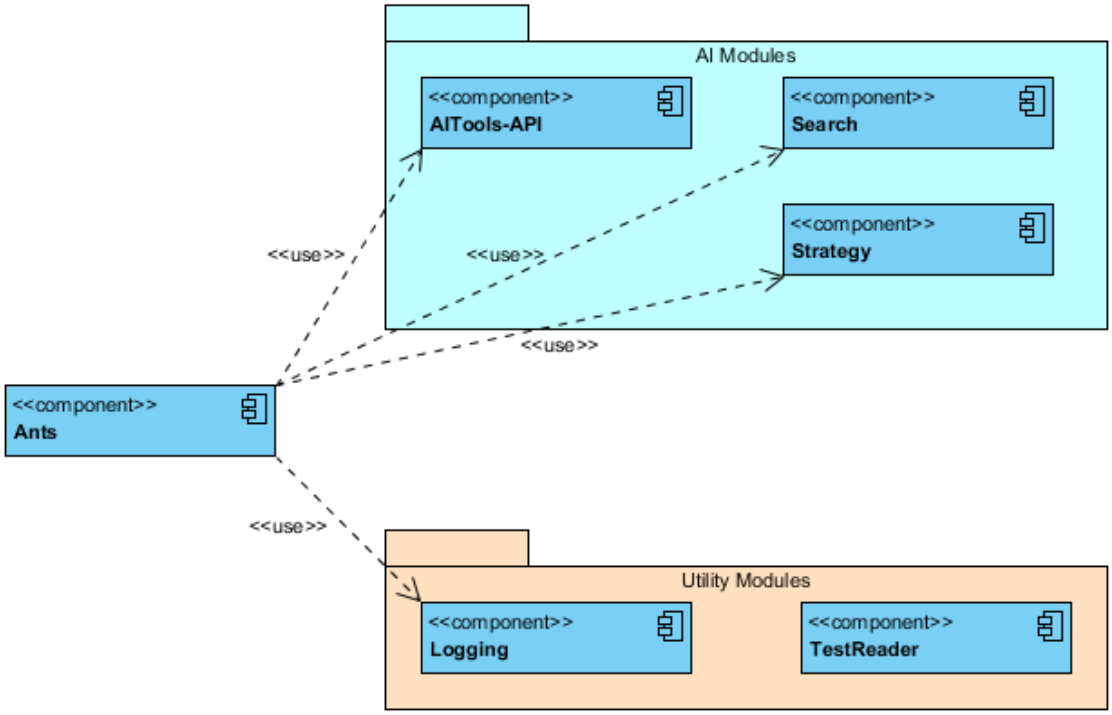

Abbildung 2.1: Module

Abbildung 2.1 zeigt die Gliederung des Bots in die verschiedenene Untermodule. Wir unterscheiden dabei zwischen den AI Modulen, welche die eigentliche AI-Logik enthalten, und den Utility Modulen, die Basisfunktionen wie Logging implementieren.

Ants: Das Ants Modul enthält das Grundgerüst des Bots und fügt die anderen Module zu einem Ganzen zusammen. Es ist das einzige Modul, in dem Ants-spezifische Funktionalitäten implementiert sind; die anderen sind generisch gehalten um die Wiederverwendbarkeit sicherzustellen. Das Ants Modul ist genauer dokumentiert im Kapitel 6.

AITools-API: Im API-Modul sind alle gemeinsamen Interfaces definiert, auf die die anderen Module zugreifen. Zum Teil ist hier auch Basis-Funktionalität implementiert, wie z.B. Distanzberechnungen auf einer Karte. Das API Modul ist genauer dokumentiert im Kapitel 3.

Search: Dieses Modul enthält unsere Implementationen zur Pfad- und Breitensuche. Ausführliche Dokumentation dazu findet sich im Kapitel 4.

Strategy: Das Strategy-Module enthält strategische und taktische Algorithmen, insbesondere die Influence Map und das CombatPositioning. Ausführliche Dokumentation dazu findet sich im Kapitel 5.

Logging: Das Logging-Modul definiert ein flexibles Logging-Framework.Weitere Informationen dazu finden sich im Kapitel 7.

TestReader: Das TestReader-Modul besteht aus einer einzelnen Klasse, die aus den Log-Files der Ants-Spielengine Auswertungen lesen kann. Es ist genauer beschrieben im Kapitel 8.5.1.

2.2 Modulabhängigkeiten

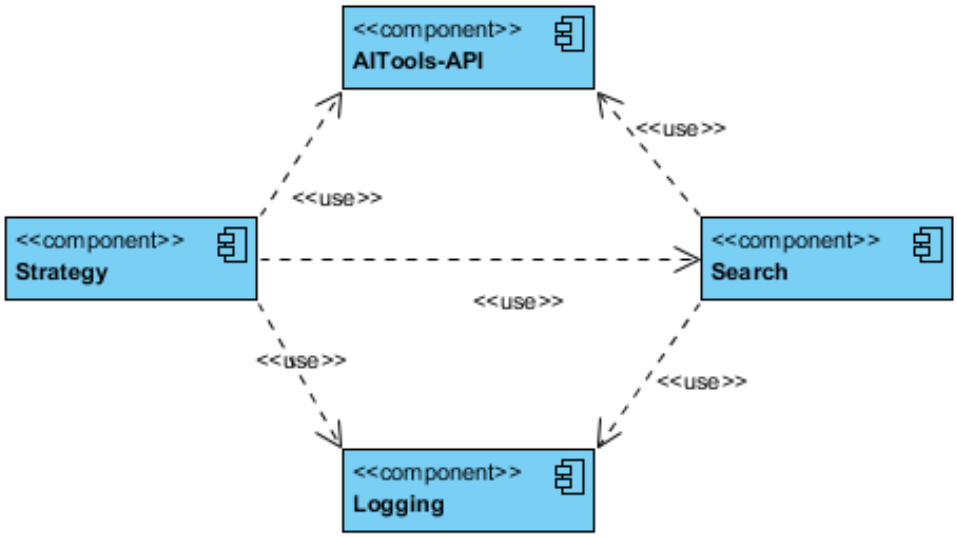

Abbildung 2.2: Modulabhängigkeiten

Abbildung 2.2 zeigt die Abhängikeiten zwischen den Modulen. Die Module API und Logging sind unabhängig, während die beiden grösseren Module Abhängigkeiten auf API und Logging haben. Die Abhängigkeit von Strategy auf Search rührt daher, dass die Influence Map mittels FloodFil Algorithmus aufgebaut wird, der im Search Modul implementiert ist.

2.3 Externe Abhängigkeiten

Der Bot selber (also der Code, der von der Spielengine aufgerufen wird) hat keine externen Abhängigkeiten - dies wäre von den Regeln des Wettbewerbs auch gar nicht erlaubt gewesen. Für ein paar andere Zwecke haben wir aber trotzdem auf externe Programmbibliotheken zurückgegriffen.

- **JUnit** `http://junit.org`**:** Für unsere zahlreichen Unit- und Funktions-Tests verwendeten wir Junit.
- **ClasspathSuite** `http://johanneslink.net/projects/cpsuite.jsp`**:** Da JUnit keine bequemen Weg bietet, alle Tests aus verschiedenen Projekten auf einmal auszuführen, verwendeten wir die ClasspathSuite, um die Tests direkt in Eclipse auszuführen.
- **Json-simple** `http://code.google.com/p/json-simple/`**:** Json-simple ist eine einfache Json-Parsing Bibliothek, die wir im TestReader verwenden, um die Json-Logs der Spielengine auszuwerten.

3 API

Unsere AITools API dient als Grundbaustein aller AI Komponenten und entkapselt die Interfaces zu unseren AI Modulen und bietet den AI Modulen sowie dem Ants Package die Basisklassen an, welche überall verwendet werden (s.a. Abbildung 2.1). Der Inhalt ist wie folgt:

3.1 Entities

Das Package Entities beinhaltet alle Klassen die im von den Such- und Strategiealgorithmen sowie vom Bot verwendet werden.

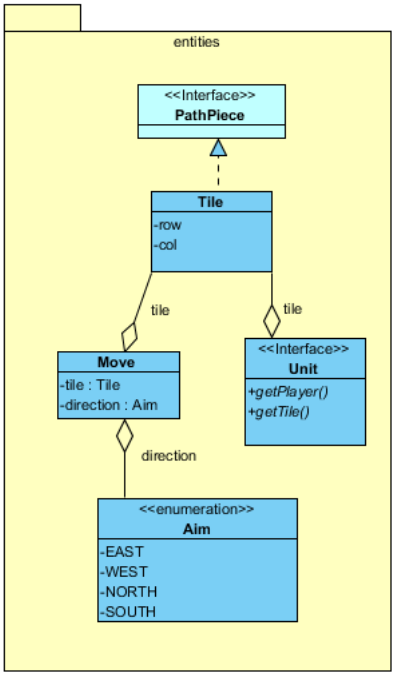

Abbildung 3.1: Entities

- **Aim**: Richtungangabe zum Beschreiben einer Bewegung der Ameise. Da sich die Ameisen nur geradeaus bewegen können, beschränken sich die gültigen Werte auf die vier Himmelsrichtungen.
- **Tile**: Repräsentiert eine Zelle auf dem Spielfeld, welche mit Row (Zeile) und Column (Spalte) definiert ist.
- **Move**: Diese Klasse beschreibt einen Spielzug mit den Eigenschaften Tile, von wo aus der Zug statt findet, und Aim, in welche Richtung der Zug ausgeführt wird.
- **Unit**: Unit besteht aus Tile und Spieler und definiert eine Einheit eines Spielers auf der Karte.

3.2 Map

Im Map-Package sind verschiedene Karten-Typen definiert, die die Interaktion mit der Spielwelt ermöglichen.

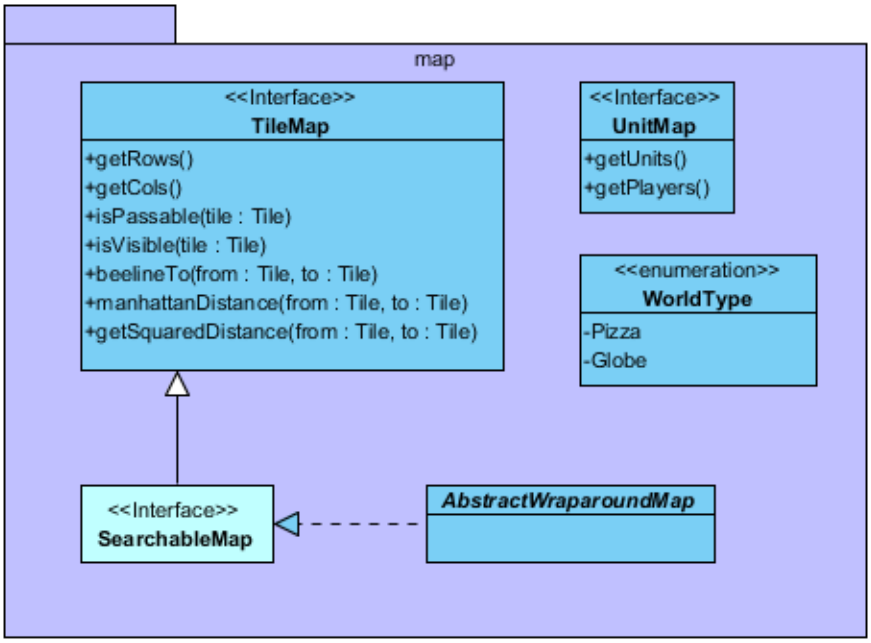

Abbildung 3.2: Map API

- **TileMap**: Dieses Interface definiert, welche Methoden eine auf Tiles aufbauende Spielkarte anbieten muss. Dazu gehören die Masse der Karte, die Geländebeschaffenheit, also ob eine Zelle für den Spieler sichtbar und passierbar ist, sowie Distanz-Messfunktionen wie ManhattanDistance, Luftlinie, quadrierte Distanz.
- **UnitMap**: Dieses Interface definiert, welche Methoden eine Spielkarte implementieren muss, um Zugriff auf die in der Karte platzieren Einheiten zu bieten. Die Methode getPlayers() gibt eine Liste der Spieler zurück, die Einheiten auf der Karte kontrollieren, die Methode getUnits() gibt alle bekannten Einheiten zurück.
- **AbstractWraparoundMap**: Implementiert das Interface SearchableMap (s. 3.3). Hier sind alle Methoden implementiert, welche die Karte anbieten muss, damit sie mit den Suchalgorithmen verwendet werden kann. Zudem sind die vordefinierten Methoden aus TileMap implementiert.
- **WorldType**: WorldType ist ein Enum und definiert die Art der Karte. Der Typ *Globus* hat keine Kartenränder, ist also ringsum begehbar. Von diesem Typ ist auch die AbstractWraparoundMap Klasse, welche diverse Methoden des TileMap Interfaces bereits implementiert und als Basisklasse unserer World-Karte[1] dient. Folglich sind auch die Karten der Ants Challenge 'WrapAround'-Karten. Der zweite Enumtyp ist *Pizza* und definiert eine Welt wie man sich unsere Erde vor 500 Jahren noch vorstellte, eine Scheibe mit Rändern, welche die Welt begrenzen. Dieser zweite Typ wurde provisorisch erstellt. Falls unsere API eine Weiterverwendung findet, kann dieser Typ zusätzlich implementiert werden.

3.3 Search

Search beinhaltet alle Basisklassen und -interfaces der Suche.

[1]Die Klasse World ist im Code unter ants.state.World eingegliedert.

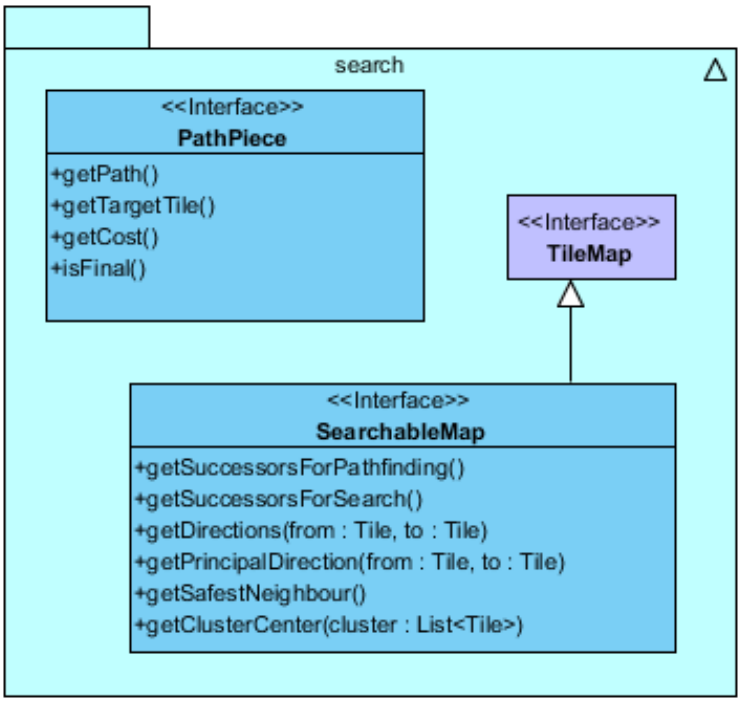

Abbildung 3.3: Search API

- **PathPiece**: Das PathPiece ist ein Interface für Strukturen, die als Suchknoten in der Pfadsuche verwendet werden können. Implementierende Klassen sind Edge (repäsentiert eine Kante in einem Cluster) und Tile. Erweiterungen dieser Klassen sind DirectedEdge (gerichtete Kanten) und Vertex (ein Knoten mit anliegenden Kanten).
- **SearchableMap**: Das Interface SearchableMap erweitert das Interface TileMap. Es beschreibt die Methoden zur Pfadsuche, wie getSuccessors(), getCost(), oder getPath(). Die Unterscheidung zwischen getSuccessorsForPathfinding() und getSuccessorsForSearch haben wir eingeführt, damit bei der Pfadsuche bestimmte Tiles von vornherein ausgeschlossen werden können. (Konkret haben wir so verhindert, dass Pfade über unsere eigenen Hügel führen).

3.4 Strategy

In diesem Package befinden sich die Karteninterfaces, welche für das Anwenden von Strategie und Taktikalgorithmen grundlegend sind.

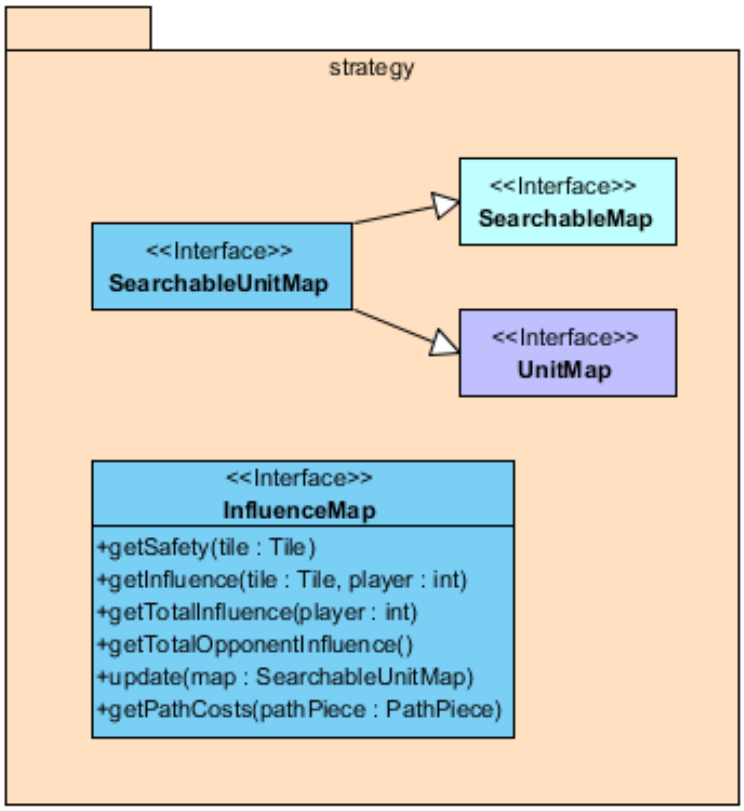

Abbildung 3.4: Strategy API

- **SearchableUnitMap**: Dieses Interface dient ausschliesslich zur Zusammenführung der beiden Interfaces UnitMap und SearchableMap.
- **InfluenceMap**: Dieses Interface definiert Methoden, die eine Influence Map anbieten muss. So muss sie für ein bestimmtes Tile und einen bestimmten Spieler den Einfluss angeben können, oder sie muss Auskunft über die Sicherheit eines bestimmten Tiles geben können. Mit der update() Methode kann die InfluenceMap aktualisiert werden.

4 Suchalgorithmen

Das erstellt Suchframework bietet eine Pfadsuche durch drei verschiedenen Algorithmen an. Zudem eine Breitensuche und eine Abwandlung davon, die Barriersuche. Als Ergänzung wurde auch ein PathSmoothing-Algorithmus implementiert.

4.1 Enities für die Pfadsuche

Abbildung 4.1 zeigt die wichtigsten Klassen, die für die Pfadsuche verwendet werden. Der Übersichtlichkeit wegen wurden nur die wichtigsten Attribute und Operationen in das Diagramm aufgenommen. Die hellgrünen Interfaces stammen von der AITools API und wurden bereits dort erläutert.

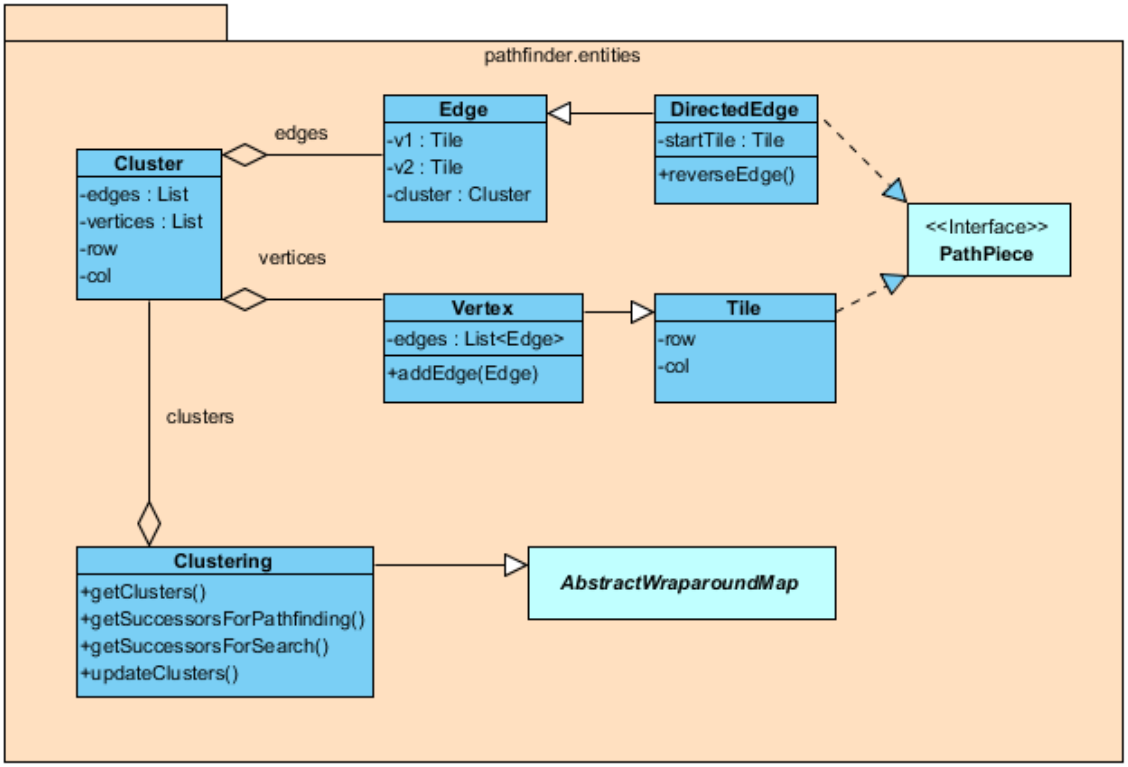

Abbildung 4.1: Spiel-Elemente für die Suche (vereinfacht)

TODO PathPiece, Tile, AbstractWrapAround sind nicht in diesem?
TODO interface grün?

- **Edge**: Repräsentiert eine Kante und wird für das Clustering verwendet.
- **DirectedEdge**: Erweitert die Klasse Edge, indem die Kante in dieser Klasse gerichtet ist.
- **Cluster**: Der Cluster, ist ein Kartenausschnitt und wird vom HPA* Algorithmus (Kapitel 4.2.3) genutzt, indem er Pfade die Pfade kennt, die durch diesen Kartenabschnitt führen.
- **Vertex**: Ein Vertex ist ein Knoten der Teil bei Pfadsuche ist. Er verbindet weitere Konten durch Katen (Edges).

- **Clustering**: Das Clustering ist auch Teil des HPA* Algorithmus. Es ist für die Aufteilung der Karte in mehrere Clusters zuständig.

Die Klasse SimplePathFinder kapselt und führt verschiedene Suchstrategien aus. Für den HPA* Algorithmus wurde diese Klasse durch ClusteringPathFinder erweitert. Diese beinhaltet zusätzlich, wie der Name schon sagt, das Clustering.

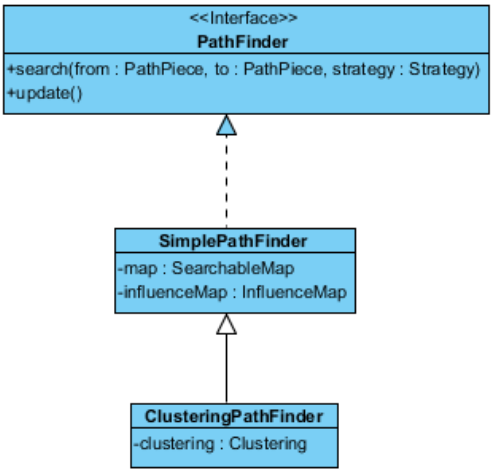

Abbildung 4.2: Klassendiagramm Pfadsuche

4.2 Pfadsuche

Wir haben drei unterschiedliche Pfadalgorithmen in unserem Code eingebaut. Via PathFinder-Klasse kann für die Pfadsuche der Algorithmus ausgewählt werden.

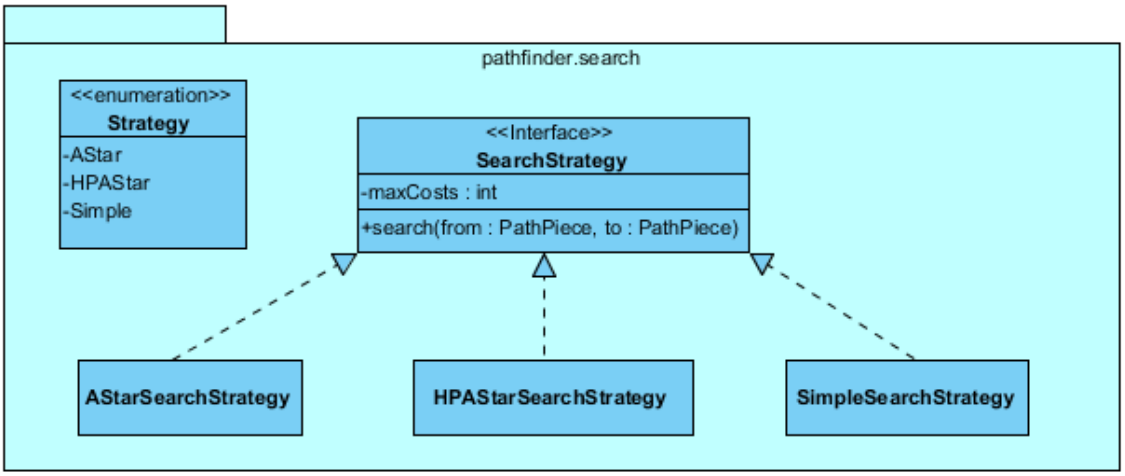

Abbildung 4.3: Suchstrategien

4.2.1 Simple Algorithmus

Der Simple Algorithmus versucht das Ziel zu erreichen indem zuerst die eine Achse, danach die andere Achse abläuft. Sobald ein Hindernis in den Weg kommt, bricht der Algorithmus ab. In der Abbildung 4.4 sucht der Algorithmus zuerst den vertikal-horizontal Pfad. Da dieser Pfad wegen dem Wasserhindernis (blau) nicht ans Ziel führt, wird der Pfad horizontal-vertikal gesucht. In dieser Reihenfolge wird ein Pfad gefunden. Dieser Algorithmus ist, wie der Name

bereits aussagt, sehr einfach aufgebaut und kostet wenig Rechenzeit. Er ist nur für kurze Distanzen praktikabel, da er keinen Hindernissen ausweichen kann.

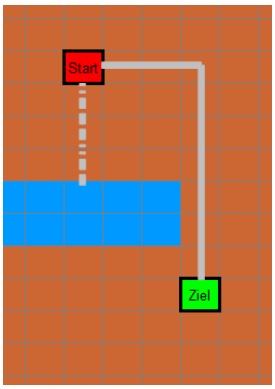

Abbildung 4.4: Simple-Path Algorithmus

Folgendes Codesnippet zeigt auf wie ein Pfad mittels Pfadsuche Simple gefunden wird. Ein SimplePathFinder wird mit der Karte initialisiert. Danach kann die Suche mit pf.search(...) gestartet werden. Als Parameter wird der Suchalgorithmus Strategy.Simple, der Startpunkt (Position der Ameise) und Endpunkt (Position des Futters), sowie die maximalen Pfadkosten (hier: 16) mitgegeben.

```
SimplePathFinder pf = new SimplePathFinder(map);
List<Tile> path = pf.search(PathFinder.Strategy.Simple, ant.getTile(), food,16);
```

4.2.2 A* Algorithmus

Den A* Algorithmus haben wir nach dem Beschrieb im Buch TODO implementiert. Beim Algorithmus werden für jeden expandierten Knoten die geschätzten Kosten f(x) für die gesamte Pfadlänge berechnet. f(x) besteht aus einem Teil g(x) welches die effektiven Kosten vom Startknoten zum aktuellen Knoten berechnet. Der andere Teil h(x) ist ein heuristischer Wert, der die Pfadkosten bis zum Zielknoten approximiert. Dieser Wert muss die effektiven Kosten zum Ziel immer unterschätzen um zu gewähren, dass der kürzeste Pfad gefunden wird. Dies ist in unserem Spiel dadurch gegeben, dass sich die Ameisen nicht diagonal bewegen können, wir aber für den heuristischen Wert die Luftlinie zum Ziel verwenden. Die Pfadsuche wird immer bei dem Knoten fortgesetzt welcher die kleinsten Kosten f(x) hat. Da wir eine TileMap verwenden, definiert jede begebare Zelle ein Knoten.

Die Abbildung 4.5 zeigt den effektiven Pfad (grau) vom zu expandierenden roten Knoten mit den minimalen Kosten von 10 Tiles. Die Luftlinie (blau) als heuristischer Wert hat aber nur eine Länge von 7.6 Tiles. Damit erfüllt unsere Heuristik die Anforderungen des Algorithmus.

Eine Pfadsuche mit A* wird gleich ausgelöst wie die Suche mit dem Simple-Algorithmus, ausser dass als Parameter die Strategy AStar gewählt wird.

```
SimplePathFinder pf = new SimplePathFinder(map);
List<Tile> path = pf.search(PathFinder.Strategy.AStar, ant.getTile(), foodTile,16);
```

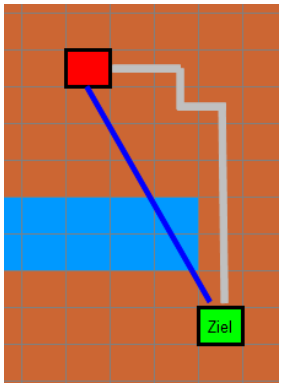

Abbildung 4.5: Heuristische Kosten (blau), Effektive Kosten (grau)

4.2.3 HPA* Algorthmus

Eine Pfadsuche A* über alle Zellen der Spielkarte ist sehr teuer, da es viel Pfade gibt, die zum Teil nur eine oder wenige Tiles nebeneinander liegen. Es werden bis zum Schluss verschiedenen Pfaden nachgegangen, die sehr ähnlich sind. Abhilfe zu dieser sehr feinmaschigen Pfadsuche bietet der Hierarchical Pathfinding A* bei welchem die Karte in Regionen (Cluster) aufgeteilt wird. Von Cluster zu Cluster werden Verbindungspfade vorberechnet, welche der Algorithmus bei der Pfadsuche verwendet.

4.2.3.1 Clustering

Das Clustering wird während dem ClusteringTask (siehe 6.4) ausgeführt, Dabei wird die Landkarte, wie bereits erwähnt, in sogenannte Clusters unterteilt. Auf dem Bild 4.7 wurde die Karte in 16 Clusters aufgeteilt.

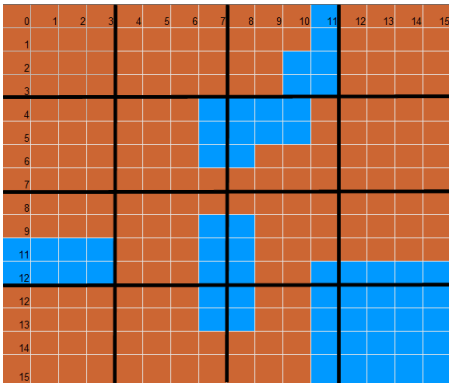

Abbildung 4.6: Clustereinteilung auf der Landkarte. Clustergrösse 4x4, Landkarte 16x16

Wir unterschieden zwischen den zwei Clusterarten *Centered* und *Corner*. Die Variante Corner wurde bereits im Vormodule 'Projekt 2' implementiert während die Variante Corner im Laufe dieser Arbeit dazu kam. Folgendes Bild zeigt den Unterschied der Varianten. Corner generiert zwei Übergangspunkte plus eine Verbindung auf der Kante zwischen zwei Clusters. Die Variante Centered generiert nur einen Übergangspunkt in der Kantenmitte der aneinander grenzenden Clusters. Die Variante Centered hat den Vorteil, dass es weniger dichtes Pfadnetz gibt, da weniger Übergangspunkte, aber sie hat auch den Nachteil, dass der gefunden Pfad, nicht in jedem Fall der Kürzeste ist. Ein Pathsmoothing muss angewendet werden.

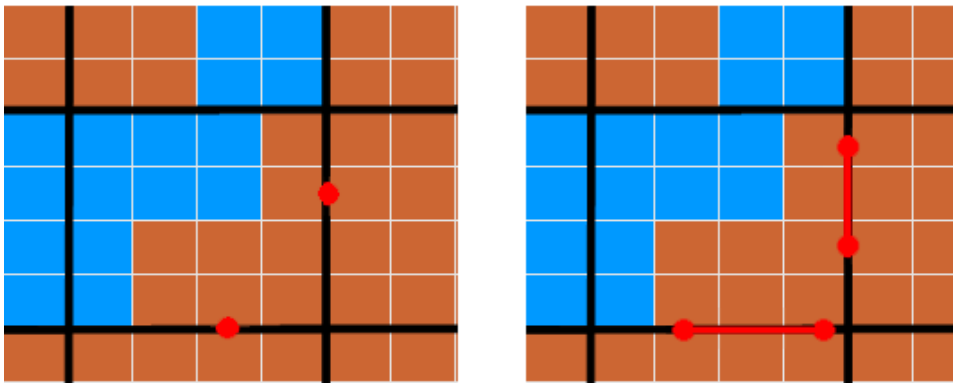

Abbildung 4.7: Vergleich der Clusterarten: Links der Typ *Corner*, Rechts der Typ *Centered*

Nachfolgend wir erläutert wie das Clustering vonstatten geht, verwendet wird die Custeringart Corner.
Nach dem Einteilen der Cluster werden für jeden Cluster und einen Nachbar-Cluster aus der Vierer-Nachbarschaft die Verbindungskanten berechnet. Dies kann natürlich nur für Clusters gemacht werden die auf einem sichtbaren Teil der Landkarte liegen, was zu Beginn des Spiel nicht unbedingt gegeben ist. Deshalb wird der ClusteringTask in jedem Spielzug aufgerufen, in der Hoffnung das der Cluster komplett sichtbar ist. Sobald eine beliebige Seite eines Clusters berechnet ist, wird diese Aussenkante im Cluster und dem anliegenden Nachbar gespeichert und nicht mehr neu berechnet.

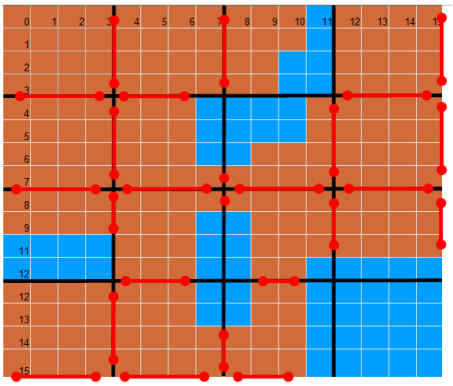

Abbildung 4.8: Die Kanten jedes Clusters wurden berechnet

Wenn ein Cluster zwei oder mehrere Aussenkanten kennt berechnet er die Innenkanten mit A*. Diese verbinden die Knoten der verschiedenen Aussenkanten. Das ergibt nun ein Pfadnetz über die Gesamtkarte. Im nachfolgenden Bild sind die Innenkanten (gelb) ersichtlich, die bei den ersten 8 Cluster berechnet wurden.

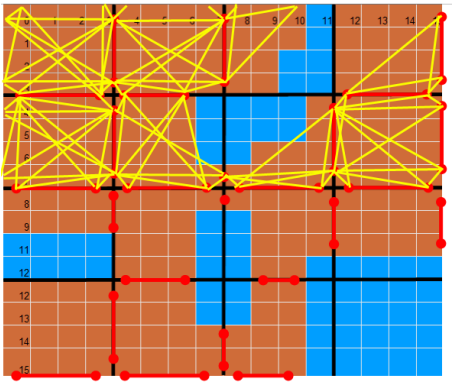

Abbildung 4.9: Darstellung der Innenkanten

Angenommen das Clustering wurde über die ganze Karte vorgenommen, kann wi in der Abbildung 4.10 ersichtlich, ein Pfad vom Pixel (3,9) nach (13,9) mittels HPA* gesucht (grüne Punkte) werden. Zuerst wird eruiert in welchem Cluster sich das Start- bzw Zielpixel befindet. Danach wird in dem gefundenen Cluster ein Weg zu einem beliebigen Knoten auf der Clusterseite gesucht. Sind diese Knoten erreicht (blaue Pfade), wird nun das vorberechnete Pfadnetz mittels A* Heuristik verwendet um die beiden Knoten auf dem kürzesten möglichen Pfad (gelb) zu verbinden. Der resultierende Pfad könnte nun via Pathsmoothing noch verkürzt werden.

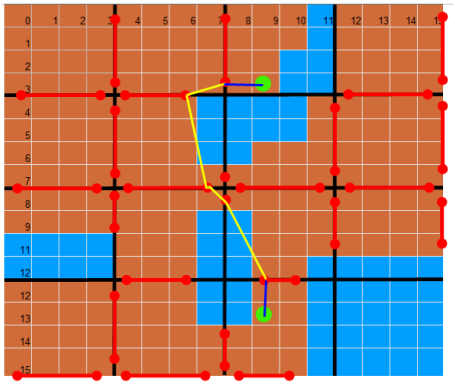

Abbildung 4.10: Errechneter Weg mittels HPA*

Um im Code mit einer Pfadsuche HPA* zu suchen, muss ein ClusteringPathFinder instanziert werden. Als Parameter erwartet der Konstruktor die Karte auf welcher das Clustering und die Pfadsuche gemacht wird, sowie die Clustergrösse (hier: 10) und den Clustertyp. Das Clustering wird mit pf.update() durchgeführt. Danach kann die Pfadsuche gemacht werden. Falls nicht alle Clusters zur Verfügung stehen, weil es noch unbekannte Flecke auf der Karte gibt, wird als Fallback versucht mit A* einen Pfad zu suchen.

```
ClusteringPathFinder pf = new ClusteringPathFinder(map, 10, type);
pf.update();
List<Tile> path = pf.search(PathFinder.Strategy.HpaStar, start, end, -1);
```

4.2.4 Pfadsuche mittels Influence Map

Die Influence Map, welche wir während der Bachelorarbeit neu implementiert haben, kann auch für die Pfadsuche verwendet werden. Dabei sind die Pfadkosten für Gebiete die vom Gegner kontrolliert sind höher als für neutrale Gebiete und tiefer für solche Gebiete die von unseren Ameisen kontrolliert werden. (Details zur Implementierung der InfluenceMap sieh Kapitel 5.1) Die Methode getActualCost(...) in der Klasse SearchStrategy wurde erweitert. Falls die Suche mit einer InfluenceMap initialisiert wurde, sind die Kosten nicht eine Einheit pro Pfadtile, sondern können zwischen 1 (sicheres Gebiet) - 4 (gefährliches Gebiet) Einheiten variieren. (Die Pfadkosten dürfen nicht negativ sein, sonst würde der A* Algorithmus nicht mehr korrekt funktionieren.) Die Kosten für jedes Pfadtile werden durch die Methode *getPathCosts(...)* der InfluenceMap berechnet.

```
protected final int getActualCost(Node current, PathPiece piece) {
    int costOfPiece = 0;
    if (useInflunceMap)
        costOfPiece = pathFinder.getInfluenceMap().getPathCosts(piece);
    else
        costOfPiece = piece.getCost();
    return current.getActualCost() + costOfPiece;
}
```

Listing 4.1: Die Kosten für das Wegstück (PathPiece) werden von der InflueceMap (falls verwendet) berechnet.

Dadurch resultiert ein Pfad der eher durch sicheres Gebiet führt. Folgende Ausgabe, welche durch einen UnitTest generiert wurde, bezeugt die korrekte Funktionalität. Der rote Punkt soll mit dem schwarzen Punkt durch einen Pfad verbunden werden. Auf der Karte sind zudem die eigenen, orangen Einheiten sowie die gegnerischen Einheiten

(blau) abgebildet. Jede Einheit trägt zur Berechnung der InfluenceMap bei. Pro Tile wird die Sicherheit auf der Karte ausgegeben, negativ für Gebiete die vom Gegner kontrolliert werden und positiv in unserem Hoheitsgebiet.

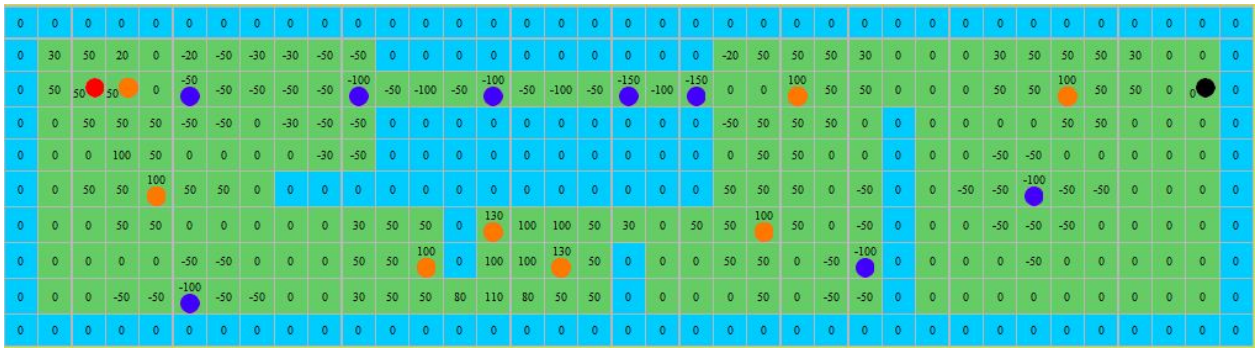

Abbildung 4.11: Ausgangslage Pfadsuche mit A* und InfluenceMap

Ohne Berücksichtigung der InfluenceMap würde der A* Algorithmus einen Pfad finden der auf direktem Weg waagrecht zum Zielpunkt führt. Sobald aber die InfluenceMap berücksichtigt wird, führt der Pfad nicht mehr auf dem direktesten Weg zum Ziel, sondern nimmt einen Umweg über sicheres Gebiet. Unten abgebildet ist der kürzeste Pfad mit Berücksichtigung der InfluenceMap (blau) und ohne InfluenceMap-Berücksichtigung (orange).

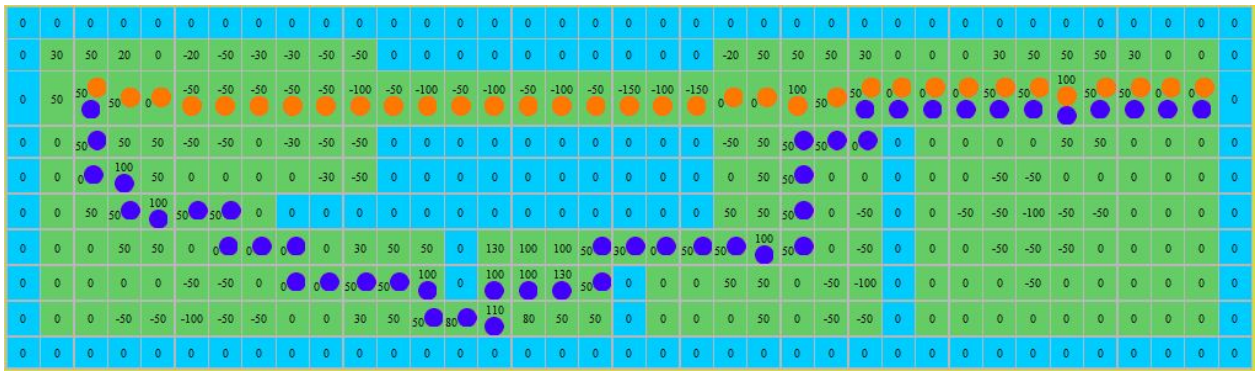

Abbildung 4.12: Resultierende Pfade mit und ohne Berücksichtigung der InfluenceMap

Die Pfadkosten für beide Pfade verglichen, legt offen, dass je nach Berücksichtigung der InfluenceMap nicht der gleiche Pfad als der 'Kürzeste' von A* gefunden wird.

Pfadkosten	**ohne InfluenceMap**	**mit InfluenceMap**
Oranger Pfad	**34**	110
Blauer Pfad	46	**106**

Tabelle 4.1: Pfadkosten mit und ohne Berücksichtigung der InfluenceMap

4.2.5 Pathsmoothing

Um unseres Search-Framework zu komplettieren bietet die Pfadsuche auch ein PathSmoothing, das 'Glätten' eines Pfades an. Wie im Clustering schon erwähnt, kann es sein, dass ein Pfad, der von dem HPA* Algorithmus gefunden wurde, nicht zwingend der Kürzeste ist. Die folgende Abbildung veranschaulicht wie der gefundene Pfad von Cluster zu Cluster (weisse Markierung), stets über die vorberechneten Verbindungspunkte (blau) verläuft. Dies ist nicht der optimale Pfad, er kann mit PathSmoothing verkürzt werden.

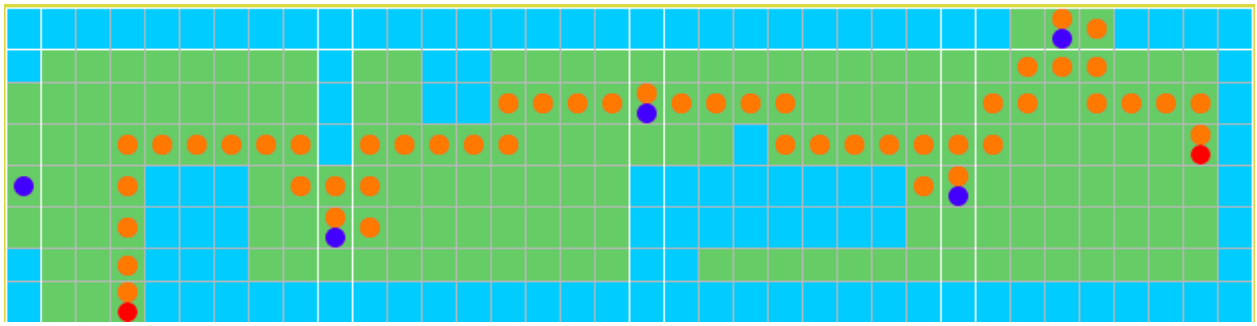

Abbildung 4.13: Der gefundene Pfad mit HPA* Clusteringart Centered, ist nicht der kürzeste.

Der Algorithmus des Pathsmoothing ist (vereinfacht) wie folgt definiert. Vom Pfad der gekürzt werden soll wird ein erster Abschnitt mit der Länge *size* genommen. Mittels manhattanDistance wird geprüft ob eine kürzere Weg für diesen Abschnitt möglich wäre. Falls ja wird mit A* ein neuer Pfad gesucht, sonst wird der alte Pfad (*subPath*) übernommen. Dieses Verfahren wird für alle nachfolgenden Pfadabschnitte gemacht, bis der ganze Pfad durchlaufen ist.

Rekursion: Der beschriebene Algorithmus, hat nicht in jedem Fall den kürzesten Pfad als Output. So sind zwar alle Pfadabschnitte optimal gekürzt, es kann aber sein, dass wenn zwei Abschnitte zusammen gefügt werden, der Pfad nicht mehr der kürzeste ist. Als Beispiel: Die Wegabschnitt Zürich-Thun und Thun-Genf mögen optimal gekürzt sein. Zusammengefügt zur Strecke Zürich-Genf, braucht es den Umweg über Thun nicht. Um Umweg zu entfernen wird der Algorithmus rekursiv aufgerufen, indem smoothPath(...) mit einer grösseren *size* als Parameter für die zusammengesetzten Abschnitte nochmals aufgerufen wird.

```
public List<Tile> smoothPath(List<Tile> path, int size, boolean recursive) {
    int start = 0;
    int current = size;
    List<Tile> smoothedPath = new ArrayList<Tile>();
    // do while every subPath of path is checked to be shorten and added to
        smoothedPath
    do {
        List<Tile> subPath = path.subList(start, current);
        int manDist = map.manhattanDistance(subPath.get(0), subPath.get(subPath.
            size() - 1)) + 1;
        List<Tile> newSubPath = null;
        if (manDist < subPath.size()) {
            newSubPath = search(Strategy.AStar, subPath.get(0), subPath.get(subPath
                .size() - 1), subPath.size() - 1);
        }
        if (newSubPath != null) {
            smoothedPath.addAll(newSubPath);
            if (recursive && newSubPath.size() < subPath.size()) {
                smoothedPath = smoothPath(smoothedPath, smoothedPath.size(), true);
            }
        } else {
            smoothedPath.addAll(subPath);
        }
        start = current;
        current = Math.min(current + size, path.size());
} while (!path.get(path.size() - 1).equals(smoothedPath.get(smoothedPath.size() -
    1)));

return smoothedPath;
}
```

Listing 4.2: JavaCode für das PathSmoothing

In der nächsten Abbildung wurde der beschriebene PathSmoothing Algorithmus angewendet, der Pfad konnte einer ursprünglichen Pfadlänge von 50 Tiles (siehe 4.13) auf eine Pfadlänge von 40 Tiles reduziert werden.

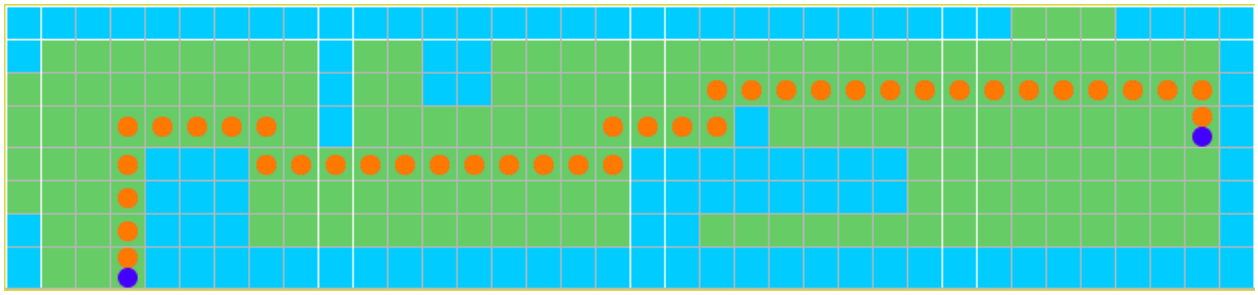

Abbildung 4.14: Der geglättete Pfad, nach Anwendung des PathSmoothing-Algorithmus

4.3 Breitensuche

Die Breitensuche (engl. Breadth First Search (BFS)) war eine der Neuimplementierungen während der Bachelorarbeit. Wir verwenden diese Suche um die Umgebung einer Ameise oder eines Hügels nach Futter, Gegnern usw. zu scannen. Man könnte die BFS auch für die Pfadsuche verwenden, dies wäre aber sehr ineffizient. Im Klassendiagramm ist zu sehen auf welchen drei Methoden die Breitensuche aufbaut.

Der Algorithmus für die Breitensuche ist in [2] beschrieben und wurde nach dieser Beschreibung implementiert.

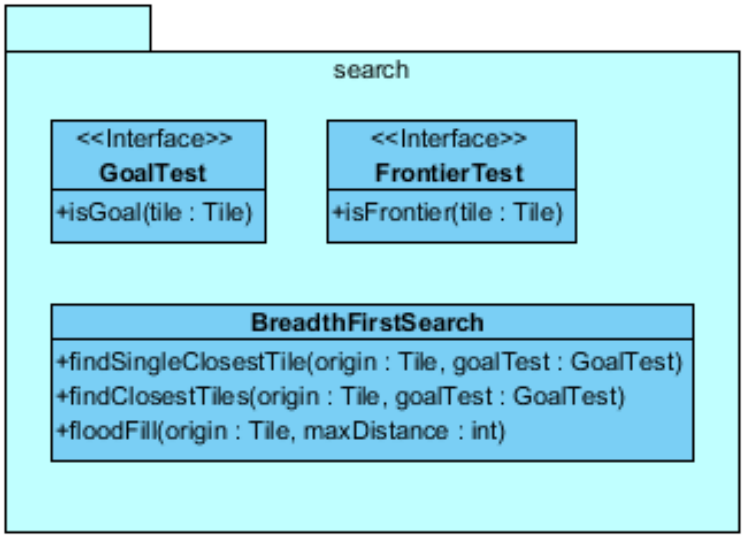

Abbildung 4.15: Breitensuche Klassendiagramm

Die Breitensuche wurde generisch implementiert, so dass sie vielseitig einsetzbar ist. So können zum Beispiel mittels 'GoalTest' je nach Anwendungsfall die Tiles beschrieben werden die gesucht sind. Folgende Breitensuche findet die Ameise welche am nächsten bei einem Food-Tile <r:20,c:16> ist. Die Suche wird initialisiert indem im Konstruktor die Spielkarte mitgegeben wird, welche durchforscht wird. Zusätzlich gilt die Einschränkung das die Breitensuche nur 40 Tiles durchsuchen darf, was einem Radius von zirka 7 Zellen entspricht. Falls keine Ameise gefunden wird gibt der Algorithmus NULL zurück.

```
AntsBreadthFirstSearch bfs = new AntsBreadthFirstSearch(Ants.getWorld());
Tile food = new Tile(20,16);
Tile antClosestToFood = bfs.findSingleClosestTile(food, 40, new GoalTest() {
      @Override
      public boolean isGoal(Tile tile) {
          return isAntOnTile(tile);
      }
  });
```

Es ist auch möglich mehrere Tiles zurück zu bekommen. Dazu wird die Methode `findClosestTiles(...)` aufgerufen.

Der gleiche Algorithmus kann aber auch alle passierbaren Tiles in einem gewissen Umkreis zurückgeben. Dies haben wir unter anderem beim Initialisieren der DefendHillMission verwendet. Wir berechnen beim Erstellen der Mission die passierbaren Zellen rund um den Hügel. Runde für Runde prüfen wir diese Tiles auf gegnerische Ameisen um die entsprechenden Verteidigungsmassnahmen zu ergreifen. Der Parameter controlAreaRadius2 definiert den Radius des 'Radars' und kann je nach Profil unterschiedlich eingestellt werden.

Dieser Flood Fill Algorithmus ist praktisch identisch mit der Breitensuche. Eine Beschreibung von Flood Fill befindet sich in [1].

```
public DefendHillMission(Tile myhill) {
    this.hill = myhill;
    BreadthFirstSearch bfs = new BreadthFirstSearch(Ants.getWorld());
    tilesAroundHill = bfs.floodFill(myhill, controlAreaRadius2);
}
```

Um die Aufrufe der Suche im Ants-Umfeld einfacher zu gestalten haben wir die Breitensuche für unseren Bot mit folgenden selbst-sprechenden Methoden erweitert.

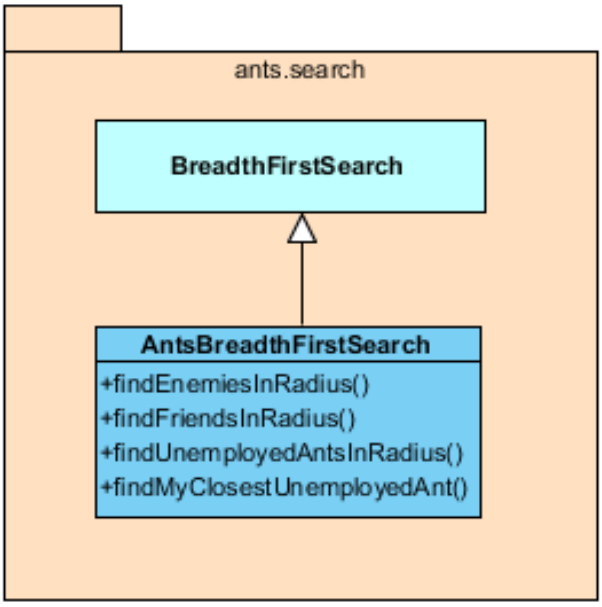

Abbildung 4.16: Breitensuche Ants-spezifisch

4.3.1 Warum Breitensuche?

Bei der Wahl des Suchalgorithmus orientierten wir uns stark an [2, S. 64-109], wo die wichtigsten Suchalgorithmen beschrieben und nach verschiedenen Kriterien beurteilt werden. Für eine detailliere Diskussion über die Eigenschaften der Algorithmen sei auf diese Seiten verwiesen; an dieser Stelle wollen wir nur kurz aufzeigen, nach welchen Kriterien wir uns für die Breitensuche entschieden haben.

Abgesehen von der Pfadsuche, bei der wir auf die heuristischen Algorithmen A* und HPA* setzten, waren unsere Anforderungen bei der Suche meist so geartet, dass eine Heuristik entweder nicht existiert, oder keine Vorteile bringt. Wenn wir beispielsweise für ein Nahrungs-Tile die nächste Ameise suchen wollen, ist das Ziel der Suche ja nicht bekannt, und daher kann keine sinnvolle Heuristik definiert werden. Es blieb also nur die Frage, welche der verfügbaren nicht-heuristischen Algorithmen wir einsetzen wollten.

Die wichtigsten Anforderungen an den Such-Algorithmus waren die Vollständigkeit (Wenn eine Lösung existiert, muss sie gefunden werden), und die Optimalität (Eine gefundene Lösung muss die kleinsten Kosten aller existierenden Lösungen haben). Damit fiel bereits die Tiefensuche weg. Die Uniform-Cost Suche machte im Kontext von Ants keinen Sinn, da in diesem Spiel jeder Schritt immer die Kosten 1 hat. Eine Bidirektionale Suche war ebenfalls nicht praktikabel, da wir ja das Ziel der Suche meist nicht kennen. Daher mussten wir uns lediglich noch zwischen der Iterativen Tiefensuche und der Breitensuche entscheiden; die Entscheidung fiel zugunsten der Breitensuche aus, weil sie einfacher zu implementieren ist, und weil wir damit auch gleich einen geeigneten Flood Fill Algorithmus implementieren konnten.

Was für unsere Entscheidung keine Rolle spielte, waren die Zeit- und Raum-Komplexität der Algorithmen; die konkreten Anwendungsfälle für die Suche waren klein genug, dass die Unterschiede vernachlässigbar waren.

4.3.2 Barrier (Sperre)

Eine Erweiterung der Breitensuche ermöglicht uns eine Sperre in der Umgebung eines Ortes zu finden. Diese Verwenden wir in der DefendHillMission zum Verteidigen des eigenen Hügels. Es kann nur eine Sperre (engl. Barrier) gefunden werden wenn das Gelände dazu passt. Die Abbildung 4.17 zeigt eine solche gefundene Sperre. Auf dieser Höhe wird der Hügel verteidigt.

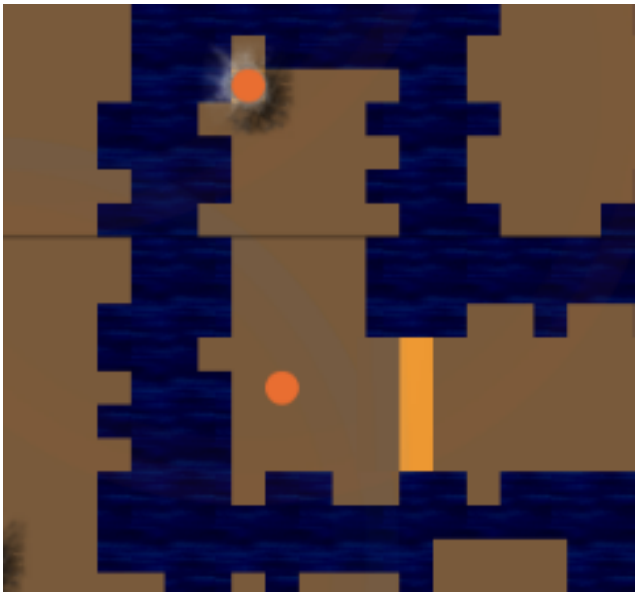

Abbildung 4.17: Auf der orangen Sperre werden die Ameisen zur Verteidigung des Hügel positioniert.

Der Algorithmus ist in der Methode `getBarrier(...)` implementiert. Diese wird aufgerufen mit den Parametern `tileToProtect` (Ort der durch eine Sperre geschützt werden soll), `viewRadiusSquared` (den Sichtradius der Einheiten), denn die Sperre soll weiter entfernt sein als der Sichtradius, damit die gegnerischen Einheiten nicht sehen was sich dahinter verbirgt. Der dritte Parameter `maximumBarrierSize` definiert, welche Breite die Sperre maximal haben darf.

```
public Barrier getBarrier(final Tile tileToProtect, int viewRadiusSquared, int
    maximumBarrierSize) {

    //BFS for getting the amount of tiles in view radius around the location to
        defend.
    int amount = [...]
    Barrier smallestBarrier = null;
    List<Tile> tiles = get (amount + 30) tiles around the location to defend.

    // for loop start at the first tile not in view radius
    for(int i = amount;i<tiles.size(); i++){
```

```
        Tile t = tiles.get(i);

        //vertical check
        if(!barrierVerticalInvalid.contains(t)){
            Barrier b = get vertical barrier on position of Tile t
            if(b is smaller than 5 Tiles && smaller than smallestBarrier){
                if(is barrier the only exit out of the location to defend){
                        smallestBarrier = b;
                }else{
                         //add all tiles of the invaild barrier
                        barrierVerticalInvalid.add(b.getTiles());
                }
            }else{
                 //add all tiles of the invaild barrier
                barrierVerticalInvalid.add(b.getTiles());
            }
        }

        //horizontal check
        if(!barrierHorizontalInvalid.contains(t)){
            Barrier b = get horiontal barrier on position of Tile t
            if(b is smaller than 5 Tiles && smaller than smallestBarrier){
                if(is barrier the only exit out of the location to defend){
                        smallestBarrier = b;
                }else{
                         //add all tiles of the invalid barrier
                        barrierHorizontalInvalid.add(b.getTiles());
                }
            }else{
                //add all tiles of the invalid barrier
                barrierHorizontalInvalid.add(b.getTiles());
            }
        }
    }
}
```

Dank dem Abspeichern der ungültigen Tiles aller zu breiten Sperren in die Listen `barrierHorizontalInvalid` und `barrierVerticalInvalid` konnte der Algorithmus markant schneller gemacht werden. Für diese Tiles muss nicht nochmals eine Sperre berechnet werden. Auch die if-Abfrage `barrier is the only exit out of the location to defend` muss nicht mehr oft aufgerufen werden, den hinter dieser Abfrage steht nämlich wiederum ein Test mit der Breitensuche. Dieser zusätzliche Test mit der Breitensuche ist viel teurer als das Zwischenspeichern der Tiles aus welchen keine gültige Sperre gemacht werden konnte.

Im Nachhinein hat sich ergeben, dass nicht unbedingt die schmalste Sperre die Beste wäre, sondern jede bei welche der Gegner, geländebedingt, weniger Einheiten aufstellen kann. So wäre in Abbildung 4.17 eine Sperre zwei Zellen östlicher besser, den der Gegner könnte beim Angriff nur mit vier Einheiten vorrücken, die Verteidigung wäre aber mit sechs Einheiten auf einer Linie deutlich stärker. Die Zeit hat aber hier leider nicht gereicht, den Algorithmus weiter zu verfeinern.

5 Strategie und Taktik

5.1 Influence Map

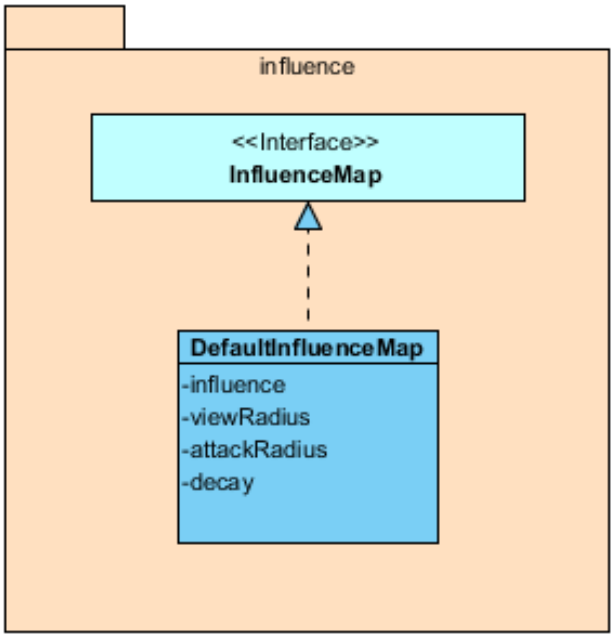

Abbildung 5.1: Influence Map Klassendiagramm

Die InfluenceMap haben wir nach den Beschreibungen in [1] implementiert. Jede bekannte Spieleinheit auf der Spielkarte 'strahlt' eine gewissen Einfluss aus. In unserer Implementation unterscheiden wir zwischen drei Einflussradien, der Angriffsradius, der erweiterte Angriffsradius und der Sichtradius. Den Radien haben wir folgende Werte zugewiesen.

Radius	Wert	Radius in Tiles*
Angriffsradius	50	2.2
Erweiterter Angriffsradius	30	5
Sichtradius	10	8.8

Tabelle 5.1: Einfluss einer Spieleinheit

* Der Radius kann je nach Spieleinstellungen ändern. Angegeben sind die Defaultwerte.

Wir verenden die InfluenceMap vorallem für die Bestimmung der Sicherheit. Abgebildet ist eine Sicherheitskarte (Desirability Map) für den orangen Spieler, wobei die Einflusswerte des Gegners von den Einflusswerten des eigenen Spielers je Tile subtrahiert werden. Positive Werte bedeuten sicheres Terrain und negative Werte unsicheres, vom Gegner kontrolliertes Gebiet.

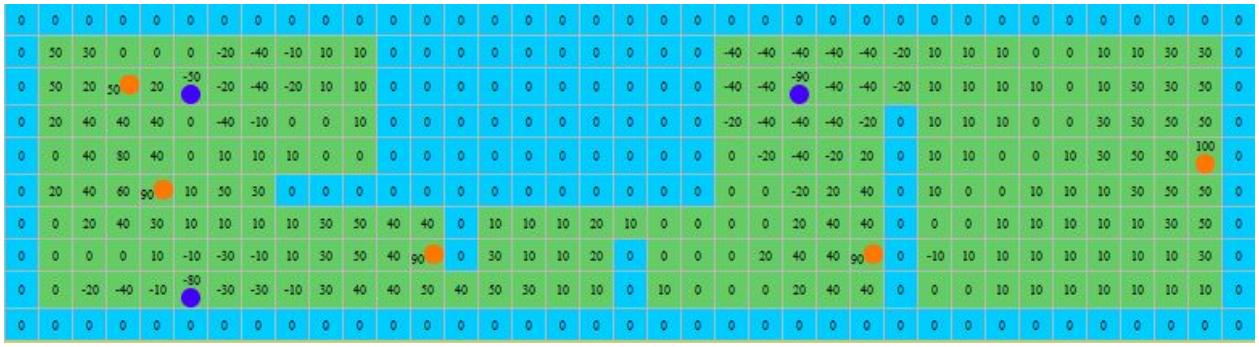

Abbildung 5.2: Influence Map, dargestellt ist die Sicherheit je Tile.

5.1.1 Update

Die InfluenceMap wird zu Beginn des Spiels initialisiert, danach wird vor jeder Spielrunde ein Update gemacht. Dabei definiert ein Decay-Wert zwischen 0 und 1, wieviel von den alten Werten beibehalten wird. Folgende Formel bestimmt den neuen Wert für jede Zelle:

$val_{(x,y)} = val_{(x,y)} * decay + newval_{(x,y)} * (1 - decay)$

5.1.2 Anwendungsfälle

In folgenden Modulen berücksichtigen wir Werte aus InfluenceMap um Entscheide zu fällen.

- **Pfadsuche mit InfluenceMap Berücksichtigung**: Siehe Kapitel 4.2.4
- **CombatSituation: Flucht**: Müssen wir die Flucht ergreifen, bewegen wir unsere Ameise auf das nächste sicherste Tile.
- **Abbruch Mission**: Falls eine Ameise auf einer andere Mission als die GatherFoodMission ist und ein FoodTile in seiner Nähe antrifft, wird abgewogen ob die Mission zu Gunsten von Futter sammeln abgebrochen werden soll. Dabei ist ein Entscheidungsfaktor auch die 'Sicherheit' des Futters. Falls das Futter nicht auf einem sicheren Weg geholt werden kann, wird die Mission nicht abgebrochen.

Natürlich könnte man die InfluenceMap auch für weitere Entscheidungen verwenden. Auch der Einsatz von Spannungskarte (Tension Map), welche auch auf den Werten InfluenceMap aufbaut, wäre denkbar. Dies wurde während dieser Arbeit nicht angeschaut bzw. implementiert.

5.2 Combat Situations

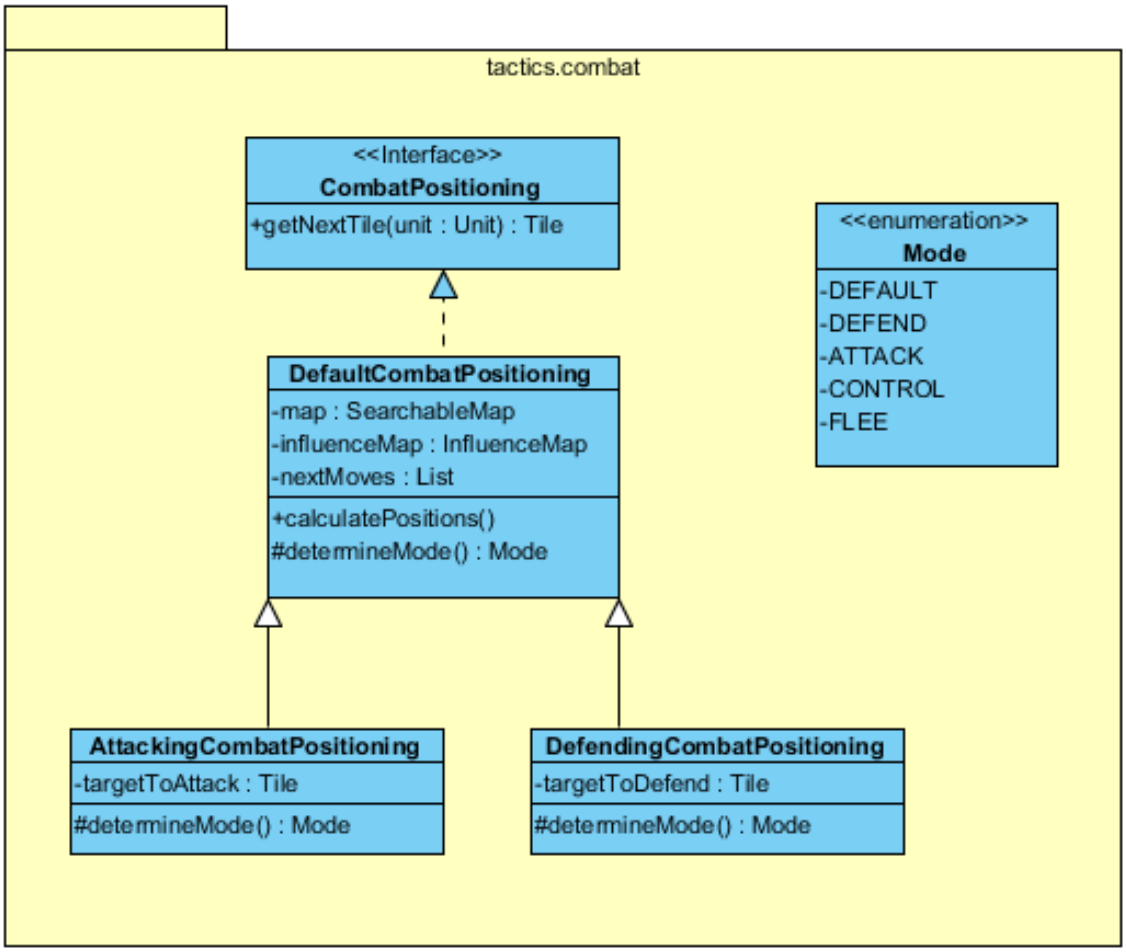

Abbildung 5.3: CombatPositioning Klassendiagramm

Kampfsituationen treten immer dann auf wenn gegnerische Ameisen auf unsere Ameisen treffen. Zum Beispiel wenn ein gegnerischer Hügel angegriffen wird, oder der eigene Hügel verteidigt werden muss. Eine Kampfsituation kann sich aber auch sonst wo auf der Karte ereignen.

5.2.1 DefaultCombatPositioning

DefaultCombatPositioning implementiert das Interface CombatPositioning und führt die Positionierung für die drei Verhalten FLEE, DEFEND, ATTACK an. Das Verhalten wird in der Methode `determineMode(...)` wie folgt bestimmt, wobei das 'DEFAULT' Verhalten dem ATTACK-Verhalten entspricht. (Das Verhalten 'CONTROL' wurde nicht implementiert.

```
protected Mode determineMode() {
    final boolean enemyIsSuperior = enemyUnits.size() > myUnits.size();
    if (enemyIsSuperior)
        return Mode.FLEE;
    return Mode.DEFAULT;
}
```

Wenn nicht ein DefaultCombatPositioning initialisiert wird, sondern ein AttackingCombatPositioning (in der AttackHillMission) oder ein DefendingCombatPositioning (in der DefendHillMission) so wird das Verhalten anders bestimmt, indem die `determineMode(...)` Methode überschrieben ist.

DefendingCombatPositioning

Hier wird immer das 'DEFEND' Verhalten ausgewählt, aus dem Grund, dass die Verteidiger sich nicht zu weit vom Hügel entfernen in dem sie den Gegner attackieren und der Hügel von einer anderen Seite eingenommen wird. In Unterzahl sollen die Verteidiger auch nicht flüchten, sondern trotzdem Versuchen den Hügel zu verteidigen. Hier könnte man sich Gedanken machen ob die Verteidiger, falls in Überzahl, die Gegner nicht bis zu einem gewissen Punkt angreifen sollen. Dieser Möglichkeit sind wir aber nicht weiter nach gegangen.

```
protected Mode determineMode() {
    return Mode.DEFEND;
}
```

AttackingCombatPositioning

Die Bestimmung des Verhalten unserer Angreifern in der AttackingCombatPositioning-Klasse ist da schon komplexer. Mittels Breitensuche werden die Gegner zwischen uns und dem Ziel ermittelt. Falls der Gegner in Unterzahl ist wird angegriffen, indem der Modus auf ATTACK gesetzt wird. Andernfalls greift die `determineMode(...)` des DefaultCombatPositioning.

TODO DISCUSS

```
@Override
protected Mode determineMode() {
    Tile clusterCenter = map.getClusterCenter(myUnits);
    BreadthFirstSearch bfs = new BreadthFirstSearch(map);
    int distanceToTarget = map.getSquaredDistance(clusterCenter, target);
    List<Tile> enemiesGuardingTarget = bfs.floodFill(target, distanceToTarget, new
        GoalTest() {
        @Override
        public boolean isGoal(Tile tile) {
            return enemyUnits.contains(tile);
        }
    });
    if ((enemiesGuardingTarget.size() * 2) <= myUnits.size())
        return Mode.ATTACK;

    // fall back to default
    return super.determineMode();
}
```

Die bereits erwähnten Verhalten, nehmen folgende Positionierung der Ameisen vor.

5.2.1.1 FLEE

Für jede Unit wird die sicherste Nachbarzelle mittels InfluenceMap bestimmt. Die Unit verschieb sich auf die sicherste Nachbarzelle, welche logischerweise vom Gegner entfernt liegt.

```
for (Tile myUnit : myUnits) {
    nextMoves.put(myUnit, map.getSafestNeighbour(myUnit, influenceMap));
}
```

5.2.1.2 DEFEND

Der Modus DEFEND wird verwendet um ein Ort, bei uns unsere eigenen Hügel, zu verteidigen. Der Algorithmus definiert sich wie folgt.

```
private void defendTarget() {
    // if no opponents are around, just position ourselves in the diagonals
    if (enemyUnits.isEmpty()) {
        move enemies to the diagonals of of the defend tile
    } else {
        // some sides mustn't be defend because they are surrounded by water
        calculate sides to defend

        foreach(side in sides to defend) {
                calculate attackers on this side
                calculate clustercenter of  enemy
                calculate defenders for this side
            calculate defend positions
            positioning of the defenders
        }
    }
}
```

Listing 5.1: Algorithmus zur Verteidigung

Das ClusterCenter des Gegners ist jeweils der Schwerpunkt aller Einheiten des Gegners und wir, wie später erklärt, für die Positionierung der eigenen Ameisen verwendet. Das Berechnen der Verteidiger je Seite beanspruchte im Code ein bisschen mehr als eine Zeile: Alle Ameisen die bereits in einer Richtung sind von welcher angegriffen wird, werden dieser Verteidigungsrichtung zugewiesen. Die restlichen Ameisen werden der Richtung zugewiesen von wo sich am meisten Angreifer nähern. So sind nun Verteidiger und Angreifer je Seite ausgemacht und es folgt die Positionierung.
Das ClusterCenter des Gegners dient als Kreismittelpunkt (siehe schwarzer Kreis in Abb. 5.4). Der Radius ergibt sich zwischen dem ClusterCenter der eigenen Verteidiger und dem ClusterCenter des Gegners. Auf diesem Kreisumfang werden, mit Start bei unserem ClusterCenter, mittels Breitensuche so viele Tiles zur Positionierung gesucht wie Verteidiger zur Verfügung stehen. Danach werden die Verteidiger auf den gefundenen Tiles positioniert.

Abbildung 5.4: DefaultCombatPositioning: Berechnung der Tiles für die Positionierung

Abbildung 5.4 zeigt wie die Tiles zur Positionierung für die südliche und die nördliche Verteidigung berechnet werden. Die schwarzen Kreise siginfieren die ClusterCenter der Kontrahenten. Die gefundenen Positionierungstiles für die nördliche Verteidigung liegen auf dem grauen Kreis und werden in der Abbildung hellorangen dargestellt. Zu sehen ist auch, dass auf dem eigenen Hügel kein Positionierung möglich ist, hier sollen neue Ameisen schlüpfen können.

Die Idee dieser Positionierung ist, dass so alle eigene Ameisen gleich weit vom Zentrum des Gegners entfernt sind. Sobald der Gegner vorrückt muss er gegen alle Ameisen gleichzeitig kämpfen. Nachteil dieser Positionierung ist, dass wenn der Gegner nicht kompakt angreift und nicht eine gleichförmige Angriffsformation hat, richten wir uns nur nach dem ClusterCenter aus, die einzelnen Gruppierungen der gegnerischen Ameisen werden nicht berücksichtigt, was ein Nachteil bei der Verteidigung zur Folge hat.
Wir haben uns überlegt, dass eine Positionierung mit Hilfe der InfluenceMap diesen Nachteil beheben würde. Man könnte die neutralen Tiles die sich zwischen den Kontrahenten befindet mittels InfluenceMap herausfinden und als Front anschauen. Nach dieser Frontlinie würden wir uns dann unsere Einheiten ausrichten Leider fehlte uns hier die Zeit dies nachträglich noch zu programmieren und auszuprobieren.

5.2.1.3 ATTACK

```
private void attackTarget() {
        find all enemies in target direction
        if(no enemies around){
            move all unit in direction of the target
        }else{
             calculate clustercenter of  enemy
             calcuate attack positions
             if(more than x friendly unit are already on the formation tiles)
                     calcuate formation tiles nearer to the enemy
             positioning of the attackers
        }
}
```

Listing 5.2: Algorithmus zur Berechnung der Angriffsformation

Im Verhalten ATTACK werden die Tiles zur Positionierung genau gleich berechnet wie vorhin im DEFEND-Modus. Nach der Berechnung wird geschaut wie viele unserer Angreifer schon auf diesen Tiles sind. Falls sich ein definierbarer Anteil der Ameisen bereits auf den Tiles befindet, gehen wir davon aus das sich unsere Ameisen formiert haben und wird der Radius verkürzt. Durch den verkürzten Radius werden Tiles berechnet die um eine Zelle näher beim Gegner sind. Danach folgt die Positionierung (Vorrücken) der Ameisen auf die berechneten Tiles.

6 Ants

Alle Klassen im Ants Package sind spezifisch für die Ants AI Challenge und repräsentieren Teile des Spiels. Nachfolgend wir deren Aufbau erläutert.

6.1 State-Klassen

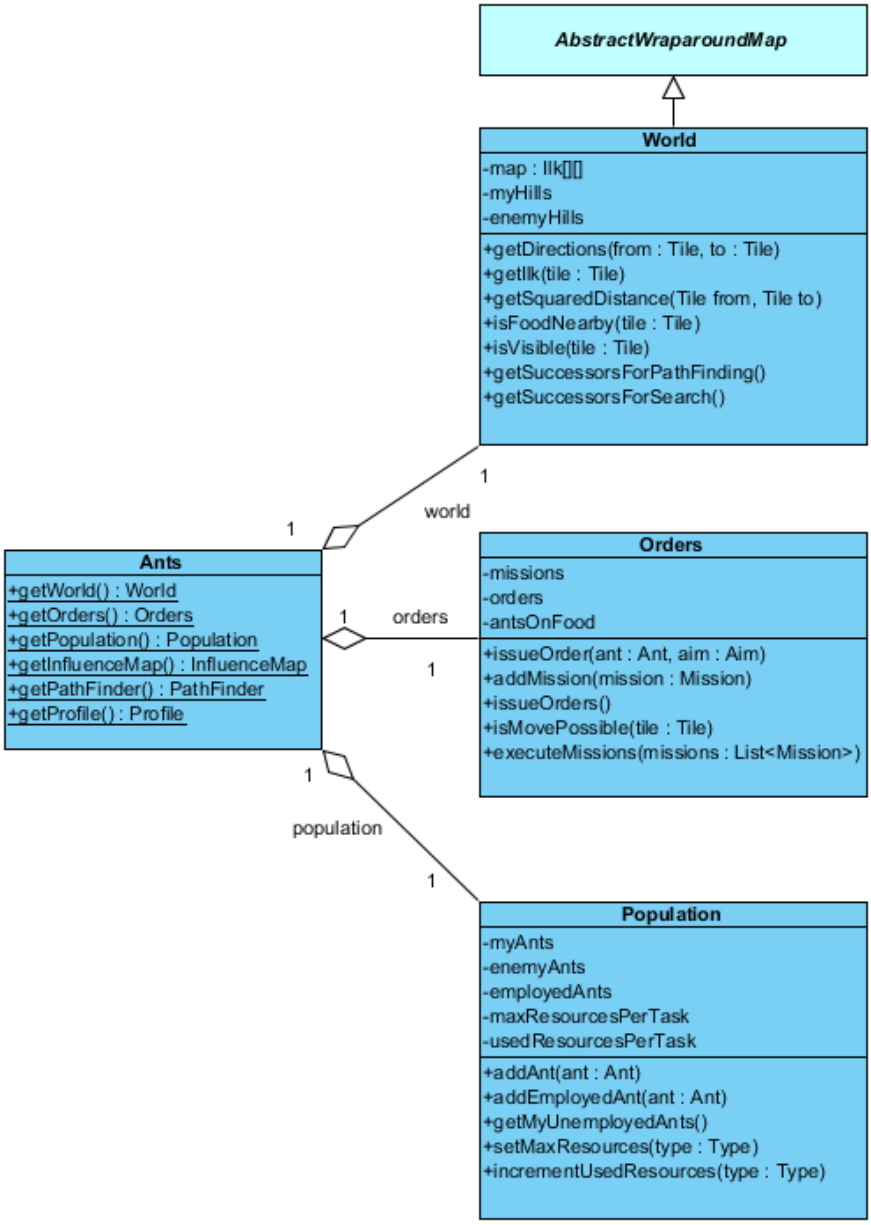

Abbildung 6.1: State-Klassen (vereinfacht)

Abbildung 6.1 zeigt eine Übersicht über der Spielzustands-Klassen. Für das Diagramm wurden lediglich die wichtigsten Methoden und Attribute berücksichtigt. Die State-Klassen implementieren alle das Singleton-Pattern.

6.1.1 Ants

Die Ants Klasse ist die zentrale State-Klasse. Sie bietet auch einfachen Zugriff auf die anderen State-Klassen. Ursprünglich hatten wir alle Methoden, die mit dem Zugriff auf den Spielzustand zu tun hatten, direkt in der Ants Klasse implementiert, haben aber schnell gemerkt, dass das unhandlich wird. Die Ants Klasse dient jetzt vor allem als Container für die anderen State-Klassen und implementiert nur noch einige Methoden, die Zustandsänderungen in verschiedenen Bereichen vornehmen.

6.1.2 World

Die World Klasse enthält Informationen zur Spielwelt und implementiert die AbstractWrapAroundMap aus der AI-Tools API. Hier wird die Karte abgespeichert, in der für jede Zelle die aktuell bekannten Informationen festgehalten werden. Das beinhaltet die Sichtbarkeit der Zelle und was die Zelle aktuell enthält (Ameise, Nahrung, Wasser, ...). Ausserdem werden Listen geführt, wo sich die eigenen und die bekannten gegnerischen Hügel befinden. Die Klasse bietet Methoden zur Distanzberechnung und gibt Auskunft über einzelne Zellen, beispielsweise ob sich Nahrung in der Umgebung einer bestimmten Zelle befindet.

6.1.3 Orders

In der Orders Klasse wird über Befehle und Missionen der einzelnen Ameisen Buch geführt. In der Liste der Befehle wird zwischengespeichert welche Ameise welche Bewegung im aktuellen Spielzug macht. Zu Beginn des Spielzuges wird diese geleert, dann von den Tasks und Missionen mit Befehlen belegt, und am Schluss des Spielzuges werden die Befehle der Spielschnittstelle übergeben. Die Liste der Missionen ist zugübergreifend geführt, da eine Mission über mehrere Spielzüge verlaufen kann. Das zentrale Verwalten der Befehle und Missionen dient dazu, sicherzustellen, dass keine widersprüchlichen Befehle ausgegeben werden wie: Mehrere Befehle für eine Ameise, gleiche Ziel-Koordinaten für mehrere Ameisen, eine Ameise ist mehreren Missionen zugeteilt etc..

6.1.4 Population

Die Population Klasse dient der Verwaltung der eigenen und der gegnerischen Ameisen-Völker. Hier werden die Ameisen mit ihren aktuellen Aufenthaltsorten festgehalten. Wenn für eine Ameise ein Befehl ausgegeben wird, wird die Ameise als beschäftigt markiert. Über die Methode `getMyUnemployedAnts()` kann jederzeit eine Liste der Ameisen abgefragt werden, die für den aktuellen Zug noch keine Befehle erhalten haben und für neue Aufgaben zur Verfügung stehen.

6.1.5 Clustering

Die Clustering Klasse dient dem Aufteilen des Spielfeldes in Clusters für die HPA*-Suche (s. Abschnitt 4.2.3). Hier werden die berechneten Clusters abgelegt, damit diese nicht bei jeder Verwendung neu berechnet werden müssen.

6.2 Spiel-Elemente (Ants-Spezifisch)

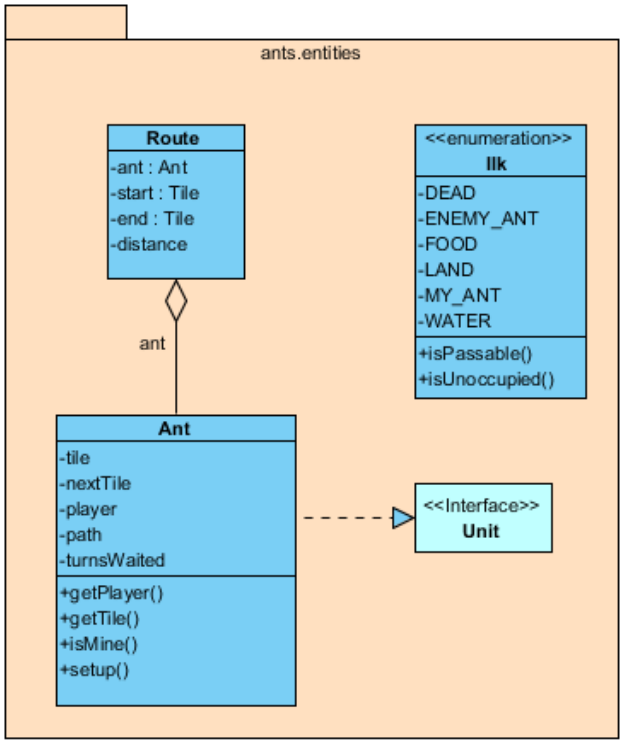

Abbildung 6.2: Ants-spezifische Elemente der Spielwelt (vereinfacht)

Abbildung 6.2 zeigt die wichtigsten Klassen, die die Elemente des Spiels repräsentieren. Der Übersichtlichkeit wegen wurden nur die wichtigsten Attribute und Operationen in das Diagramm aufgenommen.

6.2.1 Ant

Eine Ant (Ameise) gehört immer zu einem Spieler; über die Methode `isMine()` können unsere eigenen Ameisen identifiziert werden. Eine Ameise weiss jeweils in welcher Zelle sie steht. Das Feld `nextTile` dient der Verfolgung einer Ameise über mehrere Züge. Der Wert des Feld wird jeweils gesetzt, wenn der Ameise ein nächster Zug zugewiesen wird. Im nächsten Spielzug wir die Position der Ameise durch diese Information aktualisiert.

6.2.2 Route

Eine Route repräsentiert eine einfache Verbindung zwischen zwei Zellen. Die Distanz wird zwischen den Zellen wird mit der Luftliniendistanz gemessen.

6.2.3 Ilk

Ilk ist der Typ einer Zelle. Der Ilk einer Tile-Instanz gibt an, was sich gerade in der Zelle befindet. Dies kann ein Gelände-Typ sein, wenn die Zelle nicht besetzt ist, oder es kann eine Ameise, Nahrung, oder ein Hügel sein. Die Ilk-Enumeration bietet Hilfsmethoden, um festzustellen, ob eine Zelle passierbar oder besetzt ist.

```
public enum Ilk {WATER, FOOD, LAND, DEAD, MY_ANT, ENEMY_ANT };
```

Listing 6.1: Werte der Ilk-Enumeration

6.3 Aufbau Bot

Nun folgt der mehrstufige Aufbau des Bots, den wir geschrieben haben.

6.3.1 Klasse Bot

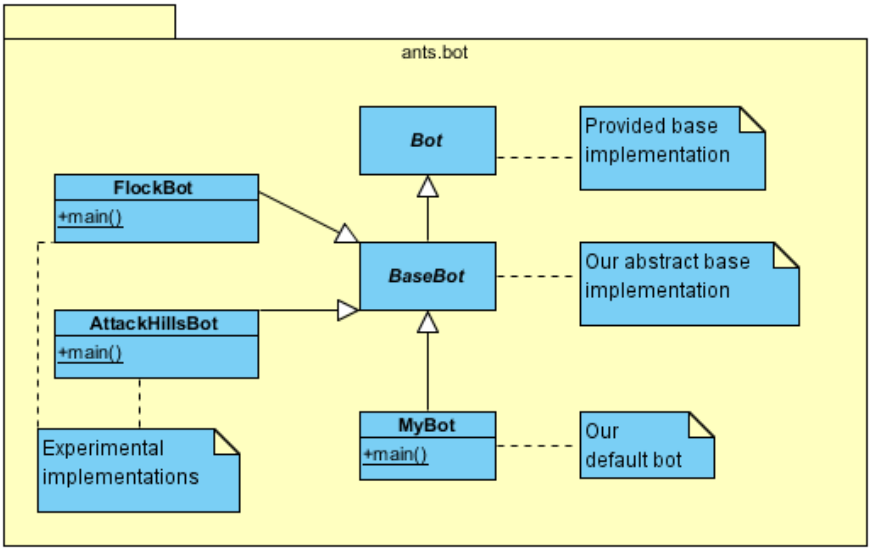

Abbildung 6.3: Vererbung der Bots wobei auf Stufe impl (Implementation) nur MyBot verwendet wird.

Als Basis für unsere Bot Implementation haben wir den Beispiel-Bot (Klasse Bot.java) verwendet, der im Java-Starter-Package enthalten ist. Das Starter-Package kann von der AI-Challenge Website[1] heruntergeladen werden. Der Bot implementiert die Klassen AbstractSystemInputReader und AbstractSystemInputParser, welche die Interaktion mit der Spielengine über die System-Input/Output Streams kapseln. Für eine optimierte Lösung könnte der Bot auch angepasst werden, indem er selber auf die Streams zugreift. Im Rahmen dieser Arbeit erschien uns das nicht nötig. Die Klasse Bot dient als Grundlage für die Klasse BaseBot, welcher wiederum Grundlage ist für die Finale Klasse MyBot. (Siehe Vererbung Abb. 6.3)

6.3.2 BaseBot

Die abstrakte Klasse BaseBot erbt vom Bot. Hier haben wird die Struktur unseres Spielzuges definiert.

[1] http://aichallenge.org/ants_tutorial.php

6.3.3 Ablauf eines Zugs

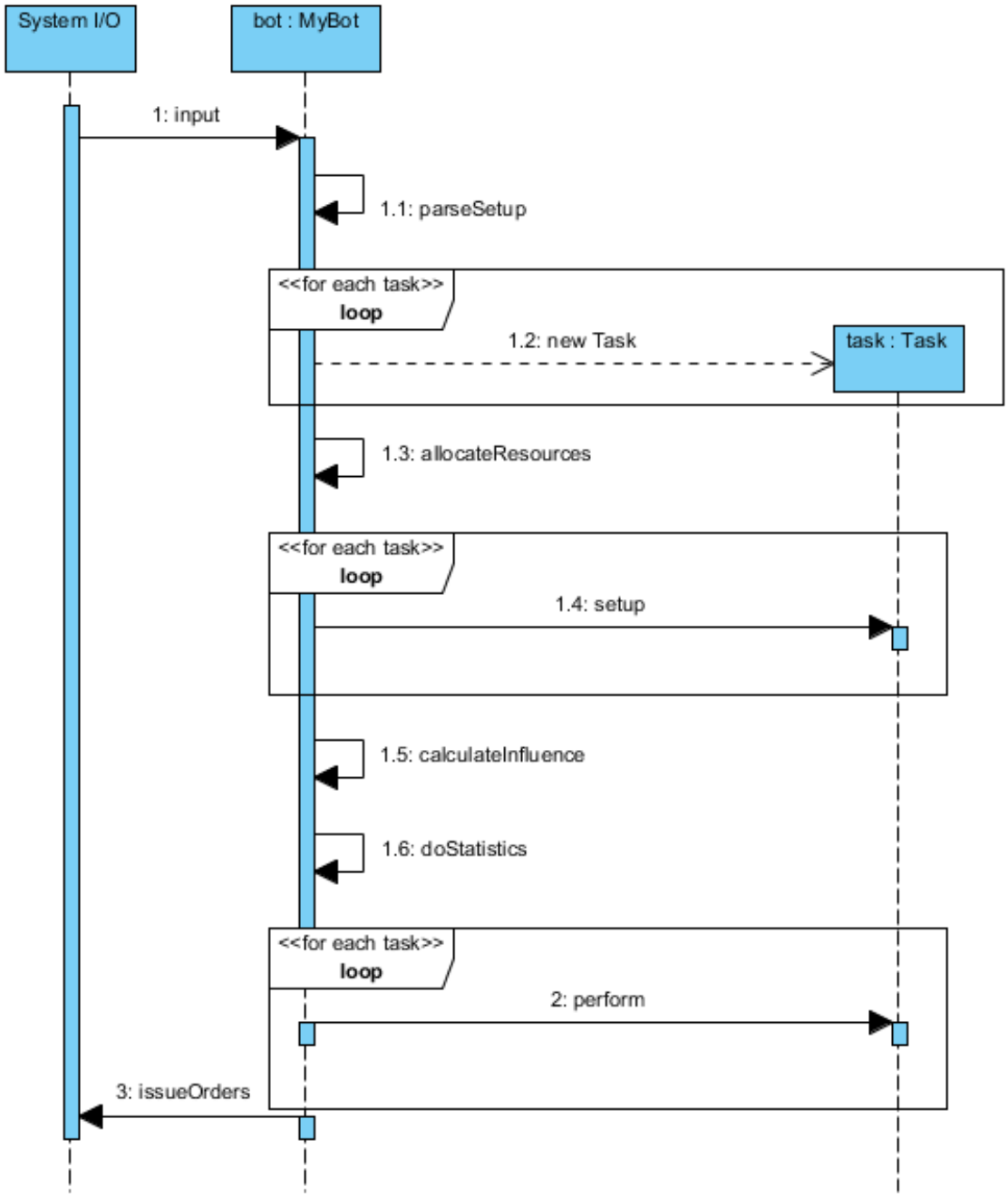

Abbildung 6.4: Ablauf des ersten Zugs des Spiels

Abbildung 6.4 zeigt den Ablauf des ersten Zugs, während Abbildung 6.5 den Ablauf aller weiteren Züge zeigt.

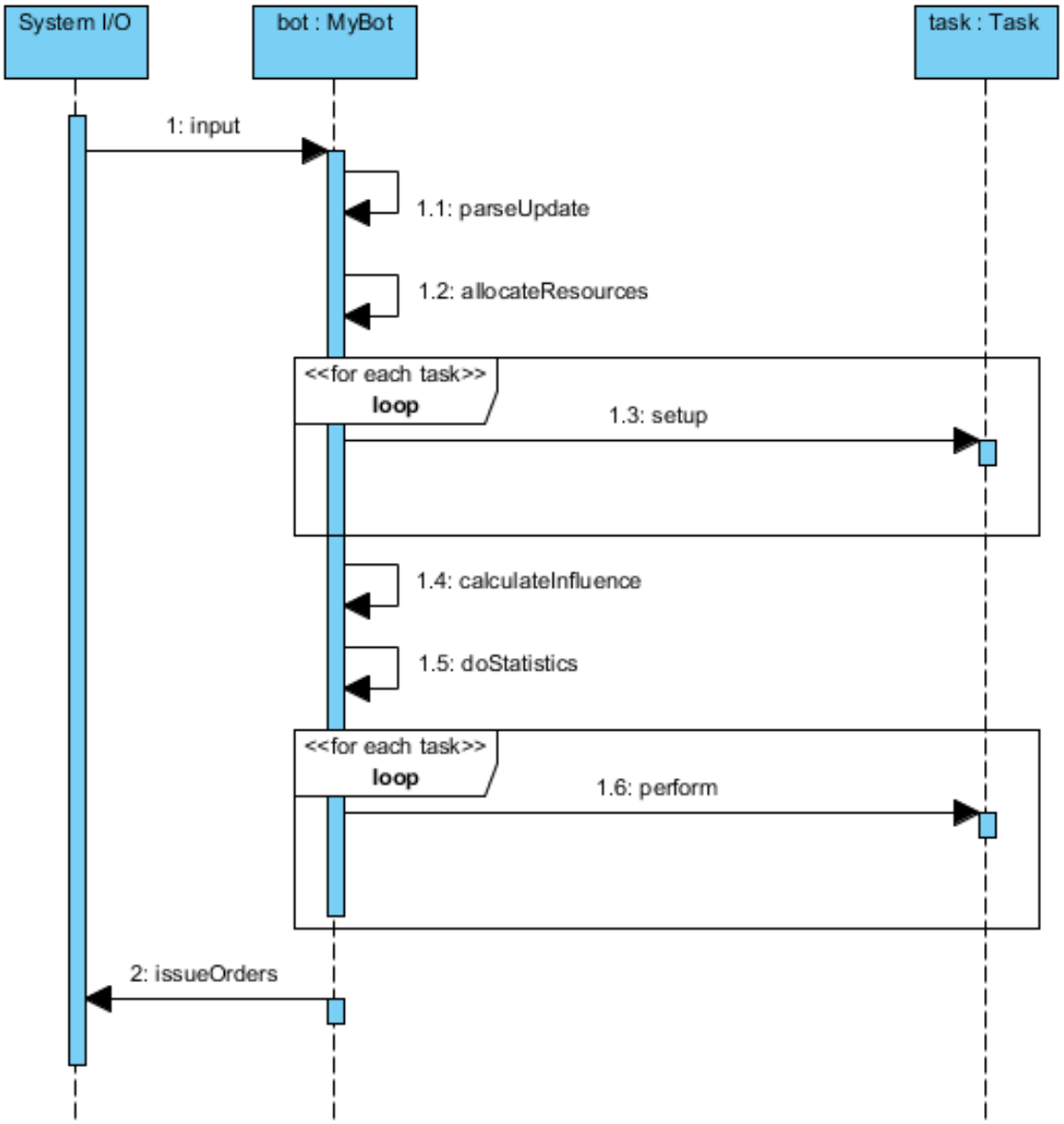

Abbildung 6.5: Ablauf der weiteren Züge des Spiels

Jeder Zug beginnt mit dem Einlesen des Inputs vom SystemInputStream. Wenn der Bot das Signal "READY" (1. Zug) oder "GO" (alle weiteren Züge) erhält, kann er den gesammelten Input verarbeiten (Methode `parseSetup()` respektiv `parseUpdate()`). Danach wird die eigentliche Logik des Bots in der Methode `doTurn(...)` ausgeführt.

Im 1. Zug werden dabei Instanzen der Tasks erstellt. Abgesehen davon unterscheidet sich der 1. Zug von diesem Punkt an nicht mehr von allen nachfolgenden Zügen. Die Tasks werden vorbereitet. (Aufruf der jeweiligen `setup()` Methode. Danach werden einige statistische Werte aktualisiert und in jedem 10. Zug auch geloggt. Dann werden die Tasks in der definierten Reihenfolge aufgerufen. Hier wird der Löwenanteil der Zeit verbracht, denn die Tasks enthalten die eigentliche Logik unserer Ameisen.

Zum Schluss werden dann mit `issueOrders()` die Züge der Ameisen über den SystemOutputStream an die Spielengine übergeben. Im Code sieht das ganze folgendermassen aus.

```
\label{lst:ablaufspielzug}
@Override
/*
 * This is the main loop of the bot. All the actual work is
 * done in the tasks that are executed in the order they are defined.
```

```
 */
public void doTurn() {
        // write current turn number, ants amount into the log file.
    addTurnSummaryToLogfiles();
        // new calculation of the influence map
    calculateInfluence();
        // write some statistics about our population
    doStatistics();
        // initialize the task (abstract method) must be implemented by the
            inherited class
    initTasks();
        // execute all task (main work to do here)
    executeTask();
        // write all orders to the output stream
    Ants.getOrders().issueOrders();
        // log all ants which didn't get a job.
    logUnemployedAnts();
}
```

Listing 6.2: Der Ablauf des Spielzuges

6.3.4 MyBot

Wie bereits im Listining **??** sichtbar, ist die Methode `initTasks()` in BaseBot abstrakt und muss von MyBot implementiert werden. `initTasks()` definiert welche Tasks, oder besser gesagt Fähigkeiten der Bot hat. Dies wurde ausgelagert, da nicht nur MyBot von BaseBot erbt, sondern auch weiter Bots die wir zu Testzwecken erstellt haben um nur gewisse Funktionalitäten zu testen. (Siehe dazu 8.3) Weiter wird in MyBot `initLogging(...)` aufgerufen. Hier definieren wir welche Logkategorien mit welchem Loglevel ins Logfile geschrieben werden. (Mehr zum Thema Logging, siehe Kapitel 7) Je nach Modul das gerade getestet wird kann die Anzahl Logeinträge justiert werden.
MyBot initialisiert folgende Tasks; es sind die Tasks die sich während der Arbeit bewährt haben. Die detaillierte Beschreibung der Tasks ist im nachfolgenden Kapitel 6.4 zu finden.

- GatherFoodTask
- AttackHillsTask
- DefendHillTask
- ExploreTask
- ClearHillTask
- CombatTask
- ClusteringTask

6.4 Tasks

Zu Beginn des Projekts haben wir die wichtigsten Aufgaben der Ameisen identifiziert. Diese Aufgaben wurden als Tasks in eigenen Klassen implementiert. Die Abbildung zeigt wie die Tasks aufgebaut sind.

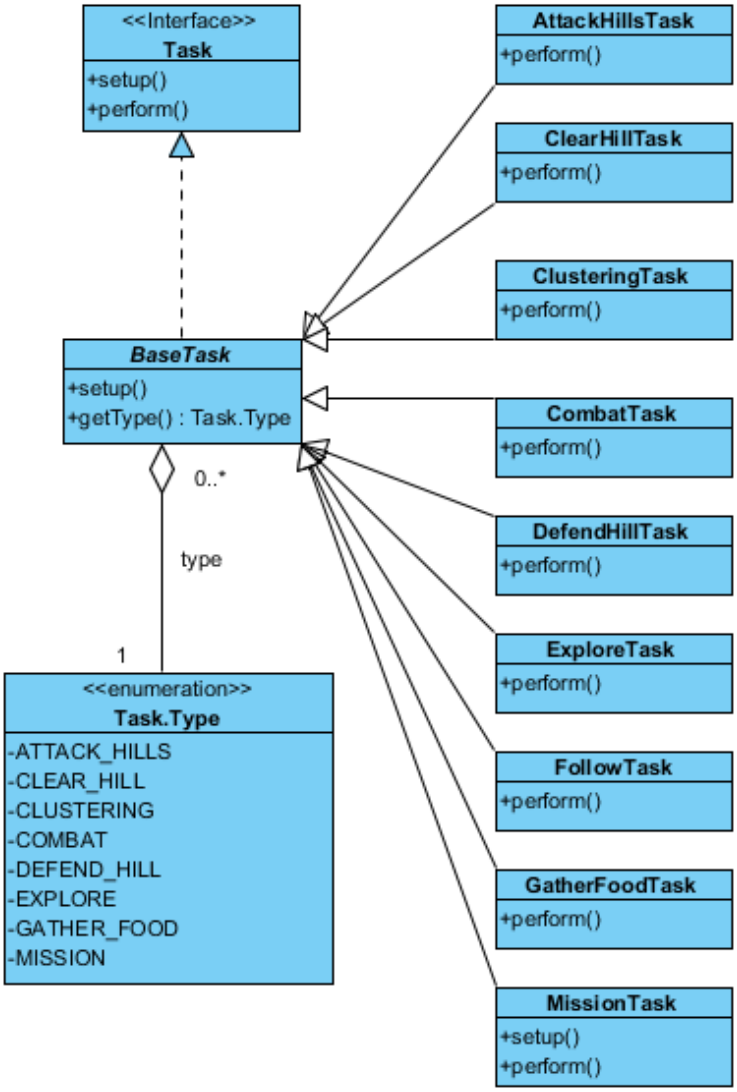

Abbildung 6.6: Tasks

Das Interface Task[2] definiert eine `setup()`-Methode welche den Task initiiert, sowie eine `perform()`-Methode welche den Task ausführt. Im Programm werden die Tasks nach deren Wichtigkeit ausgeführt, was auch der nachfolgenden Reihenfolge entspricht. Jedem Task stehen nur die unbeschäftigten Ameisen zur Verfügung, d.h. jene welchen noch keine Aufgabe zugeteilt wurde.Die zentrale Aufgabe der Tasks ist es den Ameisen Aufträge zu verteilen. Für Aufträge die länger als ein Spielzug dauern erstellt der Task eine entsprechende Mission. (siehe Abschnitt 6.5)

6.4.1 MissionTask

6.4.1.1 setup()

Dieser Task führt alle bestehenden Missionen aus. In der `setup()`-Methode des Tasks werden die bestehenden Missionen initialisiert indem die `setup()`-Methoden der vorhandenen Missionen aufgerufen werden. Diese löscht alle Ameisen aus der Mission welche den letzten Zug nicht überlebt haben. Die überlebenden Ameisen werden als 'beschäftigt' markiert, damit Sie nicht von anderen Missionen verwendet werden. Zudem werden die Ameisen, abhängig vom letzten Zug, auf die neue Spielfeldzelle gesetzt.

[2]Das Interface ist im Code unter ants.tasks.Task.Java auffindbar.

6.4.1.2 execute()

Nun, da alle Missionen inklusive Ameisen initialisiert wurde, werden diese ausgeführt. Für die Ausführungen haben wir die Missionen nach Typ in der Reihenfolge EXPLORE, COMBAT, DEFENDHILL, GATHERFOOD, ATTACKHILLS sortiert. Zuerst sollen die Missionen EXPLORE und COMBAT ausgeführt werden, aus dem Grund, dass diese Missionen keine neue Ameisen verlangen. Sie kommen mit den Ameisen aus die sie zu Beginn der Mission zugeteilt bekamen. Falls nun das Missionsziel von EXPLORE oder COMBAT erreicht wurde (`mission.isComplete()`), die Mission nicht mehr gültig ist (`!mission.isVaild()`) oder keine Züge für die Mission errechnet werden konnten (`mission.isAbandoned()`), so wird die Mission aufgelöst und die Ameisen werden für die nachfolgenden Missionen DEFENDHILL, GATHERFOOD, ATTACKHILLS freigegeben.

Die Missionen DEFENDHILL, GATHERFOOD, ATTACKHILLS sind auch überlegt angeordnet. Falls die DEFENDHILL-Missionen Verstärkung bei der Verteidigung brauchen, sind sie darauf angewiesen, dass sie Ameisen rekrutieren können die möglichst schnell zum Hügel gelangen. Es wäre ungünstig, wenn die nahe gelegenen Ameisen einer ATTACKHILLS- oder einer GATHERFOOD-Mission beitreten und so die DEFENDHILL-Mission Ameisen von weiter weg herbeirufen müsste. GATHERFOOD wird vor ATTACKHILLS aufgerufen, da Ameisen die nahe bei einer Futterzelle sind das Futter einsammeln sollen und nicht, oder erst später, am Angriff teilnehmen.

Je nach Spielsituation oder Profil des Bots, könnte man in Erwägung ziehen, die Reihenfolge vor dem Spiel oder dynamisch zu ändern. Dies wurde aber nicht weiter betrachtet.

6.4.2 GatherFoodTask

Im ersten Zug wird genau eine GatherFoodMission durch den GatherFoodTask erstellt. Diese Mission koordiniert die Futtersuche und verwaltet die Ameisen, welche Futter einsammeln.

6.4.3 DefendHillTask

Für jeden eigenen Hügel wird eine Verteidigungsmission erstellt. Ab wann und wie viele Ameisen zur Verteidigung eingesetzt werden, ist in der Mission konfiguriert.

6.4.4 AttackHillsTask

Sobald gegnerische Ameisenhaufen sichtbar sind, sollen diese angegriffen werden. Je gegnerischen Ameisenhaufen wird eine AttackHillMission erstellt. Die Mission ist für das Rekrutieren der Ameisen zuständig, die Mission kann also zu Beginn ohne Ameisen erstellt werden.

6.4.5 CombatTask

Dieser Task macht was?

6.4.6 ExploreTask

Für alle noch unbeschäftigten Ameisen wird mittels ManhattanDistance der nächste Ort gesucht der unerforscht ist. Mittels Pfadsuchalgorithmus wir der Pfad dorthin berechnet und daraus eine ExploreMission erstellt. Die Ameise wird den Pfad in den nächsten Spielzügen ablaufen. Sobald die ganze Spielkarte erforscht ist, schickt der Task die Ameisen in Gebiete die momentan nicht sichtbar sind. (Fog of War)

6.4.7 ClearHillTask

Dieser Task bewegt alle Ameisen, welche neu aus unserem Hügel 'schlüpfen', vom Hügel weg. So werden nachfolgende Ameisen nicht blockiert. Es wird keine Mission erstellt, der Task bewegt die Ameise nur eine Zelle vom Hügel weg.

6.4.8 ClusteringTask

Der ClusteringTask wird als Vorbereitung für den HPA* Algorithmus verwendet. Hier wird für alle sichtbaren Kartenregionen das Clustering vorgenommen. HPA* und das Clustering wird im Kapitel 4.2.3 im Detail beschreiben.

6.4.9 Verworfene und nicht verwendete Tasks

Leider waren nicht alle Task-Implementation von Nutzen. Sie halfen nicht den Bot zu verbessern. Hier eine Auflistung welche Tasks wir verworfen haben.

- **FollowTask:** Der FollowTask ist für Ameisen angedacht welche aktuell keine Aufgabe haben. Diese Ameisen sollen einer nahegelegenen, beschäftigten Ameise folgen. Er wurde verworfen, da wir das Problem der unbeschäftigten Ameisen minimieren konnten.
- **SwarmTask:** Hier war die Idee, dass bei einem Angriff auf einen gegnerischen Hügel die Ameisen sich auf Weghälfte zu einem Schwarm vereinen und danach den Hügel angreifen. Leider war diese Idee erfolglos, da zu viel Zeit verstrich bis sich die Ameisen gesammelt haben. Zudem war der Sammelpunkt manchmal ungünstig gelegen. (Schwer erreichbar oder in gegnerischen Gebiet).
- **FlockTask:** Geordnete Fortbewegung der Ameisen, verworfen bzw. in CombatSituation übernommen.
- **ConcentrateTask:** Ameisen an einem bestimmten Ort auf der Karte sammeln. Der Task wird beim aktuellsten Bot nicht verwendet.

6.5 Missionen

Missionen sind das Herzstück unserer Arbeit. Eine Mission dauert über mehrere Spielzüge und berechnet für jede teilnehmende Ameise welches ihre nächste Bewegung ist. Nachfolgende Darstellung zeigt, dass alle Missionen von der abstrakten BaseMission abstammen und die BaseMission das Interface Mission implementiert. Die Lebensdauer einer Mission hängt davon ab, ob sie ihr Ziel erreicht oder ob sie schon früher abgebrochen wird. Ziel und Abbruchbedingungen sind je nach Mission unterschiedlich und werden im jeweiligen Abschnitt erklärt.

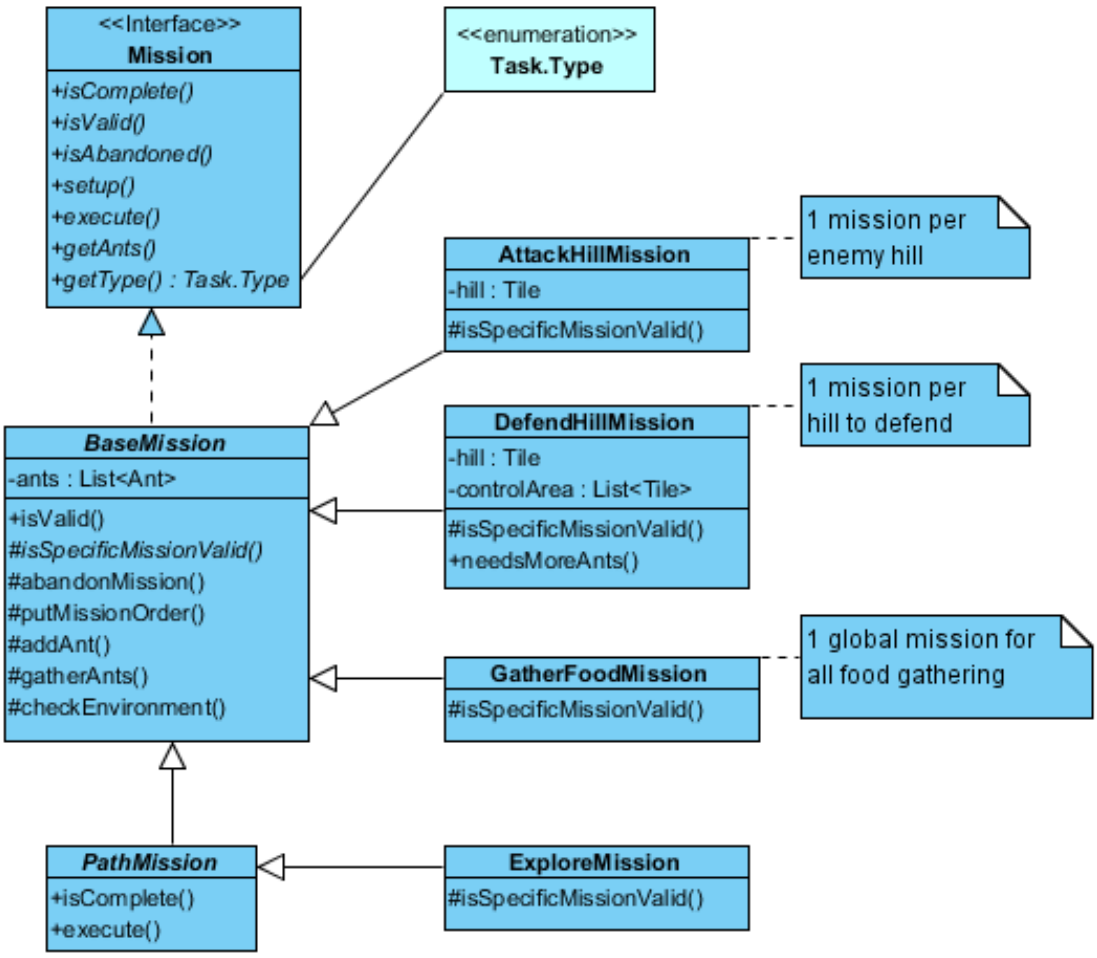

Abbildung 6.7: Missionen und ihre Hierarchie

6.5.1 BaseMission

Diese Klasse hat zwei Verwendungszwecke, erstens implementiert sie die von Interface Mission vorgegebene Methoden und zweitens stellt sie Funktionen zur Verfügung die von den spezifischen Missionen verwendet werden. Die wichtigsten Funktionalitäten sind hier mit Erklärung aufgelistet.

- **abandonMission():** Abbrechen der Mission
- **addAnt():** Ameise der Mission hinzufügen, in der Population als beschäftigt markieren.
- **doAnyMove(Ant a):** Für die mitgegebene Ameise irgendein Zug in eine der vier Richtung bestimmen.
- **doMoveInDirection(Ant ant, Tile target):** Ein Zug in eine bestimmte Richtung bestimmen.
- **gatherAnts(...):** Ameisen für die Mission rekrutieren. Die Anzahl Ameisen wird als Parameter mitgegeben.
- **moveToNextTileOnPath(Ant a):** Bewegt die Ameise ein Pfadstück weiter auf dem, der Ameisen zugewiesen, Pfad.
- **putMissionOrder(...):** Wurde ein Befehl für die Ameise gefunden, wird er der Klasse Orders (Verwaltung der Befehle) mitgeteilt.
- **releaseAnts(int amount):** Ameisen von der Mission entlassen.

- **checkEnviroment(...):** Mittels Breitensuche wird die Umgebung der Ameise nach eigenen Hügel, gegnerischen Hügel, gegnerischen Ameisen und Futterzellen gescannt. Je nach Mission wird beim Fund eines solchen Objekt die Mission abgebrochen, oder die Ameise von der Mission entlassen.

Nachfolgend werden die spezifischen Missionen erläutert. Tabellarisch werden die Eigenschaften der Missionen aufgelistet, danach folgen detaillierte Informationen zur Mission.

6.5.2 PathMission

Precondition	Der Pfad wurde vorgängig berechnet
Creator[a]	CombatTask, oder ExploreMission
Postcondition	Der Pfad ist vollständig abgelaufen
Max. Ants[b]	1
Max. Missionen	unbegrenzt
Valid[c]	siehe ExploreMission bzw. CombatTask
Gather Ants[d]	Nicht möglich
Release Ants[e]	Nicht möglich

[a]Ersteller der Mission
[b]Maximale Anzahl der Ameisen in der Mission
[c]Gültigkeit
[d]Ameisen für die Mission rekrutieren
[e]Ameisen von der Mission entlassen

Tabelle 6.1: Eigenschaften der PathMission

Die Pathmission ist eine abstrakte Klasse die von ExploreMission und als anonyme Klasse im CombatTask implementiert wird. Die einzige Funktionalität die angeboten wird ist, eine Ameise, die der Initialisierung mit Pfad mitgegeben wird, auf diesem definierten Pfad zu bewegen.

6.5.3 AttackHillMission

Precondition	Gegnerischer Hügel ist sichtbar
Creator	AttackHillsTask
Postcondition	Gegnerischer Hügel wurde erobert
Max. Ants	unbegrenzt
Max. Missionen	Je gegnerischen Hügel eine Mission
Valid	Solange der gegnerischer Hügel nicht zerstört ist.
Gather Ants	pro Zug max. fünf Ameisen, die im Umkreis von 25 Tiles des gegnerischen Hügel sind.
Release Ants	Sicheres Food-Tile in der Nähe, eigener Hügel in der Nähe der Unterstützung braucht oder wenn die Mission im Status ControlHill ist, dann werden alle Ameisen ausser zwei entlassen, diese kontrollieren den gegnerischen Hügel.

Tabelle 6.2: Eigenschaften der AttackHillMission

Diese Mission unterscheidet zwischen den drei verschiedenen Modi ControlEnemyHill,DestroyHill udn AttackEnemyHill. Der Modus wird zu Beginn jeder Runde durch die Methode determineState() definiert.

ControlEnemyHill
Der Modus ist aktiv wenn, zwei oder mehr eigene Ameise in der Nähe des gegnerischen Hügel sind. Der Gegner aber nur eine Ameise zur Verteidigung hat.
Der Status braucht nur zwei Ameisen, jene die am nächsten bei gegnerischen Hügel sind, die anderen werden entlassen. Die übrigen zwei Ameisen positionieren sich auf einer der vier diagonal gelegenen Zellen. Dank dieser Positionierung, werden alle gegnerischen Ameisen die neu aus dem Hügel schlüpfen sofort vernichtet. Der Gegner kann sich so nur durch Ameisen vermehren die aus einem anderen Hügel schlüpfen. (Die Ameisen schlüpfen zufällig aus einem Hügel.) Der Hügel wird erst zerstört wenn das Spiel endet, oder sich der Gegner mit anderen Ameisen dem Hügel nähert.

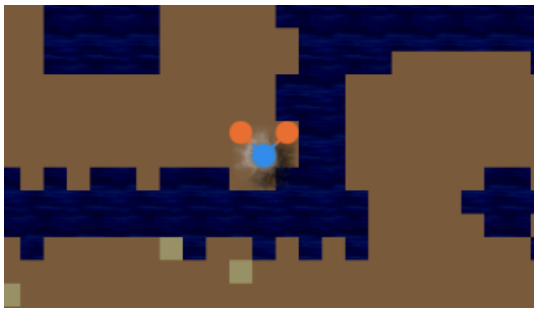

Abbildung 6.8: Ein gegnerischer Hügel wird durch unsere Ameisen (orange) kontrolliert.

DestroyHill
Der Modus ist aktiv wenn, das Spiel 5 Züge vor Spielende ist
Wenn dieser Modus eintrifft ist das Spiel fast zu Ende. Deshalb wird versucht mit der Brechstange den Hügel zu erobern indem die Ameisen auf dem kürzesten Weg und ohne Positionierung Richtung gegnerischen Hügel geschickt werden.

AttackEnemyHill
Der Modus ist aktiv in allen anderen Fällen
Alle Angreifer werden nach ihrer Position in lokalen Gruppen zusammen gefasst. Für diese Gruppe wird ein Pfad zum gegnerischen Hügel berechnet. Falls der Pfad länger ist als fünf Tiles wird ein Meilenstein (Milestone) definiert. Nun werden mittels Breitensuche die Gegner zwischen dem Meilenstein und der Gruppe ermittelt. Mit dieser Ausgangslage wird eine AttackingCombatPositioning-Klasse initialisiert. Als Parameter werden die Gruppe, die gegnerischen Ameisen sowie der Meilenstein mitgegeben. Die Klasse berechnet die nächsten Bewegungen der Gruppe. (Details siehe zur Berechnung 5.2)

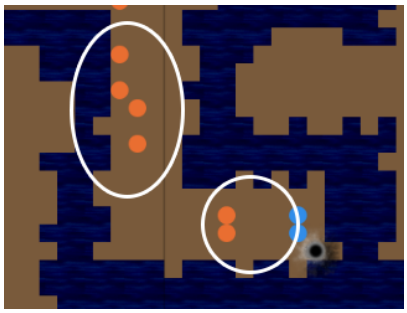

Abbildung 6.9: Die Zweiergruppe ist in Kampfstellung. Vier weitere Ameisen rücken zur Front auf.

6.5.4 DefendHillMission

Precondition	-
Creator	DefendHillTask
Postcondition	Eigener Hügel wurde zerstört
Max. Ants	unbegrenzt
Max. Missionen	Je eigener Hügel eine Mission
Valid	Solange der eigener Hügel nicht erobert ist.
Gather Ants	Mission soll immer mehr Verteidiger haben als Angreifer sich dem Hügel nähern.
Release Ants	Sind keine Angreifer in Sicht, werden die Ameisen (nicht alle) entlassen.

Tabelle 6.3: Eigenschaften der DefendHillMission

Beim Entwickeln dieser Mission haben wir uns folgende Fragen gestellt: Ab welchem Spielzug soll der Hügel bewacht werden? Wie viel Ameisen sollen für die Verteidigung eingesetzt werden? Was wenn der Hügel von mehreren Seiten angegriffen wird? Sollen alle Hügel bewacht werden? Aus diesen Frage ergab sich folgende Implementation.

Die DefendHillMission bietet zwei Arten zur Verteidigung an. Die Default-Verteidigung und die Barrier-Verteidigung. Bei der Default-Verteidigung werden die Ameise nahe um den eigenen Hügel zur Verteidigung aufgestellt. Die Barrier-Verteidigung ist nur bei Hügeln möglich die in einer Sackgasse sind, den in diesem Modus wird eine Sperre an der engsten Stelle erstellt. Nachfolgende Abbildung veranschaulicht die beiden Verteidigungsarten.

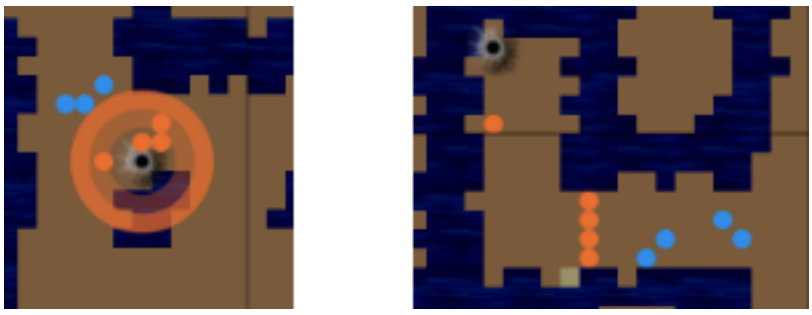

Abbildung 6.10: Links die Default-Verteidigung, rechts mit einer Barrier (Sperre).

Die Art der Verteidigung wird wie folgt definiert. Während dem zehnten Spielzug wird versucht mittels spezieller Breitensuche eine Sperre für den Hügel zu finden. Falls dies gelingt läuft die Mission wird die Sperre erstellt. Dies solange bis eine gegnerische Ameise auf die Höhe der Barriere tritt, dann gehen wir davon aus, dass die Sperre vom Gegner eingenommen wurde, wir wechseln die Verteidigungsart.

```
private void determineMode() {
  if (Ants.getAnts().getTurn() == 10) {
      AntsBreadthFirstSearch bfs = new AntsBreadthFirstSearch(Ants.getWorld());
      barrier = bfs.getBarrier(hill, Ants.getWorld().getViewRadius2(), 5);
      if (barrier != null) {
          mode = DefendMode.Barrier;
      }
  } else if (Ants.getAnts().getTurn() > 10 && mode == DefendMode.Barrier) {
    /* if we switched to barrier defend-mode we have to check if we aren't overrun.
        (enemy ant on barrier) if we are overrun, we switch back to the default
      mode. */
      for (Tile t : barrier.getBarrier()) {
          if (Ants.getWorld().getIlk(t) == Ilk.ENEMY_ANT) {
              mode = DefendMode.Default;
              break;
```

```
            }
        }
    }
```

Listing 6.3: Die Verteidigungsart wird bestimmt.

6.5.4.1 Default-Modus

Hier wird jegentlich berechnet, wie viele Ameisen den Hügel angreifen. Die Angreifer und die Verteidiger werden dem DefendingCombatPositioning übergeben, dieses berechnet die Positionen der Ameisen. Es ist einstellbar ab welchem Zug die DefendHillMission Ameisen für die Verteidigung rekrutiert. Zudem kann mit dem Parameter `DEFENDER_MORETHAN_ATTACKERS` definiert werden wie viele Ameisen mehr als die Anzahl Angreifer rekrutiert werden. Ameisen werden von der Mission entlassen wenn weniger Angreifer als Verteidiger, unter der Berücksichtigung des erwähnten Parameter, gezählt werden.

6.5.4.2 Barrier-Modus

Wie schon erwähnt, kann dieser Modus nur eintreffen wenn die Geländegegebenheiten stimmen. Ziel ist Engpässe auszunutzen um mit möglichst wenigen Ameisen zu verteidigen. Eine weitere Idee der Verwendung von einer Sperre ist, dass durch die Sperre der Gegners die Umgebung dahinter nicht erkunden kann. So bekommt er unseren Hügel nie zu Gesicht und dadurch eventuell auch kein Angriff. Diese Art zu Verteidigen hat den Nachteil, dass die Ameisen welche für die Sperre benötigt werden, entsprechend statisch platziert werden. Diese können dann nicht für andere Aufgaben verwendet werden.

Die Sperre wurde gegen Xathis getestet und sie zeigte nur eine geringe Wirkung. Der Bot von Xathis wartet vor der Sperre bis er in Überzahl ist und überrumpeln dann unsere Verteidigung. Erstaunlicher Weise wird die Sperre von Greentea (zweitbester Bot), auch wenn er in Überzahl ist, nicht angegriffen. Gegen diesen Gegner war die Sperre also gutes Mittel zur Verteidigung.

Die Funktionsweise ist wie folgt. Es werden je nach Breite der Sperre entsprechend viele Ameisen zur Position der Sperre geschickt um diese aufzubauen. Die Ameisen stellen sich paarweise hintereinander auf, solange keine Gegner in der Nähe sind. So können eigene Ameisen die einer anderen Aufgabe nachgehen die Sperre passieren. Wenn der Gegner sich in Überzahl nähert, wird die Sperre geschlossen. Danach wird das Kampfverhalten dem AttackingCombatPositioning überlassen, welches die Positionierung übernimmt.

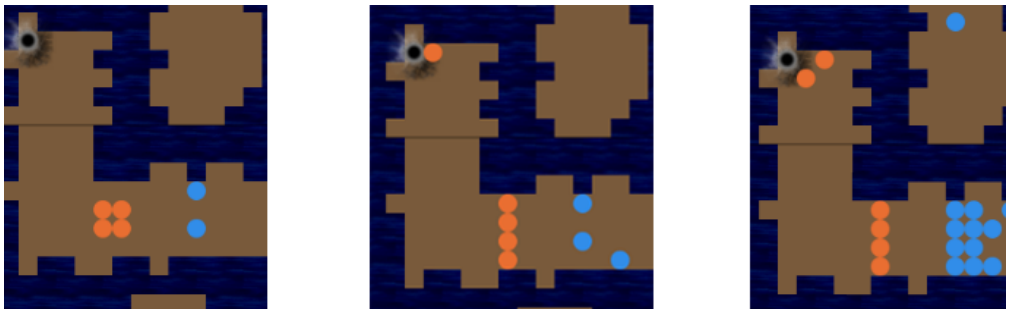

Abbildung 6.11: Die Sperre wird geschlossen und trotz gegnerischer Überzahl nicht angegriffen. (Gegner: Greentea)

6.5.5 ExploreMission

Precondition	-
Creator	ExploreTask
Postcondition	Definierter Erkundungspfad wurde zurückgelegt
Max. Ants	1
Max. Missionen	unbegrenzt
Valid	oft, siehe Beschreibung
Gather Ants	nicht möglich
Release Ants	nicht möglich

Tabelle 6.4: Eigenschaften der ExploreMission

Die ExploreMission wird verwendet um die Spielkarte zu erkunden. Die Ameise läuft den vom ExploreTask vorgegebenen Pfad ab. Falls dieser ungültig ist, weil er zum Beispiel durch Wasser führt oder die Ameise sich nicht bewegen kann, da die Ameise durch eine andere Ameise blockiert wird, so wird die Mission abgebrochen. Die Mission wird auch abgebrochen, wenn Futter in der Nähe ist, wenn ein gegnerischer Hügel oder gegnerische Ameisen auftauchen oder wenn ein eigener Hügel in der Nähe Hilfe braucht.

Abbildung 6.12: Eine Ameise auf einer ExploreMission bewegt sich in Richtung Fog of War

6.5.6 GatherFoodMission

Precondition	-
Creator	GatherFoodTask
Postcondition	Diese Mission besteht während dem ganzen Spiel.
Max. Ants	unbegrenzt
Max. Missionen	1
Valid	immer
Gather Ants	siehe Beschreibung
Release Ants	siehe Beschreibung

Tabelle 6.5: Eigenschaften der GatherFoodMission

Diese Mission koordiniert die Futtersuche und verwaltet die Ameisen, welche Futter einsammeln. Es wird nur eine GatherFoodMission zu Beginn des Spiels erstellt. Sie berechnet welche Ameise am nächsten bei einer Futterzelle

ist, berechnet den Pfad und schickt die Ameise Zug um Zug in Richtung Futter. Die execute()-Methode der GatherFoodMission sieht wie folgt aus:

```
@Override
public void execute() {
    // check the existing routes, if they are still valid.
    checkAntsRoutes();
    // gather new ants, relcalculate routes
    gatherAnts();
    // move the ants
    moveAnts();
}
```

Listing 6.4: Ablauf von execute() während der GatherFoodMission

Bei checkAntsRoutes() wird geprüft ob die Pfadrouten aller Ameisen in der Mission noch gültig sind. Ungültige Pfade sind, wenn die Futterzelle von der eigenen Ameise oder einer gegnerischen Ameise gefressen wurde. Ameisen mit einem ungültigen Pfad werden von der Mission entlassen, können aber in der nächsten Methode gatherAnts(), falls eine Aufgabe gefunden für die Ameise gefunden wird, der Mission wieder beitreten. Die Logik der gatherAnts() Methode ist in folgenden Codebeispiel vereinfacht dargestellt.

TODO CAPTION wird abgeschnitten

```
foreach(food on map){
    if(there is an ant in food mission, which is gathering this food tile with a
       path smaller than 5)
        continue;
    Ant ant = getNearestAntWithBreadthFirstSearch();
    if(ant found)
        possibleRoutes.add(new Route(ant,food));
}
foreach(Route r in possibleRoutes){
    Path path = getPathWithAStar(r);
    if(foodIsTargetedbyOtherAnt()){
            compareDistances()
            if(new path is shorter){
                    newGatherFoodRoute(r.ant,r.food,path);
                    releaseAnt(otherAnt);
            }
            continue;
    }
    if(hasAlreadyGatherFoodRoute(r.ant)
            takeSmallerRoute();
    else
            newGatherFoodRoute(r.ant,r.food,path);
}
```

Listing 6.5: Berechnung welche Ameise welches Futter einsammelt.

Zum Schluss folgt die moveAnts()-Methode. Hier werden die Ameisen auf ihrem Pfad zur Futterzelle ein Zug weiter bewegt. Die Züge werden der Orders-Klasse (Befehlsverwaltung) übergeben.

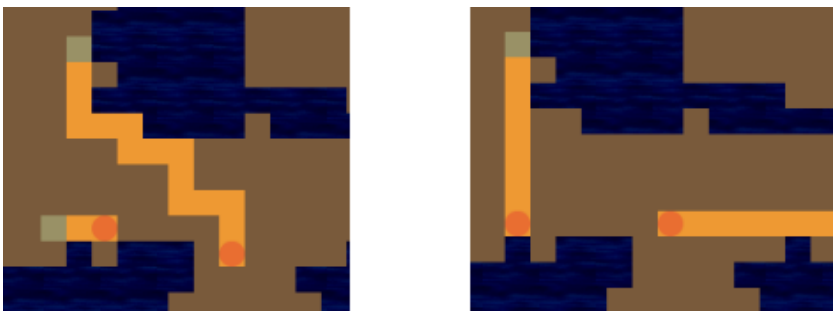

Abbildung 6.13: Ameisen am Futter einsammeln.

Links ist zu sehen wie für beide Ameisen ein Futterpfad definiert ist, beide Ameisen folgen ihrem Pfad. Zwei Züge später (rechtes Bild) hat die Ameise links ihr Futter eingesammelt und steht für eine neue Aufgabe zur Verfügung. Der beschriebene Algorithmus merkt, dass die Ameise näher an der Futterzelle oben im Bild ist und löst die andere Ameise, welche bis dahin auf das Futter zu steuerte, von der Aufgabe ab. Die rechte Ameise wird von der GahterFoodMission entlassen und geht einer anderen Beschäftigung (hier ExploreMission) nach.

6.5.7 Verworfene und nicht verwendete Mission

Wie bereits im Kapitel Task erwähnt war nicht alles programmierte erfolgreich. Hier sind die Missionen aufgelistet die zu den verworfen oder nicht verwendeten Tasks gehören. (Begründungen siehe Tasks)

- **SwarmPathMission**
- **AttackHillsInFlockMission**
- **ConcentrateMission**
- **FlockMission**

6.6 Ressourcenverwaltung

Die Ressourcenverwaltung ist naturgemäss ein wichtiger Bestandteil bei einem Spiel, in dem mit limitierten Ressourcen (Ameisen) diverse verschiedene und sich konkurrenzierende Aufgaben erledigt werden müssen. So stellen sich Fragen wie "Sollen wir eine Ameise zur Verteidigung zurückbehalten, oder schicken wir sie besser zur Futtersuche los? " oder "Lohnt es sich, die Karte weiter zu erkunden, oder gehen wir besser zum Angriff über? " Um solche Fragen beantworten zu können, haben wir ein regelbasiertes Ressourcenverwaltungs-System implementiert. Dieses ist unser wichtigstes Instrument bei der Umsetzung von strategischen Entscheidungen.

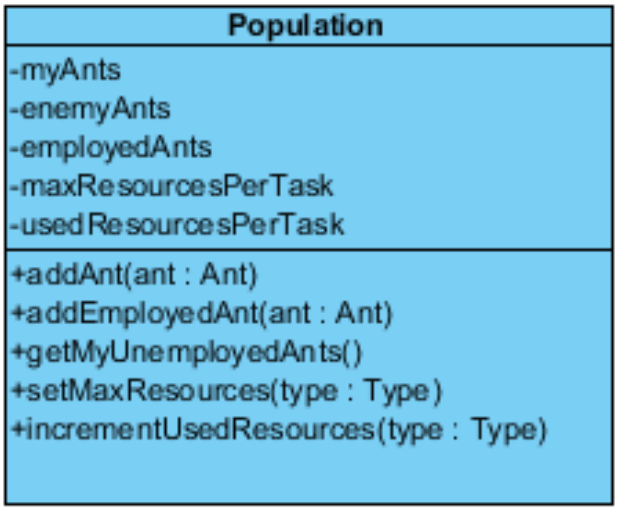

Abbildung 6.14: Population Klasse

Abbildung 6.14 zeigt unsere Population-Klasse (s. Kapitel 6.1), die als zentrale Verwaltungsinstanz für unsere Ameisen auch dafür zuständig ist, die allozierten Ressourcen zu verwalten und beim Zuweisen einer Ameise zu einer Aufgabe sicherzustellen, dass das Kontingent nicht überschritten wird.

Mit den beiden Feldern `maxResourcesPerTask` und `usedResourcesPerTask` wird hier Buch geführt darüber, welche Aufgabe wie viele Ameisen verwenden darf, und wie viele sie bereits verwendet hat. Die Aufteilung der Ressourcen, also das Initialisieren von `maxResourcesPerTask`, geschieht in unserem ResourceAllocator.

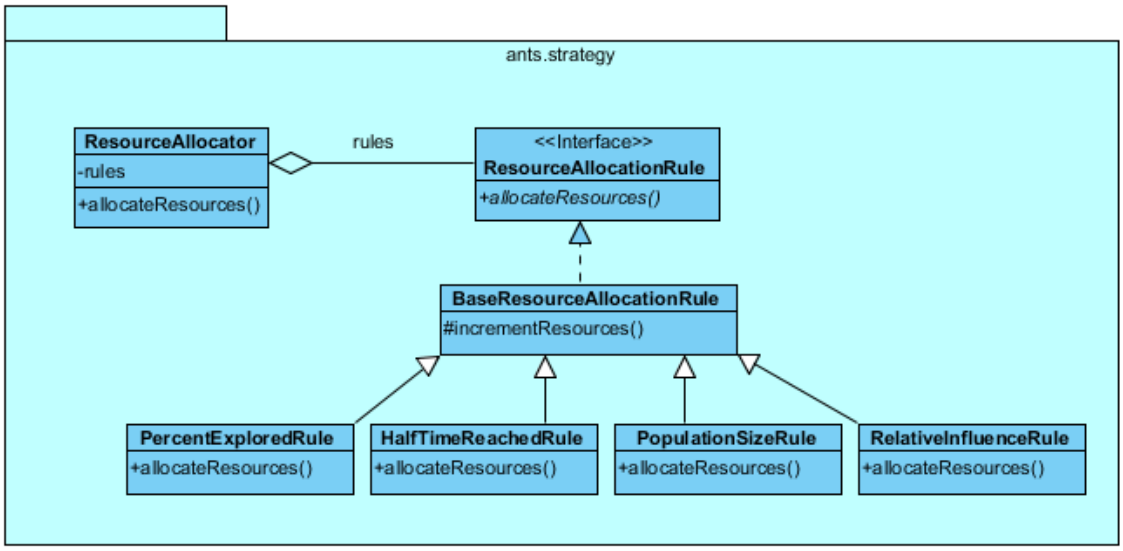

Abbildung 6.15: RessourcenManagement

Abbildung 6.15 zeigt die Klassenhierarchie unserer Ressourcenverwaltung. Die zentrale Klasse ist der ResourceAllocator; mit dessen Methode allocateResources() werden jeweils die Ressourcen neu verteilt:

```
new ResourceAllocator().allocateResources();
```

Diese Ressourcenzuteilung wird von unserem Bot jeweils zu Beginn des Zugs vorgenommen. Dazu stellt er als erstes die allozierten Ressourcen pro Aufgabe auf einen definierten Standardwert zurück. Diese Werte können über Profile (Kap. 6.8) konfiguriert werden. Dann werden der Reihe nach die konfigurierten Regeln aufgerufen, die jeweils nach ihrer eigenen Logik die Ressourcen für bestimmte Aufgaben erhöhen oder verringern. Am Ende entsteht so eine dynamische Ressourcenverteilung, die den verschiedensten Gesichtspunkten Rechnung tragen kann.

6.7 Implementierte Regeln

Wir haben 4 verschiedene Regeln implementiert, die alle von einer gemeinsamen Basisklasse ableiten. Die Basisklasse `BaseResourceAllocationRule` stellt die Methode `incrementResources(Type taskType, int increment, Type... tasksToDecrement)` zur Verfügung, die ein gegebenes Inkrement (Ressourcen-Plus) für eine Aufgabe mit gleichmässig verteilten Dekrementen (Ressourcen-Minus) auf die angegebenen anderen Aufgaben ausgleicht.

Unsere implementierten Regeln sind:

- **HalfTimeReachedRule:** Nach Erreichen der Halbzeit des Spiels sorgt diese Regel dafür, dass das Angreifen der gegnerischen Hügel höher priorisiert wird.
- **PercentExploredRule:** Diese Regel sorgt dafür, dass das Erkunden der Umgebung forciert wird, solange nicht ein gewisser Prozentsatz der Karte bereits erforscht ist.
- **PopulationSizeRule:** Diese Regel forciert die Nahrungsbeschaffung; je nach Grösse unseres Ameisenvolks tut sie dies stärker oder weniger stark.
- **RelativeInfluenceRule:** Fall wir eine dominante Position auf dem Spielfeld haben, sorgt diese Regel für zusätzliche Ressourcen für die Erkundung (um das Spielfeld weiter zu kontrollieren) und den Angriff auf gegnerische Hügel.

All diese Regeln sind über Profile konfigurierbar.

Diese Regeln sind natürlich nur eine Auswahl dessen, was mit dieser regelbasierten Ressourcenverwaltung möglich wäre. Viele weitere Regeln wären denkbar, aber wir haben in unseren Tests bereits mit diesen vier einfachen Regeln durchaus zufriedenstellende Ergebnisse erzielt. Eine Schwierigkeit bei der Implementierung neuer Regeln gilt es noch

zu beachten: Durch die grosse Dynamik des Systems sind die Auswirkungen einer Änderung manchmal nur schwer abzuschätzen; dieser Effekt wird durch die Konfigurierbarkeit des Systems noch verstärkt. Die Berechenbarkeit des Verhaltens unseres Bots ist also grösser, wenn nicht allzu viele Regeln in die Ressourcenverwaltung eingebunden werden.

6.8 Profile

Über die Profile lässt sich das Verhalten unseres Bots konfigurativ beeinflussen. Folgende Parameter stehen dazu zur Verfügung:

- **defaultAllocation.gatherFood**: Standard Ressourcenzuteilung in Prozent für die Nahrungssuche. Dieser Wert dient als Startwert für den ResourceAllocator.
- **defaultAllocation.explore**:Standard Ressourcenzuteilung in Prozent für die Erkundung der Spielwelt. Dieser Wert dient als Startwert für den ResourceAllocator.
- **defaultAllocation.attackHills**:Standard Ressourcenzuteilung in Prozent für den Angriff auf gegnerische Hügel. Dieser Wert dient als Startwert für den ResourceAllocator.
- **defaultAllocation.defendHills**:Standard Ressourcenzuteilung in Prozent für die Verteidigung der eigenen Hügel. Dieser Wert dient als Startwert für den ResourceAllocator.
- **defendHills.startTurn**: In der Anfangsphase des Spiels macht es nicht viel Sinn, Verteidiger zurückzubehalten obwohl kein Angreifer in der Nähe ist. Mit diesem Parameter kann eingestellt werden, ab welchem Zug wir pro Hügel mindestens einen Verteidiger abkommandieren.
- **defendHills.minControlRadius**: Dieser Parameter legt fest, wie gross das Gebiet um unsere Hügel ist, das wir verteidigen.
- **explore.forceThresholdPercent**: Schwellenwert für das Verhältnis von nie gesehenen Zellen zu allen Zellen auf der Karte. Wenn der Wert unter diesem Schwellenwert liegt, werden zusätzliche Ressourcen für die Erkundung bereitgestellt.
- **explore.forceGain**: Dieser Wert bestimmt, um wie viel die Ressourcen für die Erkundung erhöht werden, falls obiger Schwellenwert unterboten wird. Mögliche Werte liegen zwischen 0 und 1.
- **explore.dominantPositionBoost**: Wenn wir eine dominante Position auf dem Spielfeld haben, wird die Erkundung gestärkt, um die Karte zu kontrollieren. Dieser Parameter bestimmt, wie viel zusätzliche Ressourcen zugeteilt werden.
- **attackHills.dominantPositionBoost**: Wenn wir eine dominante Position auf dem Spielfeld haben, wird der Angriff auf gegnerische Hügel forciert, um Punkte zu sammeln. Dieser Parameter bestimmt, wie viel zusätzliche Ressourcen zugeteilt werden.
- **attackHills.halfTimeBoost**: Nach Ablauf der Hälfte der Spielzeit erhöhen wir die Aggressivität unseres Bots. Dieser Parameter bestimmt, wie stark der Angriff auf gegnerische Hügel in der 2. Halbzeit forciert wird.

6.8.1 Validierung

Die Parameter `defaultAllocation.*` müssen in der Summe 100 ergeben. Dies wird beim Start des Bots, wenn das Profil geladen wird, überprüft. Auch für die anderen Parameter sind Validierungen vorhanden (z.B. Prüfung, ob ein Prozentwert zwischen 0 und 100 liegt.)

Fehlt ein Parameter in einem Profil, so wird für diesen Parameter der Standardwert genommen.

6.8.2 Definierte Profile

Folgende Profile sind im Bot definiert und können beim Start ausgewählt werden:

Parameter	Default	Aggressive	Defensive	Expansive
defaultAllocation.gatherFood	25	20	25	30
defaultAllocation.explore	25	20	25	30
defaultAllocation.attackHills	25	45	15	20
defaultAllocation.defendHills	25	15	35	20
defendHills.startTurn	30	30	20	30
defendHills.minControlRadius	8	8	12	8
explore.forceThresholdPercent	80	80	80	90
explore.forceGain	0.25	0.25	0.25	0.4
explore.dominantPositionBoost	5	5	5	8
attackHills.dominantPositionBoost	2	5	2	2
attackHills.halfTimeBoost	20	25	15	20

Tabelle 6.6: Definierte Profile

6.8.3 Weiterentwicklungs-Potenzial

Aktuell können 11 Parameter via Profil angepasst werden, wovon die meisten einen Einfluss auf das Ressourcen-Management haben. Weitere Parameter für beliebige Teile der Logik könnten problemlos eingefügt werden und müssten sich auch nicht auf numerische Werte beschränken. Denkbar wären bspw. ganze Strategien, die per Profil einfach ausgetauscht werden könnten.

7 Logging

Für eine gründliche Analyse der Spiele sind ausführliche und konfigurierbare Logfiles unabdingbar. Zu diesem Zweck haben wir bereits im Rahmen des "Projekts 2 " ein flexibles Logging-Framework geschrieben. Dieses Framework haben wir im Rahmen der Bachelorarbeit noch einmal verbessert und erweitert. Auch die Auslagerung in eine separates, unabhängiges Modul geschah im Rahmen der Bachelorarbeit.

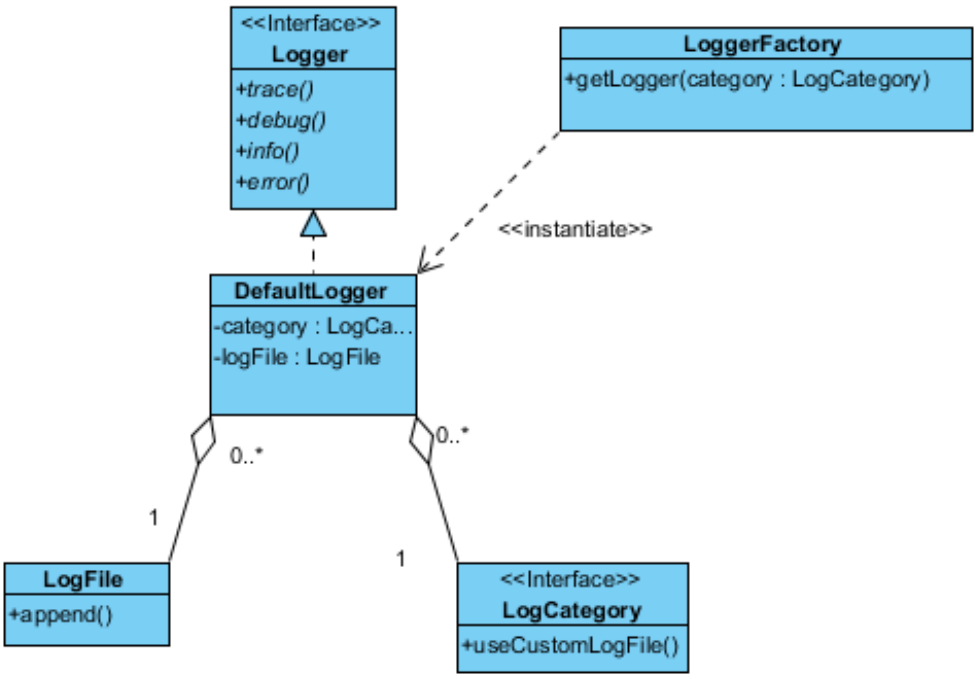

Abbildung 7.1: Logging Klassen

Abbildung 7.1 zeigt die wichtigsten Klassen des Logging-Frameworks.

- **Logger:** Das Logger Interface definiert die Methoden, die aus dem Applikations-Code heraus zum Logging verwendet werden können. Die Methodennamen bezeichnen den LogLevel, als Argumente übergeben werden ein String mit der Logmeldung sowie eine Liste von zusätzlichen Parametern, die in die Logmeldung eingefügt werden soll. Die Methoden unterstützen alle die Java String Formatter Syntax. [1]
- **DefaultLogger:** Unsere Standardimplementierung des Logger Interfaces. Der DefaultLogger hat jeweils eine LogCategory und ein LogFile, in das er seine Meldungen schreibt.
- **LogFile:** Das LogFile ist für eine einzelne Log-Datei zuständig; alle Logger, die in eine bestimmte Log-Datei schreiben wollen, greifen auf dieselbe LogFile Instanz zu. Die "physische" Datei auf dem Dateisystem wird von der LogFile Instanz erst erstellt, wenn die erste Log-Meldung geschrieben werden soll.
- **LogCategory:** Jede Log-Meldung gehört zu einer Kategorie. Dieses Interface bietet den verwendenden Modulen die Möglichkeit, eigene Kategorien zu definieren; das Logging-Framework selber definiert keine Kategorien. Mit der Methode useCustomLogFile() kann für eine Kategorie bestimmt werden, ob sie in das globale LogFile oder ein eigenes loggen soll. Dadurch kann die Übersichtlichkeit in den Log-Dateien gesteigert werden.

[1] `http://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#syntax`

- **LoggerFactory:** Sämtliches Erstellen von Logger-Instanzen läuft über die LoggerFactory. Sie stellt sicher, dass LogFile-Instanzen wiederverwendet werden und erstellt jeweils einen passenden Logger.

7.1 Konfiguration

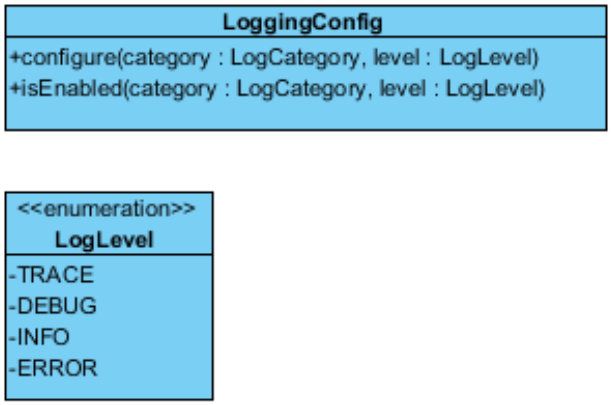

Abbildung 7.2: Logging Konfiguration

Abbildung 7.2 zeigt die Konfigurations-Klassen des Logging-Frameworks. Mit der Methode `configure()` der Klasse LoggingConfig kann der LogLevel für eine bestimmte Kategorie eingestellt werden. Mit der Methode `isEnabled()` kann abgefragt werden, ob ein bestimmter LogLevel für eine bestimmte Kategorie aktiv ist.

Der LogLevel bestimmt, ob eine Meldung überhaupt ins Log geschrieben wird. Die LogLevel haben eine implizite Hierarchie, wie man das von verbreiteteren Logging-Frameworks (z.B. Log4j, SLF4J) her kennt: TRACE > DEBUG > INFO > ERROR. Das heisst, wenn für eine Kategorie der LogLevel INFO definiert ist, werden Meldungen mit Level INFO und ERROR geloggt, DEBUG und TRACE aber nicht. Auf diese Weise ist eine feingranulare Kontrolle darüber möglich, was genau geloggt wird und was nicht.

7.2 Anwendungsbeispiele

Folgende Codezeilen zeigen, wie drei Log-Kategorien mit unterschiedlichen LogLevels konfiguriert werden können:

```
LoggingConfig.configure(LogCategory.ORDERS, LogLevel.DEBUG);
LoggingConfig.configure(LogCategory.PERFORMANCE, LogLevel.TRACE);
LoggingConfig.configure(LogCategory.SETUP, LogLevel.ERROR);
```

Folgende Codezeilen zeigen, wie ein Logger instanziert wird:

```
private static final Logger LOGGER = LoggerFactory.getLogger(LogCategory.
    ATTACK_HILLS);
```

Folgende Codezeilen zeigen, wie ein Logger verwendet wird:

```
LOGGER.info("AttackHillMission:new ants gathered for hill %s amount %s Ants: %s",
    enemyHill, newAnts.size(),
                newAnts.keySet());
```

7.3 JavaScript Addon für HMTL-Gameviewer

Das Codepaket welches von den Challenge-Organisatoren mitgeliefert wird, bietet bereits eine hilfreiche 2D-Visualisierung des Spiels, mit welchem das Spielgeschehen mitverfolgt werden kann. Die Visualisierung wurde mit HMTL und Javascript implementiert. Leider ist es nicht möglich zusätzliche Informationen auf die Seite zu projizieren. Deshalb haben wir den Viewer bereits im Projekt 2 mit einer solchen Funktion erweitert. Mit der Codezeile LiveInfo.liveInfo(...) kann eine Zusatzinformation geschrieben werden, welche auf dem Viewer später sichtbar ist. Es muss definiert werden mit welchem Zug und wo auf dem Spielfeld die Information angezeigt werden soll. Im Beispiel wird an der Position der Ameise (ant.getTile()) ausgegeben welchen Task die Ameise hat.

```
LiveInfo.liveInfo(Ants.getAnts().getTurn(), ant.getTile(),
                  "Task: %s ant: %s", issuer, ant.getTile());
```

Auf der Karte wird ein einfaches aber praktisches Popup mit den geschriebenen Informationen angezeigt. Dank solcher Zusatzinformationen muss nicht mühsam im Log nachgeschaut werden, welcher Ameise wann und wo welcher Task zugeordnet ist.

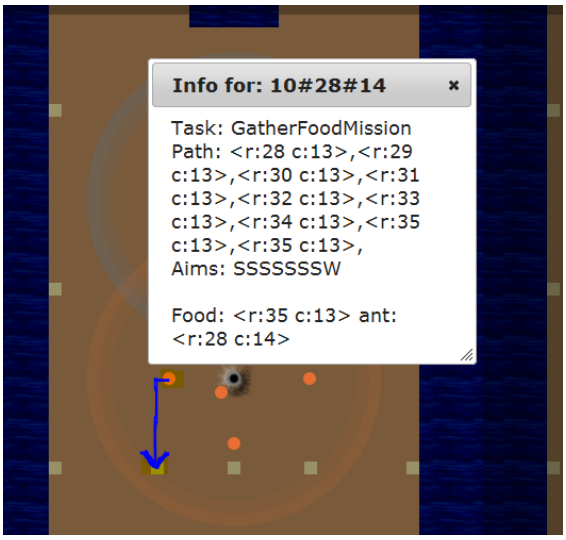

Abbildung 7.3: Im Popupfenster steht die Aufgabe der Ameise sowie die Pixel des Pfades (falls vorhanden), welcher die Ameise ablaufen wird.

Das angezeigte Popup zeigt welchen Task (GatherFoodTask) die Ameise hat, wo sie sich befindet <r:28 c:14>, welches Futterzelle angesteuert wird <r:35 c:13> und welchen Pfad dazu berechnet wurde.
Im Rahmen der Bachelorarbeit wurde dieses Addon erweitert. Nun werden alle Pixel welche in dem Popup ausgegeben werden auf der Karte markiert. (Siehe Abb 7.4)

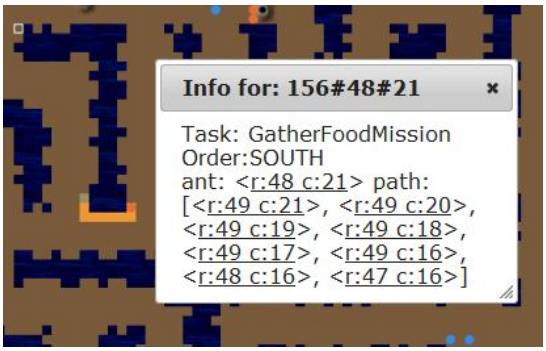

Abbildung 7.4: Mit der erweiterten Version wird der Pfad (orange) der Ameise von <r:48 c:21> nach <r:47 c:16> auf der Karte abgebildet.

7.4 Profile und Logging

Mit der Einführung der Profile gewannen wir die Möglichkeit, mehrere Kopien unseres Bots mit unterschiedlichen Konfigurationen gegeneinander antreten zu lassen. Dazu mussten wir aber die Logfiles so voneinander trennen, dass die Logs der verschiedenen Bots isoliert sind. Die Logfiles werden daher jetzt in ein Unterverzeichnis mit dem Namen des konfigurierten Profils geschrieben.

Für das LiveInfo Log, das für das Javascript Addon geschrieben wird, verzichteten wir auf eine solche Erweiterung, da so viel Information für mehrere Bots nur schwer übersichtlich darzustellen wäre. Stattdessen bauten wir einen Schalter in den Code ein, mit dem das LiveInfo vorübergehend deaktiviert werden kann, um Konflikte zwischen den Bots zu vermeiden.

8 Testing

Dieses Kapitel beschreibt wie wir unseren Bot und dessen Komponenten getestet haben. Das Testen beanspruchte ein grosser Teil unserer aufgewendeten Zeit. Die nachfolgenden Methoden zeigen auf wie wir versuchten möglichst effizient zu testen, so dass uns mehr Zeit für das implementieren von Modulen blieb. Es stellte sich heraus, dass das Implementieren und Testen von neuem Code in UnitTests, anstatt direkt im Bot, zeitsparender war und weniger Mühe bereitete Fehler zu finden.

8.1 UnitTests

Dank dem modularen Aufbau des Java-Codes war es uns möglich einzelne Module zu testen. Für unsere UnitTests verwendeten wir die JUnit 4 Library. So findet man in jedem Java Project das Code-Package und ein UnitTest-Package, welches den Code auf dessen Richtigkeit prüft. Die Module erst nach den erfolgreichen UnitTests in den Bot eingebaut. Dies hat den Vorteil, dass die Fehlern nicht im laufenden Spiel auftraten und wir die Ursachen mühsam suchen mussten.
Um das Modul CombatPositioning[1] zu realisieren haben wir sogar 'Test Driven Development' eingesetzt. Dass heisst, wir haben zuerst den Test mit Ausgangslage, Aufruf von CombatPositioning, und das erwartete Resultat als UnitTest geschrieben. Danach wurde solange an CombatPositioning programmiert, bis das erwartete Testresultat eintraf.

8.2 Visuelle Tests

In Form eines UnitTest haben wir auch unsere visuellen Tests geschrieben. Eine visuelle Überprüfung des Resultats ist meistens, vor allem bei der Pfadsuche, einfacher zu kontrollieren. Im Gegensatz zu JUnit wo das zu erwartende Resultat mit der Methode `assertEquals()` geprüft wird, haben die visuellen Test ein HTML-File als Output. In das HTML-File wird die Karte in Tabellenform gespeichert. In jeder Zelle der Tabelle können Objekte (Einheiten, Hügel, Wegpunkte etc.) durch farbige Punkte dargestellt werden. Diese Funktionalitäten bietet die extra dafür geschriebene Klasse MapOutput. Nachfolgend ist ein HTML-File abgebildet mit welchem wir die Korrektheit des Clustering und des HPA* Algorithmus visuelle überprüft haben.

[1] siehe im Code unter Startegy.tactics.combat

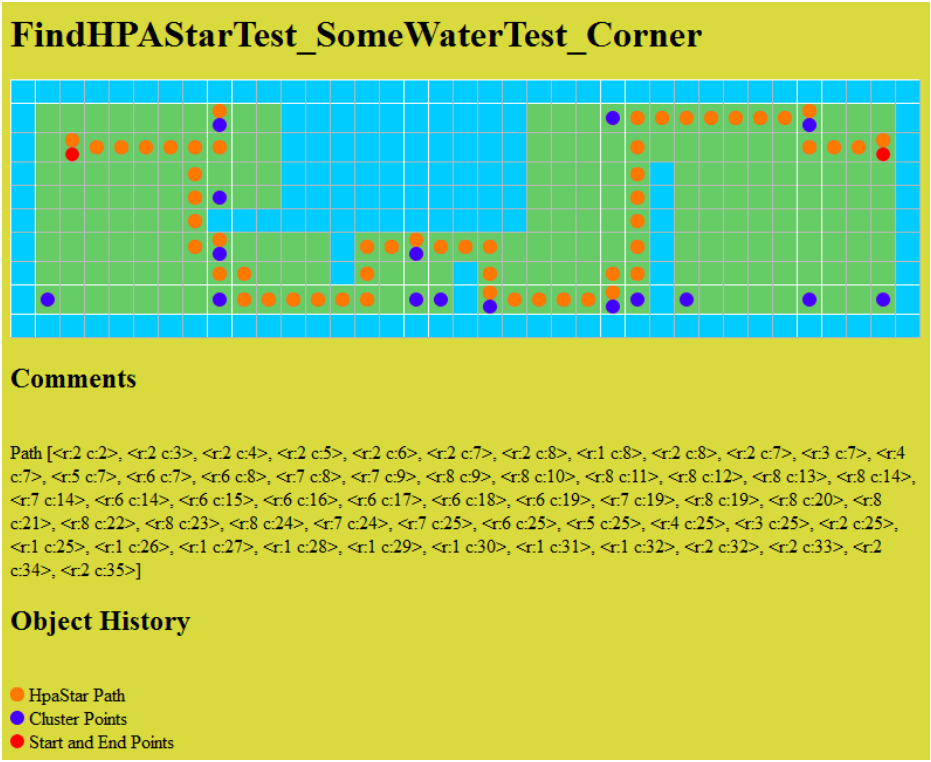

Abbildung 8.1: HTML-Ausgabe zur visuellen Überprüfung von HPA*

8.3 Testbots

Eine weitere Testmethode war Testbots zu erstellen. Zum Beispiel haben wir den DefendHillBot erstellt. Dieser hat die Eigenschaft nur zu verteidigen nicht aber anzugreifen. Wir nahmen eine kleine Karte, so dass es schnell zu Angriffen des Gegners kam. Dadurch konnten wir unseres Verteidigungvserhalten nach nur kurzer Spieldauer genau analysieren und verbessern.

Das selbe galt für den AttackHillBot. Wir haben wiederum eine kleine Karte genommen auf der viel Futter vorhanden ist, so dass sich das Ameisenvolk schnell vermehrt. Dank der beschränkten Karte kam es schnell zu Angriffen, wir konnten unsere Angriffspositionierungen testen.

Im Kapitel Task wurde beschrieben, welche Task bzw. Missionen nicht erfolgversprechend waren und wir nicht weiterverfolgt haben. Bevor wir das aber wussten, haben wir, um die Funktionen zu testen, wiederum einen speziellen Bot erstellt. Anhand der Ergebnissen konnten wir herausfinden, dass diese Methoden nicht praktikabel waren und wir die Ideen verworfen haben. So sind im Code noch folgende Bots zu finden:

- ConcentrateBot
- FlockBot
- SwarmBot

8.4 Performance Suchalgorithmen

Die entwickelten Suchstrategien werden in diesem Kapitel verglichen. Zum Vergleich wird eine Test Karte der Grösse 100 x 110 Tiles verwendet, auf welcher die Pfadsuchalgorithmen angewendet werden. Die Strategie Simple wir nicht in den Vergleich einbezogen, da diese wie beschreiben nur für kurze Pfade einsetzbar ist. Dafür wird der HPA* mit zwei verschiedenen Clustergrössen 10 & 25 angewendet.

8.5 Testreport Profile

Um die verschiedenen Profile unseres Bots zu testen, führten wir diverse Testläufe durch, in denen wir die verschieden konfigurierten Bots jeweils 100 Mal gegen verschiedene Gegner und gegeneinander antreten liessen. Für diese Testläufe wurden die in Tabelle 6.6 aufgeführten Profile verwendet.

Während der Entwicklung führten wir die meisten Tests gegen den Bot von Evan Greavette (Username egreavette) durch. Er hatte seinen Bot über GitHub veröffentlicht: `https://github.com/egreavette/Ants-AI`. Daher führten wir auch die ersten Testläufe gegen diesen Gegner durch:

Profil	Siege	Unentschieden	Niederlagen	Total Punkte
Default	62	13	25	957:579
Aggressive	61	18	21	1005:642
Defensive	53	15	32	915:726
Expansive	59	16	25	955:673

Tabelle 8.1: Bilanz der Testläufe gegen egreavette

Wie man unschwer erkennen kann, sind gegen diesen Gegner drei der Profile ungefähr gleichwertig; lediglich der Defensive Bot fällt etwas ab.

Als nächstes führten wir einen Testlauf gegen den Sieger des Wettbewerbs durch. Mathis Lichtenberger (Username xathis) hatte seinen Bot ebenfalls über GitHub zur Verfügung gestellt: `https://github.com/xathis/AI-Challenge-2011-bot`.

Profil	Siege	Unentschieden	Niederlagen	Total Punkte
Default	17	7	76	521:1094
Aggressive	23	11	66	593:1010
Defensive	11	8	81	479:1106
Expansive	24	6	70	610:988

Tabelle 8.2: Bilanz der Testläufe gegen xathis

Gegen diesen starken Gegner zeigen sich bereits deutlichere Unterschiede zwischen den Profilen. Man sieht, dass die beiden offensiver ausgerichteten Profile (Aggressive und Expansive) sich deutlich besser schlagen, während das Defensive Profil deutlich abfällt. Dies entspricht in etwa unseren Erwartungen. Die Stärken von xathis' Bot liegen eindeutig in der Offensive, während die Defensive nicht das Prunkstück unseres Bots ist. Deshalb können wir in der Verteidigung gegen xathis fast nur verlieren und haben eine grössere Chance, wenn wir selber in die Offensive gehen.

Als nächstes liessen wir unsere 4 Profile gegeneinander antreten:

Profil	Siege	Unentschieden	Niederlagen	Total Punkte
Default	22	7	71	332:891
Aggressive	23	10	67	314:909
Defensive	23	8	69	300:923
Expansive	16	7	77	277:946

Tabelle 8.3: Bilanz der Testläufe gegeneinander

Hier zeigt sich, dass die Profile im Kampf gegeneinander relativ ausgeglichen sind. Es zeigt sich auch, dass das expansive Verhalten sich gegen mehrere Gegner weniger lohnt, da sich dabei die Ameisen eher auf dem Spielfeld verzetteln und man Gefahr läuft, isolierte Ameisen zu verlieren.

Alles in allem waren aber diese Standard-Profile noch mehr oder weniger gleichwertig. Wir erstellten daher neue, stärker vom Standard abweichende Profile und führten noch einmal Testläufe gegen xathis und gegeineinander durch.

Parameter	**Default**	**Aggressive 2**	**Defensive 2**	**Expansive 2**
defaultAllocation.gatherFood	25	15	25	35
defaultAllocation.explore	25	15	25	35
defaultAllocation.attackHills	25	60	10	20
defaultAllocation.defendHills	25	10	40	10
defendHills.startTurn	30	30	20	50
defendHills.minControlRadius	8	8	20	8
explore.forceThresholdPercent	80	80	80	90
explore.forceGain	0.25	0.1	0.25	0.4
explore.dominantPositionBoost	5	3	5	10
attackHills.dominantPositionBoost	2	10	2	2
attackHills.halfTimeBoost	20	40	15	20

Tabelle 8.4: Die Profile für den 2. Testlauf

Profil	**Siege**	**Unentschieden**	**Niederlagen**	**Total Punkte**
Default	17	7	76	521:1094[a]
Aggressive	20	4	76	545:1040
Defensive	13	4	83	385:1186
Expansive	19	12	69	578:1010

[a]Da sich an diesem Profil nichts geändert hatte, wurde der Testlauf nicht noch einmal durchgeführt; die Werte stammen aus dem 1. Lauf

Tabelle 8.5: Bilanz der 2. Testläufe gegen xathis

Aus Tabelle 8.5 sieht man, dass die neuen Profile gegen xathis eher schlechter abschnitten, aber die Unterschiede zum 1. Testlauf sind nur gering.

Profil	**Siege**	**Unentschieden**	**Niederlagen**	**Total Punkte**
Default	41	9	50	428:788
Aggressive	12	7	81	276:940
Defensive	22	6	72	256:960
Expansive	11	10	79	256:960

Tabelle 8.6: Bilanz der 2. Testläufe gegeneinander

Wie Tabelle 8.6 zeigt, waren die Unterschiede zwischen den neuen Profilen im Kampf gegeneinander schon deutlich sichtbarer. Die extremeren Profile weisen alle eine deutlich schlechtere Bilanz auf, wobei die Defensive Konfiguration

noch am besten fährt. Gegen diese Profile ist unsere ausgewogene Standardkonfiguration aber bereits ein deutlicher Sieger.

8.5.1 TestReader

Für die Auswertung der Testläufe haben wir ein kleines Zusatzmodul geschrieben, das aus den Spielberichten der Spielengine die relevanten Informationen auslesen kann. Es besteht lediglich aus einer einzelnen Klasse (`TestReader` mit einer main() Methode. Es liest aus dem Log-Verzeichnis die *.replay Dateien (Spielberichte im Json-Format) ein, parsed sie mit Hilfe der Simple-Json Bibliothek und erstellt eine CSV-Datei mit den wichtigsten Ergebnissen aus den einzelnen Spielen. Ausserdem wird eine zusätzlich CSV-Datei ausgegeben, die die Ergebnisse aggregiert, und zwar ungefähr in der Form, wie sie in diesem Kapitel in den Tabellen ersichtlich ist.

8.6 Online Tests

Anfangs Dezember stellten wir fest, dass ein User aus dem alten AI-Challenge Forum mit dem Namen "smiley1983" einen TCP-Server aufgesetzt hatte, über welchen man Bots gegeneinander antreten lassen konnte. Natürlich probierten wir das aus uns führten auch um die 50 Spiele so aus. Für aussagekräftige Tests war die Teilnehmerzahl auf diesem Server aber zu gering, weshalb wir uns dagegen entschieden, diese neue Option in unser Testkonzept aufzunehmen.

9 Spielanleitung

9.1 Systemvoraussetzungen

Um ein Spiel ausführen zu können, muss auf dem Computer folgende Software installiert sein:

1. Java SE JDK Version 6 [1]
2. Python [2]
3. ANT [3]

Falls das Spiel mit anderen als den vorkonfigurierten Gegnern gestartet werden soll, muss evtl. noch andere Software installiert werden, je nachdem in welcher Programmiersprache die entsprechenden Bots geschrieben sind.

9.2 Ausführen eines Spiels

Im Ordner `Code` befinden sich unsere Eclipse-Projekte mit dem gesamten Source-Code unserer Implementation, und der offiziellen Spiel-Engine. Das Hauptprojekt befindet sich im Unterordner `Ants`. Zum einfachen Ausführen eines Spiels haben wir dort ein ANT-Buildfile (build.xml) erstellt. Dieses definiert verschiedene Targets, mit denen ein Spiel mit jeweils unterschiedlichen Parametern gestartet werden kann. Falls ANT korrekt installiert ist, können diese Targets aus dem `Ants`-Verzeichnis heraus einfach mit "`ant <targetName>`" aufgerufen werden.

1. Das Target `testBot` ist lediglich zum einfachen Testen eines Bots sinnvoll und entspricht dem Spiel, das verwendet wird, um Bots, die auf der Website hochgeladen werden, zu testen.
2. Das Target `runTutorial` führt ein Spiel mit den Parametern aus, die im Tutorial auf der Website zur Erklärung der Spielmechanik verwendet werden.
3. Die Targets `mazeAgainst*` führen jeweils ein Spiel auf einer komplexeren und grösseren, labyrinthartigen Karte aus, und zwar gegen den jeweils bezeichneten Gegner.
4. Das Target `maze4Players` führt ein Spiel gegen drei andere Bots auf einer noch etwas grösseren Karte durch.
5. Die Targets `mazeProfiles` und `mazeProfiles4Players` sind dazu da, mehrere Kopien unseres Bots mit verschiedenen Profilen gegeneinander antreten zu lassen. Wie im Kapitel 7.4 erwähnt, muss dazu erst in der Klasse LiveInfo das LiveInfo-Logging deaktiviert werden:

   ```
   private static boolean enabled = false;
   ```

6. Das Target `testOnline` lässt den Bot über einen privaten TCP-Server gegen andere Bots antreten , unter ähnlichen Bedingungen wie beim eigentlichen Wettbewerb, nur leider mit deutlich weniger Teilnehmern. (s. Kapitel 8.6)
7. Die Targets `repeat*` dienten der Durchführung der Profil-Testläufe und sind ansonsten wenig interessant.
8. Die Targets `debug*` dienten dem Starten unserer Testbots, die wir während der Entwicklung einsetzten (s. Kapitel 8.3).

[1] `http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-javase6-419409.html`
[2] `http://www.python.org/download/`
[3] `http://ant.apache.org/bindownload.cgi`

Bei den meisten dieser Targets erscheint beim Starten eine Abfrage, welches Profil man gerne starten würde. Hier stehen die in Kapitel 6.8.2 beschriebenen Profile zur Auswahl.

Im Unterordner `tools` befindet sich die in Python implementierte Spiel-Engine. Unter `tools/maps` liegen noch weitere vordefinierte Umgebungen, und unter `tools/mapgen` liegen verschieden Map-Generatoren, die zur Erzeugung beliebiger weiterer Karten verwendet werden können.

Im Unterordner `tools/sample_bots` befinden sich einige einfache Beispiel-Bots, gegen die man spielen kann.

Im Unterordner `tools/bots` haben wir Kopien der Bots von einigen Teilnehmern abgelegt, gegen die wir im Zug der Entwicklung immer wieder getestet haben.

Glossar

AI Artificial Intelligence - Künstliche Intelligenz.

API Application Programming Interface - Programmierschnittstelle.

BFS Breadth First Search - Breitensuche.

Bot AI-Agent für ein Computerspiel.

Fog of War Teil der Karte, der durch die eigenen Einheiten nicht mehr sichtbar ist.

Influence Map Datenstruktur, die zur Berechnung des Einflusses von Spieleinheiten auf die Spielkarte dient.

Lazy Initialization Bei der Lazy Initialization wird eine Ressource erst beim erstmaligen Gebrauch initialisiert (z.B. ein Logfile erst mit dem ersten Log-Eintrag erstellt).

Pair Programming Paarprogrammierung bedeutet, dass bei der Erstellung des Quellcodes jeweils zwei Programmierer an einem Rechner arbeiten. Ein Programmierer schreibt den Code, während der andere über die Problemstellungen nachdenkt, den geschriebenen Code kontrolliert sowie Probleme, die ihm dabei auffallen, sofort anspricht[4].

Tile Ortsangabe auf dem Spielfeld mit Row (Zeile) und Column (Spalte) beschrieben. Bsp: <r:12 c:10>.

UML Unified Modeling Language.

[4] Definition von `http://de.wikipedia.org/wiki/Paarprogrammierung`

Literaturverzeichnis

[1] I. Millington and J. Funge, *Artificial Intelligence For Games, Second Edition*. Morgan Kaufmann, 2009.

[2] S. Russel and P. Norvig, *Artificial Intelligence: A Modern Approach, Third Edition*. Pearson, 2010.