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: 5. 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	5. Januar 2013
Name Vorname	Lukas Kuster
Unterschrift	.....
Name Vorname	Stefan Käser
Unterschrift	.....

Inhaltsverzeichnis

Abbildungsverzeichnis

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 ist ein zugbasiertes Multiplayerspiel in welchem sich Ameisenvölker gegenseitig bekämpfen. Ziel einer 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 Futterpixel, wird dieses eingesammelt. Beim nächsten Zug entsteht bei dem Ameisenhügel eine neu Ameise.
- Die Landkarte besteht aus passierbaren Landpixel sowie unpassierbaren Wasserstellen.
- Ein Gegener wird geschlagen, wenn im Kampfradius der eigenen Ameise mehr eigene Ameise stehen als gegnerische Ameisen im Kampfradius der Ameise die angegriffen wird.
- 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] `http://www.aichallenge.org`
[2] Die Klasse ist im Code unter ants.bot.Bot.Java auffindbar

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 Komponeten zu übernehmen. Es folgt eine Auflistung was bereits bestand bzw. was wir noch erweitert haben.

Erstellt in Modul 'Projekt 2'	Erweiterung in während der Bachelorarbeit
Grundfunktionalitäten des Bots	
Pfadsuche Simple, A*, HPA*	Auslagerung in ein eigenes Framework, Performanceverbesserungen, Erweiterung des HPA* Clustering, Pfadsuche mittels InfluenceMap
Logging	Loggen in verschiedene Logfiles
Tasks	Die Funktionsweise jedes Tasks wurden nochmals umgekrempelt
Missionen	Die Missionen wurden verfeinert und mit strategischen un taktischen Entschiedungen erweitert
MinMax-Algorithmus	Bei den Kampfsituationen wurde versucht den Min-Max Alogrithmus einzubauen, welcher wir bereits im Modul Spieltheorie erarbeitet haben. (Mehr dazu siehe Kapitel Strategie)

TODO EINHEITLICHE TABELLEN FORMATIERUNG

1.3 Projektverlauf

Die Projektarbeit richtete sich nach folgendem Zeitplan:

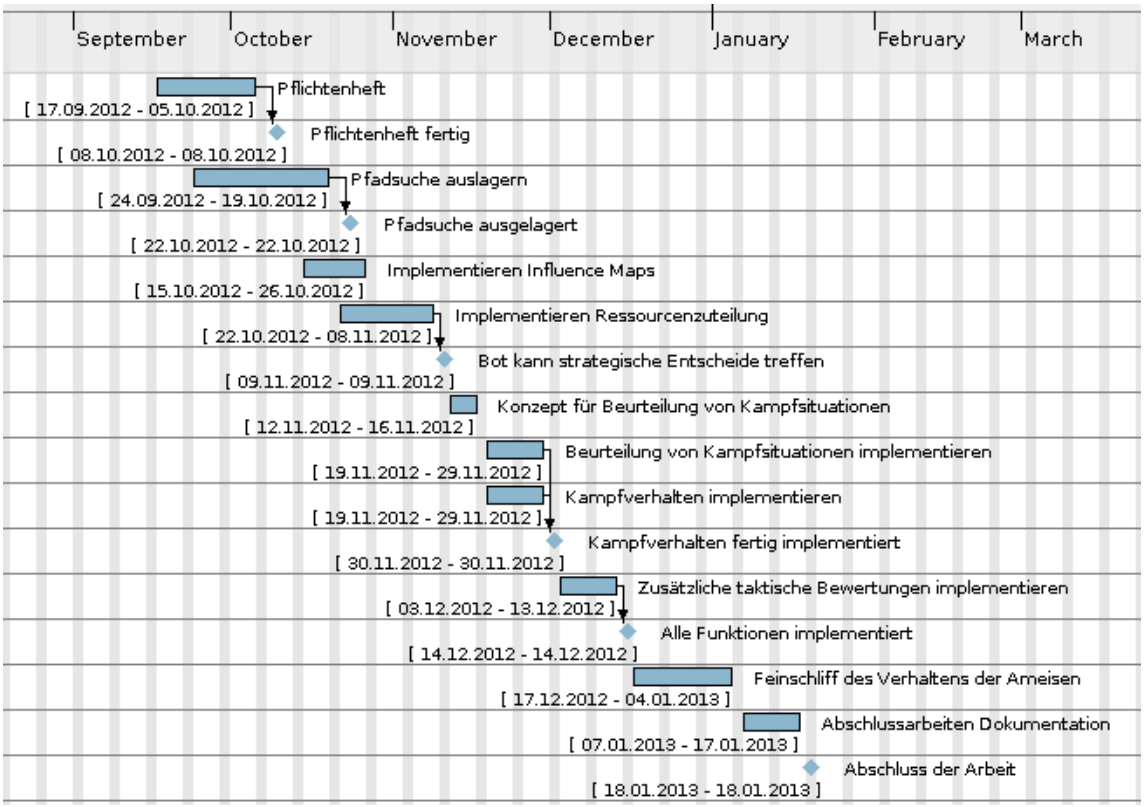


Abbildung 1.1: Projektablauf

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.

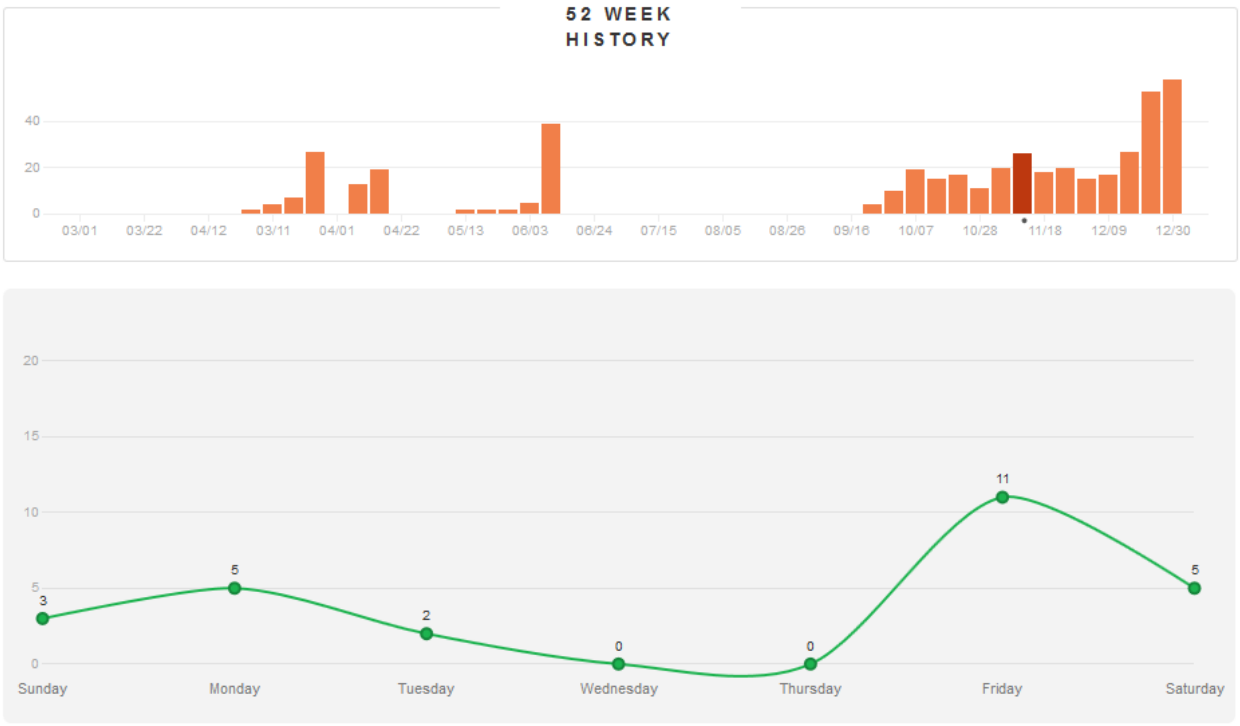


Abbildung 1.2: Commit History

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.

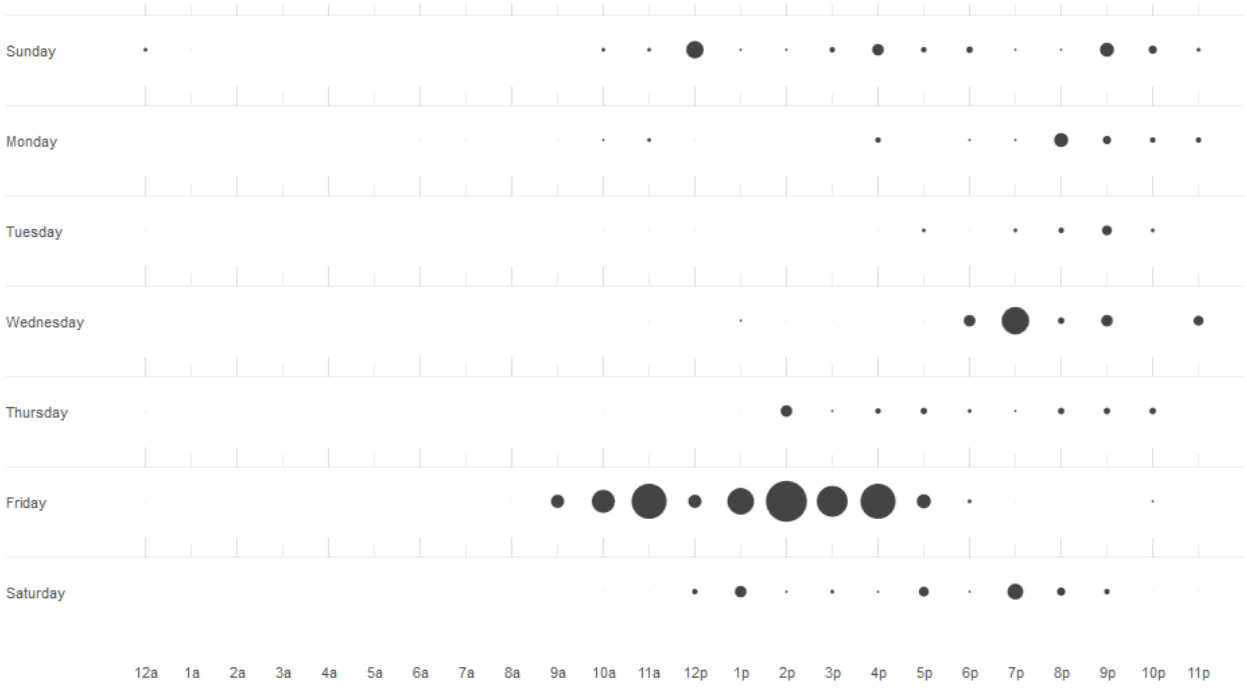


Abbildung 1.3: Aktivitäten

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 7.2) 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 6.7).

Bewertung: Das Ziel wurde erreicht. Da wir die Profile erst gegen Schluss der Arbeit eingebaut haben, beschränken sie sich aktuell auf Parameter 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 bewälitgen haben wir ein ausgeklügeltes Logging auf die Beine gestellt, in welchem wir schnell an die gewünschten Informationen gelangen. (siehe 7) Zudem können wir dank der Erweiterung des HTML-Viewer sofort sehen, welches die akutelle Aufgabe jeder einzelen Ameise ist. (siehe 7.2) Weitergeholfen haben uns auch etliche Unit- und Funktionstests, mit welchen wir neu geschriebenen Code testen und auf dessen Richtigkeit prüfen konnten. (siehe 9.1)

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 9.3.

1.9 Fazit

TODO

2 Architektur

2.1 Module

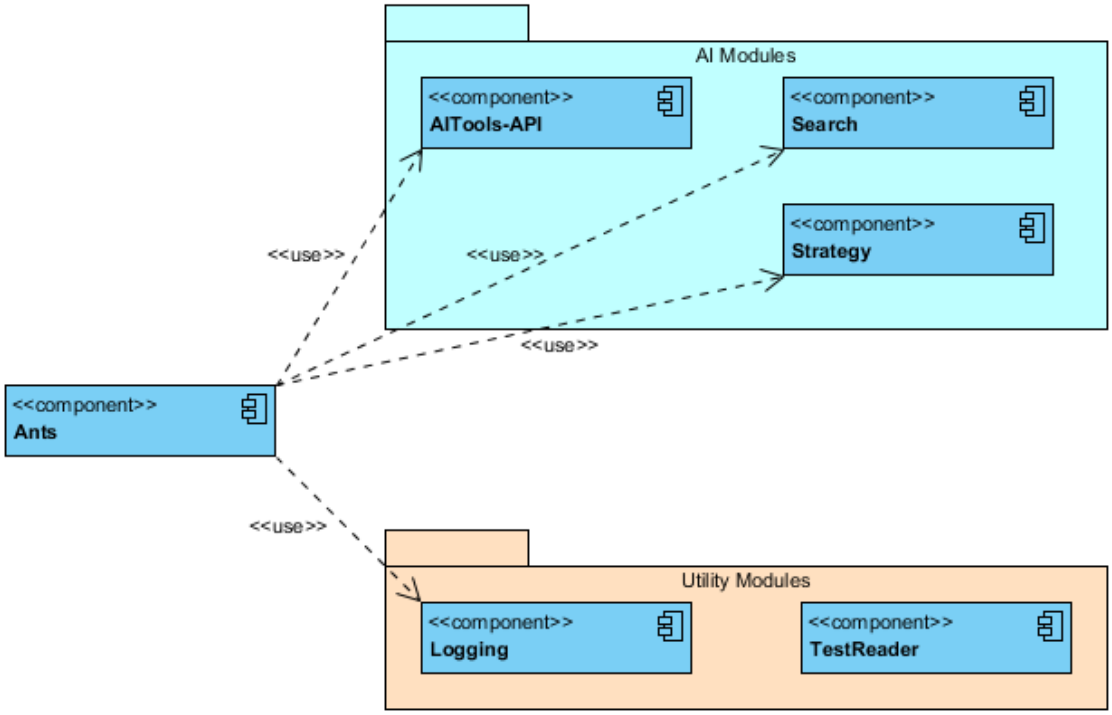


Abbildung 2.1: Module

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.

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.

Search: Dieses Modul enthält unsere Implementationen zur Pfad- und Breitensuche.

Strategy: Das Strategy-Module enthält strategische und taktische Algorithmen, insbesondere die Influence Map und das CombatPositioning.

Logging: Das Logging-Modul definiert ein flexibles Logging-Framework.

TestReader: Das TestReader-Modul besteht aus einer einzelnen Klasse, die aus den Log-Files der Ants-Spielengine Auswertungen lesen kann.

2.2 Modulabhängigkeiten

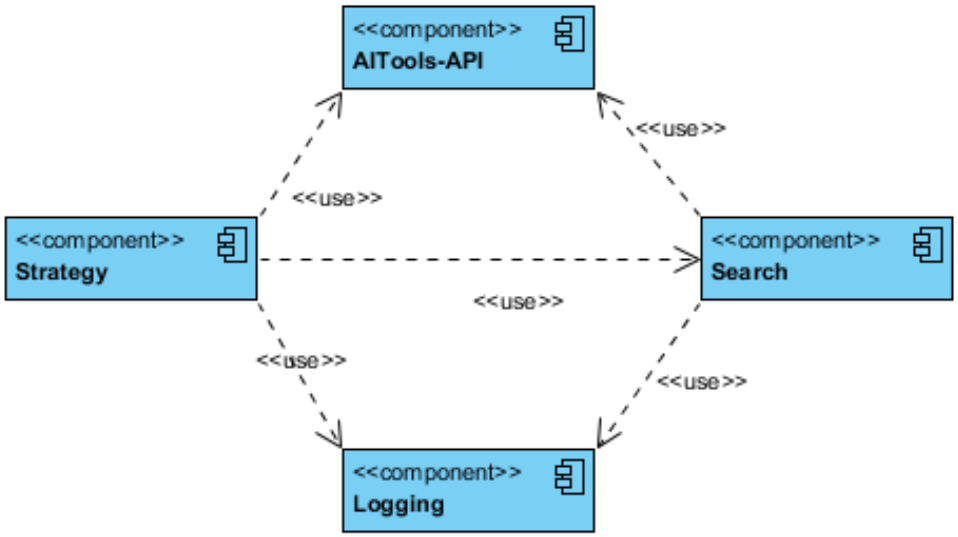


Abbildung 2.2: Modulabhängigkeiten

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.

2.4 Sequenzdiagramme

TODO macht dieses Kapitel Sinn? Was sollte hier rein?

3 API

Unsere AITools API dient als Grundbaustein des gesamten Projekt. Sie entkapselt die Interfaces zu unseren Search- und Strategieimpelementierungen. Sie beinhaltet zudem Basisklassen, welche überall verwendet werden. Der Inhalt ist wiefolgt:

3.1 Entities

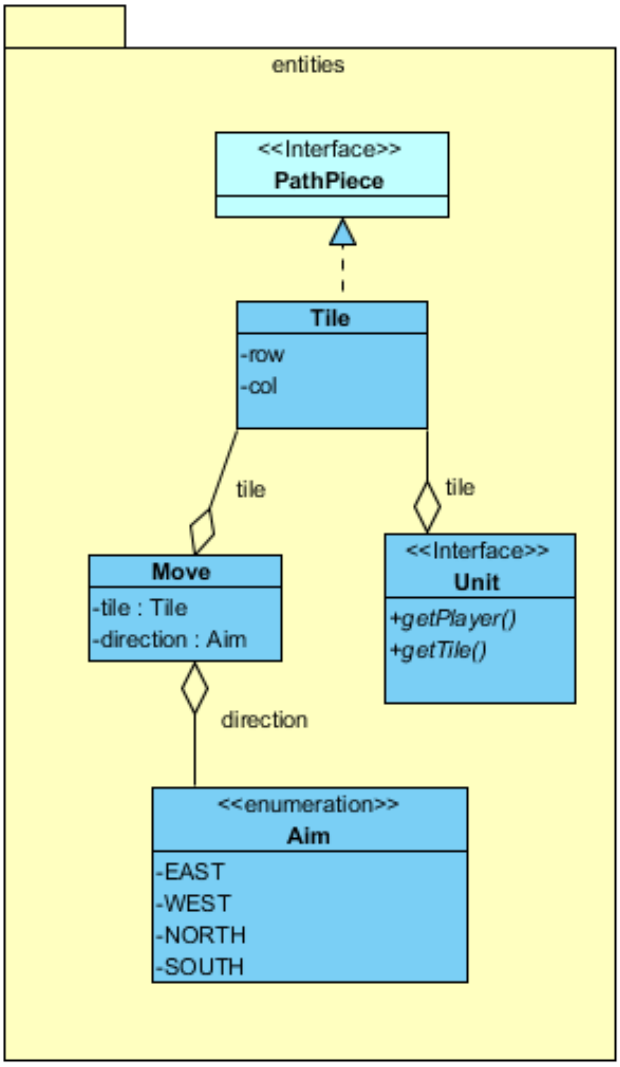


Abbildung 3.1: Entities

- **Aim**: Richtungsangabe zum Beschreiben einer Bewegung der Ameise
- **Tile**: Repräsentiert eine Zelle auf dem Spielfeld, welche mit Row (Zeile) und Column (Spalte) beschrieben 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

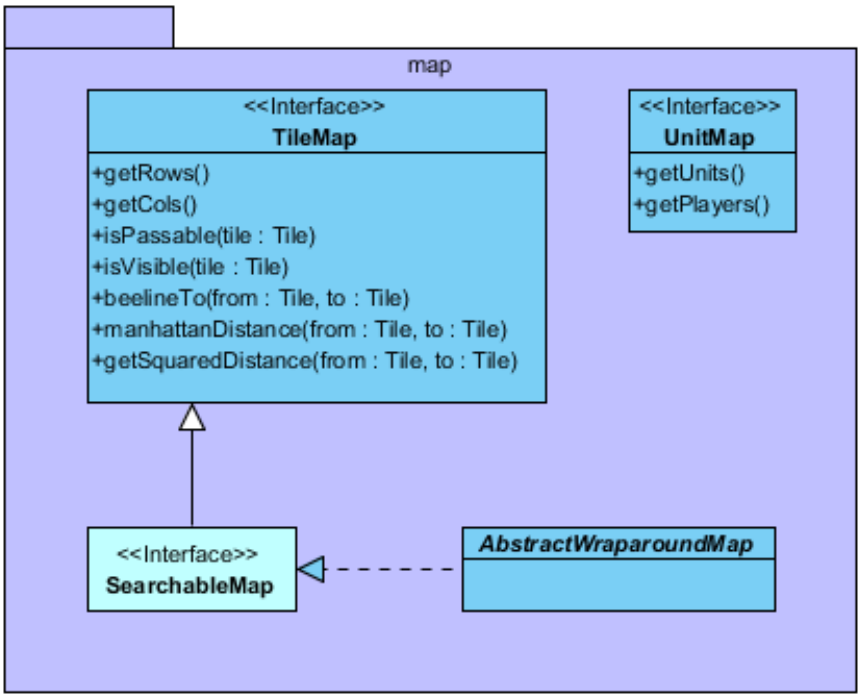


Abbildung 3.2: Map API

- **TileMap**: Dieses Interface definiert, welche Methoden eine auf Tiles aufbauende Spielkarte anbieten muss. Dazu gehören die Masse der Karte, ob ein Tile auf der Karte sichtbar und passierbar ist, sowie Distanz Messfunktion wie ManhattanDistanz, Luftlinie, quadrierte Distanz.
- **UnitMap**: UnitMap ist auch ein Interface welches auf TileMap aufbaut und zusätzlich die Methoden definiert die Einheiten und Spieler zurückzugeben.
- **AbstractWraparoundMap**: Implementiert das Interface SearchableUnitMap (siehe: Abschnitt Search). Hier sind alle Methoden implementiert, welche die Karte anbieten muss, damit sie mit den Suchalgorihmen verwendet werden kann. Zudem sind die Methoden aus TileMap implementiert. Diese geben über die Geländebeschaffenheit und Distanzen Auskunft.
- **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 Ants Challenge somit auch die Klasse AbstractWraparoundMap. Der zweite Enumtyp ist Pizza und definiert eine Welt so wie unsere Erde vor 500 Jahren noch definiert wurde, eine Scheibe mit Rändern, welche die Welt begrenzen. Dieser zweite Typ wurde provisorisch erstellt. Falls diese API eine Weiterverwendung findet, kann dieser Typ zusätzlich implementiert werden.

3.3 Search

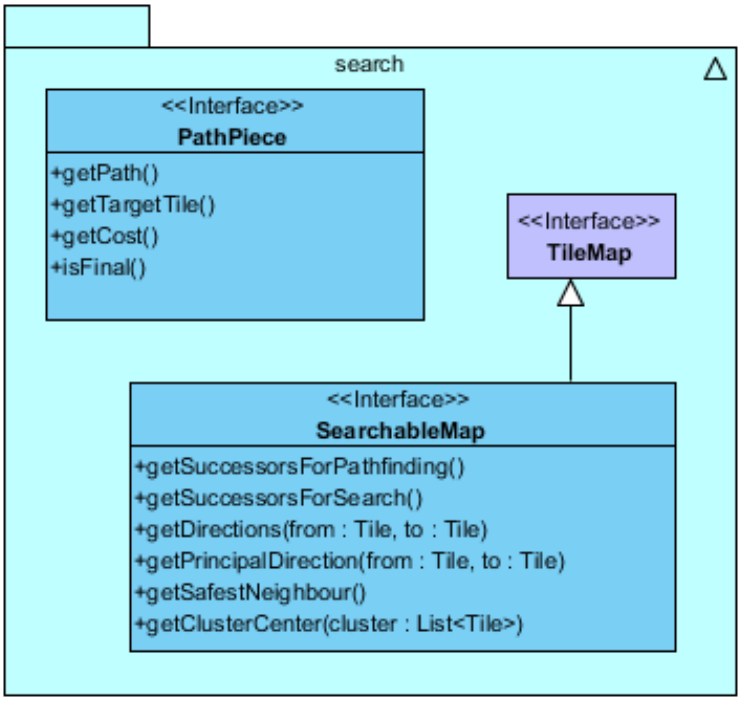


Abbildung 3.3: Search API

- **PathPiece**: Das PathPiece ist ein Interface für Strukturen, die als Suchknoten in der Pfadsuche verwendet werden können. Es definiert die für die Suche nötigen Methoden, wie getSuccessors(), getCost(), oder getPath(). Implementierende Klassen sind Edge (repäsentiert eine Kante in einem Cluster) und Tile. Erweiterungen dieser Klassen sind DirectedEdge (eine gerichtete Kante) und Vertex (eine Zelle mit zugehörigen Kanten).
- **SearchableMap**: Das Interface SearchableMap erweitert das Interface TileMap. Es beschreibt die Methoden zur Pfadsuche.
- **SearchableUnitMap**: Dieses Interface dient ausschliesslich zur Zusammenführung der beiden Interfaces UnitMap und SearchableMap.

3.4 Strategy

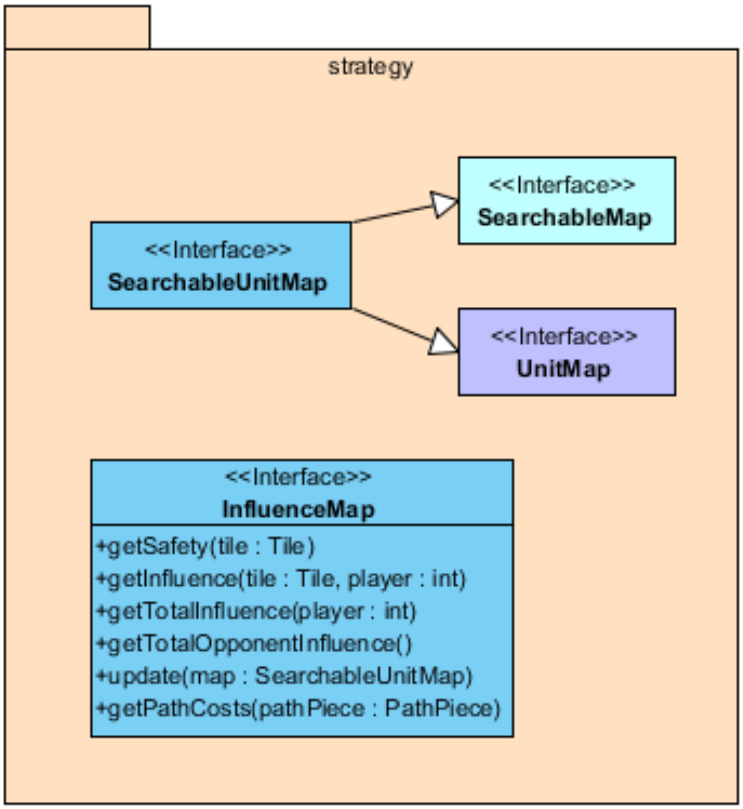


Abbildung 3.4: Strategy API

4 Suchalgorithmen

4.1 Enities für die Pfadsuche

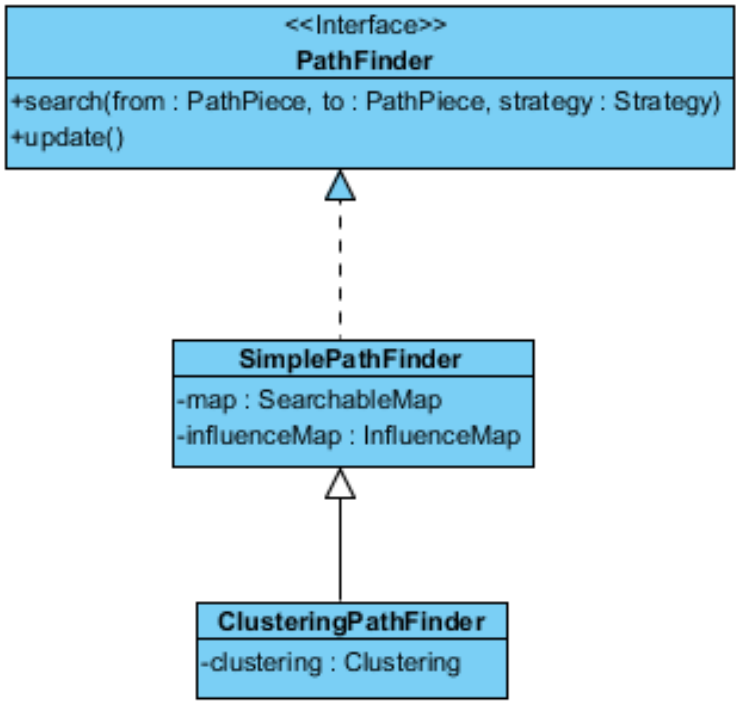


Abbildung 4.1: Klassendiagramm Pfadsuche

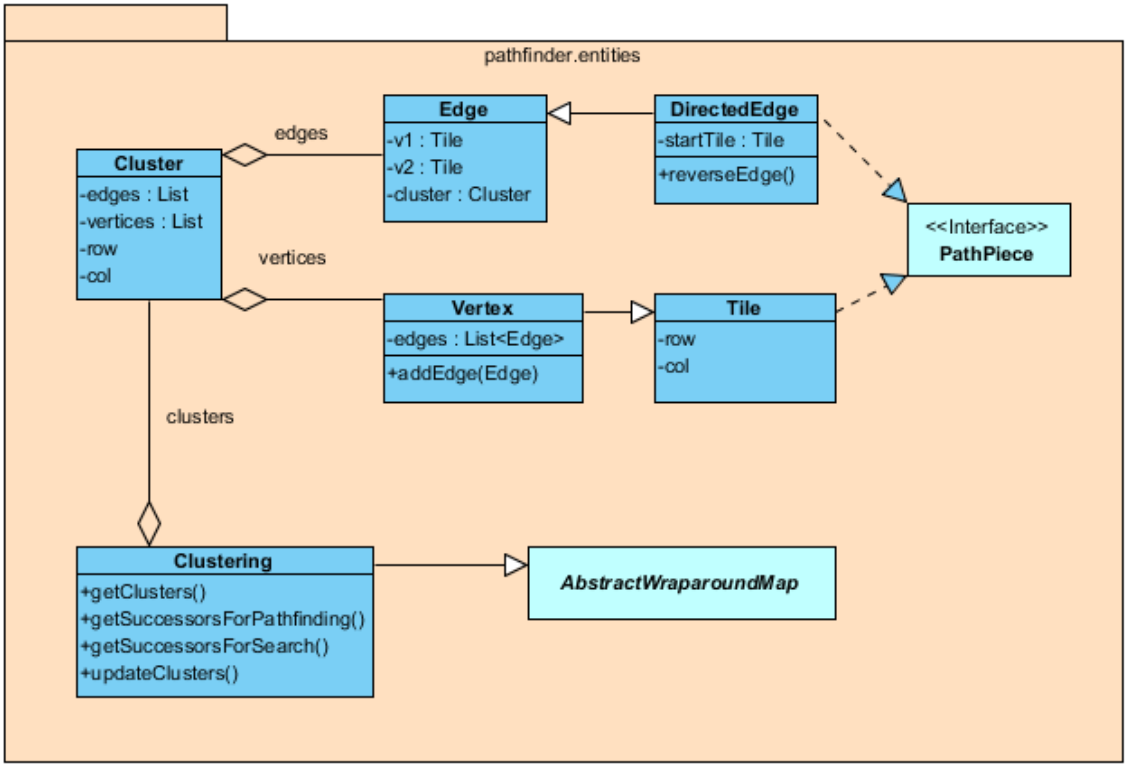


Abbildung 4.2: Spiel-Elemente für die Suche (vereinfacht)

TODO NEW IMAGE

Abbildung 4.2 zeigt die wichtigsten Klassen, die für die Pfadsuche verwendet werden. Der Übersichtlichkeit wegen wurden nur die wichtigsten Attribute und Operationen in das Diagramm aufgenommen.

4.2 Pfadsuche

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

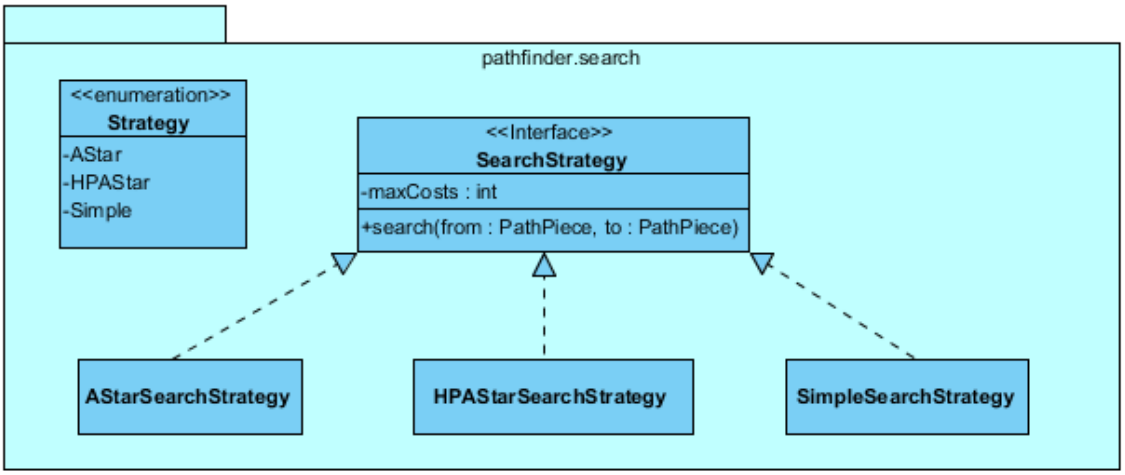


Abbildung 4.3: Suchstrategien

4.2.1 Simple Algorithmus

Der Simple Algorithmus versucht das Ziel zu erreichen indem er zuerst die eine, dann 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 via Horizontal-Vertikal Pfad gesucht. Hier wird ein Pfad gefunden. Dieser Algorithmus ist, wie der Name bereits aussagt, sehr einfach aufgebaut und kostet wenig Rechenzeit. Dafür kann er keinen Hindernissen ausweichen.

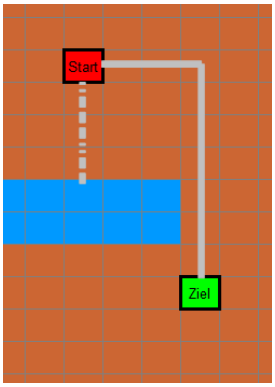


Abbildung 4.4: Simple-Path Algorithmus

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 Suchalgormithmus 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(), foodTile,16);
```

4.2.2 A* Algorithmus

Beim A* 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. 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.

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

Eine Pfadsuche mit A* wird gleich ausgelöst wie die Suche mit dem Simple-Alogirthmus, 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);
```

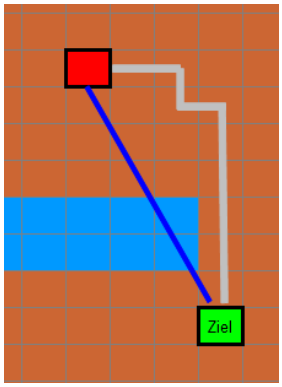


Abbildung 4.5: Heuristische Kosten (blau), Effektive Kosten (grau)

Dieser A*-Algorithmus wird in unserem Code für eine Pfadsuche über alle Pixel (jedes Pixel ist ein Node) verwendet. Der gleiche Code wir aber auch für die Pfadsuche mit dem Pfadnetz des HPA* verwendet.

4.2.3 HPA* Algorthmus

Eine Pfadsuche A* über alle Pixel ist sehr teuer, da es viel Pfade gibt, die zum Teil nur ein Pixel nebeneinander liegen. Es werden bis zum Schluss verschiedenen Pfaden nachgegangen. Abhilfe zu dieser sehr feinmaschigen Pfadsuche bietet der Hierarchical Pathfinding A* bei welchem im sogenanten Clustering über mehrere Pixel verlaufende Kanten und Knoten berechnet werden.

4.2.3.1 Clustering

Das Clustering wird während dem ClusteringTask ausgeführt, Dabei wird die Landkarte in sogenannte Clusters unterteilt. Auf dem Bild 4.6 wurde die Karte in 16 Clusters aufgeteilt.

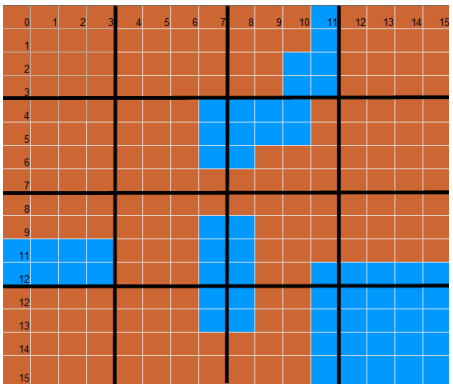


Abbildung 4.6: Clustereinteilung auf der Landkarte. Clustergrösse 4x4, Landkarte 16x16

Danach 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 Begin des Spiel nicht gegeben ist. Deshalb wird der ClusteringTask in jedem Spielzug aufgerufen, in der Hoffnung ein Cluster komplett verbinden zu können. Sobald eine beliebige Seite eines Clusters berechnet ist, wird diese Aussenkante im Cluster und dem anliegenden Nachbar gespeichert und nicht mehr neu berechnet.

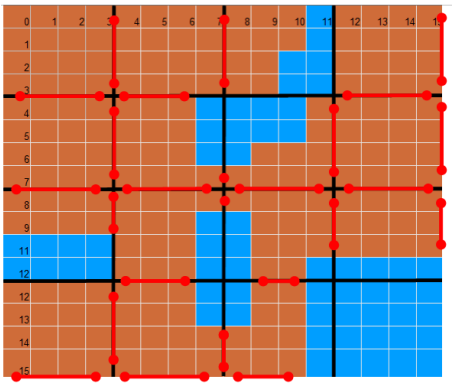


Abbildung 4.7: Die Kanten jedes Clusters wurden berechnet

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

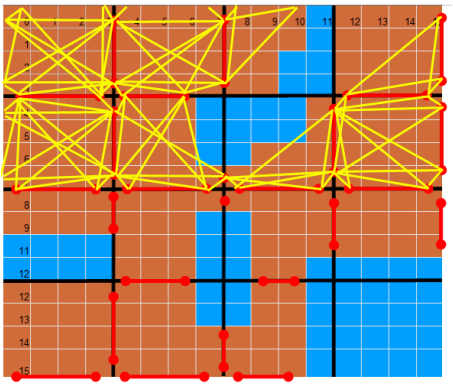


Abbildung 4.8: Darstellung der Innenkanten

In der Abbildung 4.9 wird ein Pfad vom Pixel (3,9) nach (13,9) mittels HPA* gesucht (grüne Punkte). 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 bereits beschrieben A* Algorithmus verwendet um die beiden Knoten auf dem kürzesten möglichen Pfad (gelb) zu verbinden.[1]

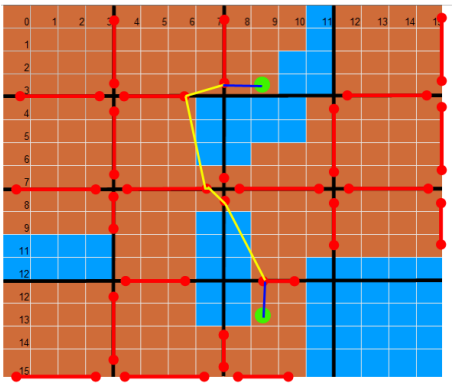


Abbildung 4.9: Errechneter Weg mittels HPA*

[1]Der resultierende Pfad könnte mittels Pathsmoothing verkürzt werden. Dies wurde aber in unserer Arbeit nicht implementiert.

Um eine Pfadsuche mit HPA* durchzuführen 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 durchgeführt werden. Falls das Clustering auf dem benötigten Kartenausschnitt nicht komplett durchgeführt werden konnte, weil nicht alle Tiles in einem Cluster sichtbar waren, wird 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 in 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 Influence Map sieh Kapitel TODO) 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 varieren. (Die Pfadkosten dürfen nicht negativ sein, sonst würde der A* Algorithmus nicht mehr korrekt funktionieren.) Die Kosten für jeden Pfadabschnitt 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;
}
```

Dadurch resultiert ein Pfad der 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 gengerischen Einheiten (blau) abgebildet. Jede Einheit trägt zur Berechnung der Influence Map bei. Pro Tile wird die Sicherheit ausgegeben, negativ für Gebiete die vom Gegner kontrolliert werden und positiv in unserem Hoheitsgebiet.

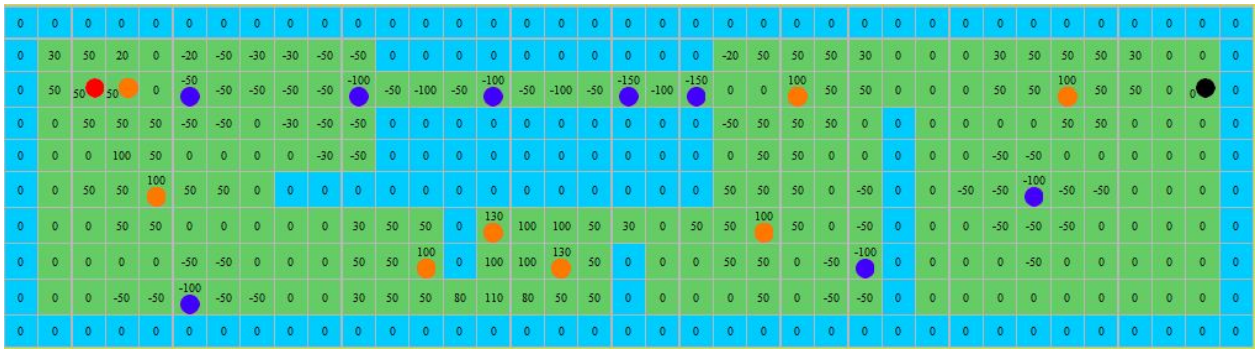


Abbildung 4.10: Ausgangslage Pfadsuche mit A* und InfluenceMap

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 direktestem Weg zum Ziel, sondern nimmt einen Umweg über sicheres Gebiet. Unten abgebildet ist der kürzeste Pfad mit Berücksichtigung der Influence Map (blau) und ohne Influence Map-Berücksichtigung (orange).

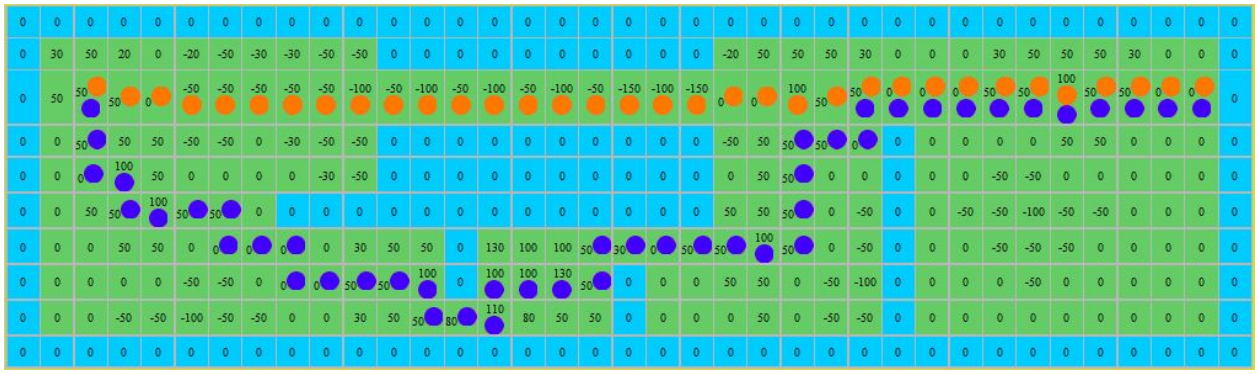


Abbildung 4.11: Resultierende Pfade mit und ohne Berücksichtigung der Influence Map

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

	Kosten ohne Influence Map	Kosten mit Influence Map
Oranger Pfad	**34**	110
BlauerPfad	46	**106**

4.3 Breitensuche

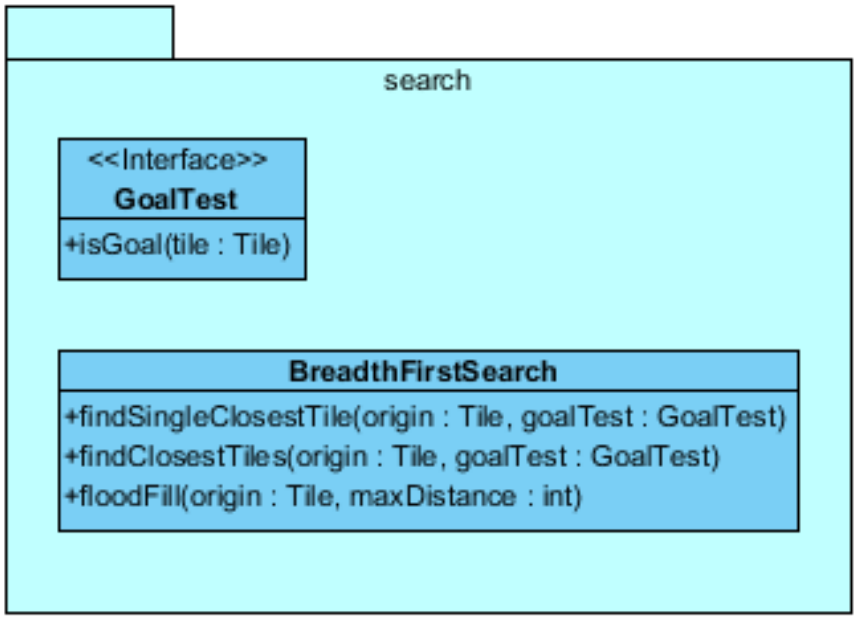


Abbildung 4.12: Breitensuche Klassendiagramm

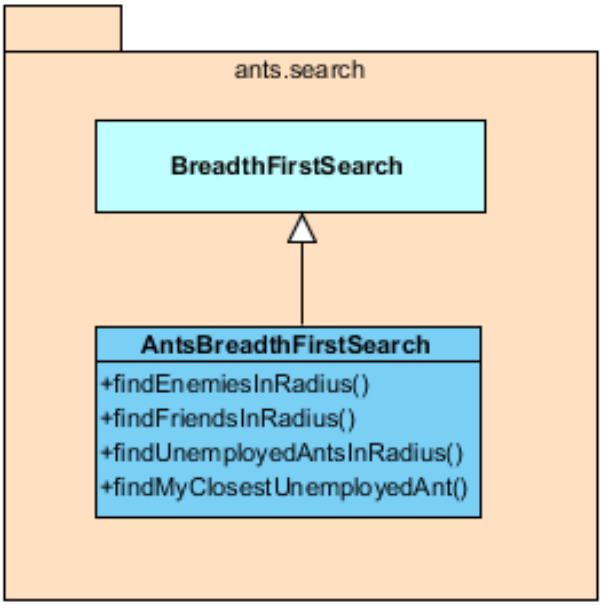


Abbildung 4.13: Breitensuche Ants-spezifisch

Die Breitensuche (engl. breadth-first search (BFS)) war eine der Neuimplementierungen während der Bachelorarbeit. Man könnte die BFS auch für die Pfadsuche verwenden, dies wäre aber sehr ineffizient. Wir verwenden diese Suche vielmehr für die Umgebung einer Ameise oder eines Hügels zu analysieren. Sie wurde generisch implementierte, so dass sie vielseitig einsetzbar ist. So können zum Beispiel mittels 'GoalTest' je nach Anwendungsfall die Tiles beschrieben werden welche gesucht sind. Folgende Breitensuche findet die Ameise welche am nächsten bei einem Food-Tile <r:20,c:16> ist. Sie 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 entspricht. Falls keien 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 Alogrithmus kann aber auch alle passierbaren Tiles in einem gewissen Umkreis zurückgeben. Dies haben wir unteranderem beim Initialisieren der DefendHillMission verwendet. Wir berechnen beim Erstellen der Mission die passierbaren Tiles rundum 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 Profile unterschiedlich eingestellt werden.

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

5 Strategie und Taktik

5.1 Influence Map

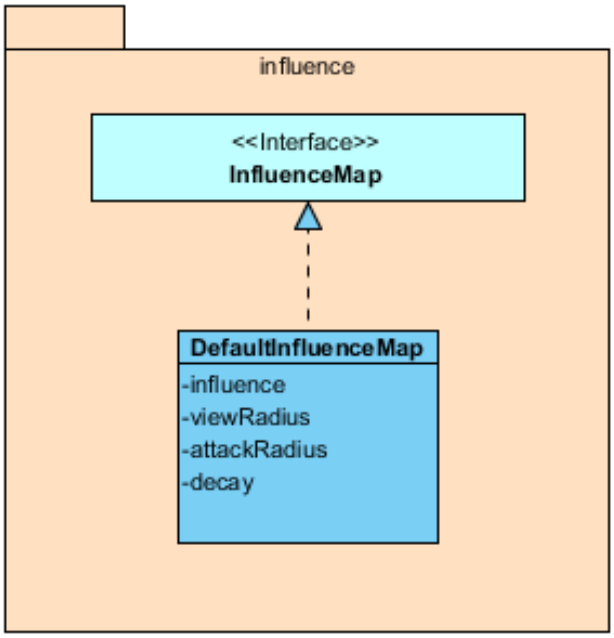


Abbildung 5.1: Influence Map Klassendiagramm

Die Influence Map haben wir nach den Beschreibungen im Buch [**?**] 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

* Die Radiuslänge kann je nach Spieleinstellung ändern. Angegeben sind die Defaultwerte.

Wir verenden die Influence Map 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.

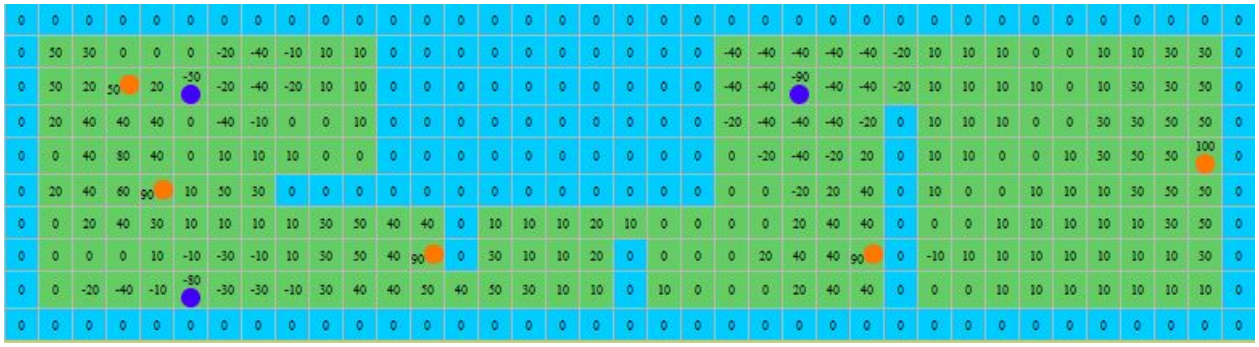


Abbildung 5.2: Influence Map, dargestellt ist die Sicherheit je Tile.

5.1.1 Update

Die Influence Map wird zu Beginn des Spiels initialisiert, danach wird vor jeder Spielrunde ein Update gemacht. Dabei definert 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 Andwendungsfälle

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

- **Pfadsuche mit Influence Map 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 Food Tile 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 Influence Map auch für weitere Entscheidungen verwenden. Auch der Einsatz von Spannungskarte (Tension Map), welche auch auf der InfluenceMap aufbaut, wäre denkbar. Dies wurde während dieser Arbeit nicht angeschaut bzw. implementiert.

5.2 Combat Situations

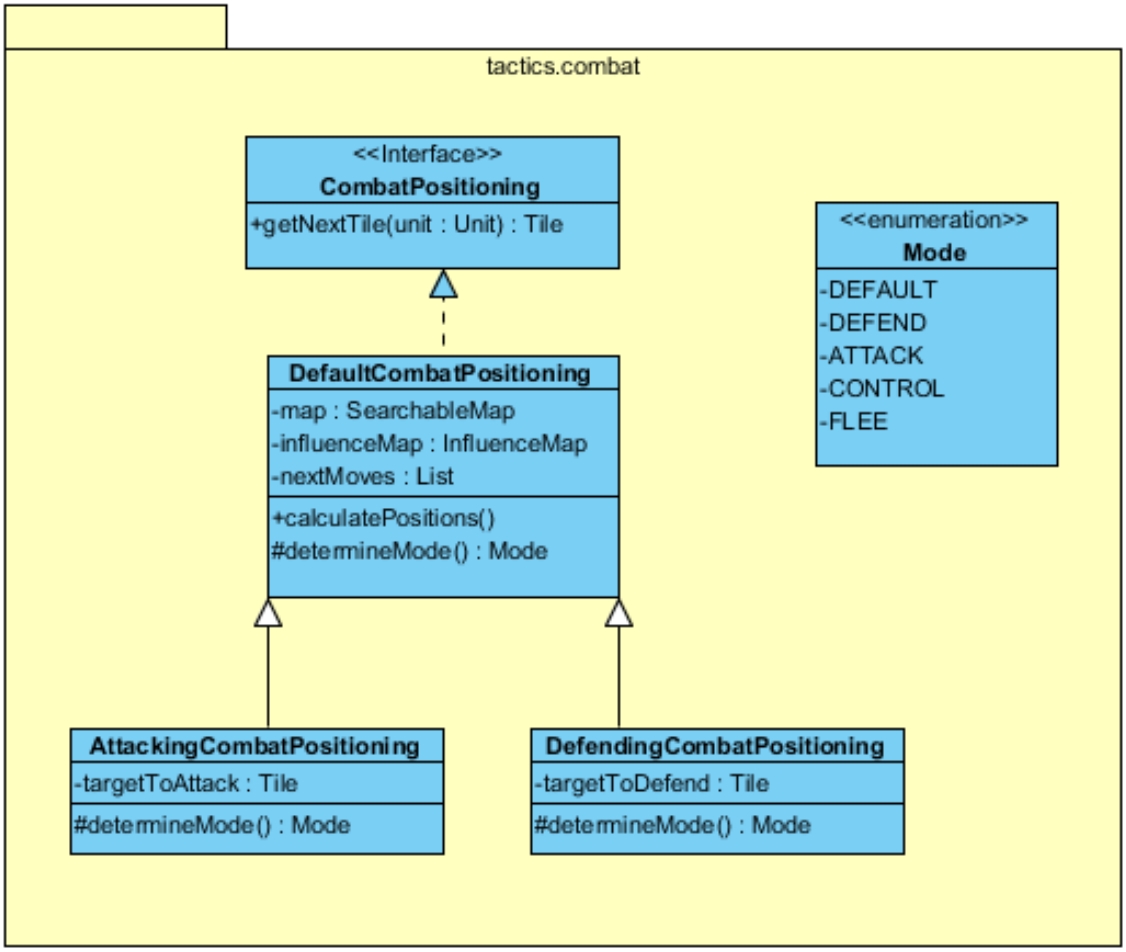


Abbildung 5.3: CombatPositioning Klassendiagramm

Kampfsiuationen werden immer dann erstellt wenn gegnerische Ameisen auf unsere Ameisen treffen. Dies ist vorallem der Fall wenn ein gegnerischer Hügel angegriffen wird, oder unserer 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 Postionierung 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.

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

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

TODO DIAGRAMM

DefendingCombatPositioning

TODO

AttackingCombatPositioning

TODO

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

5.2.1.1 FLEE

Für jede Unit wird das sicherste Nachbartile mittels Influence Map bestimmt. Die Unit verschieb sich auf das sicherste Nachbartile.

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

5.2.1.2 DEFEND

TODO

5.2.1.3 ATTACK

TODO

6 Ants

6.1 State-Klassen

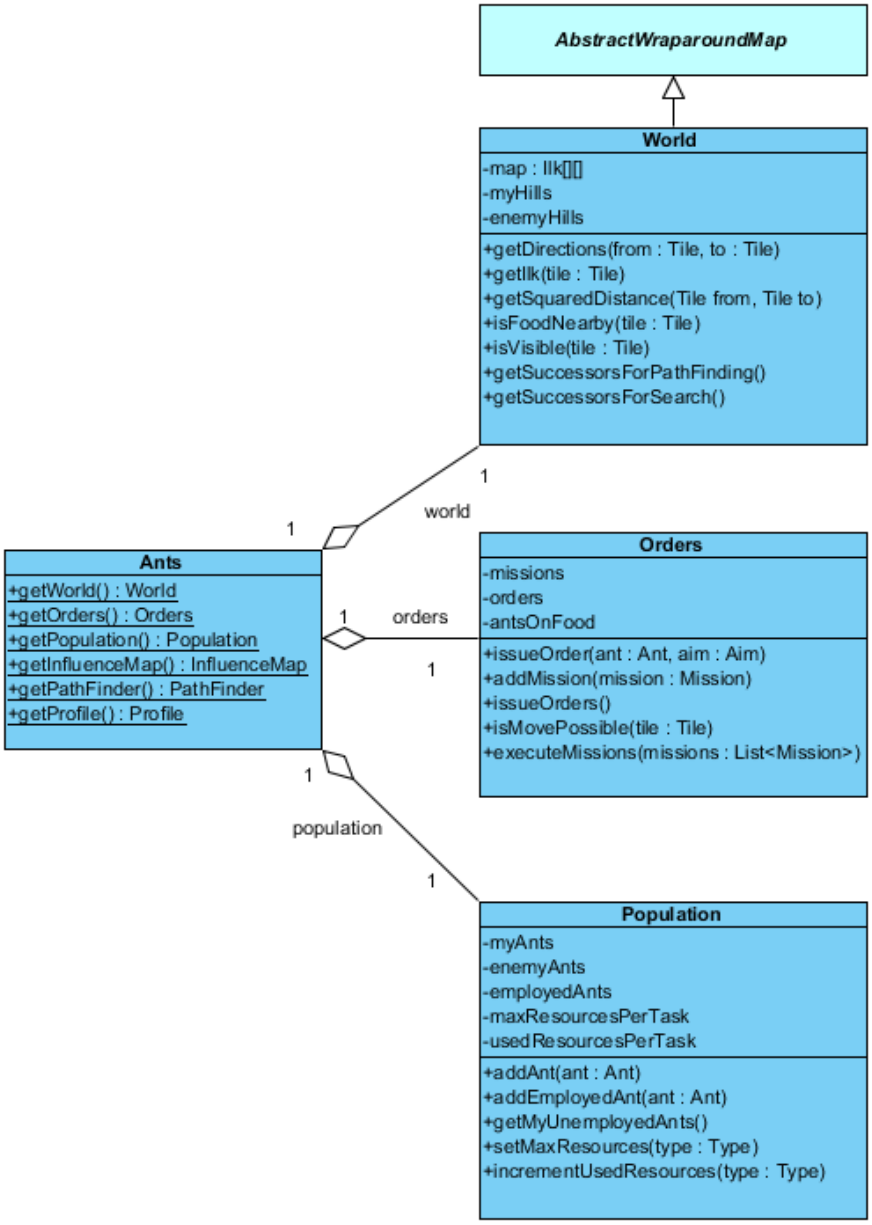


Abbildung 6.1: State-Klassen (vereinfacht)

Abbildung 6.1 zeigt eine Übersicht über die Zustands-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. 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, gibt Auskunft über einzelne Zellen und darüber, 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. Die Liste der Befehle wird dabei in jedem Zug geleert und neu befüllt, während die Liste der Missionen zugübergreifend geführt wird. 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 in mehreren Missionen 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.

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. Der Zugriff auf sie erfolgt ebenfalls über die Clustering Klasse.

6.2 Spiel-Elemente (Ants-Spezifisch)

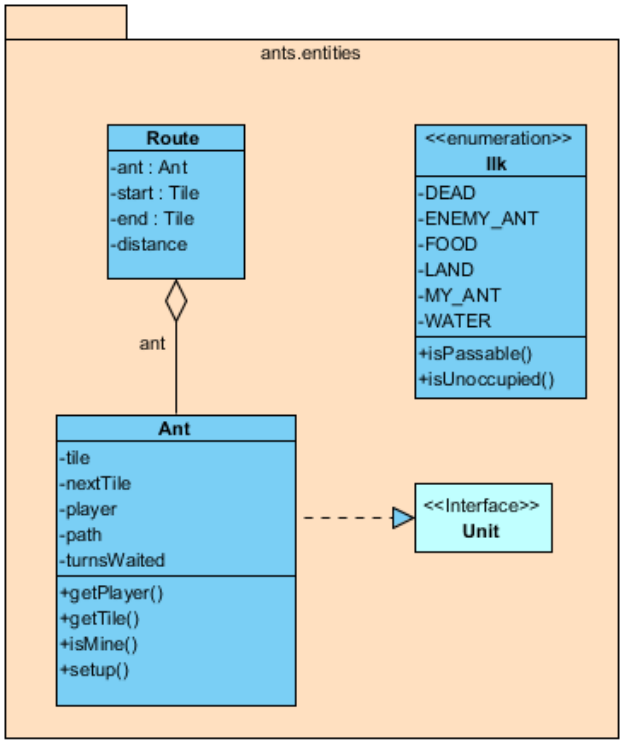


Abbildung 6.2: Ants-spezifische Elemente der Spielwelt (vereinfacht)

TODO NEW IMAGE (Searchtarget)

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 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 – das Feld wird jeweils aktualisiert, wenn der Ameise ein Befehl ausgegeben wird. Im nächsten Zug können wir dann prüfen ob die Ameise den Befehl korrekt ausführen konnte. Eine Ameise kennt auch die anderen Ameisen in ihrer Umgebung: Über die Methoden getEnemies()/FriendsInRadius() können alle bekannten Freunde und Feinde in einem bestimmten Radius ermittelt werden.

6.2.2 Route

Eine Route repräsentiert eine einfache Start-Ziel Verbindung. Sie hält für eine Ameise die Luftliniendistanz zu einem bestimmten Zielfeld fest.

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 ansonsten leer 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.

6.3 Aufbau Bot

6.3.1 Klasse Bot

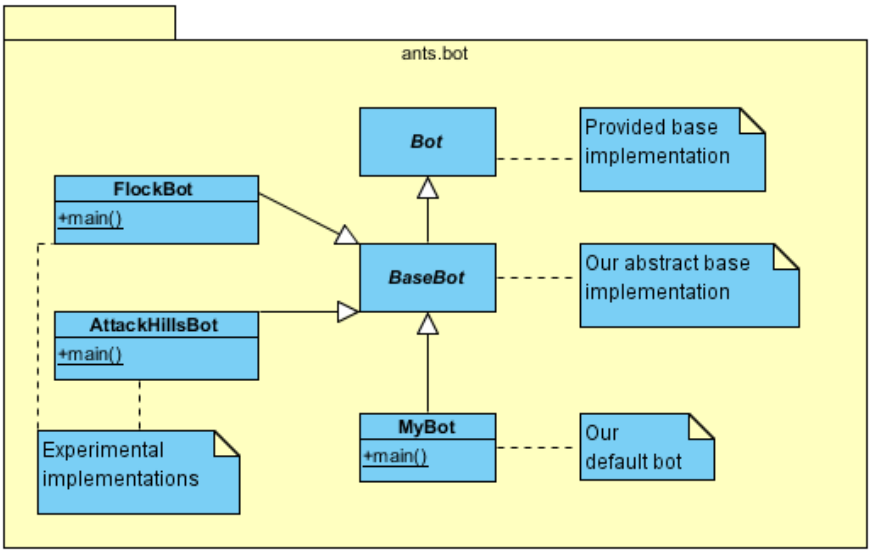


Abbildung 6.3: Vererbung der Bots wobei auf Stufe impl (Implementation) nur MyBot verwendet wird.

Als Basis für unsere Bot Implementation haben wir den Beispiel-Bot (Klasse Bot.java) verwendet, der im Java-Starter-Package enthalten ist, das von der AI-Challenge-Website heruntergeladen werden kann. Dieser erbt von den Klassen AbstractSystemInputReader und AbstractSystemInputParser, die die Interaktion mit der Spiele-Engine 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 aber noch nicht nötig. Die Klasse Bot.java dient als Grundlage für die Klasse BaseBot, welcher wiederum Grundlage ist für die Finale Klasse MyBot.java.

6.3.2 BaseBot

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

6.3.3 Ablauf eines Zugs

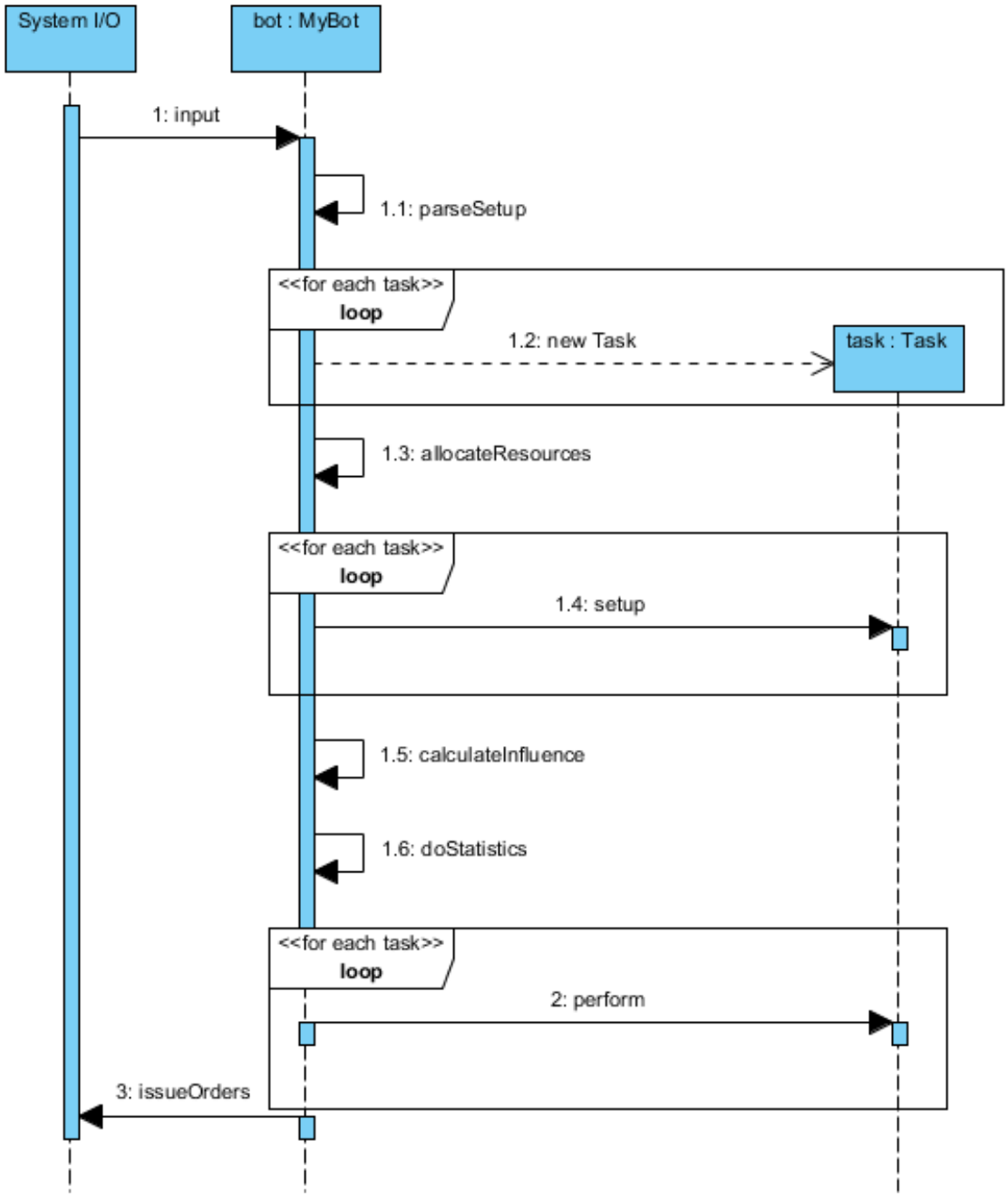


Abbildung 6.4: Ablauf des ersten Zugs des Spiels

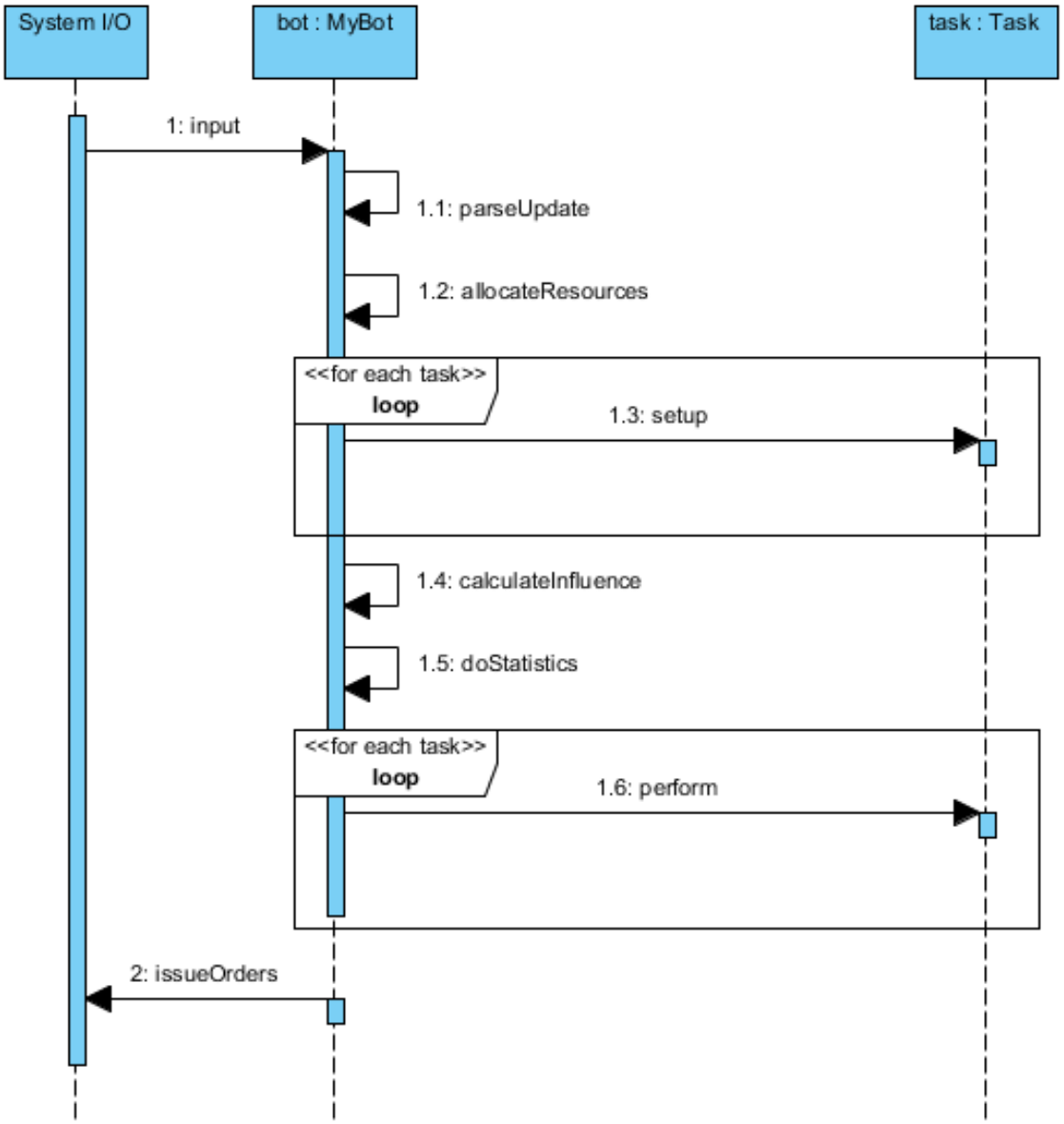


Abbildung 6.5: Ablauf der weiteren Züge des Spiels

TODO calculateInfluence() fehlt

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

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() resp. 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.

```
@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();
}
```

6.3.4 MyBot

Wie bereits erwähnt ist die Methode initTasks() in BaseBot abkstrakt 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 9.2) Weiter wird in MyBot initLogging(...) aufgerufen. Hier definieren wir welche Logkategorien mit welchem Loglevel ins Logfile geschrieben werden. Mehr zum Thema Logging ist im Kapitel Logging zu finden. Je nach Modul das gerade getestet wird können die Anzahllogeinträge justiert werden. MyBot initialisiert folgende Tasks; es sind die Tasks die sich während der Arbeit bewährt haben. Die detailierte Beschreibung der Tasks ist im Kapitel 6.4 zu finden.

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

6.4 Tasks

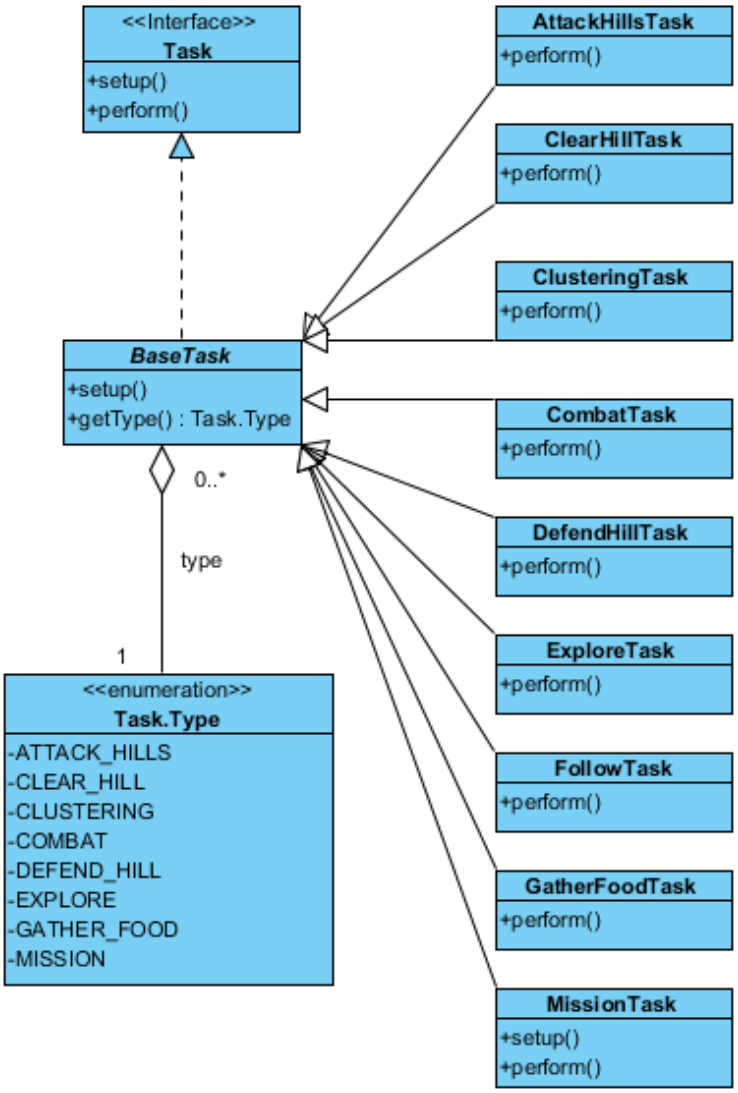


Abbildung 6.6: Tasks

Zu Beginn des Projekts haben wir die wichtigsten Aufgaben der Ameisen identifiziert. Diese Aufgaben wurden als Tasks in eigenen Klassen implementiert. Das Interface Task[1] 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 Tasks erstellen die Missionen (s. Abschnitt 6.5), welche über mehrere Züge definierte Ziele verfolgen.

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 Missionen aufgerufen werden. Diese löscht alle Ameisen

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

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.

6.4.1.2 execute()

Nun, da alle Missionen inklusive Ameisen initialisiert wurde, kännen diese ausgeführt werden. Für die Ausführugen 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 zu Beginn der Mission zugeteilt wurden. 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 nahegelegenen 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.

6.4.2 GatherFoodTask

Im ersten Zug wird genau eine GatherFoodMission erstellt, diese Mission verwaltet und koordiert die Ameisen welche auf Futtersuche sind.

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 Begin keine Ameisen enthalten.

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. Falls ein Pfad mittels Pfadsuchalgorithmus gefunden wird, wird eine ExploreMission erstellt. Die Ameise wird den gefundenen Pfad in den nächsten Spielzügen ablaufen. Sobald die ganze Spielkarte erforscht ist, schickt der Task die Ameisen in Gebiete die momentan nicht sichbar 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 durch diese 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. 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 kleine 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 in 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, 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 ihr nächster Move 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 führer abgebrochen werden muss. Ziel und Abbruchbedingungen sind je nach Mission unterschiedlich und werden im jeweiligen Abschnitt erklärt.

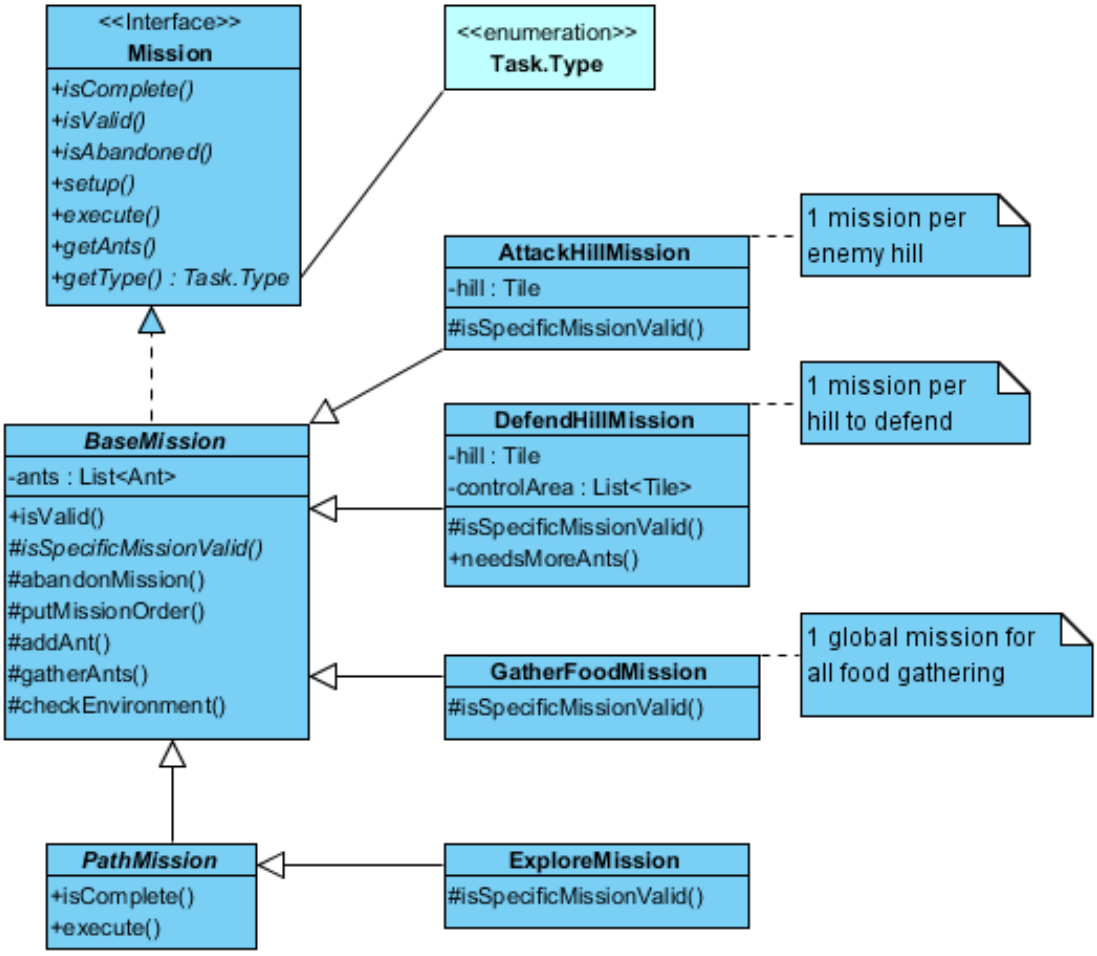


Abbildung 6.7: Missionen und ihre Hierarchie

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 spezifischen Missionen verwendet werden. Die wichtigsten 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. 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 vo 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. Tabellearisch werden die Eigenschaften der Missionen aufgelistet, danach folgen detailiierte Infomationen zur Mission.

6.5.2 PathMission

Precondition: -
Creator: (Ersteller): CombatTask, oder ExploreMission
Postcondition: Pfad vollständig abgelaufen
Max. Ants (Maximale Anzahl Ameisen): 1
Max. Missionen: unbegrenzt
Valid (Gültigkeit): siehe ExploreMission bzw. CombatTask
Gather Ants (Ameisen rektrutieren): Nicht möglich
Release Ants (Ameisen entlassen): Nicht möglich

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 sichtbar
Creator: AttackHillsTask
Postcondition: Gegnerischer Hügel erobert
Max. Ants: unbegrenzt
Max. Missionen: Je gegnerischen Hügel eine Mission
Valid: solange gegnerischer Hügel nicht zerstört ist.
Gather Ants: pro Zug max. 5 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 Unterstüztung braucht. Wenn Mission im Status ControlHill ist werden alle Ameisen entlassen ausser zwei. Diese kontrollieren den Hügel.

Diese Mission unterscheidet zwischen drei verschiedenen Modi. Der Modus wird zu Beginn jeder Runde durch die Methode determineState() definiert.

ControlEnemyHill
Status ist aktiv wenn, 2 oder mehr eigene Ameise in der Nähe des gegnerischen Hügel sind. Der Gegner aber nur eine Amesise zur Verteidigung hat.
Hier werden alle Ameisen ausser die zwei zum gegnerischen Hügel am nächsten liegenden Ameisen aus der Mission entlassen. Die übrigen zwei Ameisen positionieren sich auf einer der vier diagonal gelegenen Zellen. Soblad die Ameisen positioniert sind, fressen Sie alle gegnersichen Ameisen die neu aus dem Hügel schlüpfen. 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.

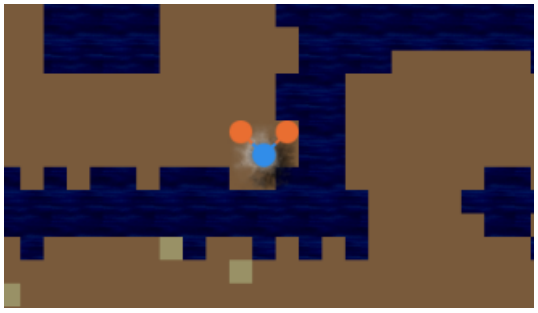


Abbildung 6.8: Ein gegnerischer Hügel wird durch unsere Ameisen (orange) kontrolliert.

DestroyHill
Status ist aktiv wenn, 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 Richtung gegnerischen Hügel geschickt werden.

AttackEnemyHill
Status ist aktiv in allen anderen Fällen
Alle Angreifer werden nach ihrer Lage in Gruppen zusammen gefasst. Für diese Gruppe wird ein Pfad zum gegnerischen Hügel berechnet. Falls der Pfad länger ist als 5 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 initialisert und die Gruppe, die gegnerischen Ameisen sowie der Meilenstein als Ziel mitgegeben. Die Klasse berechnet die Züge für die Gruppe. (Details siehe 5.2

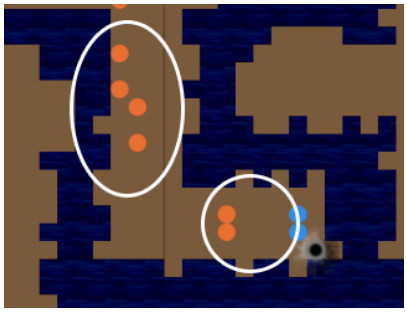


Abbildung 6.9: Die erste zweier Gruppe ist in Kampfstellung. 4 weitere Ameisen rücken zur Front auf.

6.5.4 DefendHillMission

Precondition: Keine, Mission wird zu Beginn des Spiels erstellt.
Creator: DefendHillTask
Postcondition: Gegnerischer Hügel ist erobert.
Max. Ants: unbegrenzt
Max. Missionen: Je eigener Hügel eine Mission
Valid: solange eigener Hügel nicht erobert ist.
Gather Ants: Mission soll immer mehr Verteidiger haben als Angreifer sich dem Hügel nähern.
Release Ants: Keine Angreifer in Sicht, werden die Ameisen (nicht alle) entlassen.

TODO

6.5.5 ExploreMission

Precondition: -
Creator: ExploreTask

Postcondition: Definerter Pfad zum Erkunden ist abgelaufen.
Max. Ants: 1
Max. Missionen: unbegrenzt
Valid: Solange Ameise ein gültiger Pfad hat.
Gather Ants: nicht möglich
Release Ants: nicht möglich

Die Ameise läuft den vom ExploreTask vorgegebenen Pfad ab. Die Mission wird 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 6.10: Eine Ameise auf einer ExploreMission bewegt sich in Richtung Fog Of War

6.5.6 GatherFoodMission

Precondition: Keine, Mission wird zu Beginn des Spiels erstellt.
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

Es wird nur eine GatherFoodMission zu Begin des Spiels erstellt, diese Mission erstellt und kontrolliert die Pfade der Ameisen, welche auf Futtersuche sind. 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
    gatherAnts();
    // move the ants
    moveAnts();
}
```

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 unültigen Pfad werden von der Mission entlassen, können aber in der nächsten Methode *gatherAnts()*, falls sinnvoll, der Mission wieder beitreten. Die Logik der gatherAnts() Method ist in folgenden Codebeispiel vereinfacht dargestellt.

```
foreach(foodTile on map){
```

```
if(there is an ant in food mission, which is gathering this food tile with a path smaller than 5)
continue;
Ant a = getNearestAntWithBreadthFirstSearch();
if(no Ant found)
continue;
else
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);
}
```

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.

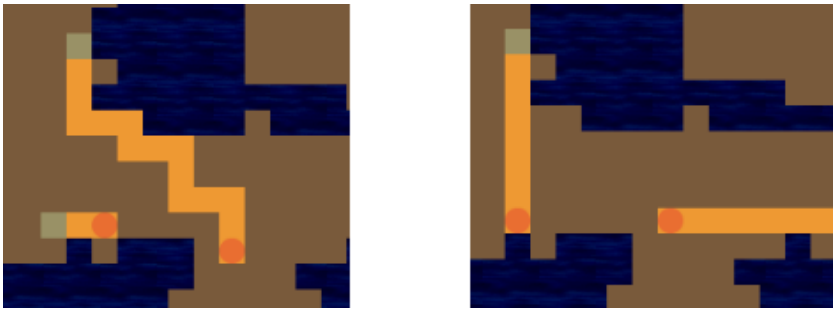


Abbildung 6.11: Ameisen am Futter einsammeln.

Links ist zu sehen für beide Ameisen ein Futterpfad definiert ist, die Ameisen folgen dem Pfad. Zwei Züge später (rechtes Bild) hat die Ameise unten 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 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 Ressourcen Management

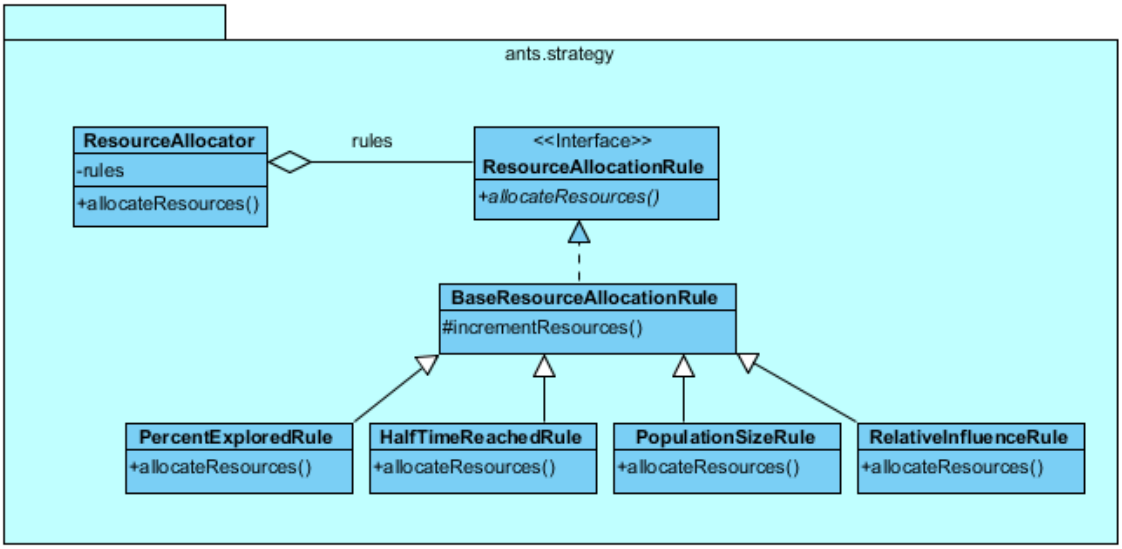


Abbildung 6.12: RessourcenManagement

Abbildung 6.12 zeigt die Klassenhierarchie unserer Ressourcenverwaltung.

6.7 Profile

7 Logging

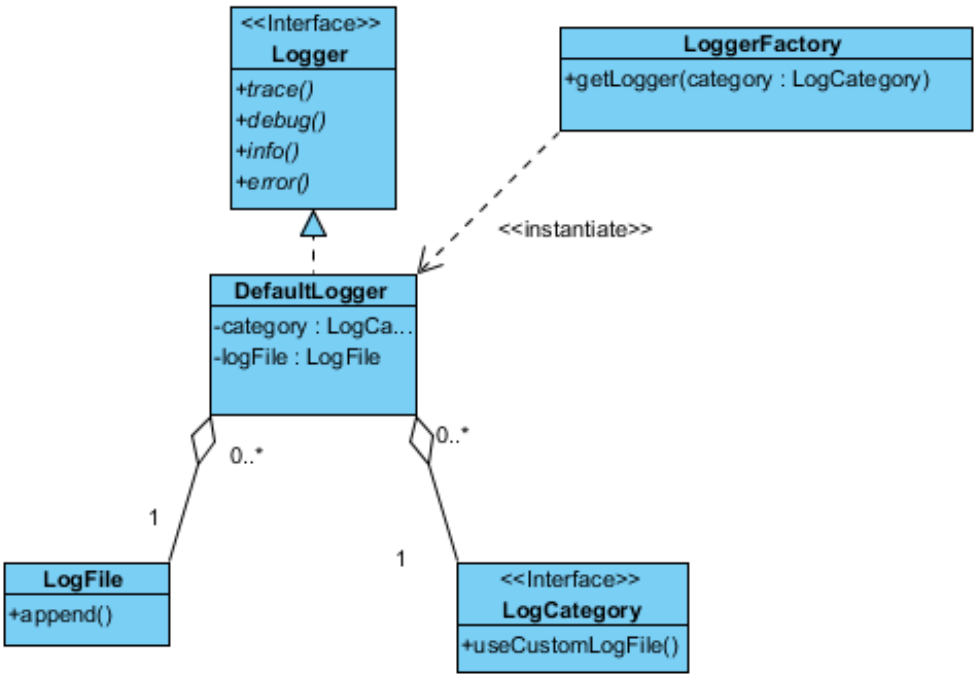


Abbildung 7.1: Logging Klassen

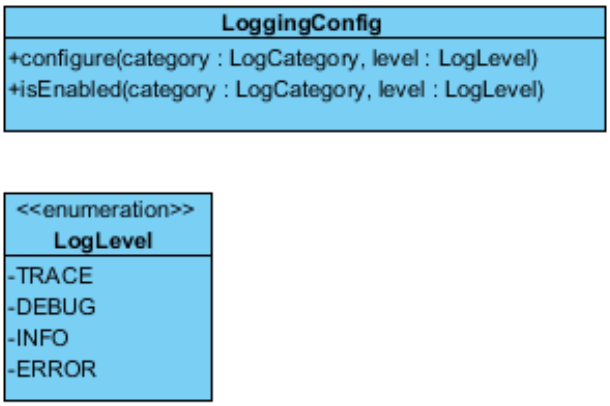


Abbildung 7.2: Logging Konfiguration

Nach einem absolvierten Spiel analysierten wir jeweils die Spielsituationen, welche sich ergeben haben. Dazu gehörte das Analysieren des geschriebenen Logs. Dabei bedienten wir uns den nachfolgenden Mechanismen.

7.1 Logkategorien und Loglevel

Jeder Logeintrag gehört einer Logkategorie an. Je Logkategorie kann der Loglevel defniert werden. Die Loglevel lauten TRACE, DEBUG, INFO und ERROR. Wenn also zum Beispiel bei der Logkategorie ATTACKHILLMISSION der Loglevel auf INFO gestellt ist, werden nur die Fehler auf Stufe INFO und ERROR in das Logfile geschieben. Zudem kann, falls erwünscht, jede Logkategorie in ein eigenes Logfile geschrieben werden. Die meisten Module habe ihre eigene Logkategorie, so kann durch korrekte Logeinstellung erzwungen werden dass nur die Logs, welche für das Analysieren eines bestimmten Spielmoduls von Bedeutung sind, ins Logfile geschrieben werden. Dadurch müssen nicht riesige Mengen an Logs druchwälzt werden um an die Informationen heran zu kommen.

7.2 JavaScript Addon für HMTL-Gameviewer

TODO Der Aufruf ist jetzt LiveInfo.live... 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 Logger.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 Infomation angezeigt werden soll. Im Beispiel wird an der Position der Ameise (ant.getTile()) ausgegeben welchen Task die Ameise hat.

```
Logger.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.

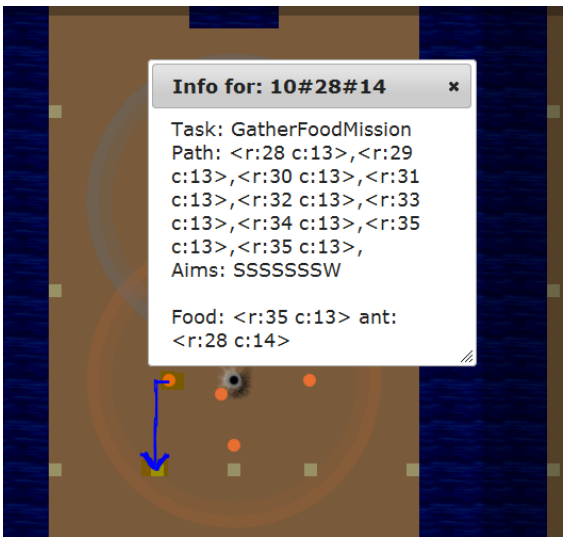


Abbildung 7.3: Im Popupfenster steht die Aufgabe der Ameise sowie die Pixel des Pfades (falls vorhanden), welcher die Ameise ablaufen wird.

Das angezeigte Popup zeigt welchen Task (GatherFoodTask) die Ameise hat, wo sie sich befindet <r:28 c:14>, welches Futterpixel 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. **??**)

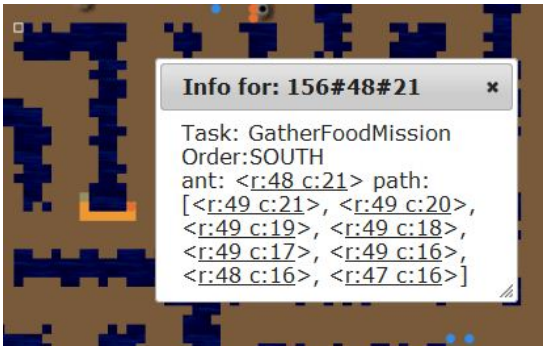


Abbildung 7.4: Mit der erweiterten Version wird der Pfad (orange) der Ameise von <r:48 c:21> nach <r:47 c:16> auf der Karte abgebildet.

8 Testreader

9 Testing

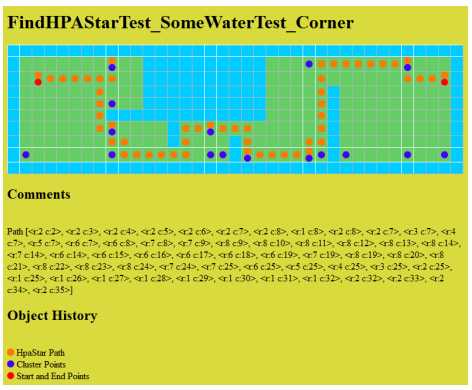
Dieses Kapitel beschreibt wie wir unseren Bot und dessen Komponenten getestet haben. Das Testen beanspruchte ein grosser Teil unserer aufgewendeten Zeit. Die nachfolgenden Methoden zeigen auf wie wir versuchten möglichst effizient zu testen, so dass uns mehr Zeit für das implementieren von Modulen blieb. 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.

9.1 Unit Tests

Danke dem modularen Aufbau des Java-Codes war es uns möglich einzelne Module zu testen. Für unsere UnitTests verwendeten wir die JUnit 4 Library. So hat jedes Java Project ein UnitTest-Package. Der Code im Java Project wird durch diverse Testklassen im UnitTest-Package auf die Richtigkeit geprüft. Die hat den Vorteil, das wir funktionierende Module in den Bot einbauen konnten, und nicht erst nach dem Einbau nach Fehlern suchen mussten. Um das Modul CombatPositioning (siehe im Code unter Startegy.tactics.combat) 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 geschrieben. Danach wurde solange an CombatPositioning programmiert, bis das erwartete Testresultat eintraf.

9.2 Visuelle Tests

In Form eines UnitTest haben wir auch unsere visuellen Tests geschrieben. Durch eine visuelle Überprüfung des Resulats ist meistens, vorallem bei der Pfadsuche, einfacher zu kontrollieren. Anstatt aber mit der JUnit Methode *assertEquals()* das Resultat zu prüfen 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.) mittels farbige Punkte dargestellt werden. Diese Funktionalitäten bietet die extra geschriebene Klasse MapOutput. Nachfolgend ist ein HTML-File abbgebildet mit welchem wir die Korrektheit des Clustering und des HPA* Algorithmus visuelle überprüft haben.



9.3 Testbots

Eine weitere Method war Testbots zu erstellen. Zum Beispiel haben wir den DefendHillBot erstellt der nur verteidigt nicht aber angreift. Wir nahmen eine kleine Karte, so dass es schnell zu Angriffen des Gegners kam. So konnten

wir unser Verhalten in der Verteidigung nach kurzer Spieldauer genau analysieren und verbessern.

Das selbe galt für den AttackHillBot. Wir haben wiederum eine kleine Karte genommen auf der viel Futter vorhand war, so dass sich das Ameisenvolk schnell vermehren konnte. Dank der beschränkten Karte kam es schnell zu Angriffen, wir konnten unsere Angriffspositionierung testen.

Im Kapitel Task wurde beschrieben, welche Task bzw. Missionen nicht erfolgsversprechend waren und wir nicht weiterverfolgt haben. Bevor wir das aber wussten, haben wir, um die Funktionen zu testen, 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

9.4 Testreport Profile

10 Spielanleitung

Dieser Anhang beschreibt kurz, wie ein Spiel mit unserem Bot ausgeführt werden kann.

Das File `Ants.zip` enthält das Eclipse-Projekt mit dem gesamten Source-Code unserer Implementation, und der offiziellen Spiel-Engine. Zum einfachen Ausführen eines Spiels haben wir ein ANT-Buildfile (build.xml) erstellt. Dieses definiert 3 Targets, mit denen ein Spiel mit jeweils unterschiedlichen Parametern gestartet werden kann.

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. Das Target `maze` führt ein Spiel auf einer komplexeren und grösseren, labyrinthartigen Karte aus und ist das interessanteste von den 3.

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. Viele der Teilnehmer haben zudem ihren Quellcode auf dem Internet publiziert, an möglichen Gegner besteht also auch kein Mangel.

Bibliography