



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: 4. 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 beherrscht. 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 4. Januar 2013

Name Vorname Lukas Kuster

Unterschrift

Name Vorname Stefan Käser

Unterschrift





Inhaltsverzeichnis

1	Einleitung	1
1.1	Projektverlauf	1
1.2	Projektorganisation	2
1.2.1	Beteiligte Personen	2
1.2.2	Projektmeetings	2
1.2.3	Dokumentation	2
1.2.4	Abgabe	3
1.3	Ziele	3
1.3.1	Funktionale Anforderungen	3
1.3.1.1	Musskriterien	3
1.3.1.2	Kannkriterien	4
1.3.2	Nicht funktionale Anforderungen	4
1.3.2.1	Musskriterien	4
1.3.2.2	Kannkriterien	4
1.4	Herausforderungen	4
1.4.1	Module testen	4
1.4.2	TODO	4
1.4.3	Vergleich mit Bots aus dem Wettbewerb	5
1.5	Fazit	5
1.6	Spielbeschreibung	5
1.6.1	Der Wettbewerb	5
1.6.2	Spielregeln	5
1.6.3	Schnittstelle	5
1.7	Abgrenzungen	6
2	Architektur	7
2.1	Modulabhängigkeiten	7
2.2	Sequenzdiagramme	8
3	API	9
3.1	Entities	9
3.2	Map	10
3.3	Search	11
3.4	Strategy	12
4	Suchalgorithmen	13
4.1	Entities für die Pfadsuche	13
4.2	Pfadsuche	14
4.2.1	Simple Algorithmus	15
4.2.2	A* Algorithmus	15
4.2.3	HPA* Algorithmus	16
4.2.3.1	Clustering	16
4.2.4	Pfadsuche mittels Influence Map	18
4.3	Breitensuche	19
5	Strategie und Taktik	21
5.1	Influence Map	21
5.1.1	Update	22
5.1.2	Anwendungsfälle	22



5.2	Combat Situations	23
5.2.1	DefaultCombatPositioning	23
5.2.1.1	FLEE	24
5.2.1.2	DEFEND	24
5.2.1.3	ATTACK	24
6	Ants	25
6.1	State-Klassen	25
6.1.1	Ants	26
6.1.2	World	26
6.1.3	Orders	26
6.1.4	Population	26
6.1.5	Clustering	26
6.2	Spiel-Elemente (Ants-Spezifisch)	27
6.2.1	Ant	27
6.2.2	Route	27
6.2.3	Ilk	27
6.3	Aufbau Bot	28
6.3.1	Klasse Bot	28
6.3.2	BaseBot	28
6.3.3	Ablauf eines Zugs	29
6.3.4	MyBot	31
6.4	Tasks	32
6.4.1	MissionTask	32
6.4.1.1	setup()	32
6.4.1.2	execute()	33
6.4.2	GatherFoodTask	33
6.4.3	DefendHillTask	33
6.4.4	AttackHillsTask	33
6.4.5	CombatTask	33
6.4.6	ExploreTask	33
6.4.7	ClearHillTask	34
6.4.8	ClusteringTask	34
6.4.9	Verworfenne und nicht verwendete Tasks	34
6.5	Missionen	34
6.5.1	BaseMission	35
6.5.2	PathMission	36
6.5.3	AttackHillMission	36
6.5.4	DefendHillMission	37
6.5.5	ExploreMission	37
6.5.6	GatherFoodMission	37
6.5.7	Verworfenne und nicht verwendete Mission	37
6.6	Ressourcen Management	38
6.7	Profile	38
7	Logging	39
7.1	Logkategorien und Loglevel	40
7.2	JavaScript Addon für HMTL-Gameviewer	40
8	Testreader	43
9	TestCenter	45
9.1	Unit- und Funktionstests	45
9.2	Testbots	45
9.3	Testreport Profile	45
10	Spielanleitung	47







Abbildungsverzeichnis

1.1	Projektablauf	2
2.1	Module	7
2.2	Modulabhängigkeiten	7
3.1	Entities	9
3.2	Map API	10
3.3	Search API	11
3.4	Strategy API	12
4.1	Klassendiagramm Pfadsuche	13
4.2	Spiel-Elemente für die Suche (vereinfacht)	14
4.3	Suchstrategien	14
4.4	Simple-Path Algorithmus	15
4.5	A* Pfadsuche	16
4.6	Clustereinteilung auf der Landkarte.	16
4.7	Cluster mit berechneten Kanten	17
4.8	Cluster mit Innenkanten	17
4.9	Errechneter Weg mittels HPA*	17
4.10	Ausgangslage Pfadsuche mit A* und InfluenceMap	18
4.11	Resultierende Pfade mit und ohne Berücksichtigung der Influence Map	19
4.12	Breitensuche Klassendiagramm	19
4.13	Breitensuche Ants-spezifisch	20
5.1	Influence Map Klassendiagramm	21
5.2	Influence Map, dargestellt ist die Sicherheit je Tile.	22
5.3	CombatPositioning Klassendiagramm	23
6.1	State-Klassen (vereinfacht)	25
6.2	Ants-spezifische Elemente der Spielwelt (vereinfacht)	27
6.3	Vererbung der Bots wobei auf Stufe impl (Implementation) nur MyBot verwendet wird.	28
6.4	Ablauf des ersten Zugs des Spiels	29
6.5	Ablauf der weiteren Züge des Spiels	30
6.6	Tasks	32
6.7	Missionen und ihre Hierarchie	35
6.8	Ein gegnerischer Hügel wird durch unsere Ameisen (orange) kontrolliert.	36
6.9	RessourcenManagement	38
7.1	Logging Klassen	39
7.2	Logging Konfiguration	39
7.3	Live-Info Popupfenster	40
7.4	Erweiterung des Live-Info Popupfenster	41





1 Einleitung

Im Rahmen des Moduls "Projekt 2" (7302) haben wir uns mit der Implementierung eines Bots für den Online-Wettbewerb AI-Challenge (Ants) beschäftigt. 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.

Wir hatten uns zum Ziel gesetzt, einen Bot zu implementieren, der möglichst alle Bereiche des Spiels beherrscht, also Nahrung sammeln, die Gegend entdecken, Hügel erobern und gegen feindliche Ameisen kämpfen. Im Gegenzug legten wir kein besonderes Gewicht darauf, dass der Bot eines dieser Verhalten besonders gut beherrschen muss. Das primäre Ziel war es, Erfahrungen zu sammeln im Hinblick auf die Bachelor-Arbeit.

Den grössten Aufwand bei der Implementierung steckten wir in die Pfadsuche, da diese eine Voraussetzung für nahezu alle Teil-Aufgaben des Bots ist. Nachdem wir mit dem bekannten A*-Algorithmus zwar kleine Erfolge erzielten, aber auch schnell Performance-Probleme bekamen, entschlossen wir uns, auf Basis eines Clustering des Spielfeldes den HPA*-Algorithmus umzusetzen. Den Algorithmus konnten wir erfolgreich implementieren, aber leider fehlte uns gegen Ende des Projekts dann die Zeit, ihn noch zu optimieren. Die Performance ist aktuell vergleichbar mit der A*-Pfadsuche.

Ein weiterer Punkt, auf den wir viel Wert legten, war die Programmstruktur. Unser Bot ist Objektorientiert aufgebaut; die zentrale Einheit sind die verschiedenen Tasks, die jeweils für eine Aufgabe der Ameisen zuständig sind.

1.1 Projektverlauf

Die Projektarbeit richtete sich nach folgendem Zeitplan:

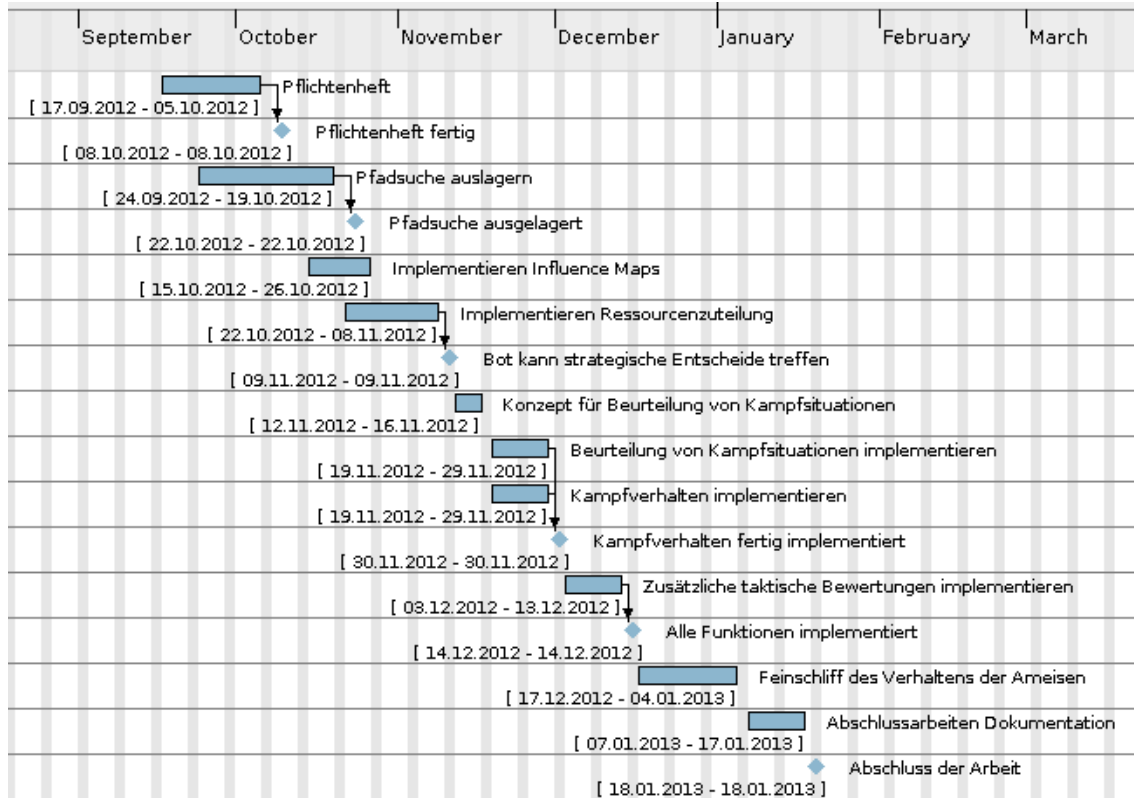


Abbildung 1.1: Projektablauf

1.2 Projektorganisation

1.2.1 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

1.2.2 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.

1.2.3 Dokumentation

Die Dokumentation soll sich am Aufbau und Inhalt des Berichts aus dem Projekt 2 anlehnen.

- Das Dokument beschränkt sich auf das Wesentliche.
- Verwendete AI-Techniken werden erläutert



- Entscheidungen und deren Grundlagen sind dokumentiert.
- Testberichte dokumentieren die durchgeführten Modultests.
- Klassendiagramme sollen einen oberflächlichen Detaillierungsgrad haben, so dass das Wichtigste auf den ersten Blick sichtbar ist.
- Anleitung zum Ausführen eines Spiels

1.2.4 Abgabe

Folgende Lieferobjekte werden am Ende der Arbeit abgegeben.

- Dokumentation
- Sourcecode

1.3 Ziele

Nachfolgend sind die Ziele aufgelistet welche wir uns vor der Arbeit gestellt und im Pflichtenheft niedergeschrieben haben. Die farbigen Pfeile zeigt den Erfüllungsgrad an. Ist ein Ziel nicht vollständig erreicht wird in *kursiver* Schrift ein Grund angegeben.

➤ Vollständig Erfüllt

➡ Teilweise Erfüllt

➤ Nicht Erfüllt

Der im Rahmen von Projekt 2 entwickelte Bot soll um Logik für taktische und strategische Entscheidungen und koordinierte Bewegung erweitert werden.

1.3.1 Funktionale Anforderungen

1.3.1.1 Musskriterien

➤ Der Bot unterscheidet zwischen diversen Aufgaben:

- Nahrungsbeschaffung
- Angriff
- Verteidigung
- Erkundung

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.
Grund: Keine Zeit für blabal...
- ➡ Schwachstellen in der gegnerischen Verteidigung ausnutzen.
- ➤ Engpässe im Terrain sichern bzw. versperren.
- Konflikzonen, d.h. viele Ameisen auf einem engen Raum, erkennen und entsprechend reagieren.

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

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

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.

1.3.1.2 Kannkriterien

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

1.3.2 Nicht funktionale Anforderungen

1.3.2.1 Musskriterien

Modularer Aufbau für eine gute Testbarkeit der Komponenten.

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.

Die Codedokumentation ist vollständig und dient der Verständlichkeit.

1.3.2.2 Kannkriterien

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

1.4 Herausforderungen

1.4.1 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 7) Zudem können wir dank der Erweiterung des HTML-Viewer sofort sehen, welches die akute Aufgabe jeder einzelnen 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)

1.4.2 TODO

lorem ipsum mehr herausforderungen ??



1.4.3 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 wage Einschätzung machen wie stark unser Bot ist. Mehr dazu unter siehe 9.3.

1.5 Fazit

1.6 Spielbeschreibung

1.6.1 Der Wettbewerb

Die AI Challenge¹ ist ein internationaler Wettbewerb 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.

1.6.2 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 Ameisenhägel eine neue Ameise.
- Die Landkarte besteht aus passierbaren Landpixel 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.
- Ein Gegner ist ausgeschieden wenn alle seine eigenen Ameisenhägel vom Gegner vernichtet wurden. Pro verlorenem Hügel gibt 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>

1.6.3 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 Bot² ab. Ein Spielzug wird im folgendem Format in den Output-Stream gelegt:

o <Zeile> <Spalte> <Richtung>

Beispiel:

¹<http://www.aichallenge.org>

²Die Klasse ist im Code unter ants.bot.Bot.java auffindbar



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.

1.7 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 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 und taktischen Entscheidungen erweitert
MinMax-Algorithmus	Bei den Kampfsituationen wurde versucht den Min-Max Algorithmus einzubauen, welcher wir bereits im Modul Spieltheorie erarbeitet haben. (Mehr dazu siehe Kapitel Strategie)

TODO EINHEITLICHE TABELLEN FORMATIERUNG



2 Architektur

2.1 Modulabhängigkeiten

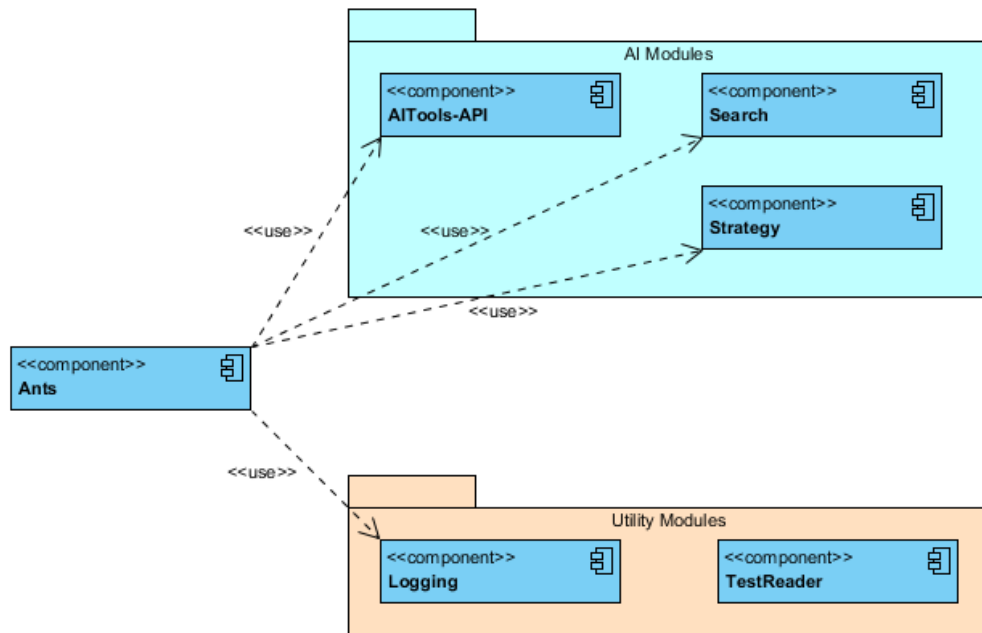


Abbildung 2.1: Module

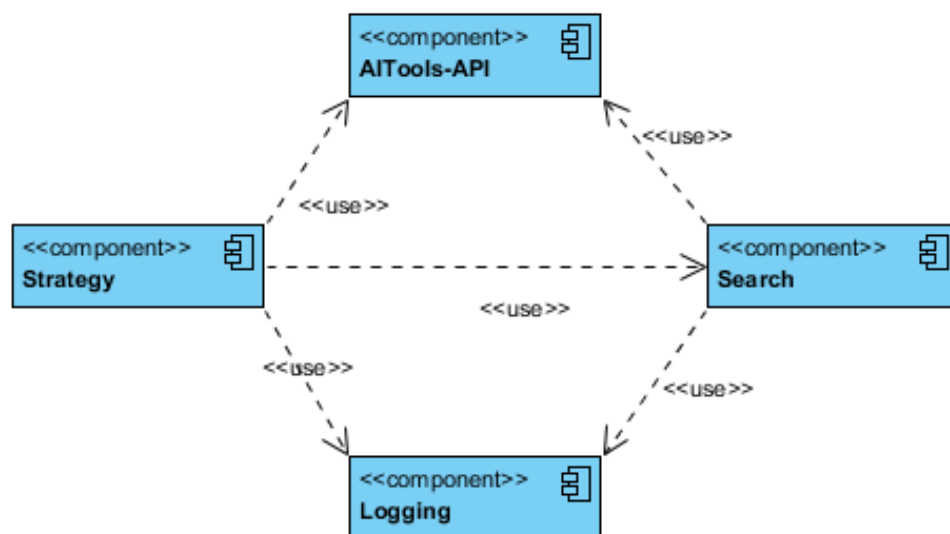


Abbildung 2.2: Modulabhängigkeiten



2.2 Sequenzdiagramme



3 API

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

3.1 Entities

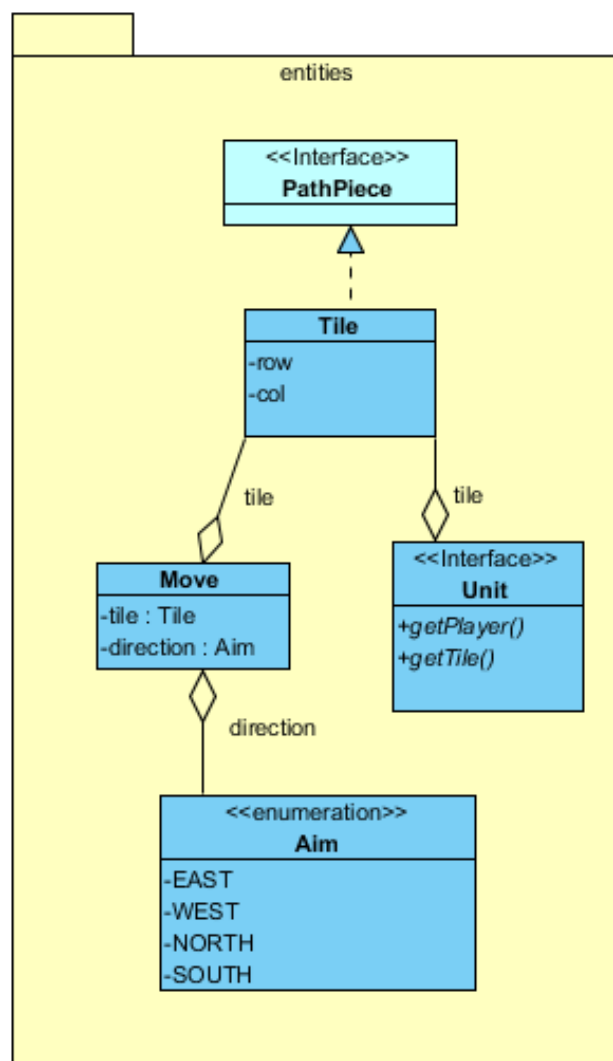


Abbildung 3.1: Entities

- **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

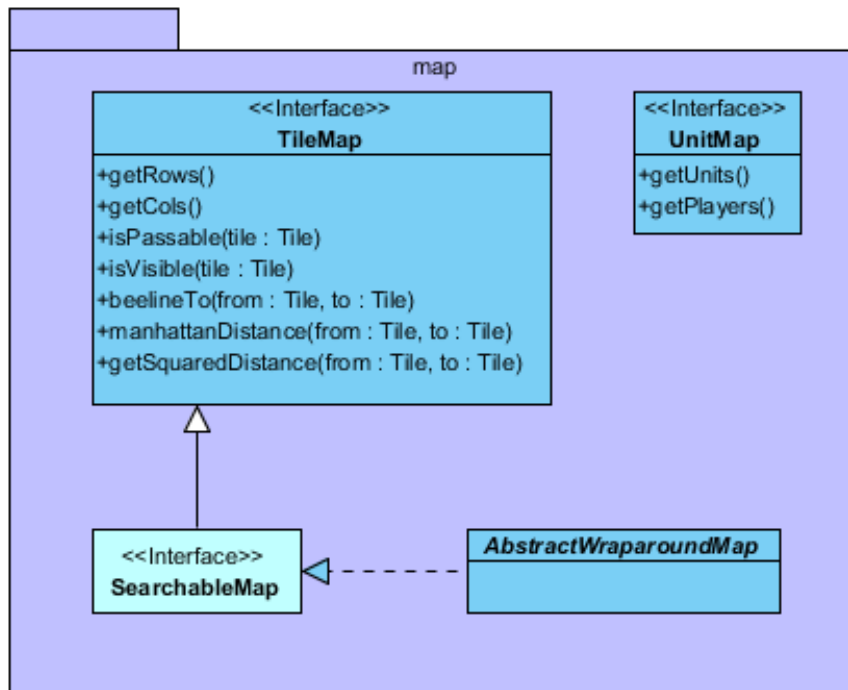


Abbildung 3.2: Map API

- **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 Suchalgorithmen 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

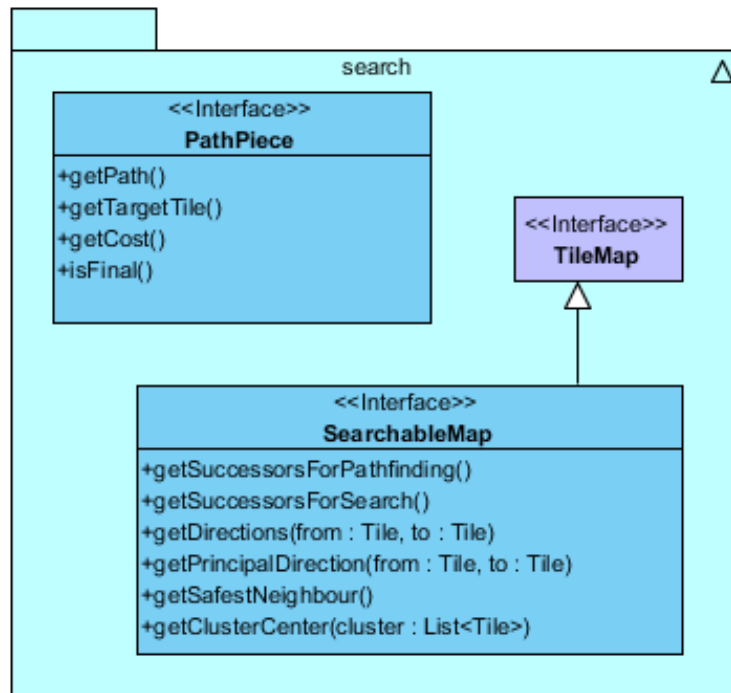


Abbildung 3.3: Search API

- **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 (repräsentiert 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

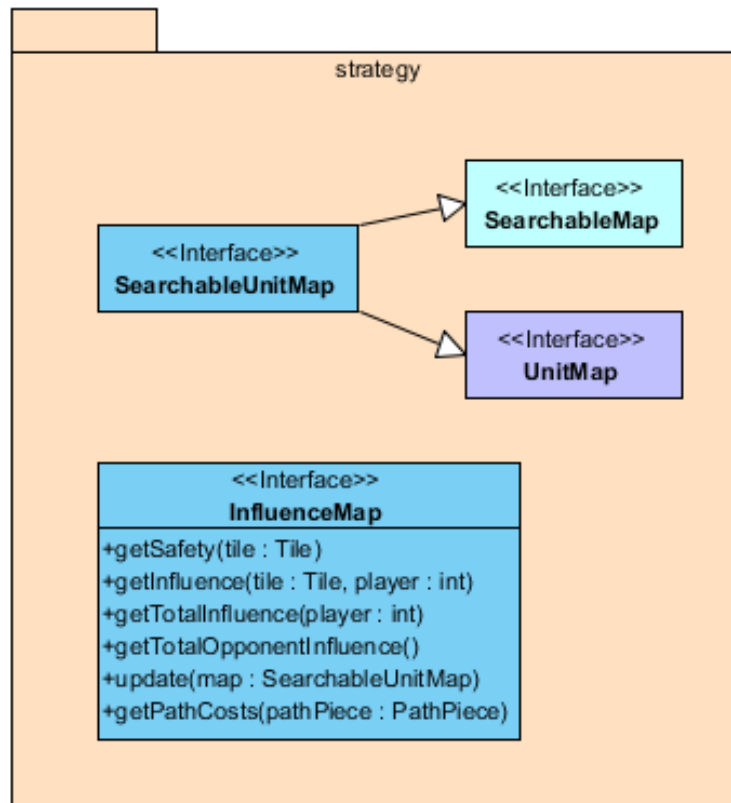


Abbildung 3.4: Strategy API



4 Suchalgorithmen

4.1 Entities für die Pfadsuche

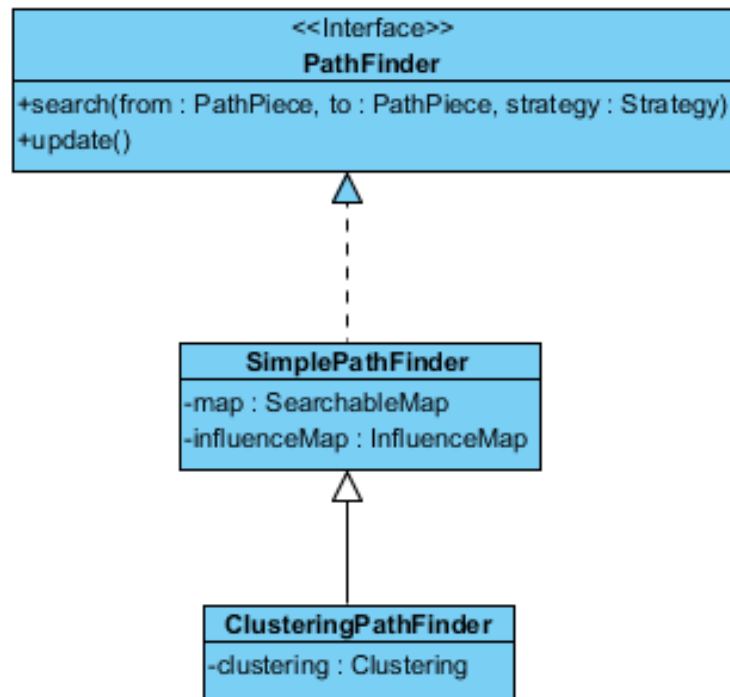


Abbildung 4.1: Klassendiagramm Pfadsuche

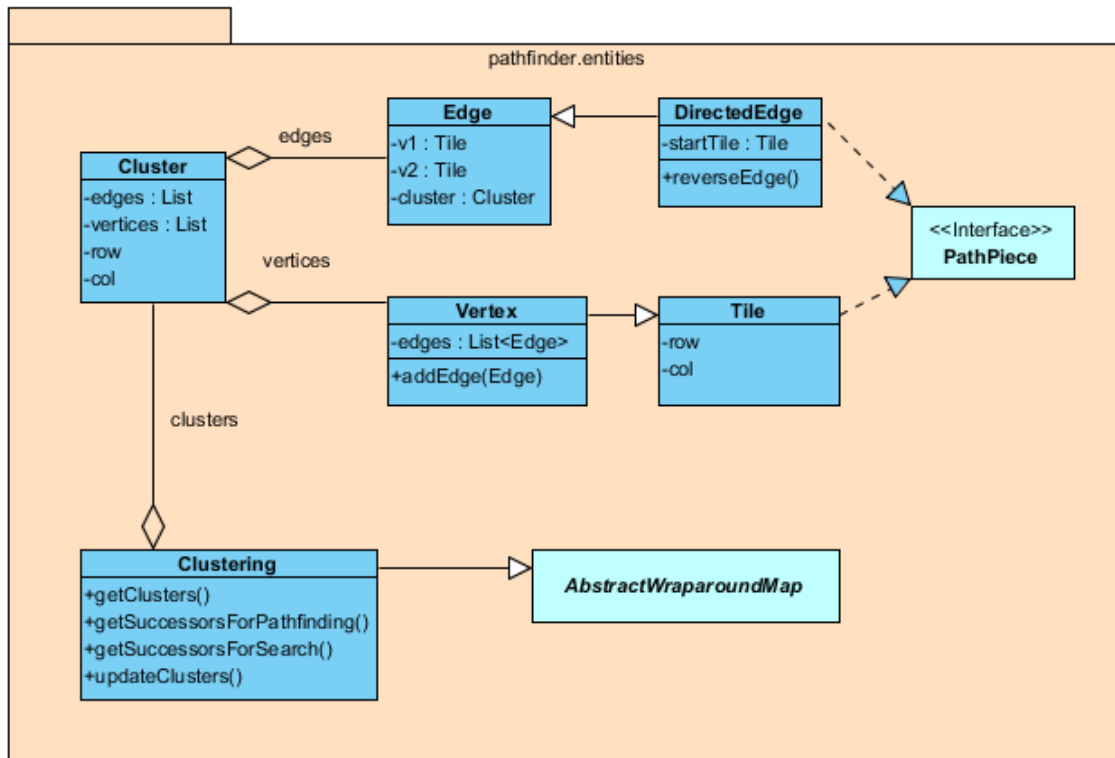


Abbildung 4.2: Spiel-Elemente für die Suche (vereinfacht)

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.

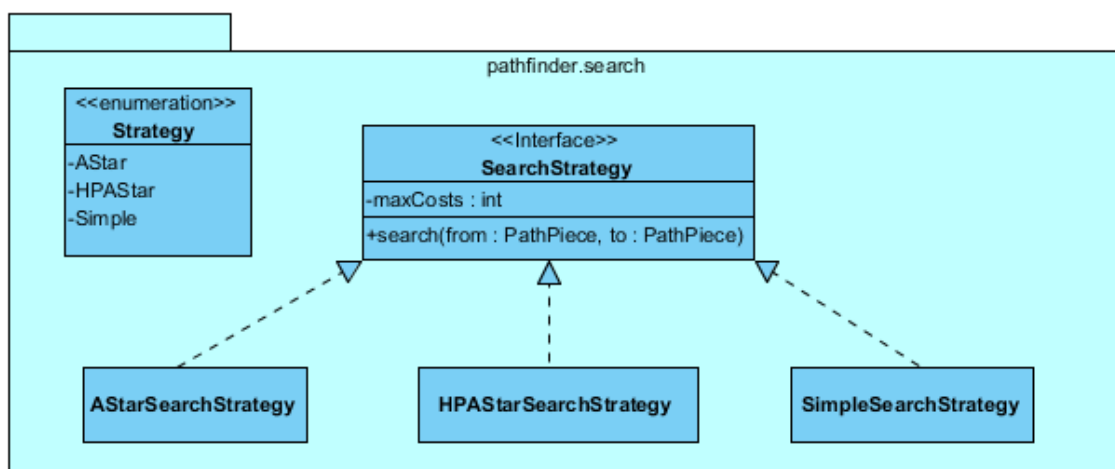


Abbildung 4.3: Suchstrategien



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.

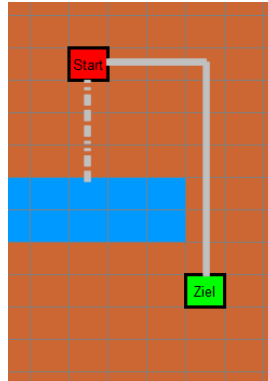


Abbildung 4.4: Simple-Path Algorithmus

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 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(), 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-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);
```

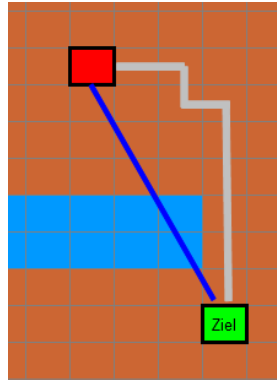


Abbildung 4.5: Heuristische Kosten (blau), Effektive Kosten (grau)

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

4.2.3 HPA* Algorithmus

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 sogenannten 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.

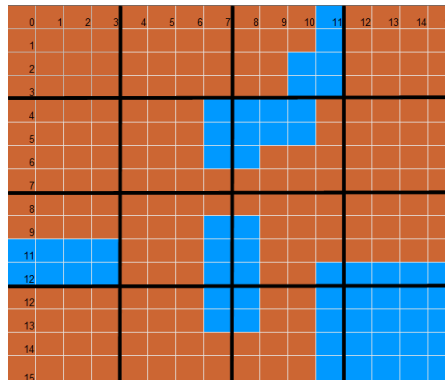


Abbildung 4.6: Clustereinteilung auf der Landkarte. Clustergrösse 4x4, Landkarte 16x16

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 Beginn 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.

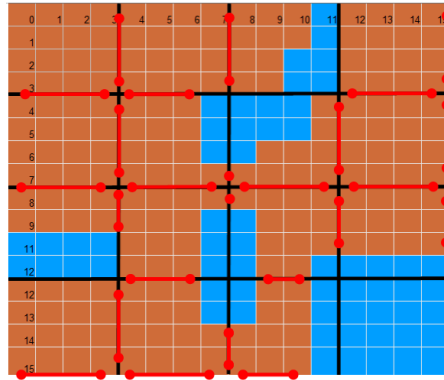


Abbildung 4.7: Die Kanten jedes Clusters wurden berechnet

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.

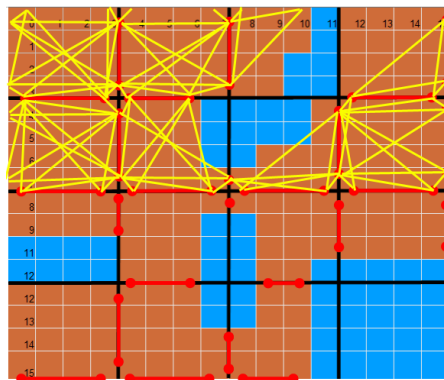


Abbildung 4.8: Darstellung der Innenkanten

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 beschriebenen A* Algorithmus verwendet um die beiden Knoten auf dem kürzesten möglichen Pfad (gelb) zu verbinden.¹

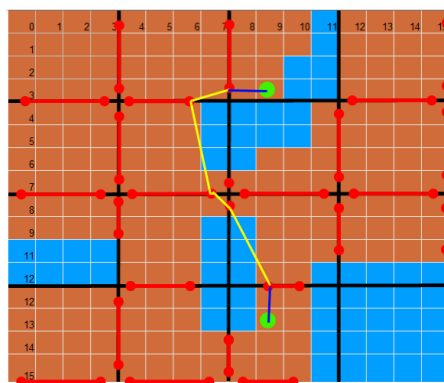


Abbildung 4.9: Errechneter Weg mittels HPA*

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



Um eine Pfadsuche mit HPA* durchzuführen muss ein `ClusteringPathFinder` instanziiert 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 variieren. (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 (useInfluenceMap)
        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.

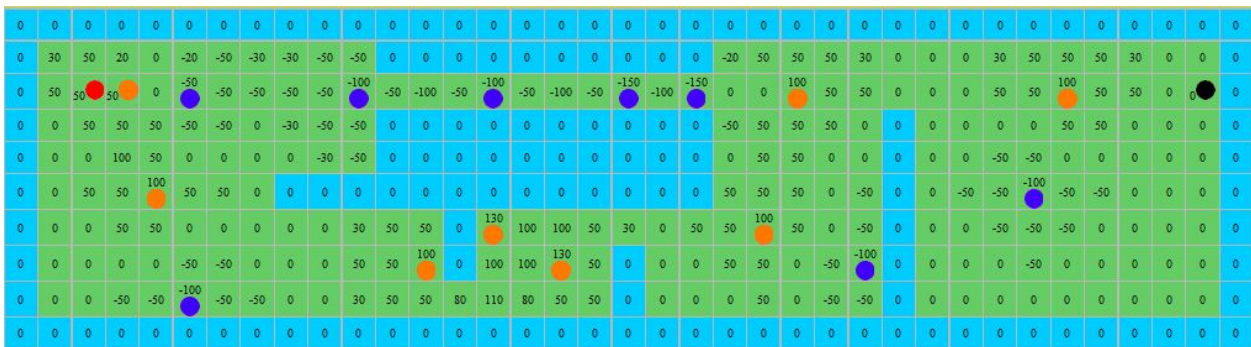


Abbildung 4.10: Ausgangslage Pfadsuche mit A* und InfluenceMap

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).

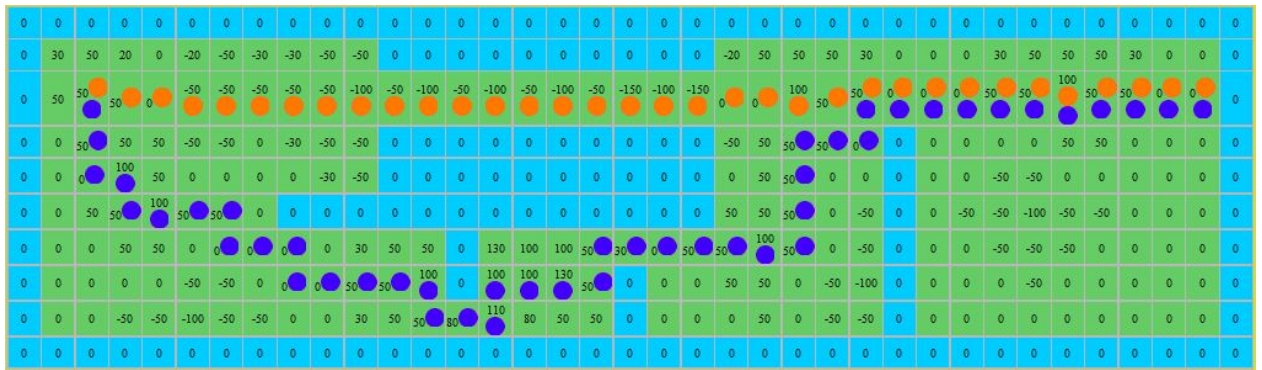


Abbildung 4.11: Resultierende Pfade mit und ohne Berücksichtigung der Influence Map

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
Blauer Pfad	46	106

4.3 Breitensuche

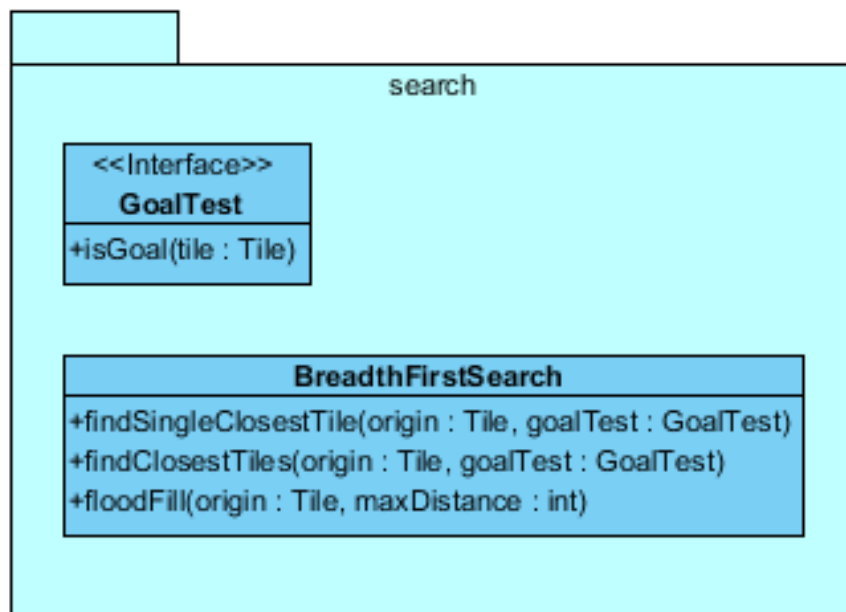


Abbildung 4.12: Breitensuche Klassendiagramm

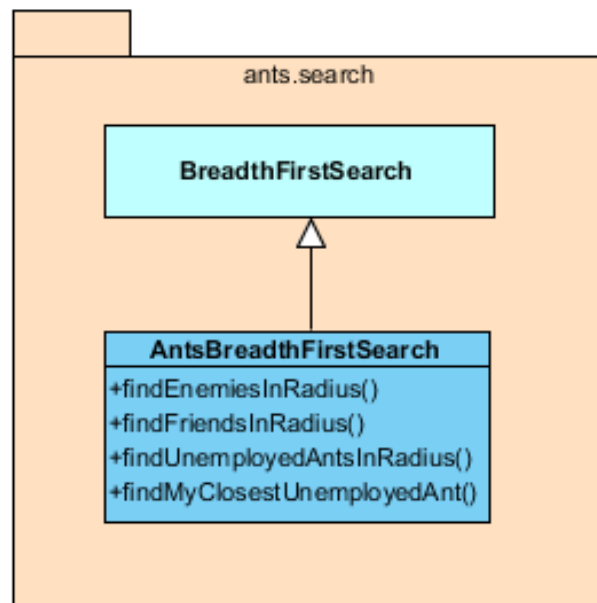


Abbildung 4.13: Breitensuche Ants-spezifisch

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 implementiert, 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 durchforstet wird. Zusätzlich gilt die Einschränkung das die Breitensuche nur 40 Tiles durchsuchen darf, was einem Radius von zirka 7 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 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

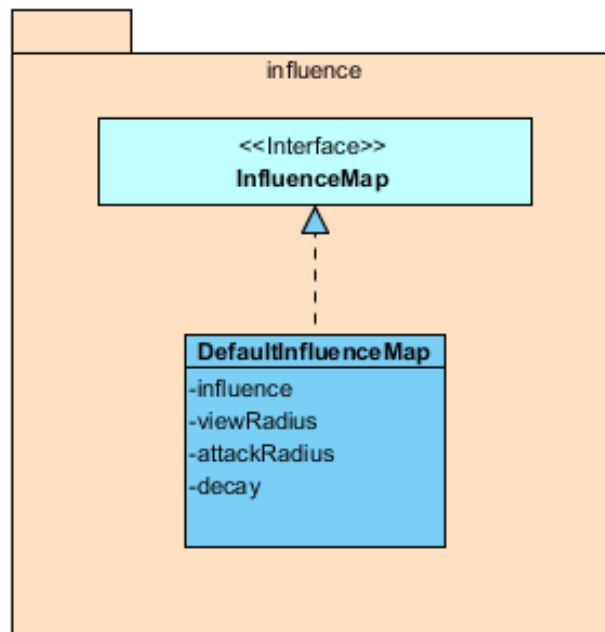


Abbildung 5.1: Influence Map Klassendiagramm

Die Influence Map haben wir nach den Beschreibungen im Buch [?] implementiert. Jede bekannte Spieleinheit auf der Spielkarte 'strahlt' einen gewissen Einfluss aus. In unserer Implementation unterscheiden wir zwischen drei Einflussradien, dem Angriffsradius, dem erweiterten Angriffsradius und dem 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 verwenden die Influence Map vor allem für die Bestimmung der Sicherheit. Abgebildet ist eine Sicherheitskarte (Desirability Map) für den orangefarbenen 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.

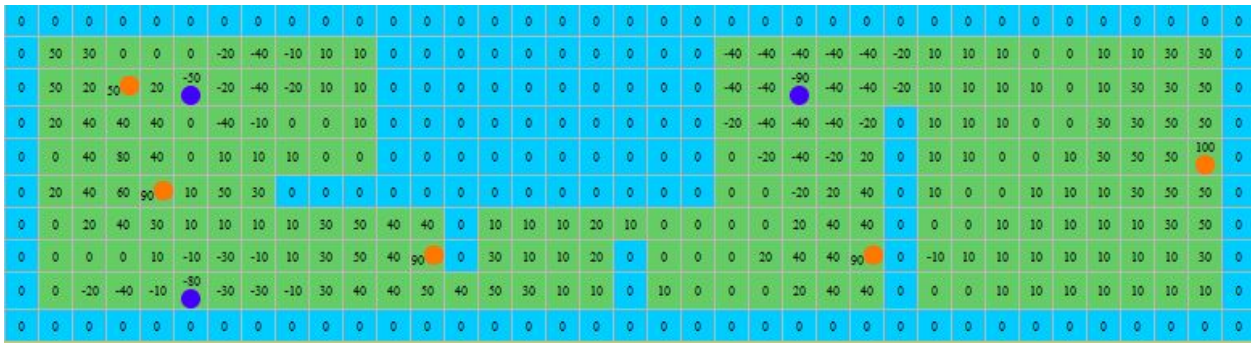


Abbildung 5.2: Influence Map, dargestellt ist die Sicherheit je Tile.

5.1.1 Update

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

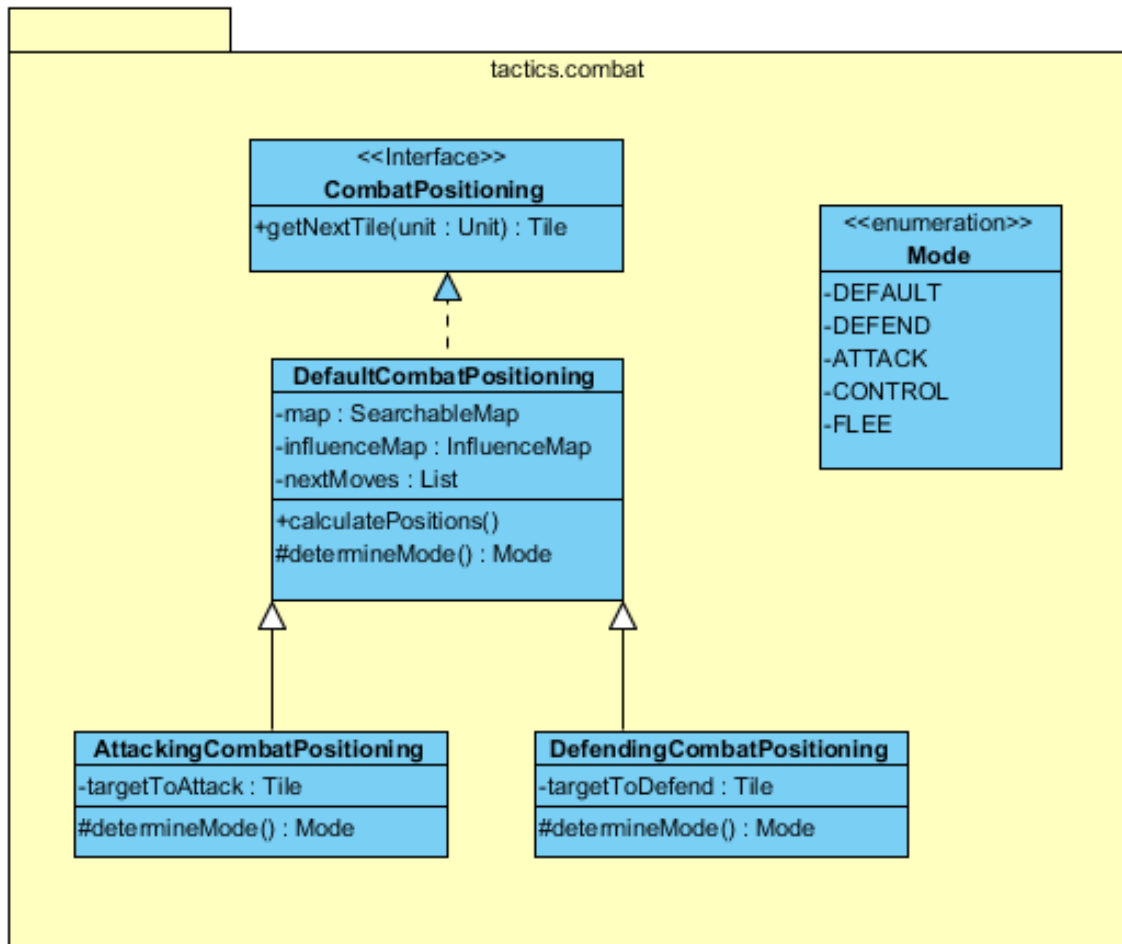


Abbildung 5.3: CombatPositioning Klassendiagramm

Kampfsituationen werden immer dann erstellt wenn gegnerische Ameisen auf unsere Ameisen treffen. Dies ist vor allem 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 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.

```
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

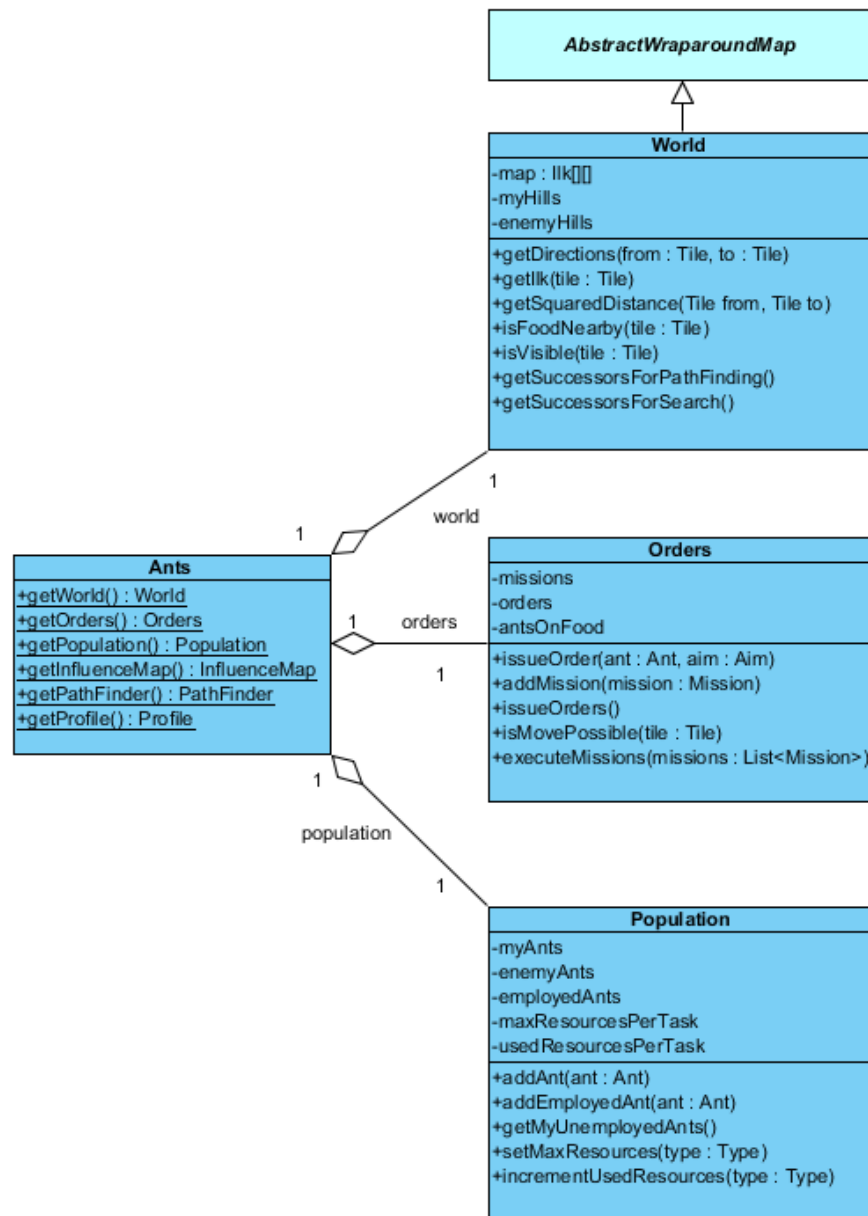


Abbildung 6.1: State-Klassen (vereinfacht)

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)

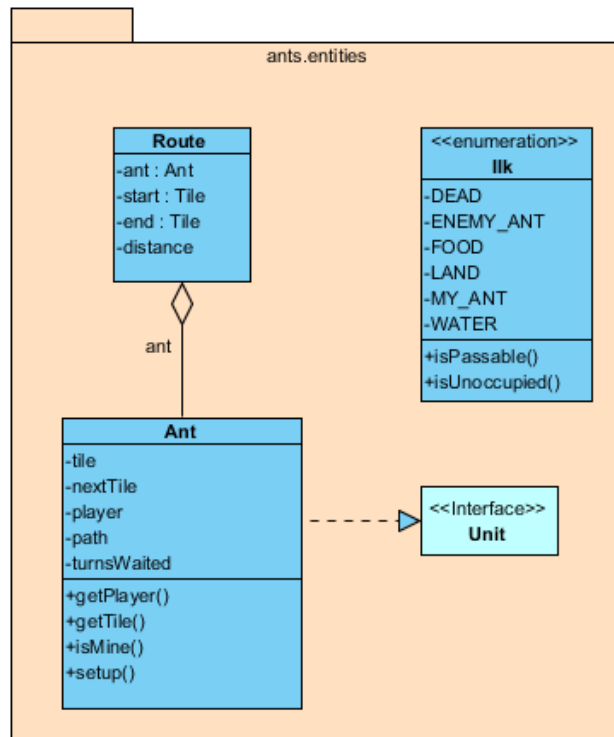


Abbildung 6.2: Ants-spezifische Elemente der Spielwelt (vereinfacht)

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

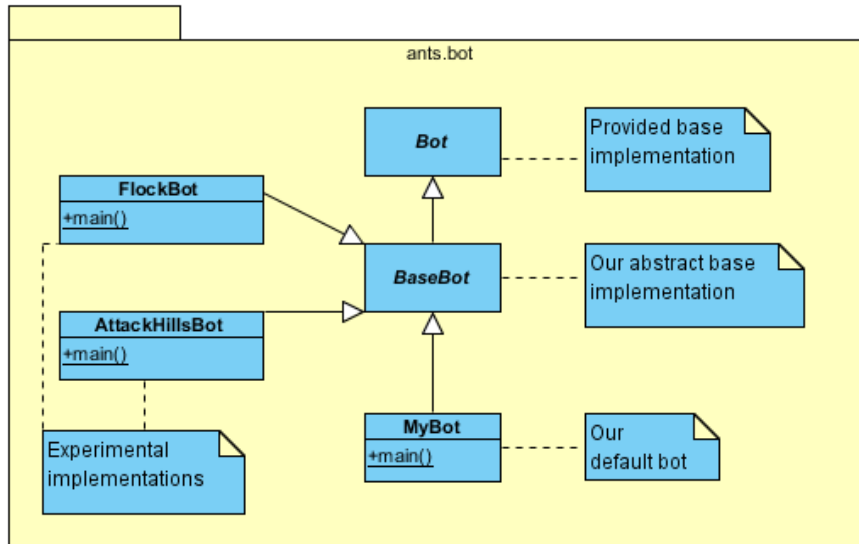


Abbildung 6.3: Vererbung der Bots wobei auf Stufe impl (Implementation) nur MyBot verwendet wird.

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

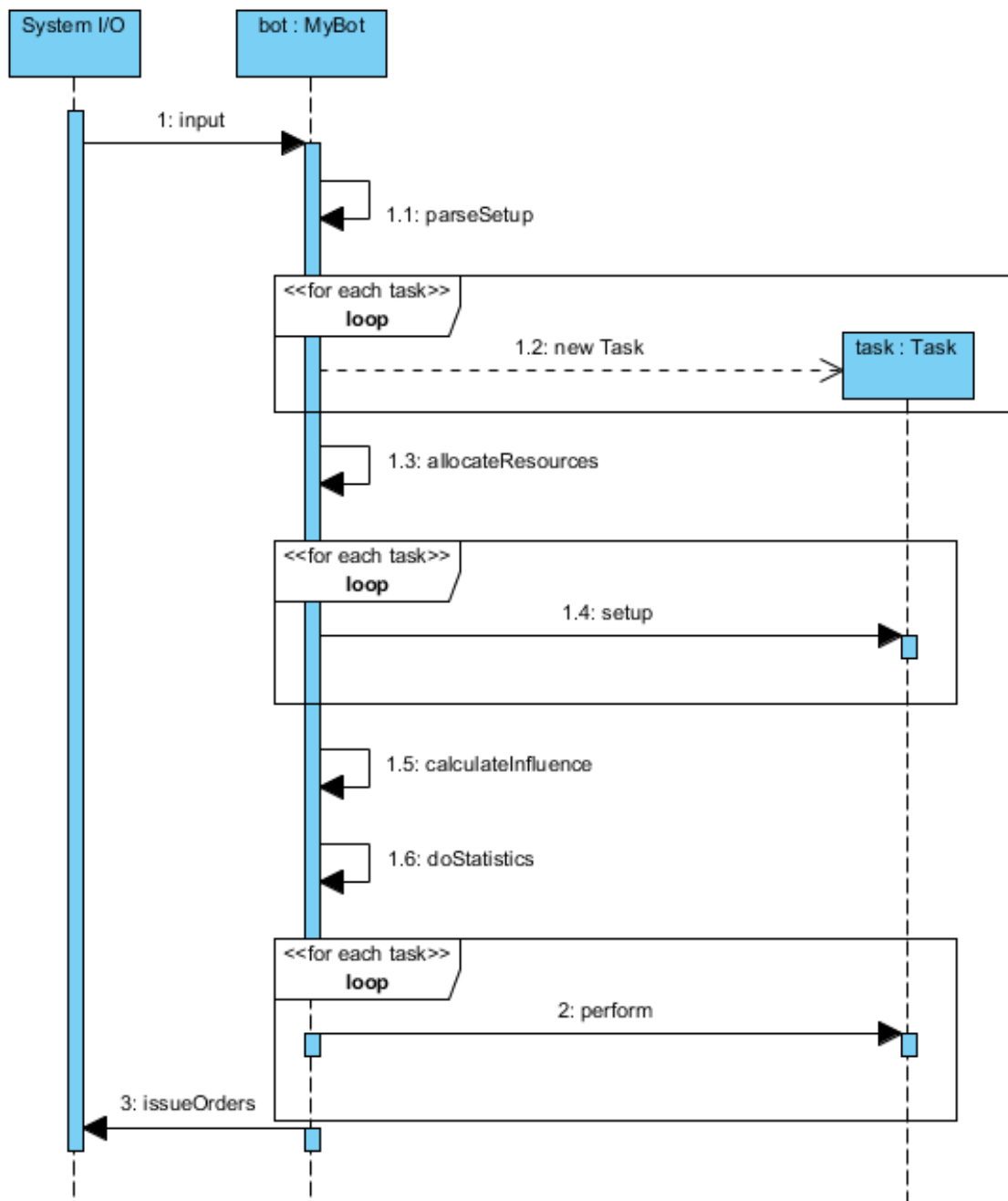


Abbildung 6.4: Ablauf des ersten Zugs des Spiels

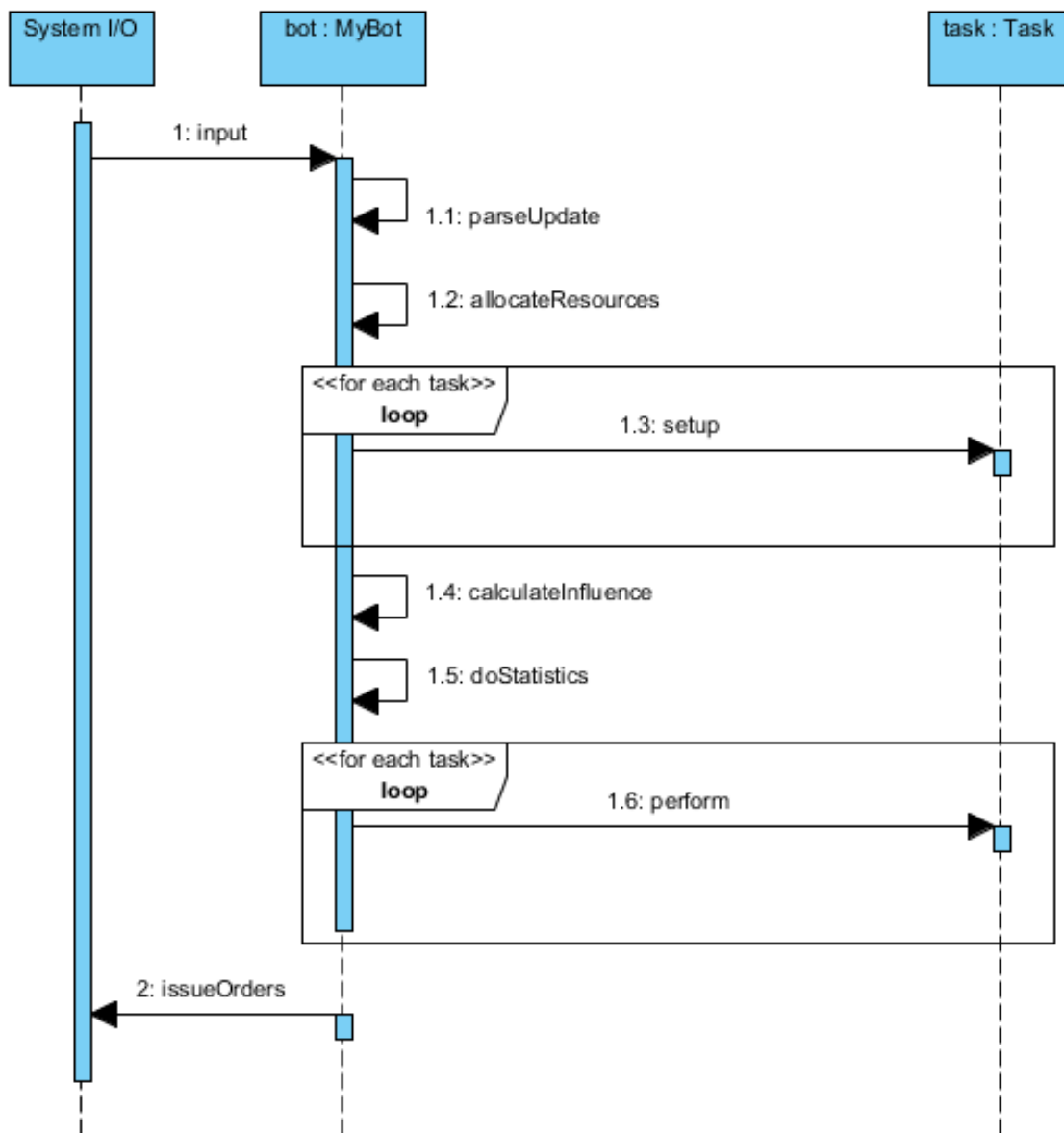


Abbildung 6.5: Ablauf der weiteren Züge des Spiels

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` 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 weitere 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 detaillierte Beschreibung der Tasks ist im Kapitel 6.4 zu finden.

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



6.4 Tasks

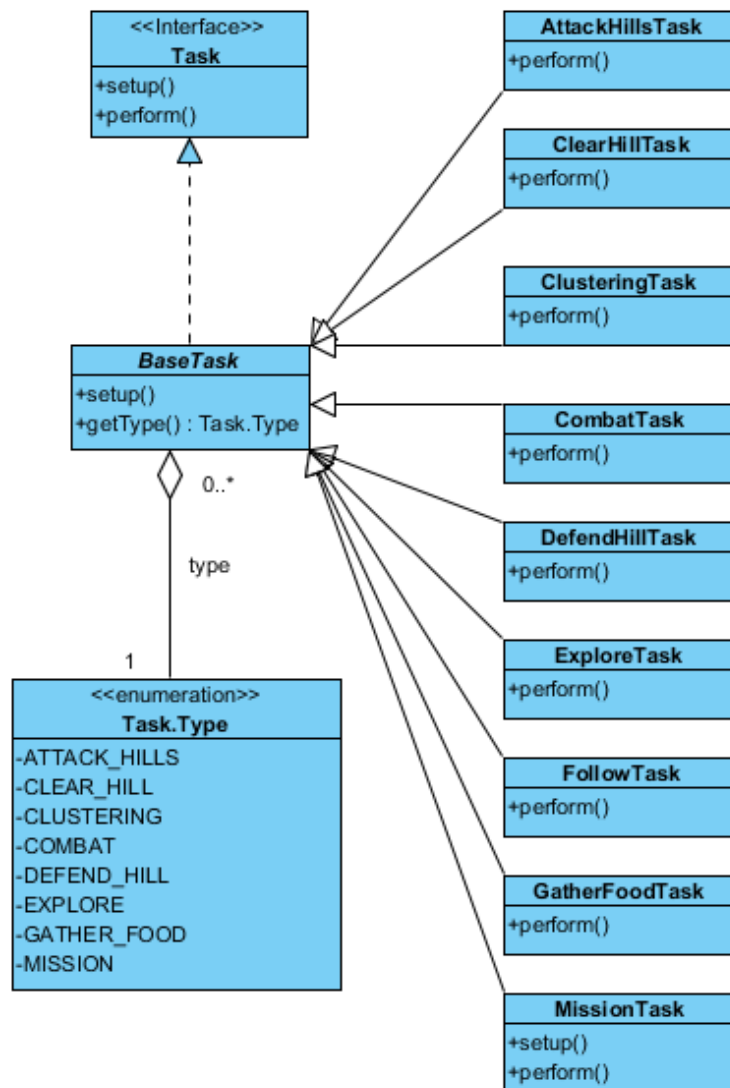


Abbildung 6.6: Tasks

Zu Beginn des Projekts haben wir die wichtigsten Aufgaben der Ameisen identifiziert. Diese Aufgaben wurden als Tasks in eigenen Klassen implementiert. Das Interface Task¹ 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

¹Das 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, können diese ausgeführt werden. 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 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.isValid()`) 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 koordiniert 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 Beginn 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 noch nicht sichtbar, also unerforscht ist. Falls ein Pfad mittels Pfadsuchalgorithmus gefunden wird, wird eine ExploreMission erstellt. Die Ameise wird den gefundenen Pfad in den nächsten Spielzügen ablaufen.



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 Verworfen 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 Weghälften 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.

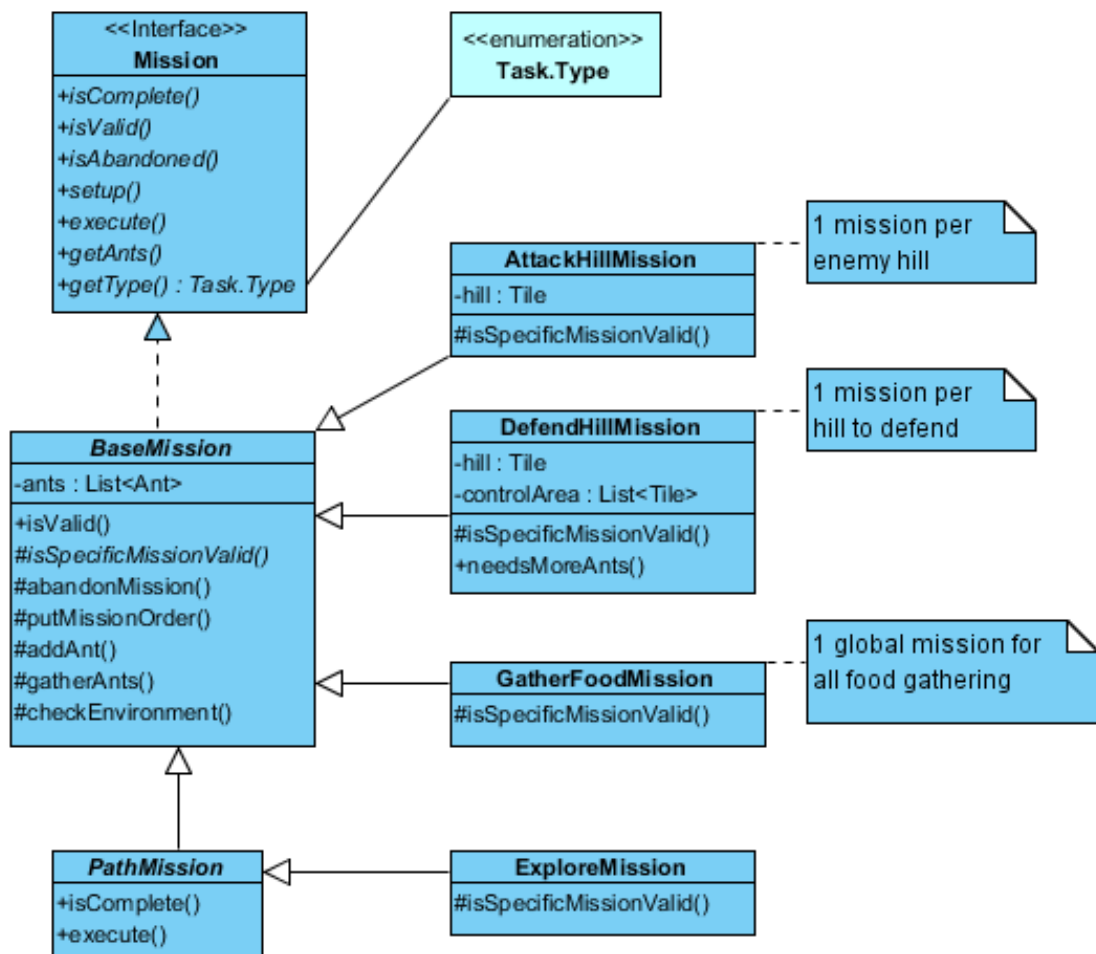


Abbildung 6.7: Missionen und ihre Hierarchie

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 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 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 rekrutieren): 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 Unterstützung 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)

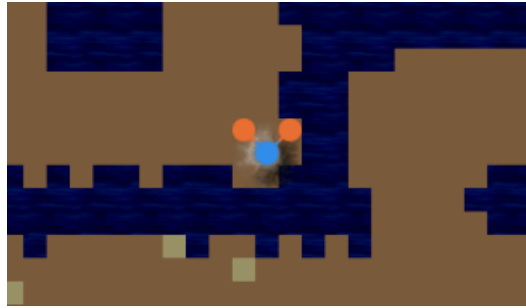


Abbildung 6.8: Ein gegnerischer Hügel wird durch unsere Ameisen (orange) kontrolliert.

DestroyHill

Status ist aktiv wenn, Spiel 5 Züge vor Spielende ist

Wenn dieser Modus eintritt 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

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.

6.5.5 ExploreMission

Precondition: -

Creator: ExploreTask

Postcondition: Definierter 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

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

6.5.7 Verworfen und nicht verwendete Mission

Wie bereits im Kapitel Task erwähnt war nicht alles programmiert erfolgreich. Hier sind die Missionen aufgelistet die zu den Tasks gehören welche verworfen wurden oder nicht verwendet werden.

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

Für jedes Food-Tile werden in einem definierbaren Radius r die nächsten Ameisen bestimmt. Danach wird nach aufsteigender Luftliniendistanz mit dem Pfadsuchalgorithmus SIMPLE (s. Abschnitt ??) oder – falls dieser keinen Pfad gefunden hat – mit A* eine passierbare Route gesucht. Wenn ein Pfad existiert, kann mit der Ameise und dem Food-Tile eine GatherFoodMission erstellt werden, welche die Ameise zum Food-Tile führt. Zu jedem Food-Tile wird immer nur eine Ameise geschickt.

6.6 Ressourcen Management

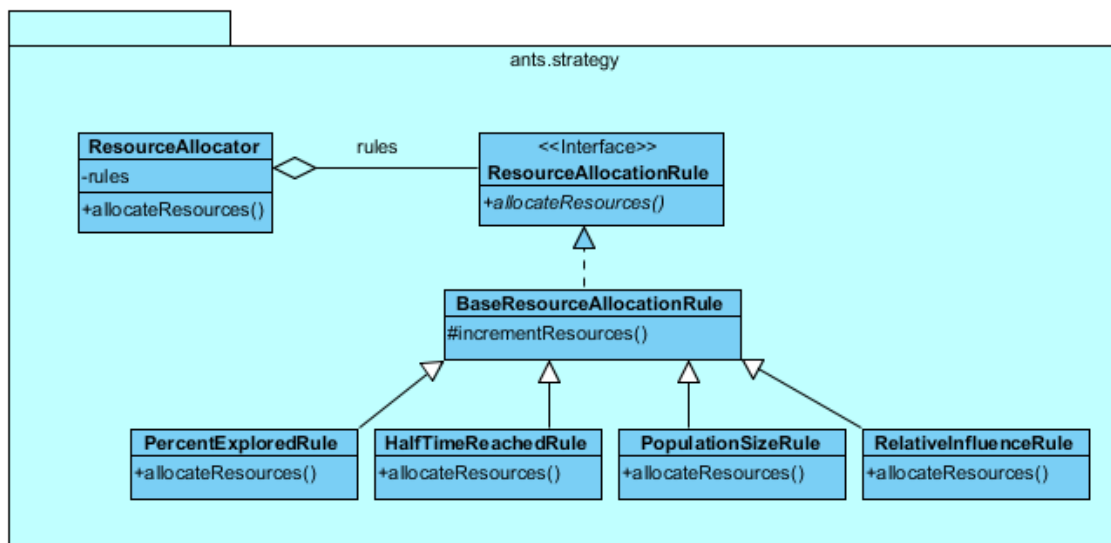


Abbildung 6.9: RessourcenManagement

Abbildung 6.9 zeigt die Klassenhierarchie unserer Ressourcenverwaltung.

6.7 Profile



7 Logging

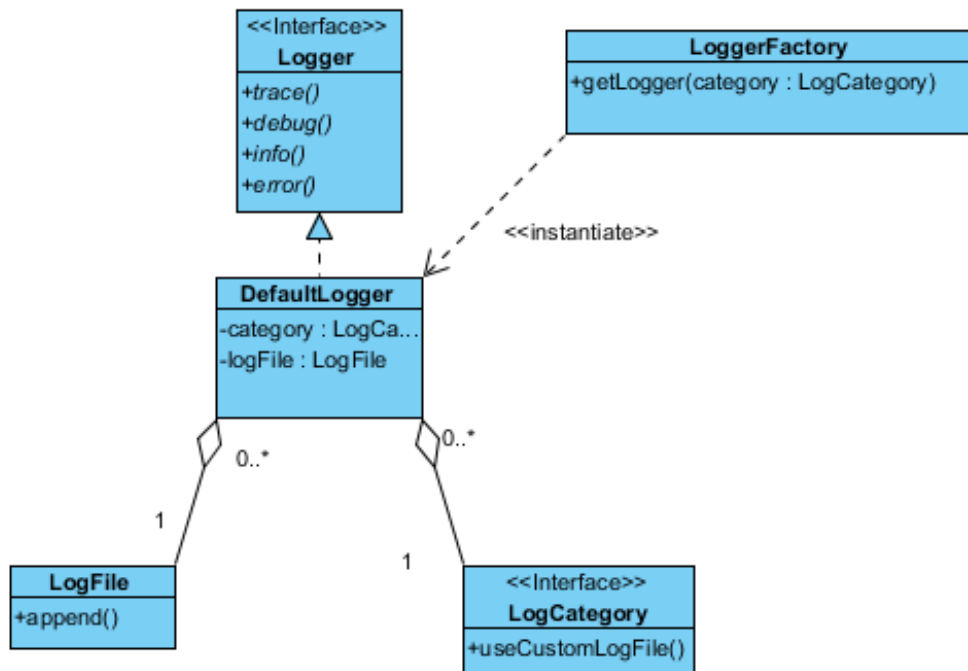


Abbildung 7.1: Logging Klassen

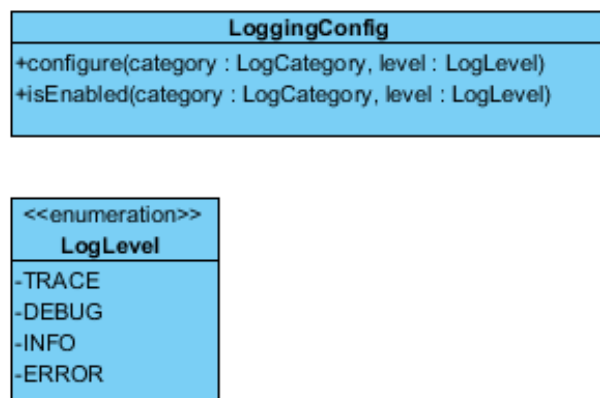


Abbildung 7.2: Logging Konfiguration

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 definiert 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 geschrieben. Zudem kann, falls erwünscht, jede Logkategorie in ein eigenes Logfile geschrieben werden. Die meisten Module haben 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 durchgewälzt 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 HTML 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 Information 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 Pop-up-Fenster 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. ??)

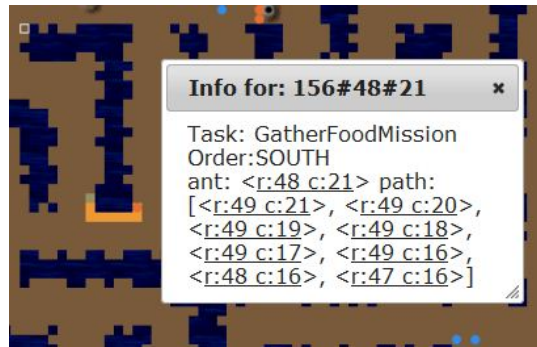
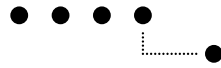


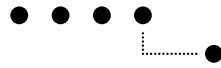
Abbildung 7.4: Mit der erweiterten Version wird der Pfad (orange) der Ameise von $\langle r:48 \text{ c}:21 \rangle$ nach $\langle r:47 \text{ c}:16 \rangle$ auf der Karte abgebildet.





8 Testreader





9 TestCenter

todo besserer name

9.1 Unit- und Funktionstests

9.2 Testbots

9.3 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 verschiedenen 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 _____

