



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: 6. 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 6. Januar 2013

Name Vorname Lukas Kuster

Unterschrift

Name Vorname Stefan Käser

Unterschrift





Inhaltsverzeichnis

1	Einleitung	1
1.1	Spielbeschrieb	2
1.1.1	Der Wettbewerb	2
1.1.2	Spielregeln	2
1.1.3	Schnittstelle	2
1.2	Abgrenzungen	3
1.3	Projektverlauf	3
1.4	Projektorganisation	4
1.4.1	Beteiligte Personen	4
1.4.2	Projektmeetings	4
1.4.3	Arbeitsweise	4
1.5	Tools	6
1.6	Artefakte	6
1.7	Zielerreichung	6
1.7.1	Funktionale Anforderungen	6
1.7.1.1	Musskriterien	6
1.7.1.2	Kannkriterien	8
1.7.2	Nicht funktionale Anforderungen	8
1.7.2.1	Musskriterien	8
1.7.2.2	Kannkriterien	9
1.8	Herausforderungen	9
1.8.1	Module testen	9
1.8.2	Kampfsituationen beurteilen	9
1.8.3	Vergleich mit Bots aus dem Wettbewerb	10
1.9	Fazit	10
1.10	Weiterführende Arbeiten	10
2	Architektur	11
2.1	Module	11
2.2	Modulabhängigkeiten	12
2.3	Externe Abhängigkeiten	12
3	API	13
3.1	Entities	13
3.2	Map	14
3.3	Search	15
3.4	Strategy	16
4	Suchalgorithmen	17
4.1	Entities für die Pfadsuche	17
4.2	Pfadsuche	18
4.2.1	Simple Algorithmus	19
4.2.2	A* Algorithmus	19
4.2.3	HPA* Algorithmus	20
4.2.3.1	Clustering	20
4.2.4	Pfadsuche mittels Influence Map	22
4.2.5	Pathsmoothing	24
4.3	Breitensuche	24



5	Strategie und Taktik	27
5.1	Influence Map	27
5.1.1	Update	28
5.1.2	Anwendungsfälle	28
5.2	Combat Situations	29
5.2.1	DefaultCombatPositioning	29
5.2.1.1	FLEE	30
5.2.1.2	DEFEND	30
5.2.1.3	ATTACK	30
6	Ants	31
6.1	State-Klassen	31
6.1.1	Ants	32
6.1.2	World	32
6.1.3	Orders	32
6.1.4	Population	32
6.1.5	Clustering	32
6.2	Spiel-Elemente (Ants-Spezifisch)	33
6.2.1	Ant	33
6.2.2	Route	33
6.2.3	Ilk	33
6.3	Aufbau Bot	34
6.3.1	Klasse Bot	34
6.3.2	BaseBot	34
6.3.3	Ablauf eines Zugs	35
6.3.4	MyBot	37
6.4	Tasks	38
6.4.1	MissionTask	38
6.4.1.1	setup()	38
6.4.1.2	execute()	39
6.4.2	GatherFoodTask	39
6.4.3	DefendHillTask	39
6.4.4	AttackHillsTask	39
6.4.5	CombatTask	39
6.4.6	ExploreTask	39
6.4.7	ClearHillTask	40
6.4.8	ClusteringTask	40
6.4.9	Verworfenne und nicht verwendete Tasks	40
6.5	Missionen	40
6.5.1	BaseMission	41
6.5.2	PathMission	42
6.5.3	AttackHillMission	42
6.5.4	DefendHillMission	43
6.5.5	Default-Modus	44
6.5.6	Barrier-Modus	44
6.5.7	ExploreMission	45
6.5.8	GatherFoodMission	45
6.5.9	Verworfenne und nicht verwendete Mission	47
6.6	Ressourcenverwaltung	47
6.7	Implementierte Regeln	48
6.8	Profile	49
6.8.1	Validierung	49
6.8.2	Definierte Profile	50
6.8.3	Weiterentwicklungs-Potenzial	50
7	Logging	51
7.1	Konfiguration	52



7.2	Anwendungsbeispiele	52
7.3	JavaScript Addon für HMTL-Gameviewer	53
7.4	Profile und Logging	54
8	Testing	55
8.1	Unit Tests	55
8.2	Visuelle Tests	55
8.3	Testbots	55
8.4	Testreport Profile	56
8.4.1	TestReader	58
8.5	Online Tests	58
9	Spielanleitung	59
	Glossar	61
	Literaturverzeichnis	63





Abbildungsverzeichnis

1.1	Projektablauf	3
1.2	Commit History	5
1.3	Aktivitäten	5
2.1	Module	11
2.2	Modulabhängigkeiten	12
3.1	Entities	13
3.2	Map API	14
3.3	Search API	15
3.4	Strategy API	16
4.1	Klassendiagramm Pfadsuche	17
4.2	Spiel-Elemente für die Suche (vereinfacht)	18
4.3	Suchstrategien	18
4.4	Simple-Path Algorithmus	19
4.5	A* Pfadsuche	20
4.6	Clustereinteilung auf der Landkarte.	20
4.7	Clustereinteilung auf der Landkarte.	21
4.8	Cluster mit berechneten Kanten	21
4.9	Cluster mit Innenkanten	21
4.10	Errechneter Weg mittels HPA*	22
4.11	Ausgangslage Pfadsuche mit A* und InfluenceMap	23
4.12	Resultierende Pfade mit und ohne Berücksichtigung der Influence Map	23
4.13	Breitensuche Klassendiagramm	24
4.14	Breitensuche Ants-spezifisch	24
5.1	Influence Map Klassendiagramm	27
5.2	Influence Map, dargestellt ist die Sicherheit je Tile.	28
5.3	CombatPositioning Klassendiagramm	29
6.1	State-Klassen (vereinfacht)	31
6.2	Ants-spezifische Elemente der Spielwelt (vereinfacht)	33
6.3	Vererbung der Bots wobei auf Stufe impl (Implementation) nur MyBot verwendet wird.	34
6.4	Ablauf des ersten Zugs des Spiels	35
6.5	Ablauf der weiteren Züge des Spiels	36
6.6	Tasks	38
6.7	Missionen und ihre Hierarchie	41
6.8	Ein gegnerischer Hügel wird durch unsere Ameisen (orange) kontrolliert.	43
6.9	Die erste zweier Gruppe ist in Kampfstellung. 4 weitere Ameisen rücken zur Front auf.	43
6.10	Links die Default-Verteidigung, rechts mit einer Sperre.	44
6.11	Eine Ameise auf einer ExploreMission bewegt sich in Richtung Fog of War	45
6.12	Eine Ameise auf einer ExploreMission bewegt sich in Richtung Fog of War	45
6.13	Eine Ameise auf einer ExploreMission bewegt sich in Richtung Fog of War	45
6.14	Ameisen am Futter einsammeln.	46
6.15	Population Klasse	47
6.16	RessourcenManagement	48
7.1	Logging Klassen	51
7.2	Logging Konfiguration	52



7.3	Live-Info Popupfenster	53
7.4	Erweiterung des Live-Info Popupfenster	53



Tabellenverzeichnis

1.1	Abgrenzungen	3
4.1	Pfadkosten mit und ohne Berücksichtigung der Influence Map	23
5.1	Einfluss einer Spieleinheit	27
6.1	Definierte Profile	50
8.1	Bilanz der Testläufe gegen egreavette	56
8.2	Bilanz der Testläufe gegen xathis	56
8.3	Bilanz der Testläufe gegeneinander	57
8.4	Die Profile für den 2. Testlauf	57
8.5	Bilanz der 2. Testläufe gegen xathis	57
8.6	Bilanz der 2. Testläufe gegeneinander	58





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 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.

1.1.2 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 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 Ameisenhügel 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>

1.1.3 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 Bot² 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>

²Die 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 Komponenten zu übernehmen. Es folgt eine Auflistung was bereits bestand bzw. was wir noch erweitert haben.

Erstellt in Modul "Projekt 2"	Erweiterung während der Bachelorarbeit
Grundfunktionalitäten des Bots	
Pfadsuche Simple, A*, HPA*	Auslagerung in ein eigenes Framework, Performanceverbesserungen, Erweiterung des HPA* Clustering, Pfadsuche 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)

Tabelle 1.1: Abgrenzungen

1.3 Projektverlauf

Die Projektarbeit richtete sich nach folgendem Zeitplan:

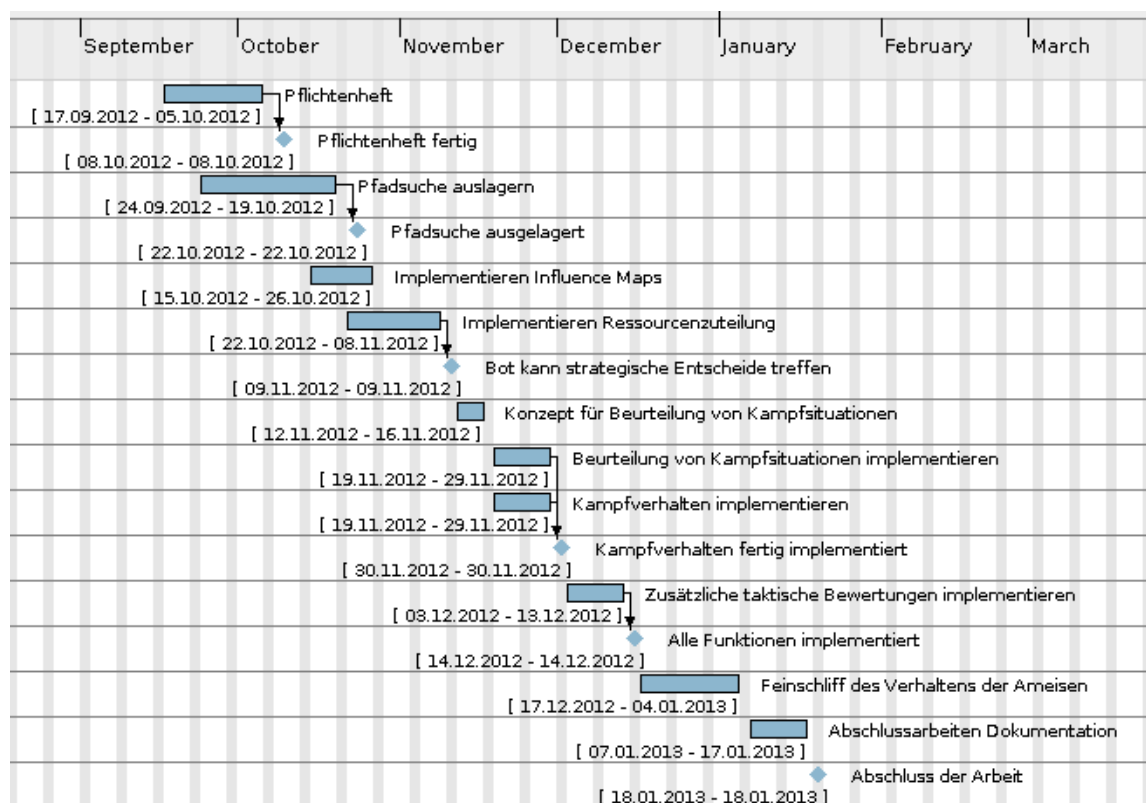


Abbildung 1.1: Projektablauf



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.

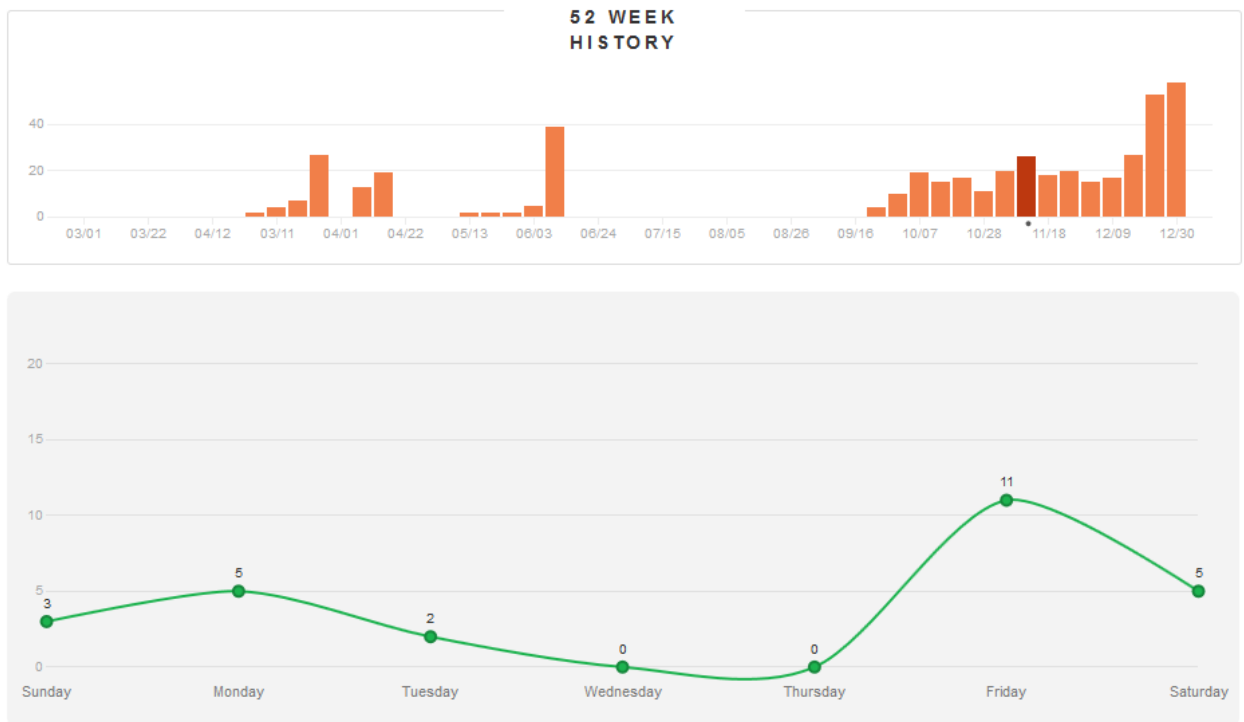


Abbildung 1.2: Commit History

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.

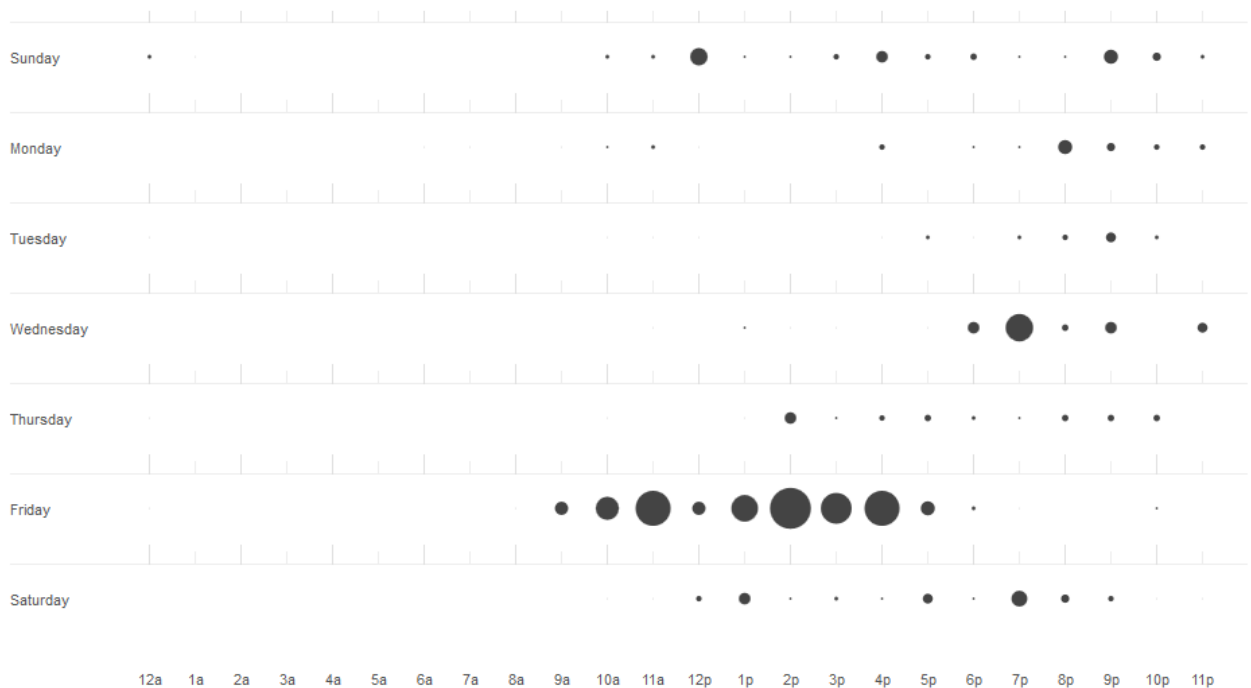


Abbildung 1.3: Aktivitäten



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/vpum1/>) 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 durch die Enumeration `Task.Type`. Sie unterscheidet bspw. zwischen `GATHER_FOOD`, `ATTACK_HILLS`, `EXPLORE`, `DEFEND_HILL` und weiteren. Alle Tasks (Kapitel 6.4) und Missionen (Kapitel 6.5) werden jeweils einem Aufgabentyp zugeordnet; auch die Ressourcenverwaltung funktioniert über diese Aufgabentypen.



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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

1.7.1.2 Kannkriterien

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

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

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

1.7.2 Nicht funktionale Anforderungen

1.7.2.1 Musskriterien

Ziel: Modularer Aufbau für eine gute Testbarkeit der Komponenten.

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

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



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

Ziel: Die Codedokumentation ist vollständig und dient der Verständlichkeit.

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

1.7.2.2 Kannkriterien

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

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

1.8 Herausforderungen

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

1.8.1 Module testen

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

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

1.8.2 Kampfsituationen beurteilen

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



1.8.3 Vergleich mit Bots aus dem Wettbewerb

Nach Ablauf des Wettbewerbs im Januar 2012, haben einige der Teilnehmer ihren Bot zugänglich gemacht. Dadurch war es uns möglich unseren Bot gegen Bots antreten zu lassen die tatsächlich am Wettbewerb teilgenommen haben. So konnten wir auch eine vage Einschätzung machen wie stark unser Bot ist. Mehr dazu unter siehe 8.4.

1.9 Fazit

TODO

1.10 Weiterführende Arbeiten

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

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

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

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

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



2 Architektur

2.1 Module

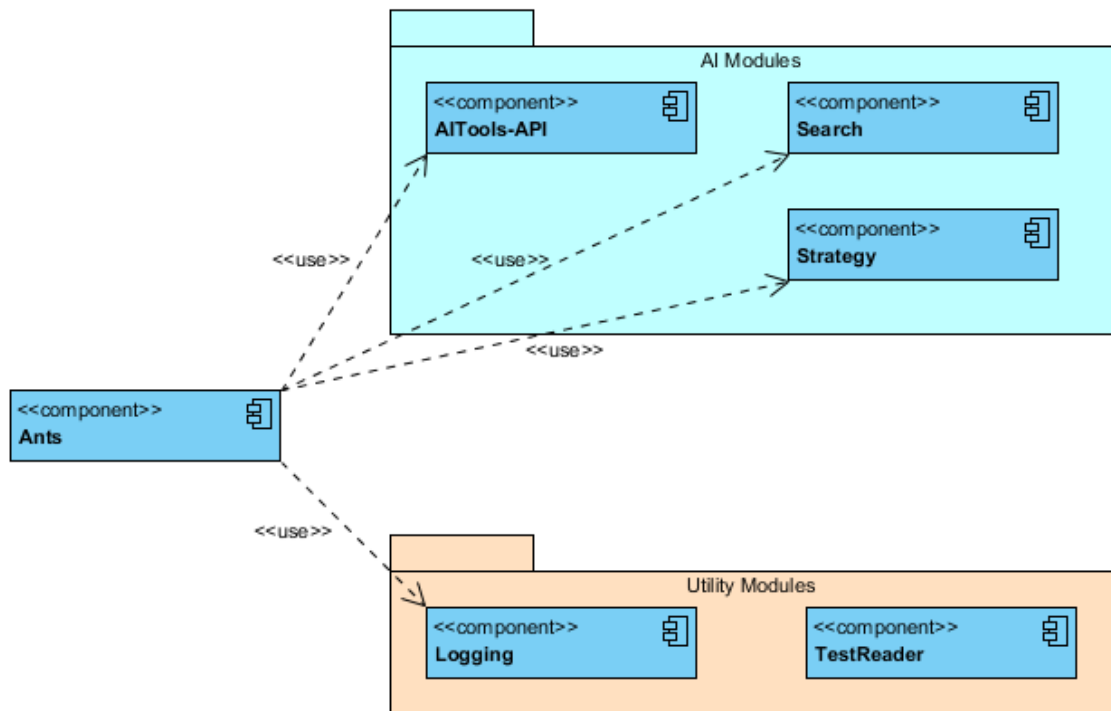


Abbildung 2.1: Module

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

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

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

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



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

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

TestReader: Das TestReader-Modul besteht aus einer einzelnen Klasse, die aus den Log-Files der Ants-Spielengine Auswertungen lesen kann. Es ist genauer beschrieben im Kapitel 8.4.1.

2.2 Modulabhängigkeiten

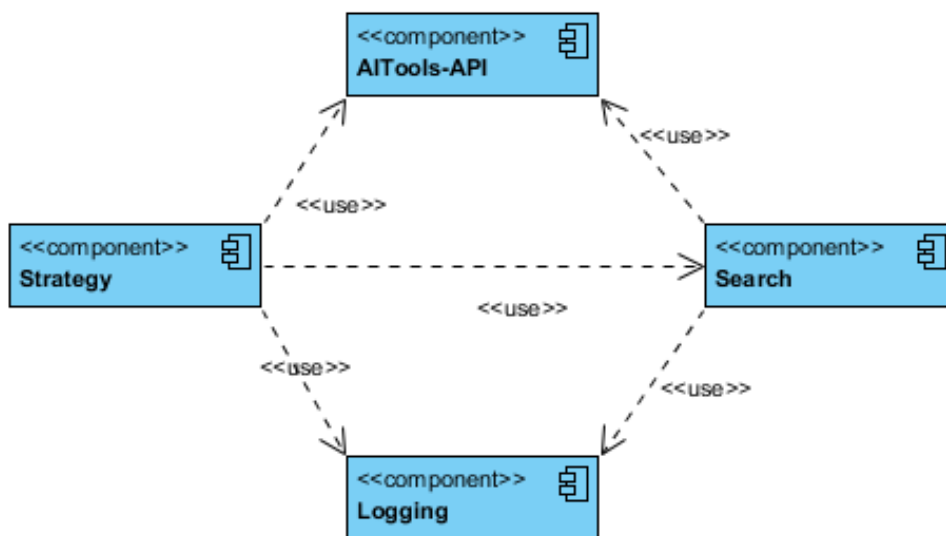


Abbildung 2.2: Modulabhängigkeiten

Abbildung 2.2 zeigt die Abhängigkeiten 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 FloodFill Algorithmus aufgebaut wird, der im Search Modul implementiert ist.

2.3 Externe Abhängigkeiten

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

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



3 API

Unsere AITools API dient als Grundbaustein 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

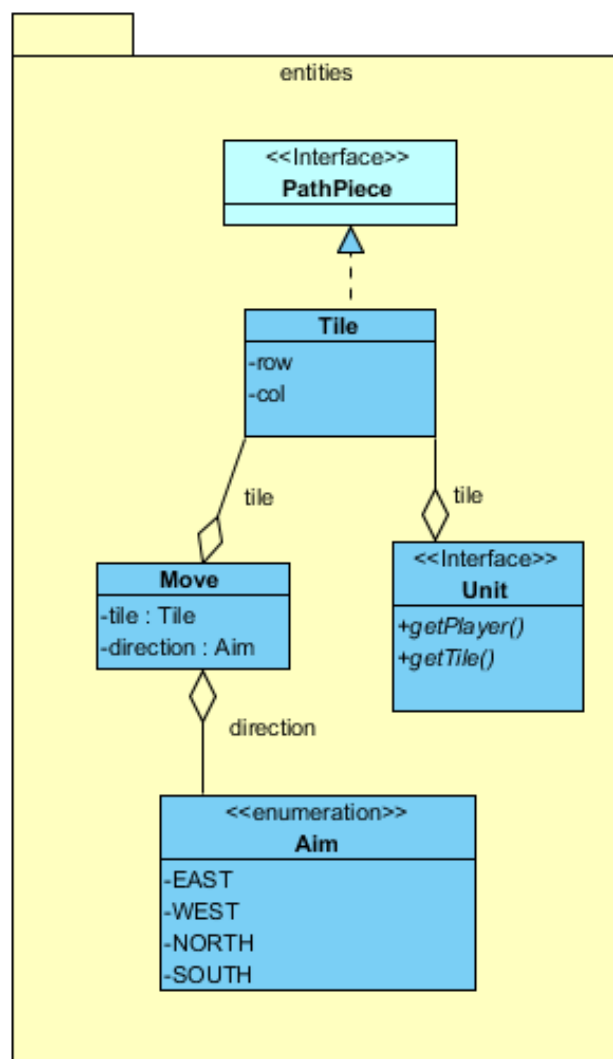


Abbildung 3.1: Entities

- **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

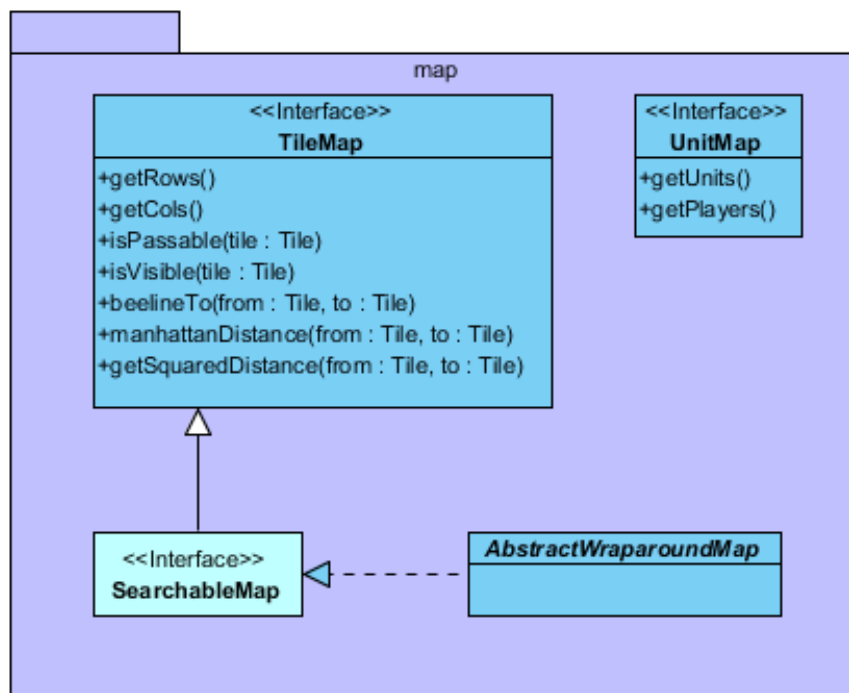


Abbildung 3.2: Map API

- **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

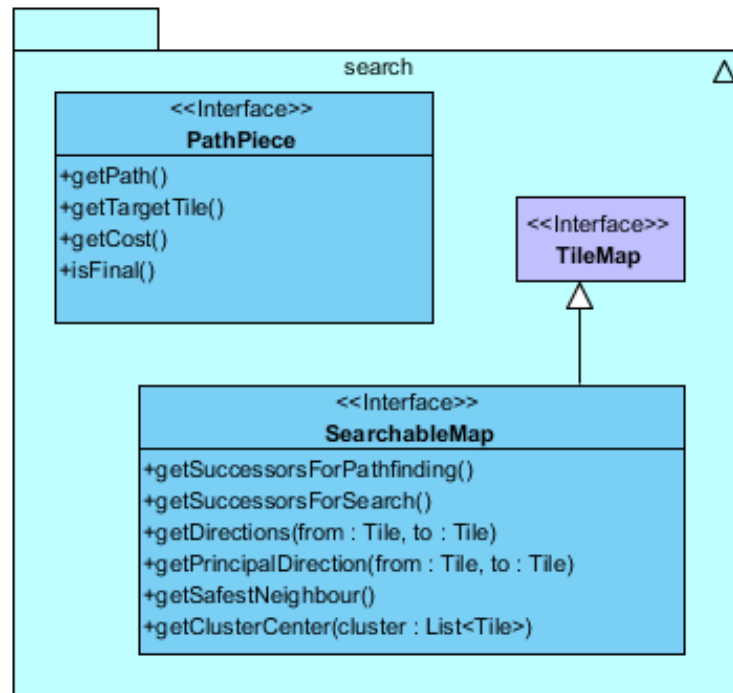


Abbildung 3.3: Search API

- **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

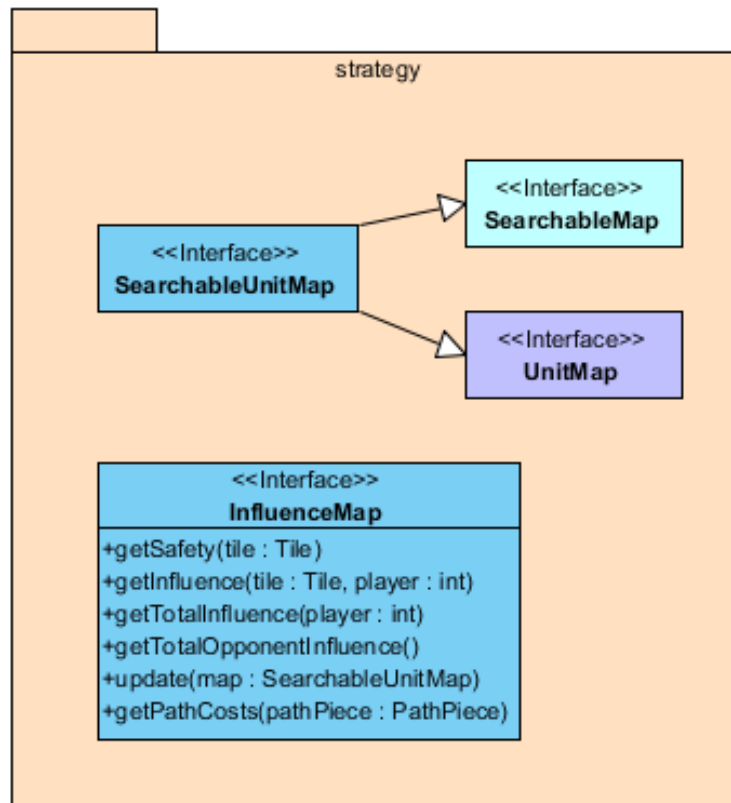


Abbildung 3.4: Strategy API



4 Suchalgorithmen

4.1 Entities für die Pfadsuche

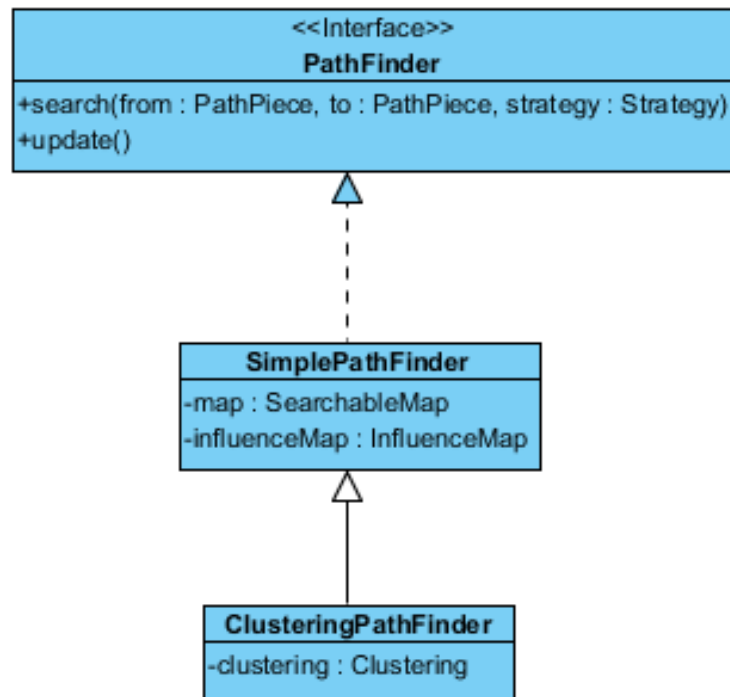


Abbildung 4.1: Klassendiagramm Pfadsuche

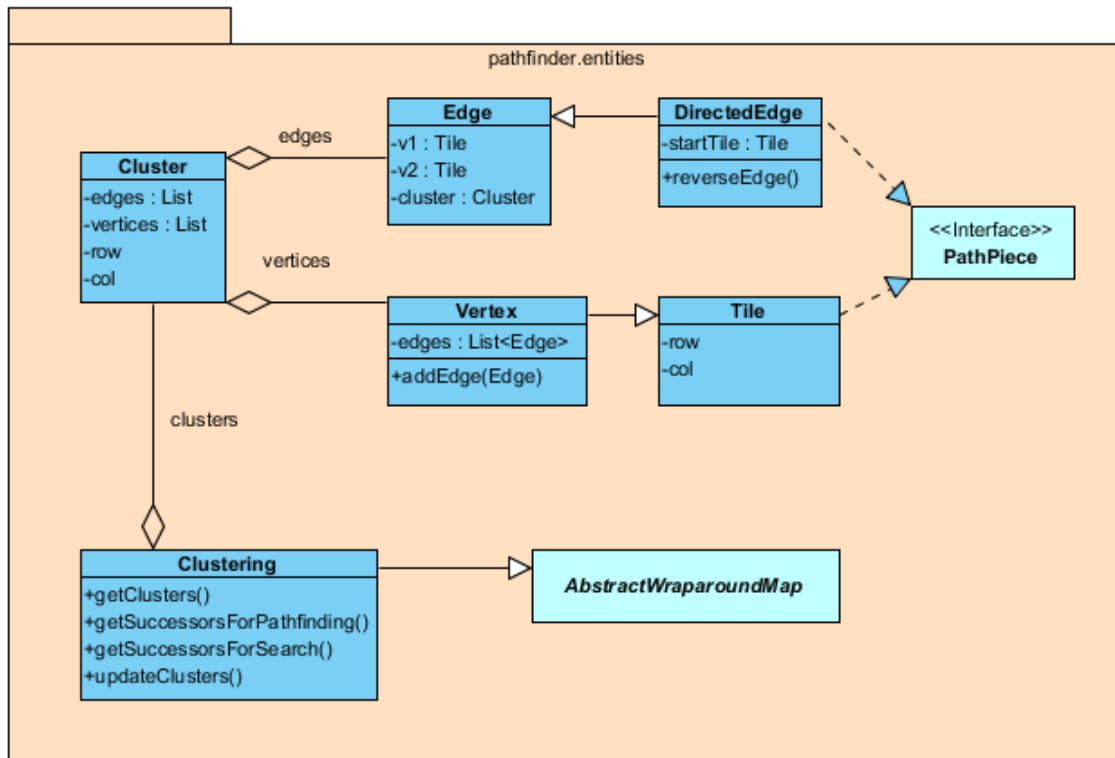


Abbildung 4.2: Spiel-Elemente für die Suche (vereinfacht)

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.

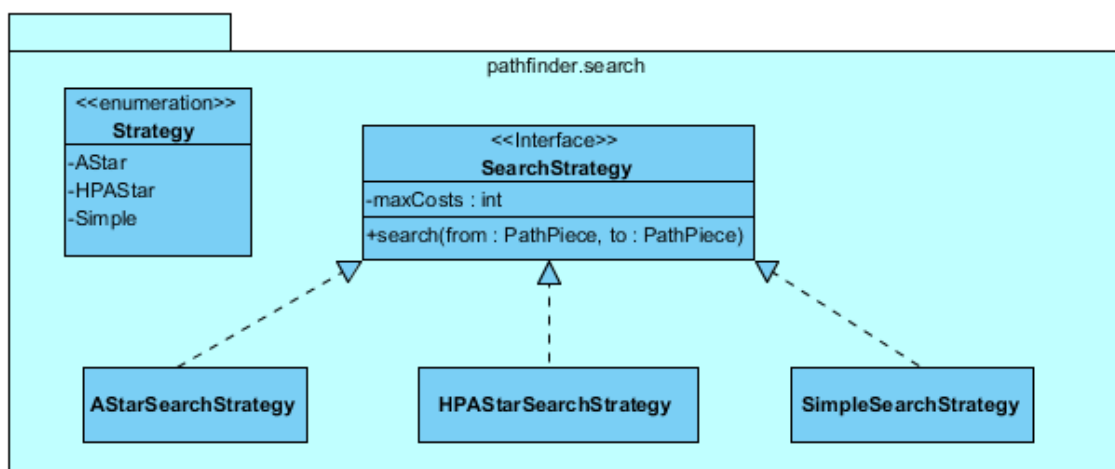


Abbildung 4.3: Suchstrategien



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.

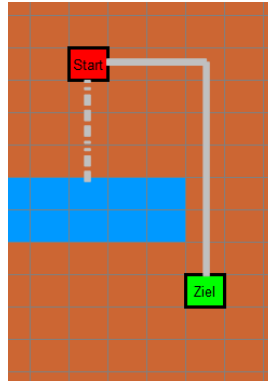


Abbildung 4.4: Simple-Path Algorithmus

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);
```

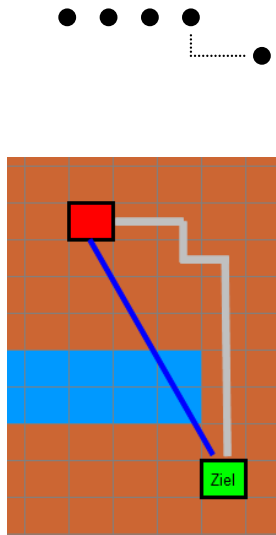


Abbildung 4.5: Heuristische Kosten (blau), Effektive Kosten (grau)

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 4.7 wurde die Karte in 16 Clusters aufgeteilt.

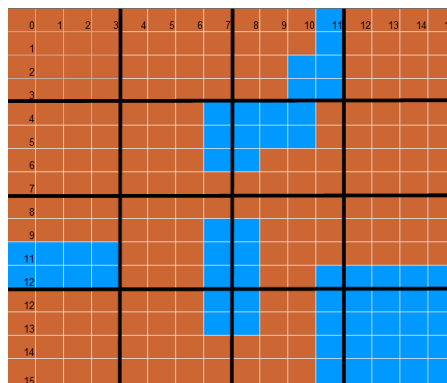


Abbildung 4.6: Clustereinteilung auf der Landkarte. Clustergrösse 4x4, Landkarte 16x16

Wir unterscheiden zwischen den zwei Clusterarten Centered und Corner. Die Variante Corner wurde bereits im Vormodule 'Projekt 2' implementiert während die Variante Centered im Laufe dieser Arbeit dazu kam. Folgendes Bild zeigt den Unterschied der Varianten. Corner generiert zwei Übergangspunkte plus eine Verbindung auf der Kante zwischen zwei Clusters während Centered nur einen Übergangspunkt in der Mitte des Übergangs generiert. Die Variante Centered hat den Vorteil, dass es weniger Pfade zum durchlaufen gibt. Aber sie hat auch den Nachteil, dass der gefundene Pfad, nicht in jedem Fall der Kürzeste ist und ein Pathsmoothing gemacht werden muss.

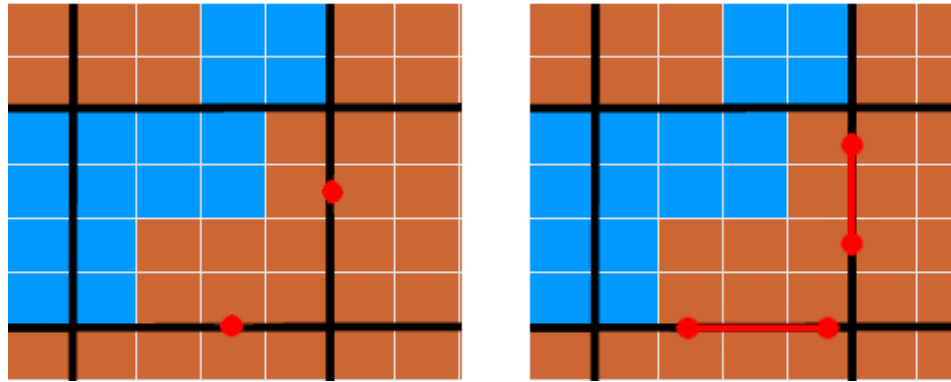


Abbildung 4.7: Vergleich der Clusterarten: Links der Typ Corner, Rechts der Typ Centered

Nachfolgend wird erläutert wie das Clustering vonstatten geht, verwendet wird die Clusteringart Corner. Nach dem Einteilen der Cluster werden für jeden Cluster und einen Nachbar-Cluster aus der Vierer-Nachbarschaft die Verbindungskanten berechnet. Dies kann natürlich nur für Clusters gemacht werden die auf einem sichtbaren Teil der Landkarte liegen, was zu Beginn des Spiel nicht 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.

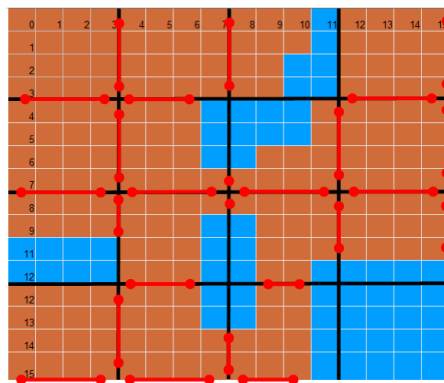


Abbildung 4.8: Die Kanten jedes Clusters wurden berechnet

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.

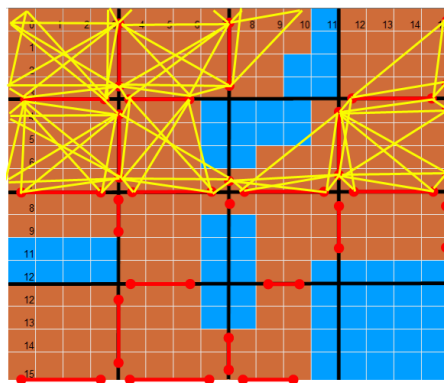


Abbildung 4.9: Darstellung der Innenkanten



In der Abbildung 4.10 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. Der resultierende Pfad könnte nun via Pathsmoothing verkürzt werden.

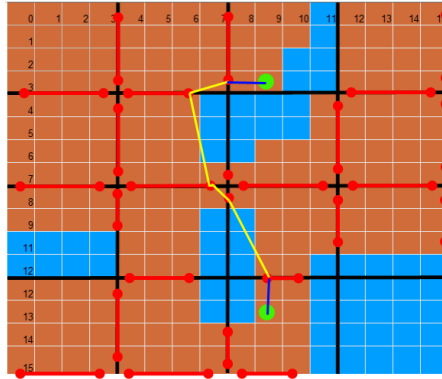


Abbildung 4.10: Errechneter Weg mittels HPA*

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.

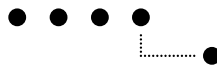
```
ClusteringPathFinder pf = new ClusteringPathFinder(map, 10, type);
pf.update();
List<Tile> path = pf.search(PathFinder.Strategy.HpaStar, start, end, -1);
```

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 gegnerischen 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.

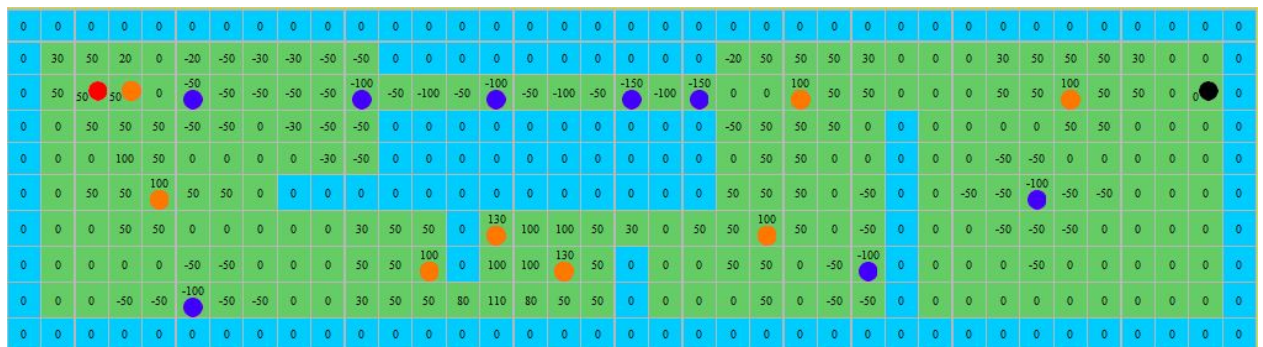


Abbildung 4.11: Ausgangslage Pfadsuche mit A* und InfluenceMap

Ohne Berücksichtigung der InfluenceMap würde der A* Algorithmus einen Pfad finden der auf direktem Weg waagrecht zum Zielpunkt führt. Sobald aber die InfluenceMap berücksichtigt wird, führt der Pfad nicht mehr auf dem 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).

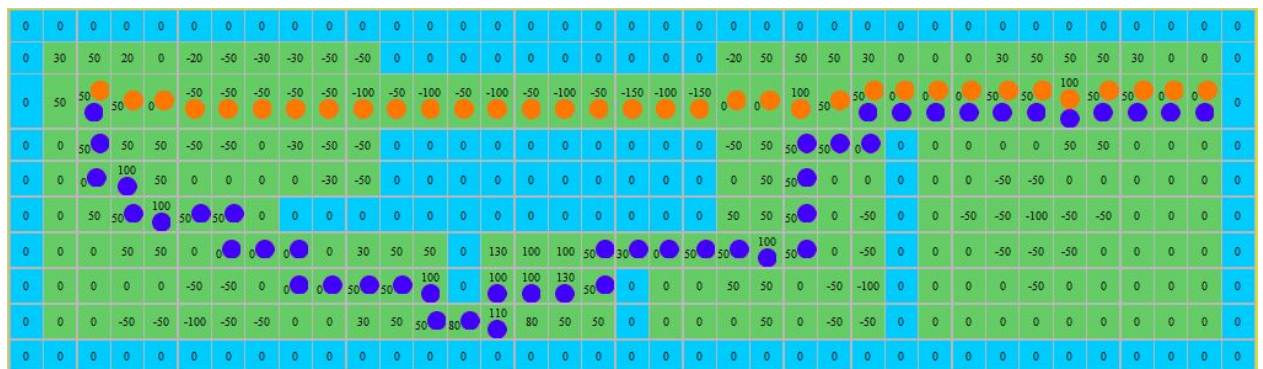


Abbildung 4.12: Resultierende Pfade mit und ohne Berücksichtigung der Influence Map

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

Tabelle 4.1: Pfadkosten mit und ohne Berücksichtigung der Influence Map

4.2.5 Pathsmoothing

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

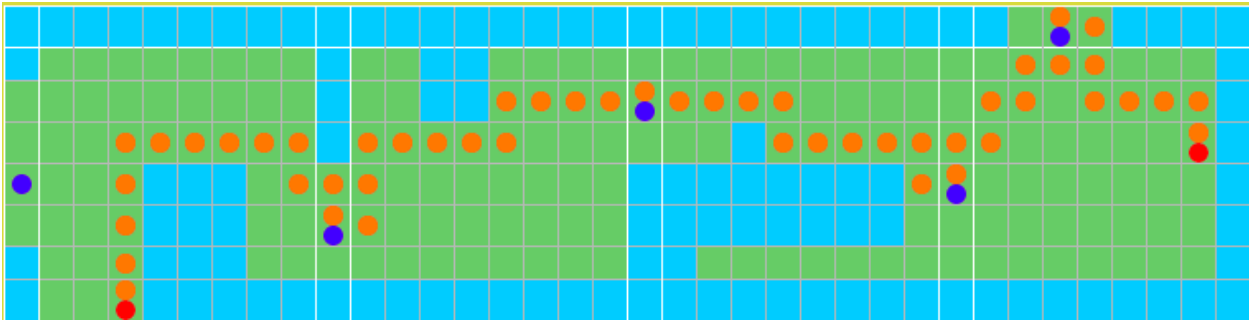


Abbildung 4.13: Der gefundene Pfad mit HPA* Clusteringart Centered, ist nicht der kürzeste.

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

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

```
public List<Tile> smoothPath(List<Tile> path, int size, boolean recursive) {
    int start = 0;
    int current = size;
    List<Tile> smoothedPath = new ArrayList<Tile>();
    // do while the last tile of the smoothed path is the same as in path
    do {
        List<Tile> subPath = path.subList(start, current);
        int manDist = map.manhattanDistance(subPath.get(0), subPath.get(subPath.size() - 1)) + 1;
        List<Tile> newSubPath = null;
        if (manDist < subPath.size()) {
            newSubPath = search(strategy.AStar, subPath.get(0), subPath.get(subPath.size() - 1), subPath.size());
        }
        if (newSubPath != null) {
            smoothedPath.addAll(newSubPath);
            if (recursive && newSubPath.size() < subPath.size()) {
                smoothedPath = smoothPath(smoothedPath, smoothedPath.size(), true);
            }
        } else {
            smoothedPath.addAll(subPath);
        }
        start = current;
        current = Math.min(current + size, path.size());
    } while (!path.get(path.size() - 1).equals(smoothedPath.get(smoothedPath.size() - 1)));

    return smoothedPath;
}
```

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

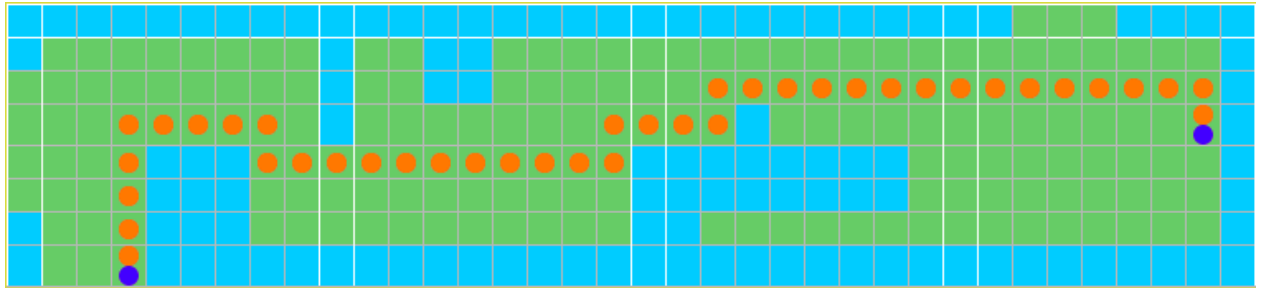
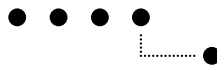


Abbildung 4.14: Der geglättete Pfad, nach Anwendung des PathSmoothing-Algorithmus

4.3 Breitensuche

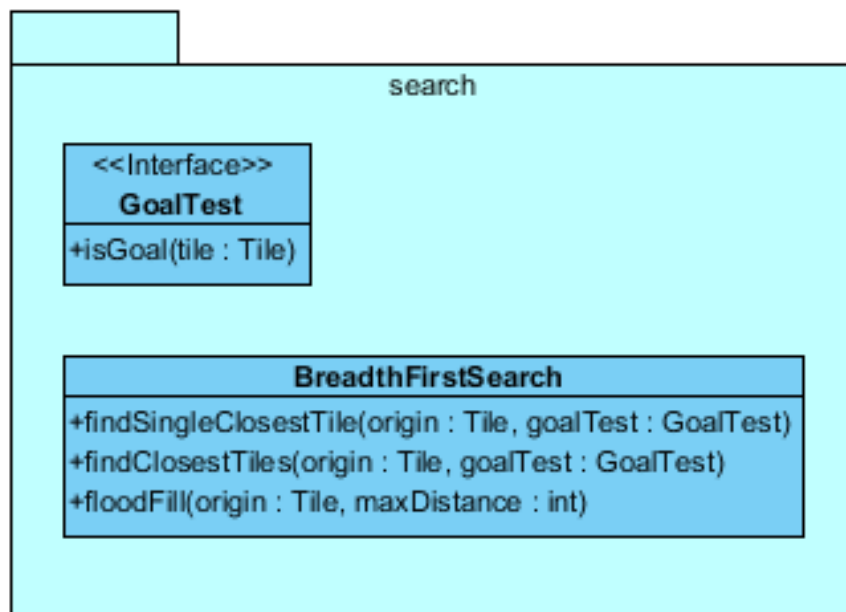


Abbildung 4.15: Breitensuche Klassendiagramm

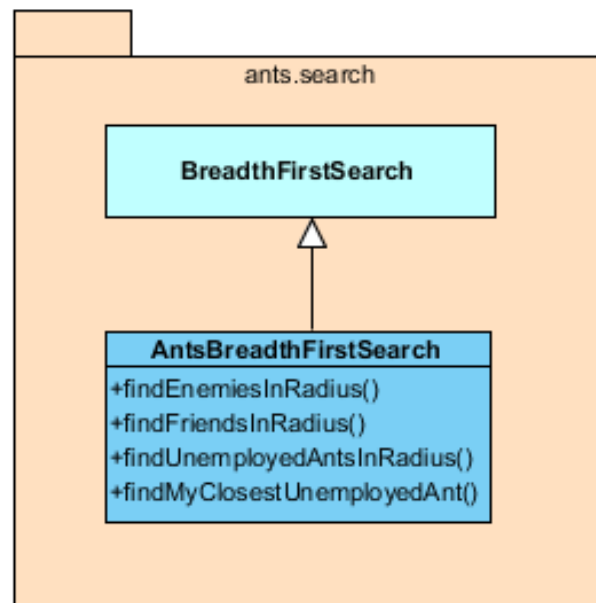


Abbildung 4.16: Breitensuche Ants-spezifisch

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.

```

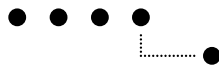
AntsBreadthFirstSearch bfs = new AntsBreadthFirstSearch(Ants.getWorld());
Tile food = new Tile(20,16);
Tile antClosestToFood = bfs.findSingleClosestTile(food, 40, new GoalTest() {
    @Override
    public boolean isGoal(Tile tile) {
        return isAntOnTile(tile);
    }
});
  
```

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

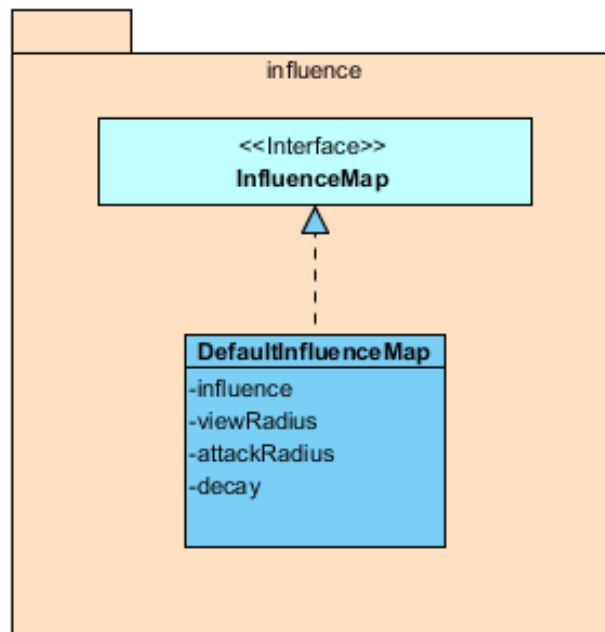


Abbildung 5.1: Influence Map Klassendiagramm

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, der Angriffsradius, der erweiterte Angriffsradius und der Sichtradius. Den Radien haben wir folgende Werte zugewiesen.

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

Tabelle 5.1: Einfluss einer Spieleinheit

* 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 orangenen 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.

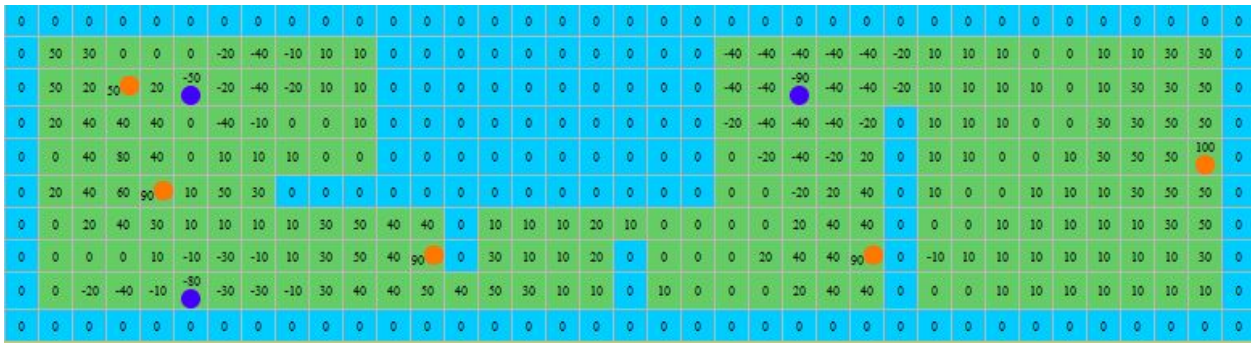


Abbildung 5.2: Influence Map, dargestellt ist die Sicherheit je Tile.

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

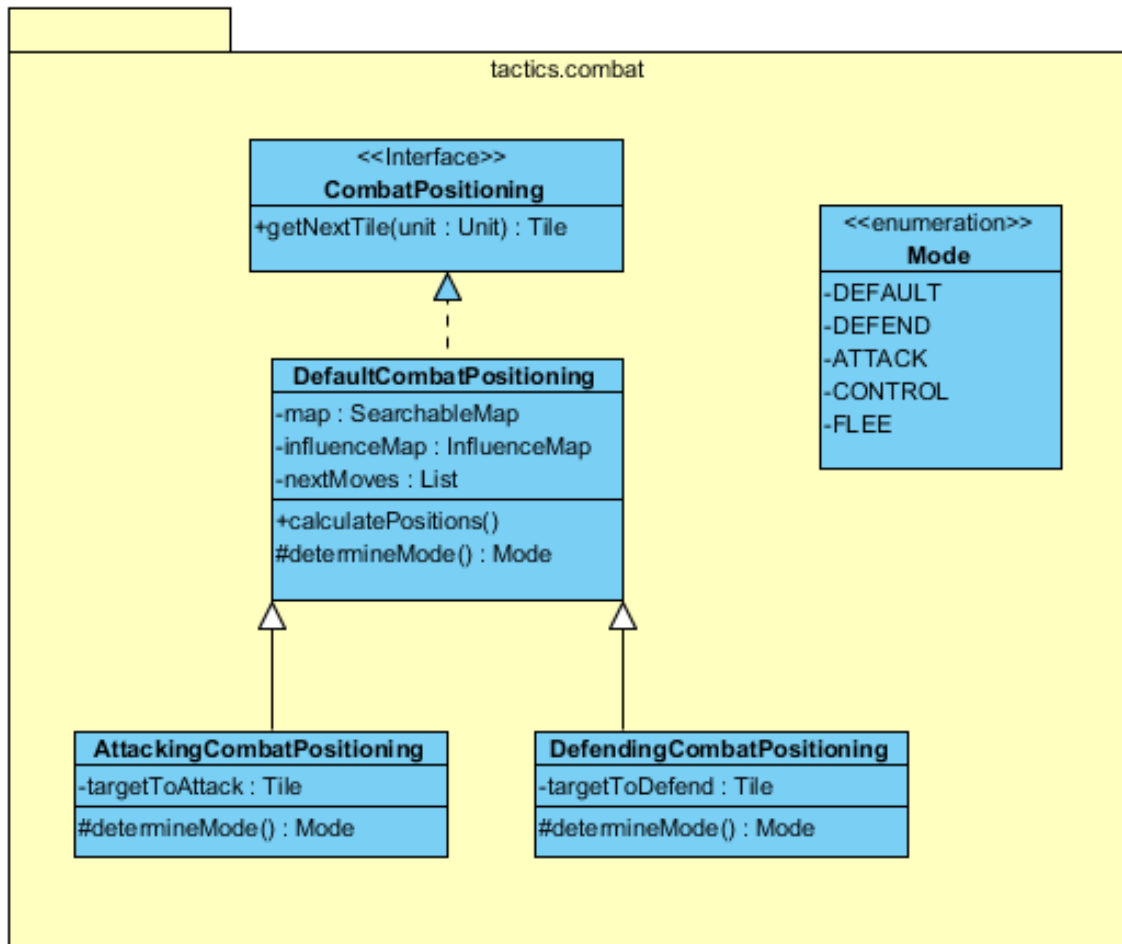


Abbildung 5.3: CombatPositioning Klassendiagramm

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

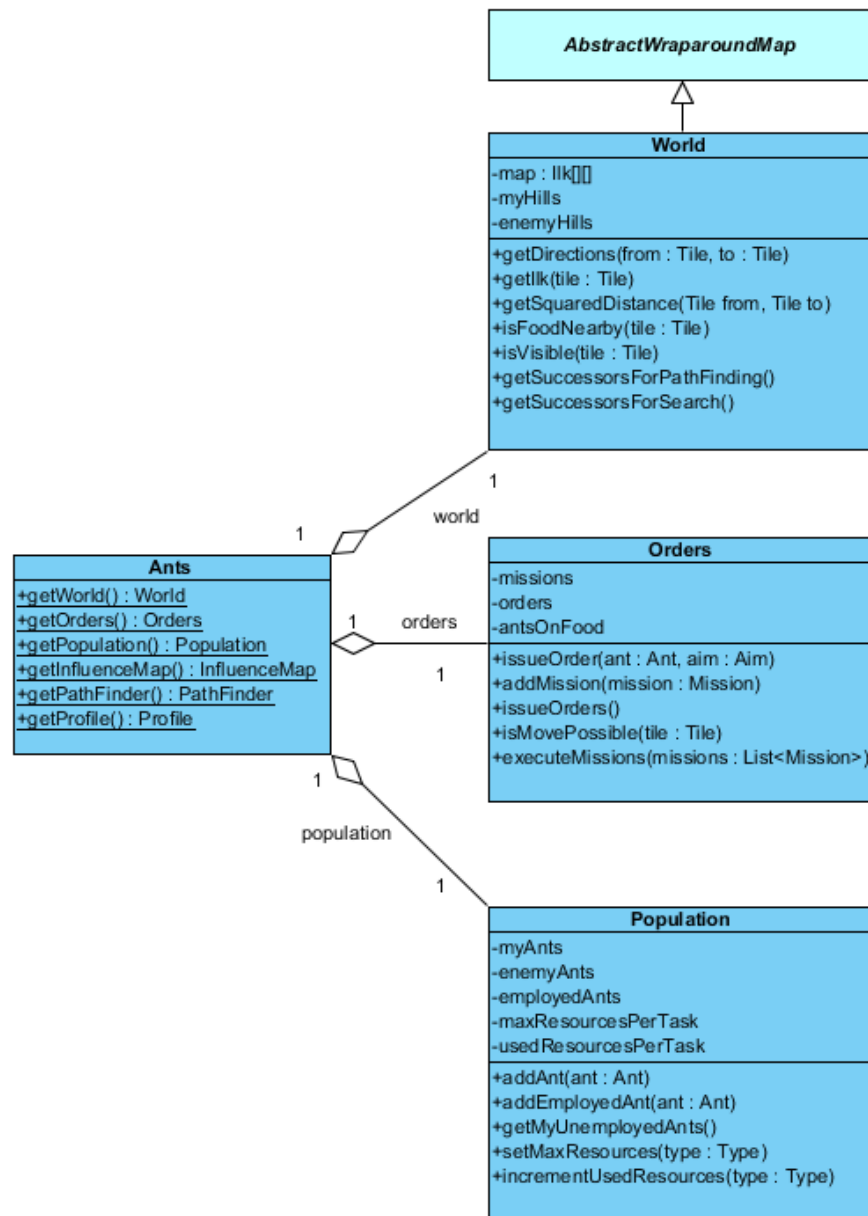


Abbildung 6.1: State-Klassen (vereinfacht)

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)

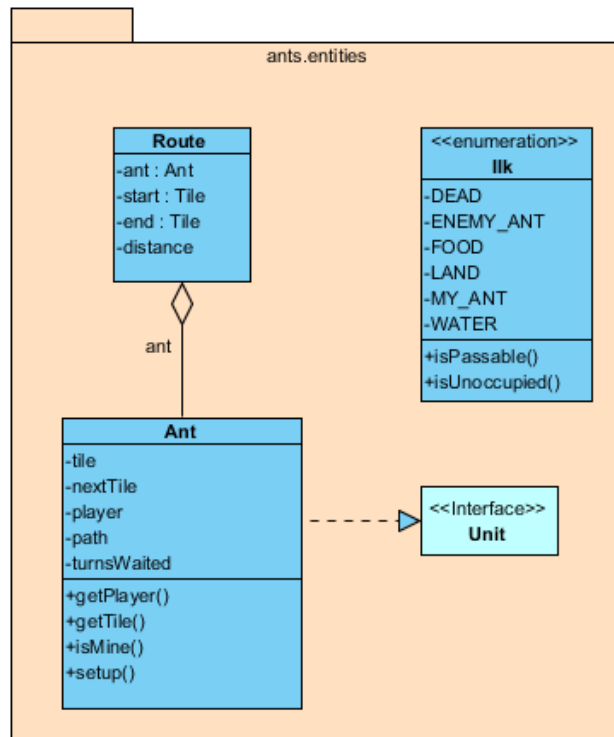


Abbildung 6.2: Ants-spezifische Elemente der Spielwelt (vereinfacht)

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

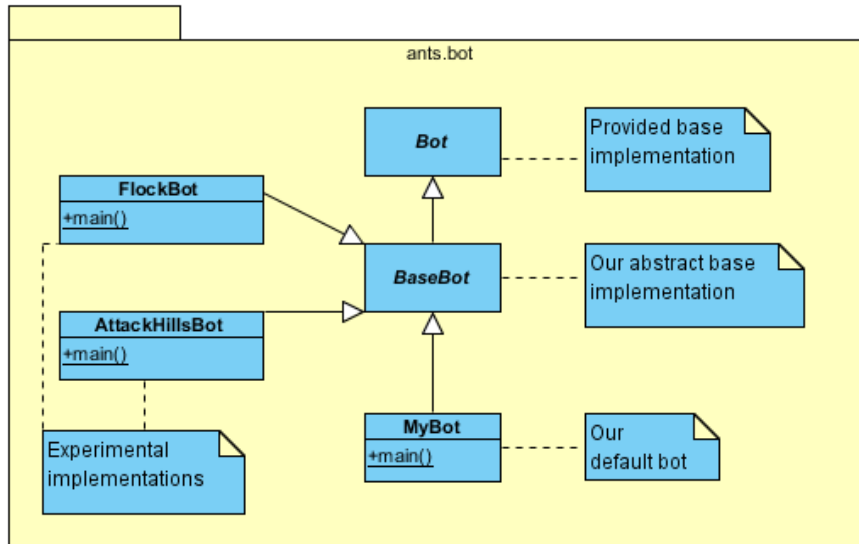


Abbildung 6.3: Vererbung der Bots wobei auf Stufe impl (Implementation) nur MyBot verwendet wird.

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

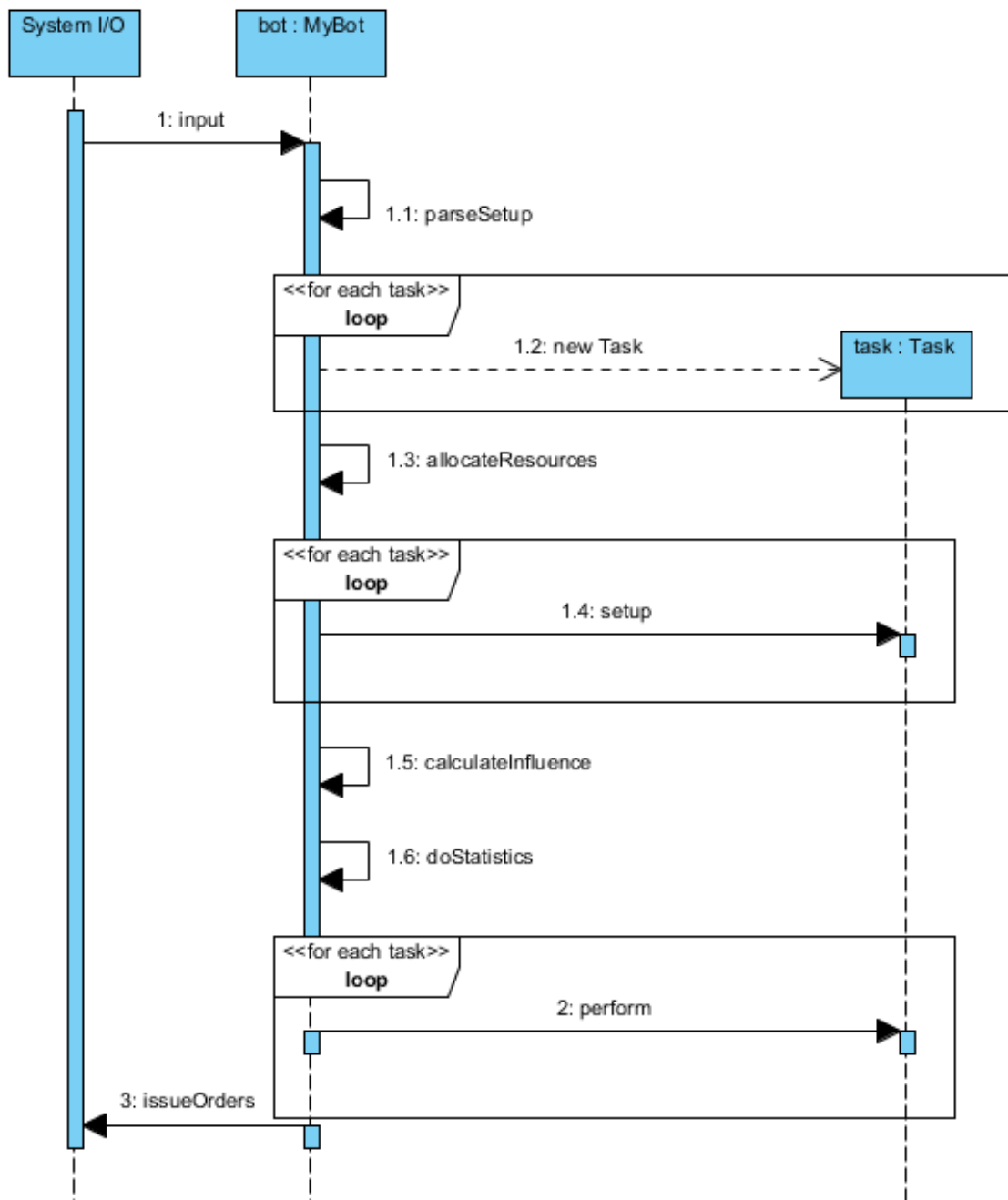


Abbildung 6.4: Ablauf des ersten Zugs des Spiels

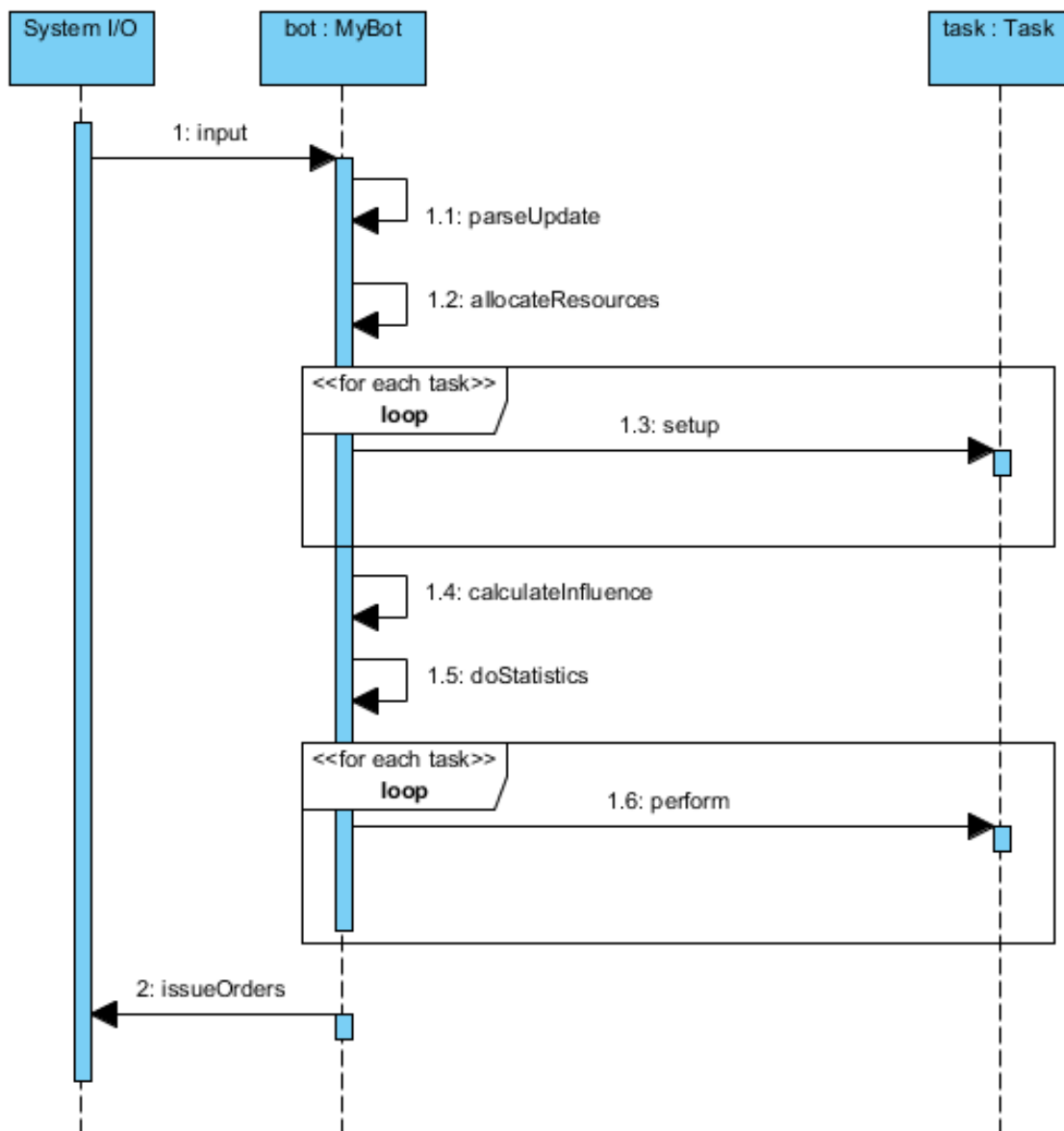


Abbildung 6.5: Ablauf der weiteren Züge des Spiels

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 8.3) 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

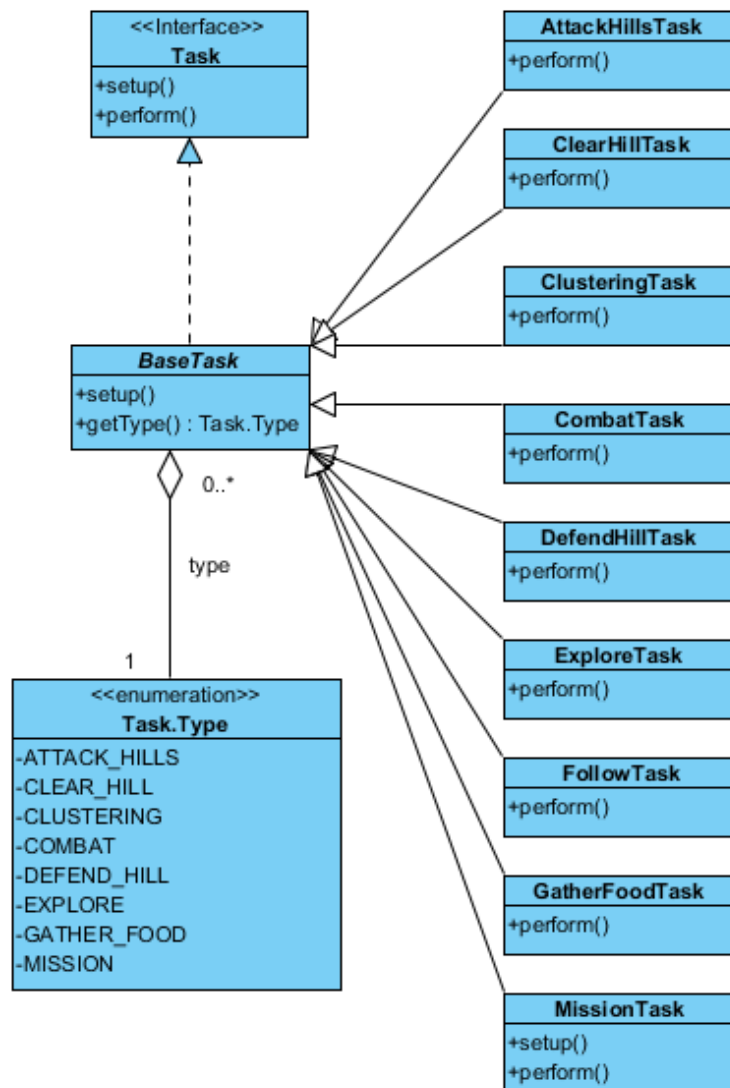


Abbildung 6.6: Tasks

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 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 sichtbar sind. (Fog of War)



6.4.7 ClearHillTask

Dieser Task bewegt alle Ameisen, welche neu aus unserem Hügel 'schlüpfen', vom Hügel weg. So werden nachfolgende Ameisen nicht 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.

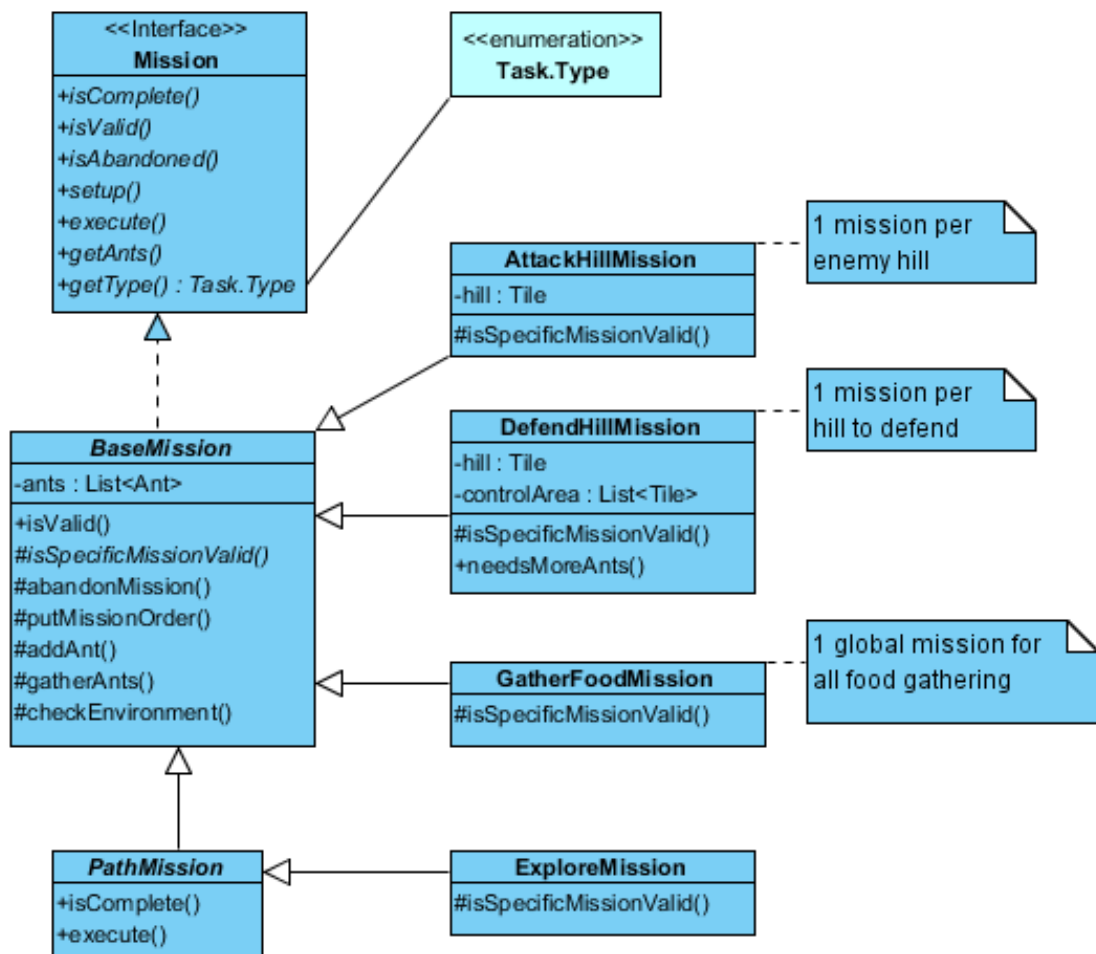


Abbildung 6.7: Missionen und ihre Hierarchie

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) Der Hügel wird erst zerstört wenn das Spiel endet, oder sich der Gegner mit anderen Ameisen dem Hügel nähert.

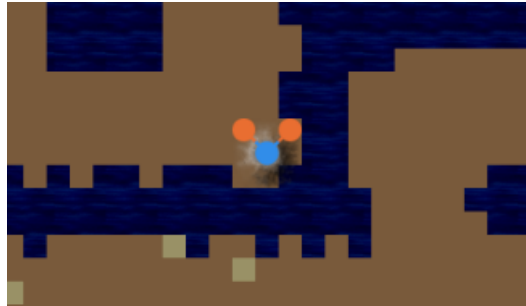


Abbildung 6.8: Ein gegnerischer Hügel wird durch unsere Ameisen (orange) kontrolliert.

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 initialisiert und die Gruppe, die gegnerischen Ameisen sowie der Meilenstein als Ziel mitgegeben. Die Klasse berechnet die Züge für die Gruppe. (Details siehe 5.2

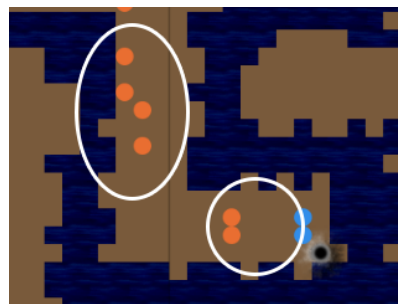


Abbildung 6.9: Die erste zweier Gruppe ist in Kampfstellung. 4 weitere Ameisen rücken zur Front auf.

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.

Beim Entwickeln dieser Mission sind wir auf folgende Fragen gestossen: Ab welchem Spielzug soll der Hügel bewacht werden? Wieviel Ameisen sollen für die Verteidigung eingesetzt werden? Was wenn der Hügel von mehreren Seiten angegriffen wird? Sollen alle Hügel bewacht werden? Aus diesen Frage ergab sich folgende Implementation:

Die DefendHillMission bietet zwei Modi zur Verteidigung an. Die Default-Verteidigung und der Barrier-Modus. Bei der Default-Verteidigung werden die Ameise nahe um den eigenen Hügel zur Verteidigung aufgestellt. Der



Barrier-Modus ist nur bei Hügelmöglich die in einer Sackgasse sind, den in diesem Modus wird eine Sperre an der engsten Stelle erstellt. Nachfolgende Abbildung veranschaulicht die beiden Modi.

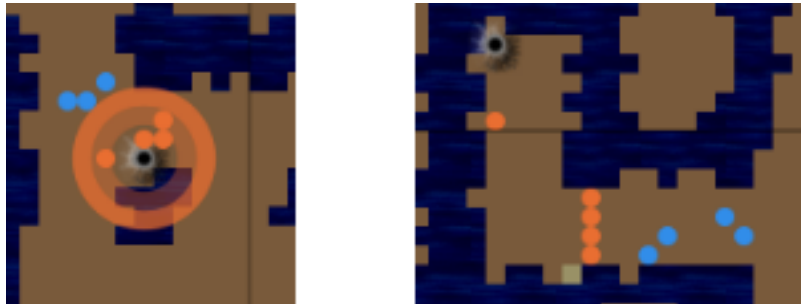


Abbildung 6.10: Links die Default-Verteidigung, rechts mit einer Sperre.

Der Modus wird wie folgt definiert. Während dem 10. Spielzug wird versucht mittels spezieller Breitensuche eine Sperre für den Hügel zu finden. Falls dies gelingt läuft die Mission im Barrier-Mode. Dies solange bis eine gegnerische Ameise auf die Höhe der Barriere tritt, dann gehen wir davon aus, dass die Sperre vom Gegner eingenommen wurde, wir gehen in den Default-Modus zurück.

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

6.5.5 Default-Modus

Hier wird jегentlich berechnet, wieviele Ameisen den Hügel angreifen. Die Angreifer und die Verteidiger werden dem DefendingCombatPositioning übergeben, dieses berechnet die Positionen der Ameisen. Es ist einstellbar ab welchem Zug die DefendMission Ameisen für die Verteidigung rekrutiert. Zudem kann mit dem Parameter `textifDEFENDERMORETHANATTACKERS` definiert werden wieviel Ameisen mehr als die Anzahl Angreifer rekrutiert werden.

6.5.6 Barrier-Modus

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

Die Sperre wurde gegen Xathis getestet und sie zeigte nur eine geringe Wirkung. Der Bot von Xathis wartet vor der



Sperre bis er in Überzahl ist und überrumpeln dann unsere Verteidigung. Erstaunlicher Weise wird die Sperre von Greentea (zweitbesten Bot), auch wenn er in Überzahl ist, nicht angegriffen. Gegen diesen Gegner war die Sperre also gutes Mittel zur Verteidigung.

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



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

6.5.7 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

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

6.5.8 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 Beginn 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.

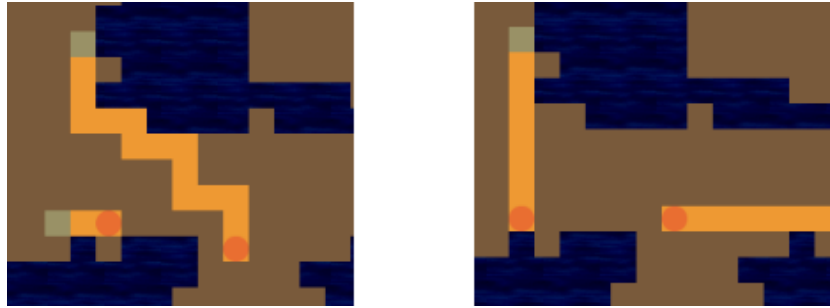


Abbildung 6.13: Ameisen am Futter einsammeln.

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.

6.5.9 Verworfen und nicht verwendete Mission

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

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

6.6 Ressourcenverwaltung

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

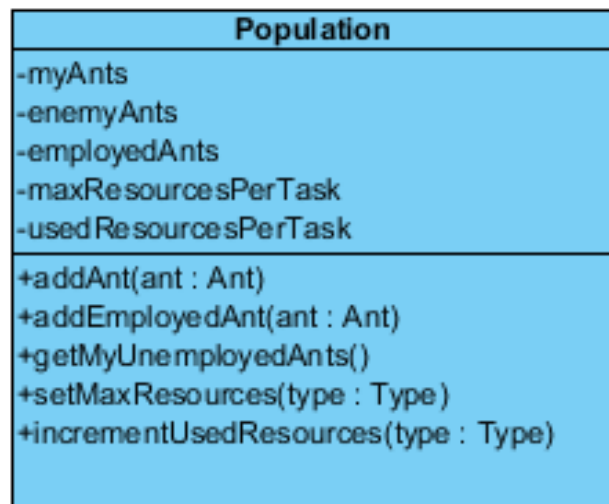


Abbildung 6.14: Population Klasse

Abbildung 6.15 zeigt unsere Population-Klasse (s. Kapitel 6.1), die als zentrale Verwaltungsinanz für unsere Ameisen auch dafür zuständig ist, die allozierten Ressourcen zu verwalten und beim Zuweisen einer Ameise zu einer Aufgabe sicherzustellen, dass das Kontingent nicht überschritten wird.

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

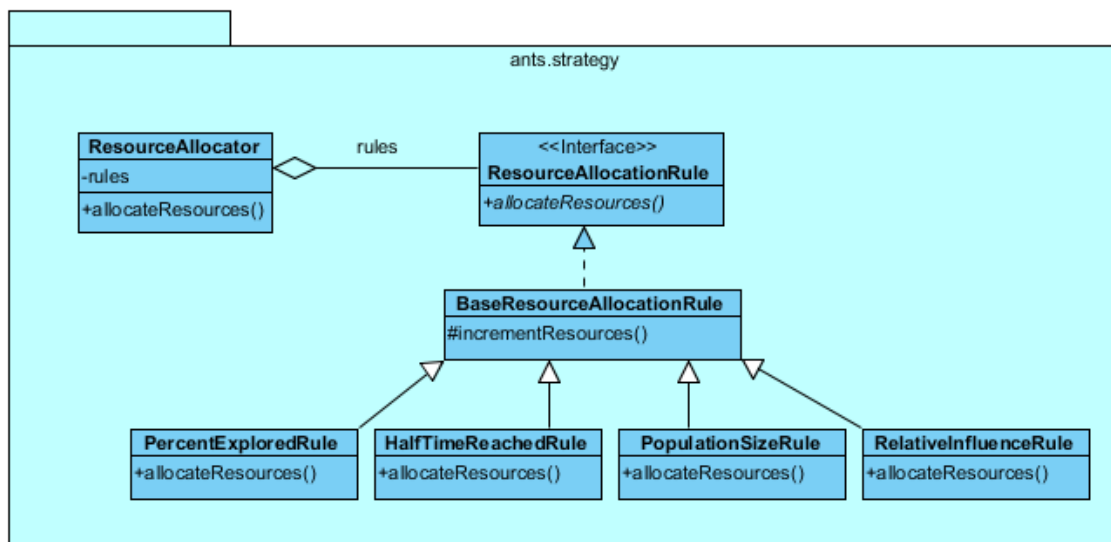


Abbildung 6.15: RessourcenManagement

Abbildung 6.16 zeigt die Klassenhierarchie unserer Ressourcenverwaltung. Die zentrale Klasse ist der `ResourceAllocator`; mit dessen Methode `allocateResources()` werden jeweils die Ressourcen neu verteilt:

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

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



6.7 Implementierte Regeln

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

Unsere implementierten Regeln sind:

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

All diese Regeln sind über Profile konfigurierbar.

Diese Regeln sind natürlich nur eine Auswahl dessen, was mit dieser regelbasierten Ressourcenverwaltung möglich wäre. Viele weitere Regeln wären denkbar, aber wir haben in unseren Tests bereits mit diesen vier einfachen Regeln durchaus zufriedenstellende Ergebnisse erzielt. Eine Schwierigkeit bei der Implementierung neuer Regeln gilt es noch zu beachten: Durch die grosse Dynamik des Systems sind die Auswirkungen einer Änderung manchmal nur schwer abzuschätzen; dieser Effekt wird durch die Konfigurierbarkeit des Systems noch verstärkt. Die Berechenbarkeit des Verhaltens unseres Bots ist also grösser, wenn nicht allzu viele Regeln in die Ressourcenverwaltung eingebunden werden.

6.8 Profile

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

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



- **explore.dominantPositionBoost:** Wenn wir eine dominante Position auf dem Spielfeld haben, wird die Erkundung gestärkt, um die Karte zu kontrollieren. Dieser Parameter bestimmt, wie viel zusätzliche Ressourcen zugeteilt werden.
- **attackHills.dominantPositionBoost:** Wenn wir eine dominante Position auf dem Spielfeld haben, wird der Angriff auf gegnerische Hügel forciert, um Punkte zu sammeln. Dieser Parameter bestimmt, wie viel zusätzliche Ressourcen zugeteilt werden.
- **attackHills.halfTimeBoost:** Nach Ablauf der Hälfte der Spielzeit erhöhen wir die Aggressivität unseres Bots. Dieser Parameter bestimmt, wie stark der Angriff auf gegnerische Hügel in der 2. Halbzeit forciert wird.

6.8.1 Validierung

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

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

6.8.2 Definierte Profile

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

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

Tabelle 6.1: Definierte Profile

6.8.3 Weiterentwicklungspotenzial

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



7 Logging

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

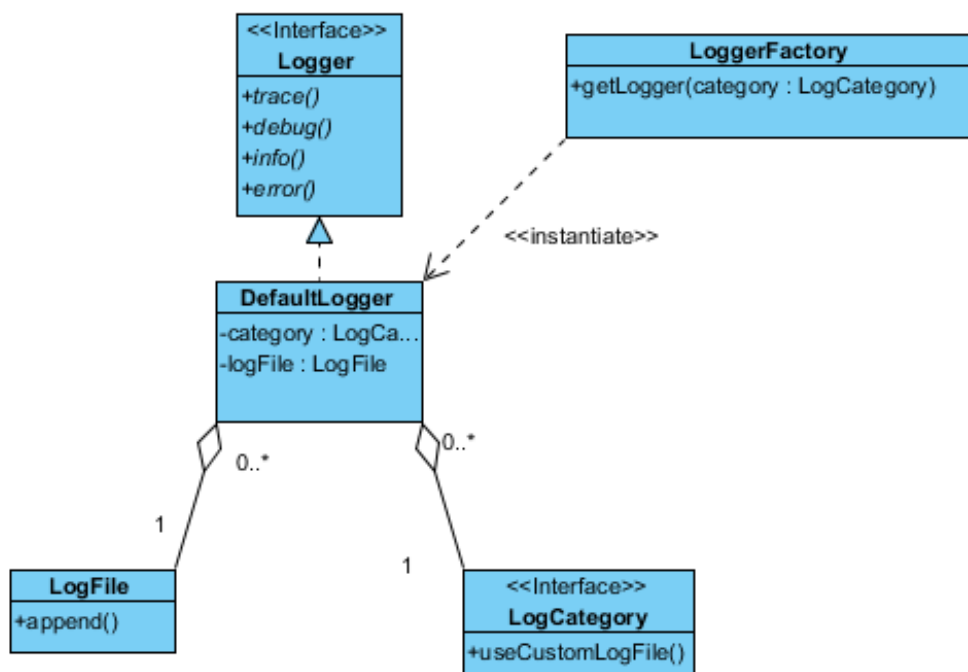


Abbildung 7.1: Logging Klassen

Abbildung 7.1 zeigt die wichtigsten Klassen des Logging-Frameworks.

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

¹<http://docs.oracle.com/javase/6/docs/api/java/util/Formatter.html#syntax>



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

7.1 Konfiguration

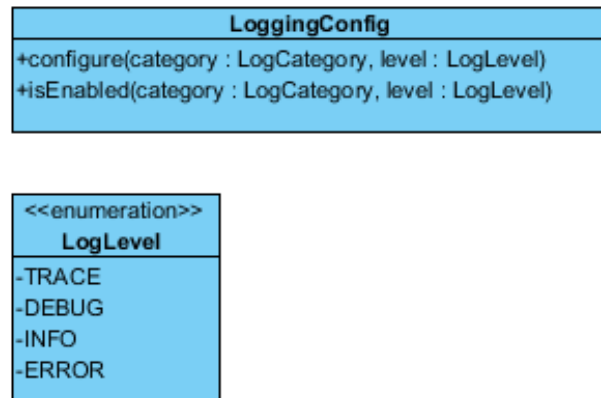


Abbildung 7.2: Logging Konfiguration

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

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

7.2 Anwendungsbeispiele

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

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

Folgende Codezeilen zeigen, wie ein Logger instanziiert wird:

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

Folgende Codezeilen zeigen, wie ein Logger verwendet wird:

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




7.3 JavaScript Addon für HMTL-Gameviewer

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

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

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



Abbildung 7.3: Im Popufenster 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 7.4)

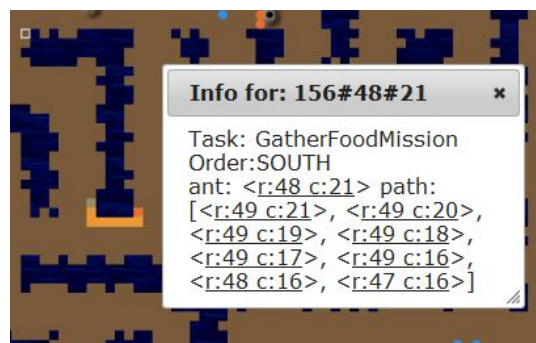


Abbildung 7.4: Mit der erweiterten Version wird der Pfad (orange) der Ameise von `<r:48 c:21>` nach `<r:47 c:16>` auf der Karte abgebildet.



7.4 Profile und Logging

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

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



8 Testing

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

8.1 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.

8.2 Visuelle Tests

In Form eines UnitTest haben wir auch unsere visuellen Tests geschrieben. Durch eine visuelle Überprüfung des Resultats 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 abgebildet mit welchem wir die Korrektheit des Clustering und des HPA* Algorithmus visuelle überprüft haben.

TODO mapoutput01?

8.3 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

8.4 Testreport Profile

Um die verschiedenen Profile unseres Bots zu testen, führten wir diverse Testläufe durch, in denen wir die verschiedenen konfigurierten Bots jeweils 100 Mal gegen verschiedene Gegner und gegeneinander antreten liessen. Für diese Testläufe wurden die in Tabelle 6.1 aufgeführten Profile verwendet.

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

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

Tabelle 8.1: Bilanz der Testläufe gegen egreavette

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

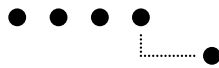
Als nächstes führten wir einen Testlauf gegen den Sieger des Wettbewerbs durch. Mathis Lichtenberger (Username xathis) hatte seinen Bot ebenfalls über GitHub zur Verfügung gestellt: <https://github.com/xathis/AI-Challenge-2011-bot>.

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

Tabelle 8.2: Bilanz der Testläufe gegen xathis

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

Als nächstes liessen wir unsere 4 Profile gegeneinander antreten:



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

Tabelle 8.3: Bilanz der Testläufe gegeneinander

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

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

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

Tabelle 8.4: Die Profile für den 2. Testlauf

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

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

Tabelle 8.5: Bilanz der 2. Testläufe gegen xathis

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



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

Tabelle 8.6: Bilanz der 2. Testläufe gegeneinander

Wie Tabelle 8.6 zeigt, waren die Unterschiede zwischen den neuen Profilen im Kampf gegeneinander schon deutlich sichtbarer. Die extremeren Profile weisen alle eine deutlich schlechtere Bilanz auf, wobei die Defensive Konfiguration noch am besten fährt. Gegen diese Profile ist unsere ausgewogene Standardkonfiguration aber bereits ein deutlicher Sieger.

8.4.1 TestReader

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

8.5 Online Tests

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



9 Spielanleitung

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 verschiedene 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. Die Targets `mazeAgainst*` führen jeweils ein Spiel auf einer komplexeren und grösseren, labyrinthartigen Karte aus, und zwar gegen den jeweils bezeichneten Gegner.
4. Das Target `maze4Players` führt ein Spiel gegen drei andere Bots auf einer noch etwas grösseren Karte durch.
5. Die Targets `mazeProfiles` und `mazeProfiles4Players` sind dazu da, mehrere Kopien unseres Bots mit verschiedenen Profilen gegeneinander antreten zu lassen. Wie im Kapitel 7.4 erwähnt, muss dazu erst in der Klasse `LiveInfo` das `LiveInfo-Logging` deaktiviert werden:

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

6. Das Target `testOnline` lässt den Bot über einen privaten TCP-Server gegen andere Bots antreten, unter ähnlichen Bedingungen wie beim eigentlichen Wettbewerb, nur leider mit deutlich weniger Teilnehmern. (s. Kapitel 8.5)
7. Die Targets `repeat*` dienen der Durchführung der Profil-Testläufe und sind ansonsten wenig interessant.
8. Die Target `debug*` dienen dem Starten unserer Testbots, die wir während der Entwicklung einsetzten (s. Kapitel 8.3).

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

Im Unterordner `tools` befindet sich die in Python implementierte Spiel-Engine. Unter `tools/maps` liegen noch weitere vordefinierte Umgebungen, und unter `tools/mapgen` liegen 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.

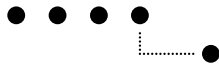
Im Unterordner `tools/bots` haben wir Kopien der Bots von einigen Teilnehmern abgelegt, gegen die wir im Zug der Entwicklung immer wieder getestet haben.





Glossar

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





Literaturverzeichnis
