Tigers Mannheim

Taktische Spielfeldanalyse im Robocup mittels Rasterung des Spielfelds

Teil A

STUDIENARBEIT
des Studienganges Informationstechnik
an der Dualen Hochschule Baden-Württemberg Mannheim

Oliver Steinbrecher 933748
Paul Birkenkampf 4278912

Mannheim, den 09. Januar 2012

DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

Bearbeitungszeitraum:	01.10.2011 - 09.01.2012
Kurs:	TIT09ANS
Ausbildungsfirma:	Deutsches Zentrum für Luft- und Raumfahrt
Betreuer:	Prof. Dr. Rainer Colgen

Ehrenwörtliche Erklärung

Hiermit versichern wir, dass wir die vorliegende Arbeit selbstständig
und nur unter Benutzung angegebener Quellen und Hilfsmittel angefertigt haben.

Oliver Steinbrecher

Paul Birkenkampf

Mannheim, den 09. Januar 2012

Inhaltsverzeichnis

Abbildungsverzeichnis **ii**

1 Abstract **1**

2 Einleitung **2**
2.1 Roboterfuball . . . 2
2.2 Das Team Tigers Mannheim . . . 3
2.3 Ziele und Anforderungen . . . 3

3 Entwurf und Implementierung der Spielfeldanalyse **5**
3.1 Einordnung in die Zentralsoftware Sumatra . . . 5
3.2 Aufteilen des Spielfeldes in Rechtecke . . . 6
3.2.1 Rechteck-Klassen . . . 8
3.2.2 Performanceverbesserung bei der Arbeit mit Rechteckklassen . . . 10
3.3 Definition der Güte eines Rechtecks . . . 10
3.4 Entwurf zur Bewertung einzelner Rechtecke . . . 10
3.4.1 Ansätze für Bewertungsalgorithmen . . . 11
3.4.2 Echtzeitproblematik . . . 13
3.5 Ablauf der Spielfeldanalyse . . . 14

4 Entwurf und Implementierung der Visualisierung **15**
4.1 Grundstruktur in der Zentralsoftware . . . 15
4.1.1 Modell View Presenter (MVP) . . . 15
4.1.2 Umsetzung von MVP in Sumatra . . . 16
4.2 Die grafische Komponente im Detail . . . 17
4.2.1 Bisherige Struktur . . . 17
4.2.2 Notwendige Änderungen . . . 18
4.3 Java AWT Komponenten dekorieren . . . 18
4.4 Implementierung . . . 18
4.4.1 Design einer Zeichenebene . . . 18
4.4.2 Zusammenfassen von Zeichenebenen . . . 20
4.4.3 Gesamtdarstellung . . . 20

5 Zusammenfassung und Ausblick **21**

Literaturverzeichnis **I**

Abbildungsverzeichnis

2.1 Zwei Roboter der Small Size League der Tigers Mannheim 3

3.1 Modularer Aufbau von Sumatra 6
3.2 UML-Diagramm der Rechteckstruktur 9
3.3 Ergebnis der ersten Rasteranalyse 12
3.4 Ergebnis der zweiten Rasteranalyse 13
3.5 UML-Diagramm der Feldanalysestruktur 14

4.1 MVP Konzept in Sumatra 17
4.2 UML-Diagramm der Layer-Struktur 19
4.3 *FieldPanel* mit Steuerungsmöglichkeiten für unterschiedliche Layer 20

Kapitel 1

Abstract

Wie beim echten Fußball geht es auch im Roboterfußball darum, mehr Tore als der Gegner zu schießen um zu gewinnen. Die Erfahrungen des Teams Tigers der DHBW Mannheim bei der Roboterfußballweltmeisterschaft im Jahr 2011 zeigten, dass dies eine große Herausforderung ist. Zur Unterstützung der Offensivbemühungen des Teams soll eine automatische Spielfeldanalyse implementiert werden. Diese soll den Entwicklern der Spielzüge neue Möglichkeiten bieten, da so unter anderem freie Räume erkannt werden können. Dazu wird das Feld in viele Rechtecke unterteilt und jedes dieser Rechtecke bewertet. In die Bewertung des Rechtecks geht dabei der Abstand der Roboter zum jeweiligen Rechteck ein. Die so gewonnenen Daten können dann von den Entwicklern von Spielzügen genutzt werden.

As in real football, it is also in robot soccer to score more goals then your opponent to win. The experience of the team Tigers from the DHBW Mannheim in the robot soccer world championship in 2011 had shown that this is a big challenge. To support the offensive efforts of the team, an automatic field analysis should be implemented. This should give the developers of plays new options, as e.g. open spaces can be detected. For this, the field is divided into many rectangles and each of these rectangles is evaluated. In the evaluation of the rectangle, the distance of the robot to the rectangle is taken into account. The obtained data can then be used by the developers.

Kapitel 2

Einleitung

2.1 Roboterfuball

Mensch gegen Computer - Dieser Wettkampf fasziniert die Menschen schon seit den ersten Schritten in der Informationstechnik. Zuerst nur visuell im Spiel Pong, später dann intellektuell im Schach und in naher Zukunft auch physisch in einer der beliebtesten Sportart des Menschen - Fußball.
Beim RoboCup treffen sich Menschen aus aller Welt, um ihre selbstentwickelten Fußballroboter gegeneinander antreten zu lassen und damit auf spielerische Art immer neue Fortschritte in verschiedenen Bereichen, wie z.B. der Elektrotechnik, Bildverarbeitung oder Künstlichen Intelligenz zu entwickeln, testen und vorzuführen. Dabei verfolgen die Entwickler ein hochgestecktes Ziel:

> *"bis zum Jahre 2050 ein Team von autonomen, humanoiden Robotern zu entwickeln, das in der Lage ist, den zu diesem Zeitpunkt amtierenden menschlichen Fußballweltmeister schlagen zu können."* [1]

Der RoboCup und die Weltmeisterschaft der Federation of International Robot-Soccer Association (FIRA) sind die wichtigsten internationalen Wettkämpfe im Roboterfußball. Beide werden finanziell zum größten Teil von Sponsoren getragen ohne die Roboterfußball nicht möglich wäre.
Beim RoboCup gibt es sechs verschiedene Ligen in denen Fußball gespielt wird, sowie zwei die sich nicht mit Fußball beschäftigen. Diese beiden Ligen sind die Rescue League, die sich mit der Entwicklung von Rettungsrobotern beschäftigt und die @Home League, die Roboter als Haushaltshilfen entwickelt. Die Hauptaufmerksamkeit beim RoboCup liegt jedoch auf den Fußballligen. Diese unterscheiden sich nach den verwendeten Robotern (u.a. auch humanoide Roboter verschiedener Größen). Dabei ist das dynamischste Spiel bei den Robotern in der Small Size League (siehe Abbildung 2.1) zu beobachten.

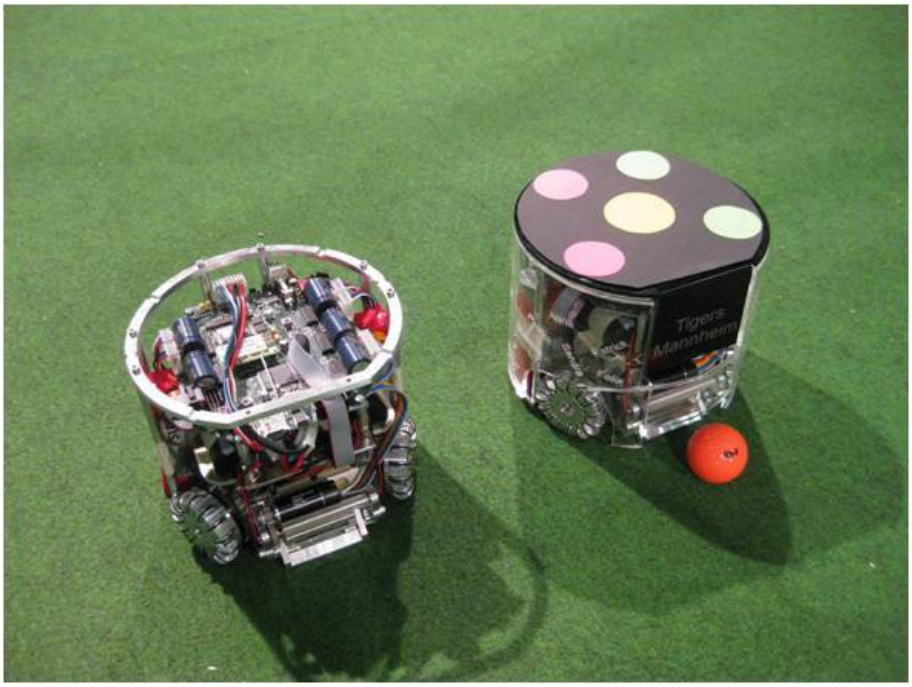

Abbildung 2.1: Zwei Roboter der Small Size League der Tigers Mannheim

2.2 Das Team Tigers Mannheim

Die Tigers (**T**eam **I**nteracting and **G**ame **E**volving **R**obot**S**) der DHBW Mannheim sind ein Team das in der Small Size League aktiv ist. Sie nahmen im Jahr 2011, nach dreijähriger Projektarbeit, erstmals an der Roboterfußballweltmeisterschaft in Istanbul teil. Dort wurden sie inoffiziell als bester Newcomer gehandelt, da sie ein Spiel gewannen und kein Spiel 10:0 verloren (sofortiger Spielabbruch). Im Gegensatz zu vielen anderen Teams bestehen die Tigers Mannheim nur aus Studenten. Die betreuenden Professoren sind nicht aktiv an der Entwicklung des Projekts beteiligt, stehen aber beratend und unterstützend zur Seite.

2.3 Ziele und Anforderungen

Die Teilnahme an der Weltmeisterschaft in Istanbul erwies sich als eine gute Testplattform und offenbarte, dass bereits gute Ansätze zur Verteidigung vorhanden waren, es jedoch an strategisch offensiven Spielzügen mangelte. Dazu zählt beispielsweise der gezielte Pass und dann sofortiger Schuss auf das gegnerische Tor.
Ziel dieser Studienarbeit ist es, ein System zu entwickeln das eine Spielfeldanalyse durchführt. Die so gewonnenen Daten sollen dann als Grundlage zur Planung und Ausführung von offensiven Spielzügen verwendet werden können. Um diese Analyse durchzuführen soll das

Spielfeld gerastert werden, um somit klar definierte Bereiche zu erhalten. Den so entstandenen Rechtecken soll ein Wert hinzugefügt werden, der besagt wie gut oder schlecht das Rechteck für das eigene Team ist (im weiteren Güte genannt). Des Weiteren soll diese Analyse in der Zentralsoftware Sumatra (siehe Abschnitt 3.1) visualisiert werden. Dafür soll ein neues System zur Zeichnung des Spielfelds realisiert werden, das es ermöglicht mit verschiedenen Layern zu arbeiten. Grundsätzlich ist es wichtig, dass die Analyse sehr performant arbeitet und das Gesamtsystem nicht stark verlangsamt.
In dieser Studienarbeit soll lediglich die Datengrundlage geschaffen werden die es den Entwicklern z.B. ermöglicht, beim Offensivspiel die Roboter taktisch günstig zu positionieren. Grundsätzlich wäre es auch möglich die gewonnen Informationen zu Nutzen, um grundlegende strategische Spielentscheidungen zu treffen. Weder die strategische Spielplanung noch die Entwicklung entsprechender Spielzüge, die die Daten nutzen, sollen Bestandteil dieser Arbeit sein.

Kapitel 3

Entwurf und Implementierung der Spielfeldanalyse

3.1 Einordnung in die Zentralsoftware Sumatra

Sumatra ist die Zentralsoftware der Tigers Mannheim. Sie besteht aus verschieden Modulen die jeweils verschiedene Aufgaben haben. Das Cam-Modul wertet die von den Kameras kommenden Daten aus und schickt diese als CamFrame an das WorldPredictor-Modul. Dieses Modul versucht vorauszusagen wie die Spielsituation nach dem Durchlauf aller Module aussieht. Diese Information wird dann als WorldFrame an das AI-Modul weitergegeben. Hier entscheidet die künstliche Intelligenz unter Berücksichtigung des WorldFrames und der Schiedsrichterentscheidungen was die Roboter als Gemeinschaft für Spielzüge anwenden sollen. Um diese gemeinsame Bewegung zu ermöglichen, erhält jeder Roboter einen so genannten Skill der vom SkillSystem-Modul in, für den Roboter verständliche, BotCommands umgewandelt wird. Diese BotCommands werden dann vom Botmanager-Modul an die Roboter gesendet die diese Befehle ausführen. Dieser Ablauf ist in Abbildung 3.1 visualisiert. Der gesamte Durchlauf (im folgenden Frame genannt) wird immer wieder wiederholt und dauert beim aktuellen Stand ca. 10ms.

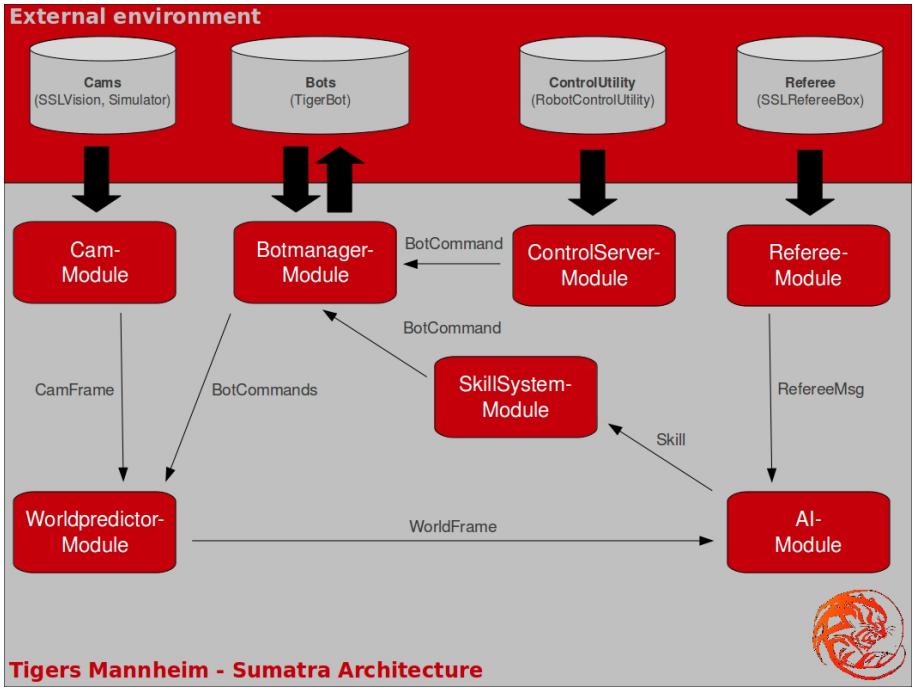

Abbildung 3.1: Modularer Aufbau von Sumatra

Welche Taktiken und Spielzüge im Spiel angewendet werden, wird im AI-Modul entschieden. Dafür gibt es verschiedene Metriken die auswerten, ob das eigene Team oder die Gegner in Ballbesitz sind - also eine Offensiv- oder Defensivtaktik angewendet wird - und welcher Spielzug am erfolgversprechendsten ist. Wurde so ein Spielzug ausgewählt, verteilt das AI-Modul die verschiedenen Rollen (Torhüter, Verteidiger, Passempfänger, usw.) an die einzelnen Bots und weist ihnen Skills (Fahren, Zielen, Schießen, usw.) zu. Neben diesen Entscheidungen erfolgt im AI-Modul auch die Pfadplanung. Hier wird sichergestellt das der Bot seine Zielposition erreicht ohne gegen andere Bots oder, wenn gewünscht, den Ball zu fahren. Da die Studienarbeit bei der Spielzugauswahl behilflich sein soll wird sie im AI-Modul entwickelt.

3.2 Aufteilen des Spielfeldes in Rechtecke

Der Grundgedanke bei der Rasterung des Spielfelds besteht darin, dieses in eine definierte Zahl kleinerer Rechtecke zu unterteilen. In den ersten Versionen wurden bereits alle grundlegenden Funktionen zur Erstellung von Rastern zur Verfügung gestellt, jedoch wurde Performanceaspekte bei der Entwicklung nicht berücksichtigt. So wurde beispielsweise bei jedem ansprechen eines Rechtecks dieses zunächst berechnet (Position und Größe) und anschließend übergeben. Dies wurde im Rahmen der Spielfeldanalyse zusätzlich geändert. Grundsätzlich ist die Verwendung von zwei Rastern vorgesehen. Dies lässt sich darin be-

gründen, dass unterschiedliche Rasterauflösungen von Vorteil sind. Soll ein Roboter positioniert werden, so wird ein gewünschtes Teilfeld ausgewählt und ein zufälliger Punkt innerhalb dieses Bereiches ausgewählt. Das Ganze dient einer groben Positionierung, die somit als Basis nur ein grobes Raster voraussetzt. Andere Verfahren ermöglichen im Anschluss daran eine exakte Ausrichtung des Roboters nach weiteren Kriterien. Zur genauen Analyse ist es jedoch wichtig exakte und detailiertere Informationen zu Berechnung zu nutzen die ein Raster mit einer hohen Auslösung notwendig machen. Zusammenfassend wird also zwischen den folgenden Rastern unterschieden:

Positionierungraster: Kann zur Positionierung der Roboter auf dem Spielfeld verwendet werden. Für den Entwickler von Spielzügen wird so die Positionierung erleichtert, indem dieser nur noch ein explizites Feld angeben muss in das sich der Roboter bewegen soll.

Analyseraster: Wird verwendet, um die aktuelle Spielsituation zu analysieren und kann von den Entwicklern ausgewertet werden. Auch eine Positionierung der Roboter ist über dieses Raster möglich, aber nicht vorgesehen.

Die Klasse *FieldRasterGenerator* stellt die zentrale Anlaufstelle zur Rasterung des Spielfelds dar. Sie wurde mit dem Singleton-Pattern realisiert um zu gewährleisten, dass es nur eine zentrale Instanz zur Erstellung eines Rasters vorhanden ist, schließlich soll pro Spiel auch nur ein Raster pro Rasterart existieren. Die Konfiguration wird mittels einer xml-Datei spezifiziert, die beim Starten von Sumatra eingelesen wird. Mittels der folgenden drei Parametern kann das Raster konfiguriert werden:

1. Anzahl der Spalten
2. Anzahl der Zeilen
3. Analysefaktor - Hierbei handelt es sich um einen Faktor, der ein Maß für die Verfeinerung des Analyserasters im Vergleich zum Positionierungsraster ist.

Für die Anzahl der Spalten und Zeilen sind nur Zweierpotenzen gültig, um eine symmetrische Aufteilung des Spielfelds in Rasterbereiche zu erhalten.
Aus Performancegründen wird bei der Instanziierung der Klasse pro Teilrechteck des Rasters ein eigenes Objekt von Typ Rechteck (*Rectangle*) angelegt, das den jeweiligen Bereich auf dem Spielfeld exakt beschreibt. Die Menge aller berechneter Rechtecke wird dann in einer Map hinterlegt, die einen schnellen Elementzugriff, ohne weitere Berechnungen im späteren Verlauf, ermöglicht. Es existiert dabei pro Rasterart ein eigener Map-Buffer.
Um eine bessere Unterscheidung zwischen den Rechtecken zu ermöglichen wurde eine Spezialisierung der Klasse *Rectangle* vorgenommen, die eine eindeutige Kennzeichnung jedes Teilrechtecks des Rasters ermöglicht. Eine detailierte Beschreibung des Aufbaus der Rechteckklassen ist in Abschnit 3.2.1 zu finden.
Beginnend in der linken oberen Ecke des 4. Quadranten (Ursprung liegt im Mittelpunkt) wird zeilenweise von links nach rechts der Index für die Rechtecke aufgebaut. Mittels entsprechender Methoden des *FieldRasterGenerators* kann der Entwickler, unter Angabe eines

bestimmten Indexes, auf jedes beliebige Teilrechteck zugreifen und es für weitere Zwecke verwenden. Wird ein Element außerhalb des Gültigkeitsbereiches [0, $\text{Rectangle}_{\text{max}}$ - 1] angesprochen, so wird eine *IllegalArgumentException* geworfen und der Programmfluss somit unterbrochen.
Nach der Initialisierung der Map-Buffer werden jedem dort gespeicherten Rechteck alle Nachbarn berechnet und in einer Map gespeichert. Befindet sich ein Rechteck am Rand und hat somit weniger als acht Nachbarn, werden die nicht vorhandenen Nachbarn auf *NULL* gesetzt. Da sich die Nachbarn eines Rechtecks nicht verändern ist es so später möglich sehr schnell über einen Index die Nachbarrechtecke ansprechen. Dies wird für die Auswertung der Analysedaten von Vorteil sein.

3.2.1 Rechteck-Klassen

Aufgrund der geometrischen Beschaffenheit eines Rasters wurde eine geometrische Klasse *Rectangle* erstellt, die es ermöglicht Bereiche eines Spielfelds exakt zu beschreiben. Ein Rechteck wird zunächst über seine Eckpunkte und Seitenlängen charakterisiert. Diese Eigenschaften sind über das Interface *IRectangle* ansprechbar. Durch diese Art der Implementierung auf Interfaces wird gewährleistet, dass konkrete Umsetzungen leichter ausgetauscht werden können. Dieses Interface wird wiederum von der abstrakten Klasse *ARectangle* verwendet. Sie umfasst spezifische Rechteckmethoden die allgemeingültig für alle Arten von Rechtecken sind. Dazu gehören Funktionalitäten zur Flächenberechnung oder Methoden zur Bestimmung von Hashes, Erstellung von Zeichenketten oder Vergleichs-Operationen. Es wird außerdem zwischen zwei Arten von Rechteckklassen differenziert, die veränderliche oder unveränderliche Zugriffsmöglichkeiten auf die privaten Member ermöglichen. Diese Beschränkungen werden in den beiden Klassen *Rectangle* und *Rectanglef* (f = final) realisiert, wodurch versucht wird die fehlerhafte Verwendung dieser Klassen zu minimieren.
Zur Bewertung einzelner Bereiche des Raster wurde eine weitere Klasse namens *AIRectangle* definiert. Sie erbt von *Rectanglef* und erweitert diese um einen eindeutigen Identifier und ein Attribut, das zu Speicherung von Informationen verwendet werden kann. In dieser Arbeit sollen in diesem Member die Güte des Rechtecks hinterlegt werden. Genaueres dazu wird in Kapitel 3.3 angesprochen.
Das UML-Diagramm zur beschriebenen Rechteckstruktur ist in Abbildung 3.2 dargestellt.

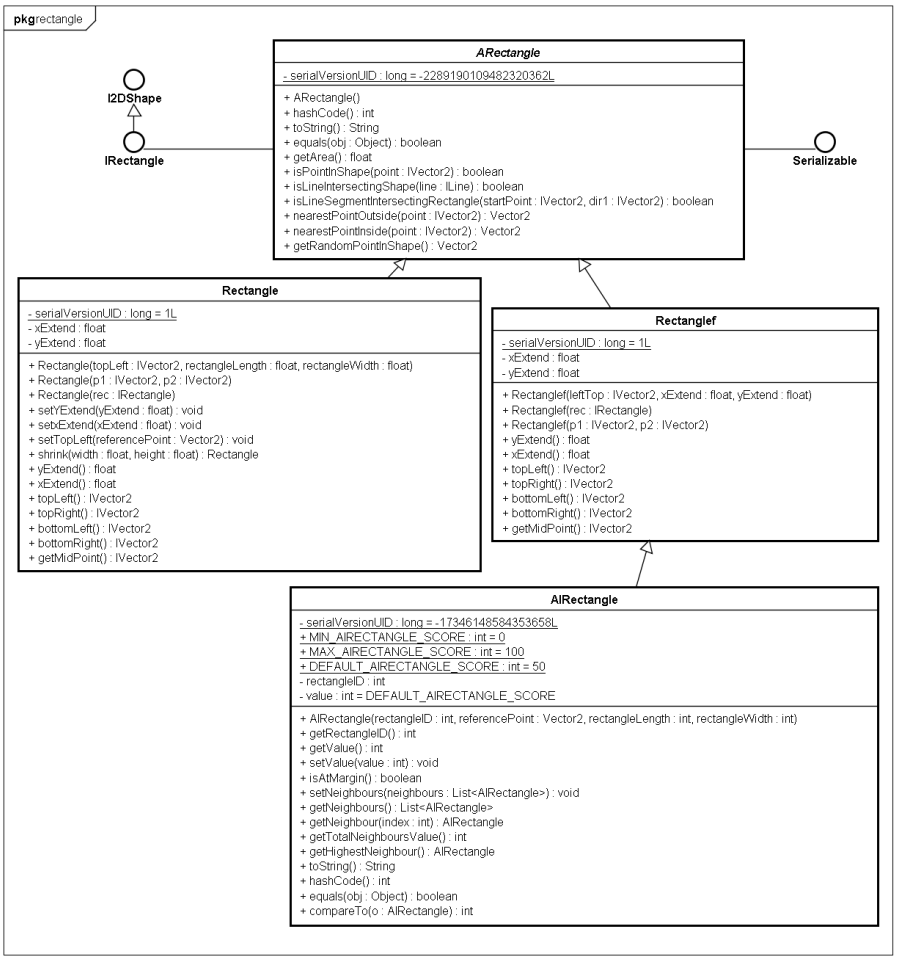

Abbildung 3.2: UML-Diagramm der Rechteckstruktur

3.2.2 Performanceverbesserung bei der Arbeit mit Rechteckklassen

Die Klasse *AIRectangleVector* stellt einen Speichervektor dar, der *AIRectangle*s enthält. Bei der Konstruktion des Objekts wird der Vektor mit Kopien aller Analyserechtecken, aus dem *FieldRasterGenerator*, befüllt. Auf diesem Vektor können dann verschiedene Analyseverfahren angewendet werden, wobei entstehende Ergebnisse in dem Member *value* des *AIRectangle* hinterlegt werden können. Sollen Berechnungen durchgeführt werden die unabhängig und in keiner Relation zueinander stehen, so würde dies, zur besseren Transparenz, auf unterschiedlichen Vektoren umgesetzt.
Um Fehlverhalten bei der Nutzung des Vektors ausschließen zu können wurde dieser so konstruiert, dass er unveränderlich ist. Das bedeutet das Rechtecke nicht verändert, gelöscht oder hinzugefügt werden können. Die Entwickler können nur auf die Rechtecke zugreifen und den Wert des Members *value* verändern.

3.3 Definition der Güte eines Rechtecks

Die Güte eines Rechtecks wird durch seine Position zu den eigenen (Tigerbots) und den gegnerischen Robotern (Foebots) bestimmt. Dabei markiert ein hoher Wert einen von den Gegnern gut abgedeckten Bereich und ein niedriger Wert einen von den eigenen Bots besetzten Raum. Ein Rechteck besitzt dann einen hohen Gütewert (schlecht für das eigene Team), wenn ein gegnerischer Bot näher an diesem Rechteck steht als ein eigener. So könnte der Foebot schneller im betrachteten Rechteck sein als ein Tigebot und somit einen geplanten Pass in den freien Raum abfangen. Ein niedriger Gütewert (gut für das eigene Team) hingegen wird dann erreicht, wenn ein Tigerbot näher am Rechteck steht als ein Foebot. Je größer der Unterschied der beiden Entfernung ist, desto höher bzw. niedriger wird auch der Gütewert. Numerisch beginnt der Gütewert bei 0 (sehr gut) und geht bist 100 (sehr schlecht). Der Standardwert für alle Rechtecke ist bei 50 festgelegt.

3.4 Entwurf zur Bewertung einzelner Rechtecke

Um die Güte eines Rechtecks zu bestimmen, können mehrere Berechnungsschritte und -ansätze möglich sein. Das Paket *scoreAlgorithms* enthält alle Berechnungsalgorithmen die auf ein Rechteck angewendet werden können. Zur Behandlung verschiedener Algorithmen wurde auf das Strategie-Entwurfsmuster zurückgegriffen. Für jede Berechnungsart wird eine Klasse angelegt die von *AScoringAlgorithm* erbt. Diese enthält die Methode *scoreSituation* die den eigentlichen Berechnungsalgorithmus umsetzt und die der akutelle Frame sowie das zu nutzende Rechteck übergeben werden. Als Ergebnis wird die Güteinformation des Rechtecks, als Ganzzahl, zurückgegeben und von der aufrufenden Klasse weiterverarbeitet.

3.4.1 Ansätze für Bewertungsalgorithmen

Der erste umgesetzte Ansatz war es, über die Position der Bots die Güte zu bestimmen. Dazu wurde über alle Bots iteriert und überprüft ob sich die Position im aktuellen Rechteck befindet. Für jeden so gefundenen eigenen Bot wurde die Güte verringert und für jeden gegnerischen Bot erhöht.
Dies sollte durch einen weiteren Bewertungsalgorithmus erweitert werden, der die aktuelle Bewegungsrichtung und -geschwindigkeit berücksichtigen sollte. Da die Bots diese allerdings im Bruchteil einer Sekunde ändern können, wären die so erfassten Daten nur wenig aussagekräftig. Deshalb wurde ein anderer Algorithmus gesucht.
Als geeigneter Algorithmus wurde eine Berechnung mithilfe der Normalverteilung gefunden. Dabei wurde wieder über alle Bots iteriert und der Abstand vom Rechteckmittelpunkt zum Bot (in mm) bestimmt. Dieser wurde dann durch die längere Seite des Rechtecks geteilt um so einen skalierten Wert zu erhalten. Dieser skalierte Wert wurde benötigt, da die Normalverteilung für große Werte gegen Null geht. War der skalierte Wert größer als 3,5 (empirisch ermittelt), galt der Bot als nicht wichtig für dieses Feld da er zu weit entfernt ist. War er jedoch kleiner als 3,5, wurde dieser in eine Normalverteilung mit N(0,2) (diese besitzt bei 3,5 einen Wert von ca. 0,04) eingesetzt und der so erhaltene Wert mit einem vom Benutzer bestimmbaren Faktor erweitert.
In Pseudocode ergab sich also folgender Algorithmus:

```
1  Pseudocode pro gegnerischem Bot:
2  laenge = (bot.position - rechteck.mittelpunkt).laenge;
3  skalierteLaenge = laenge / rechteck.langeSeite;
4  Wenn skalierteLaenge < 3,5
5  guete += normalverteilung(skalierteLaenge) * FAKTOR;
6
7  Pseudocode pro eigenem Bot:
8  laenge = (bot.position - rechteck.mittelpunkt).laenge;
9  skalierteLaeange = laenge / rechteck.langeSeite;
10 Wenn skalierteLaenge < 3,5
11 guete -= normalverteilung(skalierteLaenge) * FAKTOR;
```

Listing 3.1: Pseudocode für die Score-Berechnung

Um eine noch bessere Gütebewertung zu erhalten, wurde der Algorithmus um eine Fließkommavariable latex *tigerFactor* erweitert. Diese Variable diente dazu, Bereiche, die von den eigenen Bots besetzt sind und keine Gegner in der Nähe haben, hervorzuheben.
Der *tigerFactor* sowie die folgenden Werte wurden dabei empirisch bestimmt.
Bei der Initialisierung des *tigerFactor* wurde dieser auf 5 gesetzt und für jeden gegnerischen Bot der einen geringen skalierten Abstand als 3,5 hat mit 0,7 multipliziert (je mehr gegnerische Bots in der Nähe sind desto schlechter). Zusätzlich wurde der tigerFactor bei einem skalierten Abstand von weniger als 1,5 auf 1 gesetzt (alle Felder um den gegnerischen Bot herum sind schlecht). Abschließend wurde dieser Faktor dann für jeden eigenen Bot auf die Gütebewertung multipliziert. Dadurch ergaben sich bei der Visualisierung der

Güte in der Software klar gute und schlechte Bereiche (siehe Abbildung 3.3).

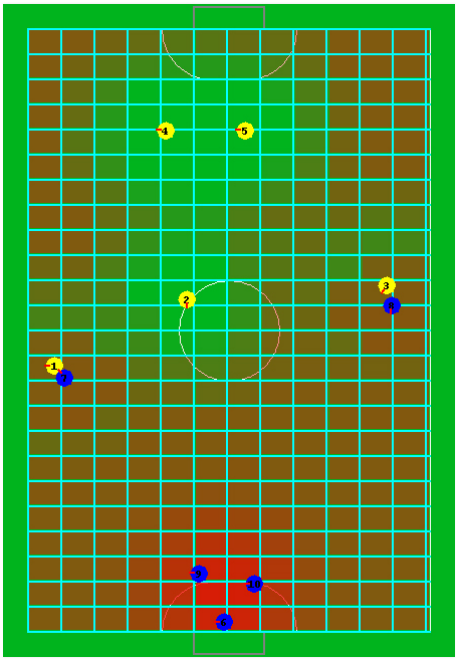

Abbildung 3.3: Ergebnis der ersten Rasteranalyse

Die so gewonnenen Daten waren aber noch nicht zufriedenstellend. In Abbildung 3.3 ist zu sehen das z.B. freie Wege nicht optimal erkannt wurden und manche Bereiche eine bessere Güte (im Vergleich zu der Standardgüte) haben die ein eigener Bot nur durch umfahren eines gegnerischen Bots erreichen kann.
Um diese Daten zu verbessern wurde ein weiterer Ansatz implementiert. Dafür wurde im Vorfeld eine klare Gütedefinition aufgestellt und mithilfe dieser ein Algorithmus umgesetzt der ausschließlich die Abstände von Gegner und eigenen Bots berücksichtigt. Dabei wird über alle Bots iteriert und so pro Team der Bot bestimmt, der den kürzesten skalierten Abstand zum betrachteten Rechteck hat. Von diesen beiden Abständen wird dann die Differenz gebildet und mit 10 (empirisch bestimmt) multipliziert. Der so erhaltene Wert wird auf den Standardwert des Rechtecks addiert und die Güte ermittelt. So ergab sich ein deutlich konsistenteres Bild (siehe Abbildung 3.4).

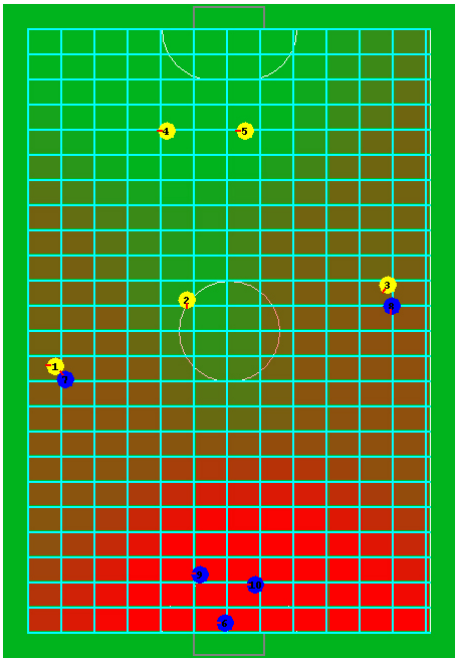

Abbildung 3.4: Ergebnis der zweiten Rasteranalyse

3.4.2 Echtzeitproblematik

Da beim RoboCup eine Echtzeitanalyse der Daten sehr wichtig ist, um schnell auf die sich veränderten Situationen reagieren zu können, darf die Spielfeldanalyse keine große Rechenleistung beanspruchen. Daher sollte in den ersten Ansätzen nicht über alle Rechtecke iteriert werden sondern nur über eine vom Benutzer festgelegte Anzahl. Um trotzdem eine möglichst ausgeglichene Information über das gesamte Spielfeld zu erreichen, mussten diese Rechtecke zufallsmäßig über das Spielfeld verteilt werden. Daher wurde in der Klasse *FieldAnalyser* eine globale Liste von Ganzzahlen angelegt und zufallsmäßig mit allen Identifiern der Recktecke befüllt. Diese Liste wurde zum Programmstart einmal angelegt und danach nicht wieder verändert. Sie diente dazu, dass die Rechtecke des Feldes immer in der gleichen, einmal zufällig bestimmten Reihenfolge abgearbeitet wurden aber nicht der Reihe nach (z.B. 12, 4, 42, 24, ... und nicht 1, 2, 3, 4, ...). Würden die Rechtecke der Reihe nach abgearbeitet werden, so könnte es zu dem Fall kommen das zwar aktuelle Informationen über die eine Spielfeldhälfte da sind aber veraltete über die andere. Bei einer zufälligen Reihenfolge ist dieser Fall statistisch ausgeschlossen. Die Liste wurde nur einmal angelegt damit auch ausgeschlossen ist das einzelne oder mehrere Rechtecke auch nach mehreren Frames nicht durch den Zufallsgenerator ausgewählt werden. Dies hätte zu Folge gehabt das diese Rechtecke veraltete Informationen enthielten und somit keine korrekte Aussage über die aktuelle Spielfeldsituation möglich wäre. Mit der oben erwähnten Liste konnten die Informationen nur ein bestimmtes Alter erreichen und die älteste wurde überschrieben.

Nach der Implementierung des letztendlichen *ScoringAlgorithm* wurde aber festgestellt das selbst die Aktualisierung des gesamten Spielfelds keine spürbaren Auswirkungen auf die Rechenzeit der restlichen Software hatte. Da eine konsistente Information über das gesamte Spielfeld zum aktuellen Zeitpunkt besser ist, wurde der vorher beschriebene Algorithmus daraufhin gelöscht.

3.5 Ablauf der Spielfeldanalyse

Zur Bewertung der Güte aller Rechtecke wird der *FieldAnalyser* aufgerufen. Dieser erbt von der abstrakten Klasse *ACalculator*, die die *calculate* Methode enthält die zu jedem neuen Frame aufgerufen wird.
Des Weiteren enthält die Klasse *FiledAnalyser* eine Instanz des *AIRectangleVector* und eine Liste von allen benötigten *ScoringAlgorithms*. Diese beiden Attribute werden im Konstruktor initialisiert. In der vorher angesprochenen *calculate* Methode wird dann über alle vorhandenen Rechteck iteriert und alle *ScoringAlgorithms* auf das jeweilige Rechteck angewandt. Ein vorher lokal angelegtes Attribut speichert die so erhaltene Güteinformation zwischen und legt diese nach Berechnung aller *ScoringAlgorithms* in dem entsprechenden Rechteck ab (im Member *value*). Hier sei zu erwähnen, dass alle Algorithmen gleich gewichtet sind und deren einzelnen berechneten Werte nur akkumuliert werden. Es kann aber durchaus später sinnvoll sein dies zu ändern, wodurch auch Anpassungen an der Struktur notwendig würden.
Als UML-Diagramm dargestellt ergibt sich folgende Abbildung:

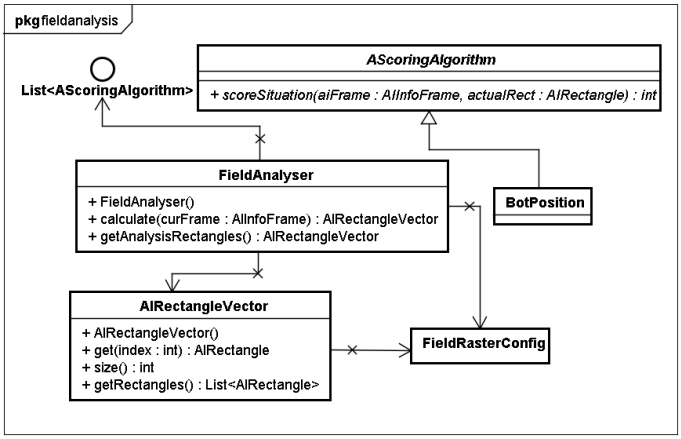

Abbildung 3.5: UML-Diagramm der Feldanalysestruktur

Somit ist das gesamte Spielfeld ausgewertet und für jedes Rechtecke eine aktuelle Güteinformation vorhanden.
Über die Methode *getAnalysisRectangles* wird der *AIRectangleVector* zurück gegeben und kann von anderen Klassen weiter verwendet werden. Hauptsächlich dient sie dazu, den *AIRectangleVector* auszulesen und ihn im aktuellen Frame abzulegen.

Kapitel 4

Entwurf und Implementierung der Visualisierung

In dem folgenden Abschnitt soll die Visualisierung der gewonnen Analysedaten thematisiert werden. Damit jedoch die entsprechenden Funktionen implementiert werden konnte sollte zunächst die Grundstruktur zur grafischen Darstellung von Informationen, bedingt durch die unstrukturierte Entwicklung der grafischen Oberfläche, überarbeitet und vereinfacht werden. Zum besseren Verständis wird zunächst die Grundstruktur mit ihren Problemen dargelegt und darauf aufbauend die Veränderungen erläutert.

4.1 Grundstruktur in der Zentralsoftware

Innerhalb der eigenen Team-Software gibt es auch eine Visualisierung des aktuellen Zustands des Gesamtsystems. Dies beinhaltet zum einen die Darstellung der aktuell geplanten Spielstrategie und berechnete Trajektorien zur Wegplanung, aber auch Statusinformationen über die jeweiligen Roboter, wie den Ladezustand der Akkumulatoren, der Schussvorichtung oder Informationen über die Funkverbindung. Innerhalb von Sumatra ist die grafische Oberfläche mittels dem Modell View Presenter (MVP) umgesetzt.

4.1.1 Modell View Presenter (MVP)

Bei dem Model View Presenter Konzept handelt es sich um ein spezielles Entwurfsmuster aus der Softwareentwicklung mit dem Ziel zusammengehörige Komponenten strikt voneinander zu trennen [3]. Dies hat den Vorteil, dass die einzelnen Teile unabhängig voneinander getestet und verifiziert werden können . Außerdem wird so auch die Austauschbarkeit gesteigert. Im folgenden sollen nun die einzelnen Bestandteile dieses Entwurfsmuster näher

gebracht werden.*Damit MVP seine eigentlichen Vorteile gegenüber MVC entfalten kann, werden für Modell und Ansicht jeweils Schnittstellen (engl. Interfaces) verwendet. Die Schnittstellen definieren den genauen Aufbau beider Schichten und der Presentator verknüpft lediglich die Schnittstellen miteinander* [4].

Model

Bei dem Model handelt es sich um die Logik mit den dazugehörigen Daten die illustrierten werden sollen. Dabei ist vor allem wichtig, dass das Modell alle nötigen Methoden und Funktionalitäten bereit stellt die es den anderen Komponenten ermöglicht alle notwendigen Informationen auszulesen. Ein allgemeines Beispiel dafür wäre die Geschäftslogik bei einer Unternehmenssoftware. Im vorliegenden Fall handelt es sich aber um die eigentlichen spielentscheidende Logik mit ihren Steuerungsfunktionen und weiteren zusätzlichen Komponenten.

View

Bei der View handelt es sich um die eigentliche grafische Oberfläche. Sie enthält keine Logik und stellt nur Funktionen zur Darstellung von Informationen/Daten aus dem Modell bereit. Das Entwurfsmuster sieht im Allgemeinen nicht vor, dass die View Kenntnis und Zugriff auf den Presenter oder das Modell hat. In der Praxis ist es manchmal nur mit einem sehr großen Aufwand vollständig realisierbar weshalb dann einen Kompromiss eingegangen wird und ggf. aus dem Design-Muster ausbricht. Diese Abkehr vom eigentlichen Muster sollte aber wohl überlegt vorgenommen werden und sollte nicht dem Standard entsprechen.

Presenter

Der Presenter ist das Verbindungsglied zwischen dem Modell und der grafischen Oberfläche. Er ist dafür verantwortlich, dass die View mit den notwendigen Daten befüllt wird. Häufig übernimmt diese Komponente dann auch Formatierungsaufgaben abhängig von der verwendeten grafischen Oberfläche.

4.1.2 Umsetzung von MVP in Sumatra

Wie bereits in Abschnitt 3.1 erwähnt ist Sumatra mittels eigenen Modulen umgesetzt worden. Jedes Modul stellt dabei das eigentliche Modell dar und besitzt zudem seinen eigenen Presenter und eine grafische Komponente. Der Datenfluss bzw. die Kommunikation von dem Modell zum Presenter wurde dabei häufig mittels dem Oberserver-Entwurfsmuster realisiert [2]. So beispielsweise auch bei dem Modul für die künstlichliche Intelligenz (AI-Modul). Dabei wird der AIInfoframe, welcher alle aktuellen Spielinformationen enthält, an den Presenter übergeben. Dieser kann dann mit der Aufbereitung der Daten beginnen und die entsprechenden Felder der View befüllen. Die Umsetzung des MVP-Konzeptes ist in der folgenden UML-Grafik, siehe Abb. 4.1, illustriert. Die Klasse *VisualizerPresenter*

entspricht dem Presenter das AI-Moduls, der entsprechende Interfaces implementiert. Die Klasse *AAgent* entspricht dem Modell und die View-Komponente wird mit dem *VisualizerPanel* realisiert.

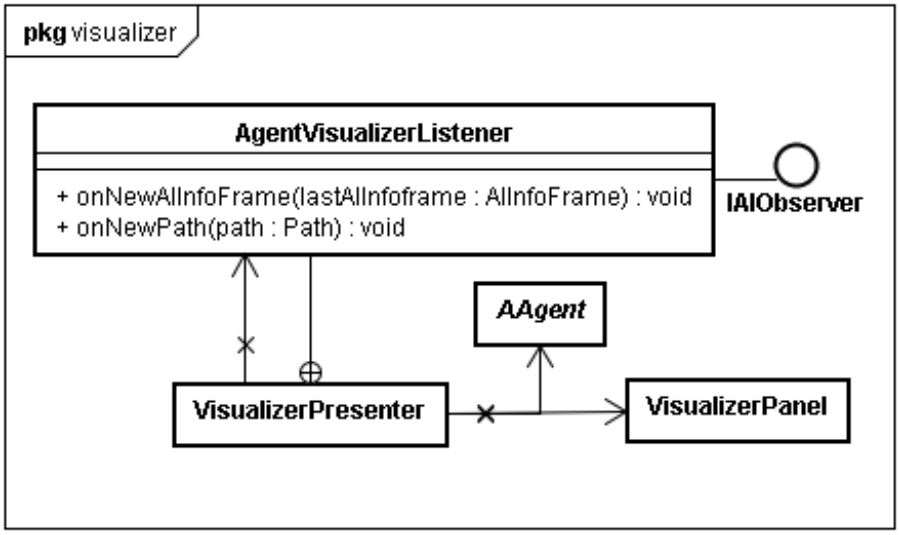

Abbildung 4.1: MVP Konzept in Sumatra

Bei der dem Erstellen des Presenters registriert sich dieser als Observer beim *AAgent*. Mittels dem Interface *IAIObserver* bzw. der inneren Klasse *AgentVisualizerListener* kann dann das AI-Modul dem Presenter neue Daten zur Darstellung mitteilen.

4.2 Die grafische Komponente im Detail

Da im weiteren Verlauf die Veränderungen in der grafischen Komponente thematisiert werden sollen nun zunächst die Probleme mit der gewachsenen GUI angesprochen werden. Die hier behandelte View beschäftigt sich vorallem mit der grafischen Darstellung der Roboter-Positionen auf dem Spielfeld, der Illustration von taktischen Informationen sowie Steuerungsmöglichkeiten.

4.2.1 Bisherige Struktur

Bedingt durch die starke Weiterentwicklung der Software Sumatra wurden viele grafische Funktionalitäten zur Darstellung nach und nach in die Klasse *FieldPanel* implementiert, wodurch diese stark anwuchs und sehr an Übersichtlichkeit verloren hat. Dabei wurden nicht darauf geachtet, dass es einen Standardablauf zum Darstellen von Strukturelement, wie Kreisen, Rechtecken etc., gibt. Bisher können folgende Elemente auf dem Spielfeld gezeichnet werden:

- Position der Roboter
- Position des Balls
- Geschwindigkeits- und Beschleunigungsvektoren
- Feldraster
- Debug-Punkte (z.B. Ziel eines Passes)

4.2.2 Notwendige Änderungen

Um die Übersichtlichkeit zukünftig zu fördern und die Anforderungen zur Darstellung von der Rasteranalyse zu erfüllen soll nun ein Layer-System eingeführt werden indem in verschiedenen Ebenen überhalb des Spielfeld Strukturelemente gezeichnet werden können. Zur Umsetzung des Layers-Systems soll mit dem in Java 1.7 neu eingeführten JLayern gearbeitet werden.

4.3 Java AWT Komponenten dekorieren

Die *JLayer* Klasse kann in Kombination mit dem *LayerUI* verwendet werden um AWT Komponenten zu dekorieren. Innerhalb des *JLayers* werden dann die paint()-Methoden der *LayerUI* sowie die der eigentlichen ursprünglichen Komponente verschachtelt, sodass diese dekoriert wird. Mittels der *LayerUI* Klasse wird dabei festgelegt, wie die Komponente verändert werden soll. D.h. sie enthält den eigentlichen Code zur Manipulation.

4.4 Implementierung

Nun werden die wichtigsten Klassen zur Umsetzung des Layer-Systems dargestellt und ihre Funktionsweise näher beschrieben.

4.4.1 Design einer Zeichenebene

Die abstrakte Klasse *AFieldLayer*, siehe Abbildung 4.2, erbt von *LayerUI<JComponent >* und stellt die Basis-Klasse dar, die eine Zeichenebene repräsentiert. über die abstrakte Methode paintLayer(Graphics g, JComponent j) werden die gewünschten Elemente auf der Zeichenebene, mittels dem *Graphics* Parameter untergebracht. Der *JComponent* Parameter liefert zusätzliche, nicht zwingend notwendige, Informationen über die grafische Komponente auf der die zu zeichnenden Elemente dargestellt werden sollen. Unter Verwendung entsprechender Settern kann zur Laufzeit eingestellt werden, ob der Layer gezeichnet und ob ggf. Debug-Informationen visualisiert werden sollen. Für die Darstellung aller relevanter Informationen wurden insgesamt die folgenden Layer definiert:

1. Border-Layer
2. Defense-Goal-Points
3. Debug Points
4. Ball-Bot-Layer
5. Positioning Raster
6. Analysing Raster

Damit die Raster nach dem gleichen Algorithmus dargestellt werden wurde eine zusätzliche, von *AFieldLayer* erbende, abstrakte Klasse *ARasterLayer* erstellt, die eine entsprechende Methode zum Zeichnen von Rastern bereitstellt. Die beidenen davon erbenden Klassen *AnalysingRasterLayer* und *PositioningRaster* nutzen diese Methode und besitzen noch eigene Debug-Visualisierungen. Mittels von Enums kann zwischen den verschiedenen Layern unterschieden werden.

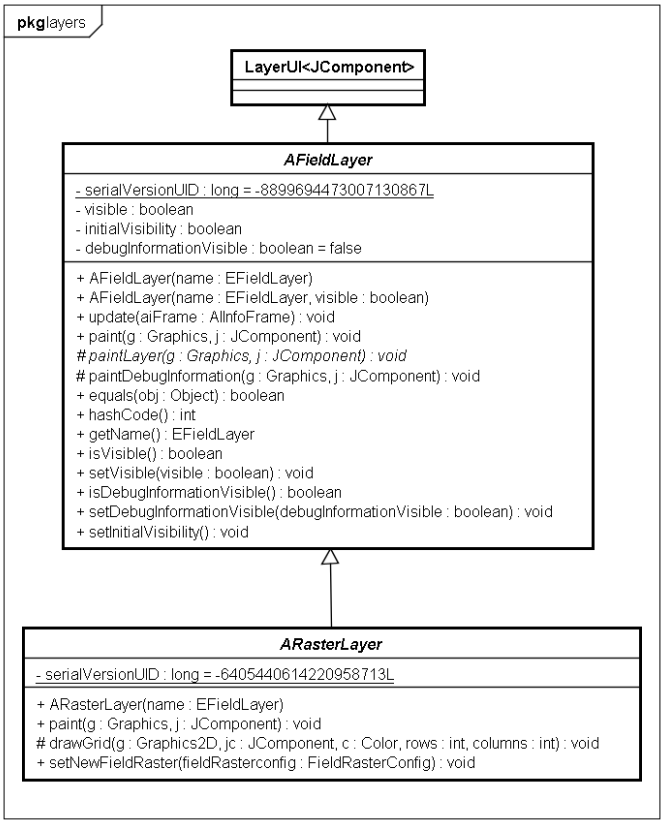

Abbildung 4.2: UML-Diagramm der Layer-Struktur

4.4.2 Zusammenfassen von Zeichenebenen

Die Klasse *MultiFieldLayerUI* dient dazu die verschiedenen *AFieldLayer* zusammenzufassen, da der *JLayer* nur in Kombination mit einer *JComponent* und einer *LayerUI* arbeitet. Sie erbt auch von der Klasse *LayerUI<JComponent >* und stellt eine Liste bereit, die alle zu illustrierenden *AFieldLayer* beinhaltet. Iterativ werden die Elemente der Liste auf die grafische Komponente gezeichnet. über spezielle Methoden können mittels der eindeutigen Enums gezielt Layer innerhalb dieser Liste über ihre Methoden verändert werden. Die Eindeutigkeit in der Liste wird dadurch gewährleistet, dass jedes Layer nur einmal hinzugefügt werden kann.

4.4.3 Gesamtdarstellung

Das *FieldPanel* ist vom Typ *JPanel* und stellt die grüne Grundfläche für die Layer zur Verfügung. Diese grüne Ebene wird mit Hilfe eines geladenen Bildes umgesetzt und entsprechend mit dem *MultiFieldLayerUI* und dem *JLayer* dekoriert. Im Konstruktor werden die einzelnen *AFieldLayer* in die Liste des *MultiFieldLayerUI* eingefügt. Außerdem beinhaltet diese Klasse globale Methoden zur Transformation von Koordinaten und Längen zwischen der realen Welt und der GUI. Die folgende Abbildung 4.3 zeigt die finale Struktur der neuen grafischen Oberfläche.

Abbildung 4.3: *FieldPanel* mit Steuerungsmöglichkeiten für unterschiedliche Layer

Kapitel 5

Zusammenfassung und Ausblick

Das Ziel der Studienarbeit, eine automatischen Spielfeldanalyse im Rahmen des Teams Tigers Mannheim zu implementieren, wurde erreicht. Dafür wurden vorhandene Strukturen übernommen, gegenbenenfalls angepasst und neue erstellt. Die Rasterung des Spielfelds wurde geringfügig im Hinblick auf Performance angepasst. Es wurde darauf geachtet, dass eventuelle Erweiterungen bei der Implementierung ohne großen Aufwand eingebaut werden können und dem Entwickler eine intuitive Benutzung der Daten geboten wird. Mehrere Bewertungsalgorithmen wurden erstellt bis die beste Version gefunden wurde. Diese zeichnet sich durch eine sehr klare und einfache Gütedefinition aus und ist leicht verständlich. Zusätzlich wurde die grafische Darstellung überarbeitet und angepasst, um eine Überprüfung der Berechneten Werte durchführen zu können.
Noch sind keine Implementierungen mit der so gewonnenen Datengrundlage vorgenommen worden aber ein Einsatz für verschiedene Aufgaben ist vorstellbar. Dabei steht vorallem die Spielzug- oder Taktikauswahl sowie die Spielzugausführung (taktische Positionierung der Roboter) im Vordergrund.

Literaturverzeichnis

[1] ELEKTRONIKPRAXIS: *Bremer Studenten wurden Europameister bei den RoboCup German Open 2009*. http://www.elektronikpraxis.vogel.de/index.cfm?pid=4800&pk=185583&sector=2, Abruf: 09. Januar 2012

[2] Fowler, Martin: *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002. – ISBN 978–0–321–12742–6

[3] Potel, Mike: *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java*. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, Abruf: 09. Januar 2012

[4] Wikipedia: *Model View Presenter — Wikipedia, Die freie Enzyklopdie"*. http://de.wikipedia.org/w/index.php?title=Model_View_Presenter&oldid=94260082, Abruf: 09. Januar 2012