*Todo list

- Vic - Reicht das so!? . . . 3
- Vic - hier auch den Server erwähnen? . . . 3
- Bogi - Vervollständigen . . . 4
- Da wird plötzlich vom Server angefangen, ohne jemals erwähnt zu haben, dass wir einen haben! Den Absatz vllt in eine Server - section packen? . . . 5
- Beschreibung der Schnittstellen, z.B. Dateien und ihre Struktur, Benutzte Module/Bausteine und ihre Einbindung, Installationsvorschriften . . . 6
- Vic - Bemerkung zur Grafik: es gibt doch garkeinen OPCODE == LOST oder WON? Meiner Meinung nach wird der Zustandsübergang wird durch 'letztes Schiff versenkt' ausgelöst und dann hängt es davon ab, ob es das eigene ist oder nicht..? gleiches gilt für die folgende abbildung . . . 8
- Vic - Ich befürchte, dass das Klassendiagramm kaum lesbar ist, macht es dann sinn das reinzunehmen? oder siehst du ne möglichkeit das größer zu machen? 12
- Bogi - Bild zum Modulzusammenspiel zusammenklickern ;) . . . 17
- Spellcheck from here, please. . . . 19
- Bogi - Schreib bitte erstmal verständliche Sätze ;) . . . 19
- Sequenzdiagramm/Ablaufdiagramm, evtl. . . . 20
- Bogi - TODO
  ACK! Begonnen, Muse verloren. Später gehts weiter . . . 20
- Bogi - ich glaub das 'normalerweise' aus dem vorherigen satz gilt jetzt nicht mehr oder? entweder im Standardcode ändern oder hier im text - was denst du? . . . 22
- spellcheck?? . . . 23
- Entwicklertest, Testbarkeit der Software (Treiber,...), Positiv- und Negativtests, Stresstest (nicht funktional) . . . 24
- Ausblick - was fehlt noch? bekannte Fehler, spätere Verbesserungen, Fazit der Teilnehmer . . . 24
- Ausblick: Überprüfung adden z.B. wenn 2er Versenkt, können alle 2er Lücken als Wasser markiert werden, etc. . . . 24
- Ausblick: Schachbrettangriff . . . 24

Wahlpflichtfach KI

Projekt: Schiffe Versenken

Autoren: Victor Apostel, Stefan Bogdanski, Sibille Ritter
Studiengang: (Internationaler) Studiengang Technische Informatik B.Sc.

Inhaltsverzeichnis

1 Einleitung **3**
1.1 Anforderungen 3

2 Spielregeln **4**

3 Kommunikationsprotokoll **5**

4 Softwarearchitektur **6**
4.1 Spielserver 10
4.2 Client - Java 11
4.2.1 View 12
4.2.2 Controller / Model 15
4.3 Client - Prolog 17
4.3.1 Hauptmodul 18
4.3.2 Initialisierungsmodul 19
4.3.3 Modul zur Plazierung von Schiffen 20
4.3.4 Verteidigungsmodul 20
4.3.5 Angriffsmodul 21
4.3.6 Strategiemodul 22
4.3.7 Ausgabemodul 22
4.3.8 Ausgabeverhalten 23

5 Evaluation **24**
5.1 Testkonzept 24
5.2 Testfälle 24

6 Benutzungshinweise für Endbenutzer **24**

7 Ausblick **24**

Abbildungsverzeichnis

1 Darstellung der möglichen Kommunikationsteilnehmer 7
2 Zustandsübergänge des Clients 8
3 Interne Zustandsübergänge während des laufenden Spiels 9
4 Sequenzdiagramm der Kommunikation 10
5 Klassendiagramm des Servers 11
6 Klassendiagramm des Java Clients 12
7 Gegnerisches Spielfeld mit einem zerstörten (dunkelgrau) und einem beschädigten Schiff (rot) 13
8 Eigenes Spielfeld mit versenkten Schiffen (dunkelgrau), einem beschädigten Schiff (rot) und misglückten Angriffen des Gegners (dunkelblau) 14
9 Gesamte Benutzeroberfläche 15

Tabellenverzeichnis

1 Kommunikationsprotokoll 6
2 Ergebniskodierung 6
3 Prädikate für globale Spielvariablen 18

1 Einleitung

Vic - Reicht das so!?

Im Wahlpflichtfach *Künstliche Intelligenz* wurde während der Vorlesungen und den vorlesungsbegleitenden Beispielen ein solides Verständnis und Wissen über die Implementierung einer künstlichen Intelligenz in Prolog erarbeitet. Zum Ende der Veranstaltung soll das angeeignete Wissen nun in Form eines Projektes angewendet und vertieft werden.

Die vorliegende Dokumentation beschreibt die Implementierung des Spiels Schiffe-Versenken (Battleship) in Java und Prolog.

Ziel des Projektes ist es *Schiffe-Versenken* gegen "den Computer" spielen zu können. Hierfür muss das Spiel mit allen Spielregeln in Prolog abgebildet werden. Ebenso muss eine Spielstrategie für den Computerspieler konzeptioniert und implementiert werden, welche dafür sorgt das die künstliche Intelligenz Siegesorientiert spielt. Eine Benutzungsfreundliche Oberfläche für menlische Spieler wird durch ein Java GUI realisiert.

Vic - hier auch den Server erwähnen?

1.1 Anforderungen

Die Software unterliegt den folgenden Anforderungen:

- Es gelten die allgemeinen Regeln des Spiels *Schiffe-Versenken*, Besonderheiten werdnen in Abschnitt 2 erläutert.
- Der Prolog Client kommuniziert über Sockets mit dem zu entwickelnden Java Server.
- Benötigte Module für den Prolog Client zum Regeln:
  - des generellen Spielablaufs,
  - der Platzierung von Schiffen auf dem Spielfeld,
  - der Antworten auf gegnerische Angriffe,
  - der eigenen Angriffe nach einer
  - Siegstrategie.

- Es soll möglich sein, selbst gegen die künstliche Intelligenz zu spielen.
- Es soll möglich sein, dass zwei Instanzen der künstlichen Intelligenz eine bestimmte Anzahl von Spielen gegen einander austragen.

Neben diesen Anforderungen ist eine weitere, spezielle, Anforderung definiert:

Da eine weitere Gruppe von Studenten dieses Moduls ebenfalls das Spiel *Schiffe-Versenken* als Projektthema wählten, soll es möglich sein, dass die künstlichen Intelligenzen beider Gruppen gegeneinander antreten können. Hierfür wurde das, in Abschnitt 3 dokumentierte Kommunikationsprotokoll unter beiden Gruppen vereinbart.

2 Spielregeln

Da die allgemeinen Regeln des Spiels *Schiffe-Versenken* bekannt sein sollten, werden an dieser Stelle nur die relevanten Eckdaten und besondere Vereinbarungen aufgeführt:
Bogi - Vervollständigen

- Spielfelder
  - 10 x 10 Felder gross
  - Eigenes Feld entweder: Wasser, Schiff, Getroffen
  - Gegnerisches Feld entweder: Unbekannt, Treffer, Versenkt
- Schiffe
  - 1 x 5er, 1 x 4er, 2 x 3er, 1 x 2 er
  - Schiffe dürfen sich nicht berühren
  - Eine *schräge* Platzierung der Schiffe **zu einander** ist gültig. Anders ausgedrückt dürfen Schiffe sich **nicht** in der 4er Nachbarschaft (horizontal und vertikal benachbarte Felder) zu einander befinden, jedoch in den diagonal benachbarten Feldern.
- Spielablauf
  - Die beiden Spieler schießen abwechselnd bis sämtliche Schiffe eines Spielers versenkt sind.

3 Kommunikationsprotokoll

Die Kommunikation zweier Spieler erfolgt auf Basis von Textnachrichten, den sogenannten Kommandos, die einer fest definierten Codierung genügen. Das Ziel dieses Kapitels ist die Erläuterung des Nachrichtenaufbaus und deren Interpretation.

Jedes Kommando, das es zu interpretieren gilt, ist aus Gründen der Prologkompatibilität mit runden Klammern umgeben und endet mit einem ".". Da zudem stets ganze Zeilen verarbeitet werden sollen, wird der Nachricht das nicht druckbare Zeichen "\n" angehängt. Damit wird signalisiert, dass die gesamte Nachricht übermittelt wurde.

Innerhalb der umschließenden runden Klammern werden zwei Angaben erwartet. Die erste Angabe wird als *Opcode* bezeichnet und dient der eindeutigen Bestimmung des Kommandotyps. Die zweite Angabe des Kommandos ist eine, von eckigen Klammern umgebene Parameterliste und kann null bis drei, durch Kommata getrennte, nummerische Elemente beinhalten. Die Anzahl der erwarteten Parameter richtet sich nach dem übermittelten Opcode.

Die formale Beschreibung des Kommandoaufbaus liegt im Folgenden in Form von regulären Ausdrücken vor.

$$
\begin{aligned}
COMMAND &:= \backslash(OPCODE, LIST\backslash)\backslash.NEWLINE \\
LIST &:= \backslash[[PARAMS]?\backslash] \\
NEWLINE &:= \backslash\backslash r \\
OPCODE &:= [1-5] \\
PARAMS &:= [0-9] + [, [0-9]+]\{0,2\}
\end{aligned}
$$

Der Spielablauf durchläuft mehrere Zustände, die durch das Kommunikationsprotokoll abgedeckt werden müssen. So startet jeder Spieler in einem Initialisierungszustand. In diesem Zustand wird festgelegt, welcher der beiden Spieler den ersten Angriff ausführt.

Da wird plötzlich vom Server angefangen, ohne jemals erwähnt zu haben, dass wir einen haben! Den Absatz vllt in eine Server - section packen?

Stellt der Server eine gültige Anzahl an verbundenen Spielteilnehmern fest, sendet dieser ein Startsignal an alle Teilnehmer und die Clients wechseln ihrerseits vom Initia-

lisierungszustand in den Angriffs-, bzw. Verteidigungszustand. Weitere Nachrichten die es zu übertragen gilt, sind zum Einen der Angriff auf eine Feldkoordinate, sowie zum Anderen das Ergebnis des Angriffs.

Die Abbildung der genannten Aktionen in Opcodes, sowie die erwarteten Parameter können der Tabelle 1 entnommen werden.

Opcode	**Bedeutung**	**Param 1**	**Param 2**	**Param 3**
1	Angriff	X	Y	-
2	Ergebnis des Angriffs	X	Y	Ergebnis
3	Spieler startet im Verteidigungszustand	-	-	-
4	Spieler startet im Angriffszustand	-	-	-
5	Startsignal	-	-	-

Tabelle 1: Kommunikationsprotokoll

Das Ergebnis eines Angriffs wird ebenfalls kodiert übertragen und kann den Zustand des Clients beeinflussen. Im Regelfall sendet bzw. empfängt dieser die Ereignisse “Wasser“, “Schiff wurde getroffen“, oder “Schiff wurde versenkt“. Des Weiteren kann als Reaktion auf einen Angriff die Nachricht eintreffen, dass das letzte Schiff versenkt wurde. In diesem Fall wechseln die Clientzustände in Abhängigkeit vom Sender und Empfänder dieser Nachricht in “gewonnen“ bzw. “verloren“.

Eine Auflistung der möglichen Ergebnisse eines Angriffs ist in Tabelle 2 gegeben.

Code	**Bedeutung**
1	Wasser
2	Schiff wurde getroffen
3	Schiff wurde getroffen und versenkt
4	Letztes Schiff wurde versenkt. Das Spiel ist beendet

Tabelle 2: Ergebniskodierung

4 Softwarearchitektur

Beschreibung der Schnittstellen, z.B. Dateien und ihre Struktur, Benutzte Module/-Bausteine und ihre Einbindung, Installationsvorschriften

Die Software des Spiels “Schiffe versenken“ wurde als Client-Server Anwendung gemäß Abbildung 1 entworfen. Die zentrale Kommunikationsschnittstelle stellt der Kommunikationsserver dar, der auf dem Port 54321 eingehende Verbindungen annimmt. Verbindungen können sowohl mittels eines Java-Clients hergestellt werden, die von einem Menschen bedient werden, als auch von Prolog-Clients, die vollständig autonom agieren. Es liegen hinsichtlich der Clientkombinationen keine Beschränkungen vor, sodass auch z.B. ein Prolog-Client gegen einen anderen Prolog-Client antreten kann.

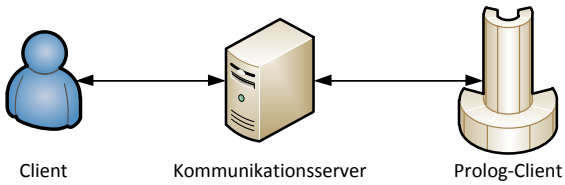

Abbildung 1: Darstellung der möglichen Kommunikationsteilnehmer

Der gesamte Spielverlauf lässt sich anhand der Statusdiagramme in den Abbildungen 2 und 3 beschreiben. Diese sind sowohl für den Javaclient, als auch für das Prologprogramm gültig.

Unmittelbar nach Start des Clients befindet sich dieser im Initialisierungszustand *INITIALIZATION*. Während dieser Phase obliegt es dem Client seine Schiffe gemäß den Regeln zu platzieren. Des Weiteren hat er eine Nachricht des Servers zu empfangen, die angibt, ob sein initialer Zustand *DEFENCE* oder *ATTACK* sein soll, wenn er in die *RUNNING*-Phase übergeht. Dieser Übergang erfolgt durch den Empfang des Startkommandos, mit dem Opcode = 5.

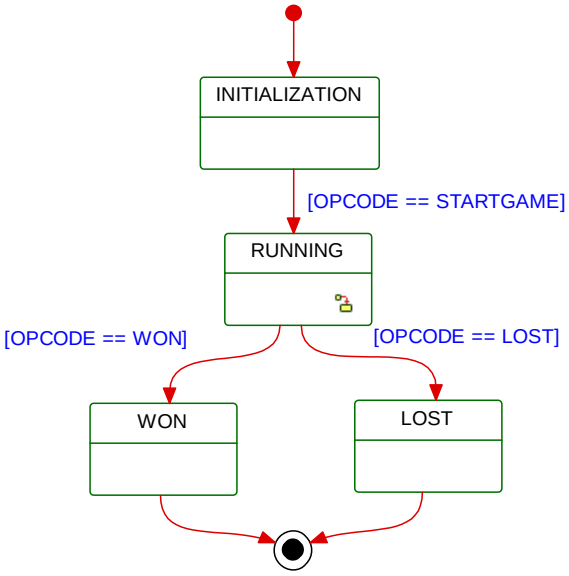

Abbildung 2: Zustandsübergänge des Clients

Vic - Bemerkung zur Grafik: es gibt doch garkeinen OPCODE == LOST oder WON? Meiner Meinung nach wird der Zustandsübergang wird durch 'letztes Schiff versenkt' ausgelöst und dann hängt es davon ab, ob es das eigene ist oder nicht..? gleiches gilt für die folgende abbildung

Befindet sich der Client im *RUNNING*-Status, so wechselt er zwischen seinen internen Zuständen *ATTACK* und *DEFENCE* hin und her. Dieser Wechsel erfolgt immer dann, wenn eine *ATTACKRESPONSE* Nachricht mit dem Opcode = 2 übertragen wurde. Der primäre Unterschied zwischen beiden Zuständen ist die Reihenfolge der erwarteten Nachrichten. Befindet sich der Client im Subzustand *DEFENCE*, so erwartet er von seinem Kontrahenten eine Nachricht mit dem Opcode = 1 und beantwortet diese seinerseits mit dem Opcode = 2. Sollte sich der Client im Subzustand *ATTACK* befinden, so sendet er zuerst die Nachricht mit dem Opcode = 1 und erwartet im Anschluss eine Nachricht seines Gegners.

Das Alternieren der Zustände erfolgt solange, bis in der Antwortnachricht mit dem Opcode = 2 eine Meldung über den Verlust aller Schiffe transferiert wird. Dieses Ereignis wird gemäß Tabelle 2 auf Seite 6 mit dem Ergebniscode = 4 beschrieben. Empfängt der Client die Nachricht, so gilt das Spiel als gewonnen. Umgekehrt verliert der Client das Spiel, wenn er diese Nachricht verschickt.

Im Rahmen dieses Kontexts wechselt der Spielzustand in *WON* bzw. *LOST* und das Programm kann beendet werden.

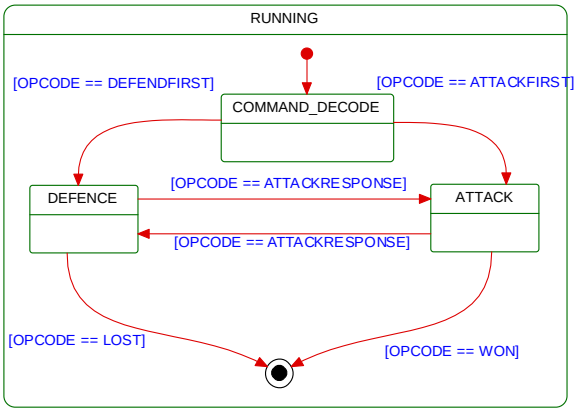

Abbildung 3: Interne Zustandsübergänge während des laufenden Spiels

Die vorgesehene Kommunikationsabfolge wird ebenfalls in Abbildung 4 in Form eines Sequenzdiagramms dargestellt. Das Diagramm stellt einen beispielhaften Spielablauf mit einem Prolog-Client und einem Java-Client dar. Der Client, der sich zuerst mit dem Server verbindet, beginnt per Konvention im Verteidungsmodus und der zweite verbundene Client startet im Angriffsmodus. Erst wenn sich zwei Teilnehmer am Server verbunden haben, sendet dieser das Startsignal an alle Clients.

Die Spielphase wird durch eine Schleife bestimmt, in der die Teilnehmer von den Angriffs- in den Verteidigungszustand wechseln und umgekehrt. Die dabei übertragenen Nachrichten werden stets an den Server übertragen, der diese an den jeweils anderen Client weiterleitet. Dadurch kommt keine direkte Verbindung beider Spieler zustande.

Das Programmende aus Sicht der Kommunikation wird erreicht, wenn ein Spieler die Nachricht überträgt, dass er über keine Schiffe mehr verfügt. In diesem Fall werden die Verbindungen getrennt und der Server kann neue Verbindungen von Spielern zum Ausrichten eines neuen Spiels annehmen.

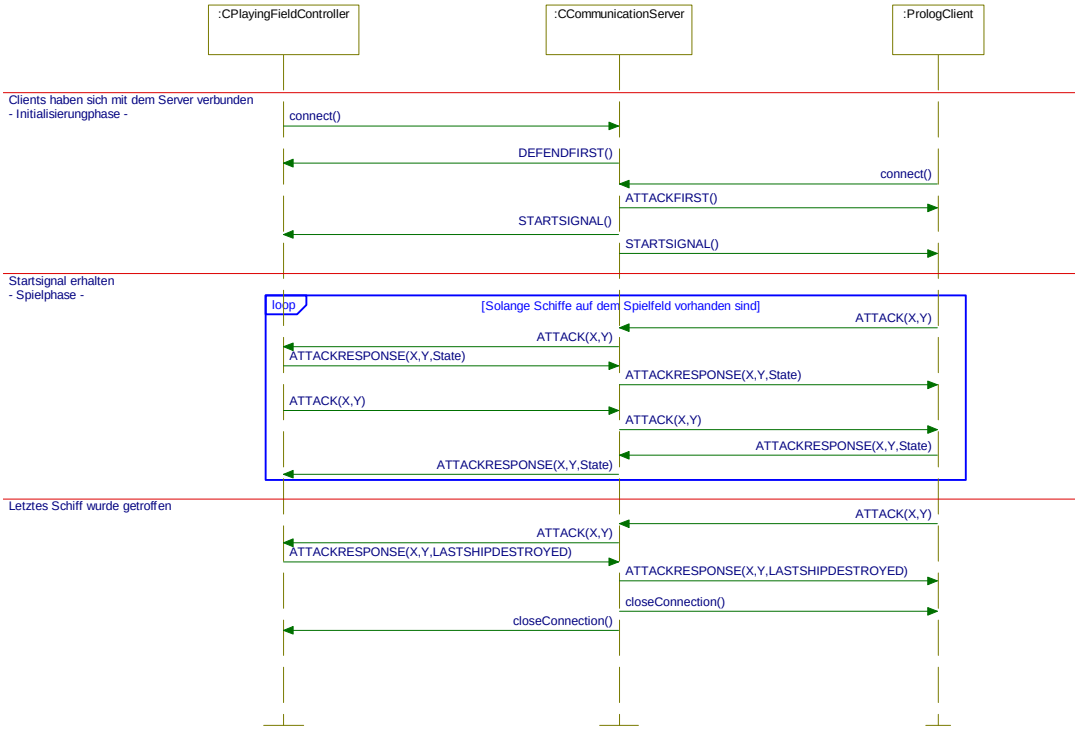

Abbildung 4: Sequenzdiagramm der Kommunikation

4.1 Spielserver

Der Server bildet die Schnittstelle zwischen den beiden Kommunikationspartnern, indem er eingehende Nachrichten eines Clients an den jeweils anderen verbundenen Teilnehmer weiterleitet. Somit limitiert der Server die maximale Spielerzahl. Des Weiteren generiert der Server das Startsignal, sobald sich zwei Teilnehmer mit ihm verbunden haben und legt fest, welcher der Spieler im Verteidigungs- bzw. Angriffsmodus startet.

Die Serverlogik wird maßgeblich durch die beiden Klassen `CCommunicationServer` und `CClientHandler` gesteuert (vgl. Abbildung 5). Die Aufgabe der Klasse `CCommunicationServer` ist die Annahme eingehener Verbindungen. Für jeden verbundenen Client wird ein separater Thread vom Typ `CClientHandler` erzeugt und gestartet.

Diese Threadobjekte führen die eigentliche Serverlogik aus. Sie überwachen den eingehenden Datenstrom und sobald eine Nachricht eingegangen ist, wird diese kopiert und an den jeweils anderen Client weitergeleitet. Für die Überwachung des Eingangs-

stromes ist die Methode `run()` verantwortlich und das Duplizieren der Nachricht wird in `notifyAllOtherClients(String line)` gehandhabt. Der eigentliche Sendevorgang wird durch den Aufruf der Methode `send(String msg)` ausgeführt.

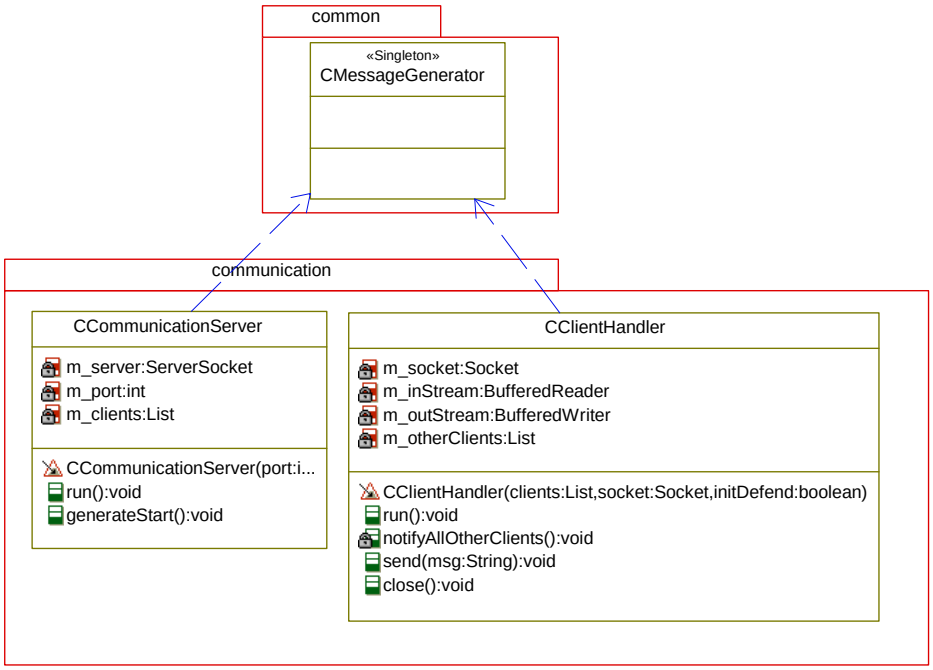

Abbildung 5: Klassendiagramm des Servers

4.2 Client - Java

Der hier beschriebene Java-Client ist ein möglicher Teilnehmer, der sich mit den Kommunikationsserver verbinden und ein Spiel austragen kann. Sein Design orientiert sich am Model-View-Controller (MVC) Prinzip, wobei das Spielbrettmodell auf Grund seines geringen Umfangs im Controller eingebettet wurde.

Eine Übersicht des Entwurfs wird in Abbildung 6 dargestellt. Die beiden Klassen `CBattleShipGUI` und `CPlayingFieldPanel` bilden den *View* Bestandteil des MVC Prinzips ab.

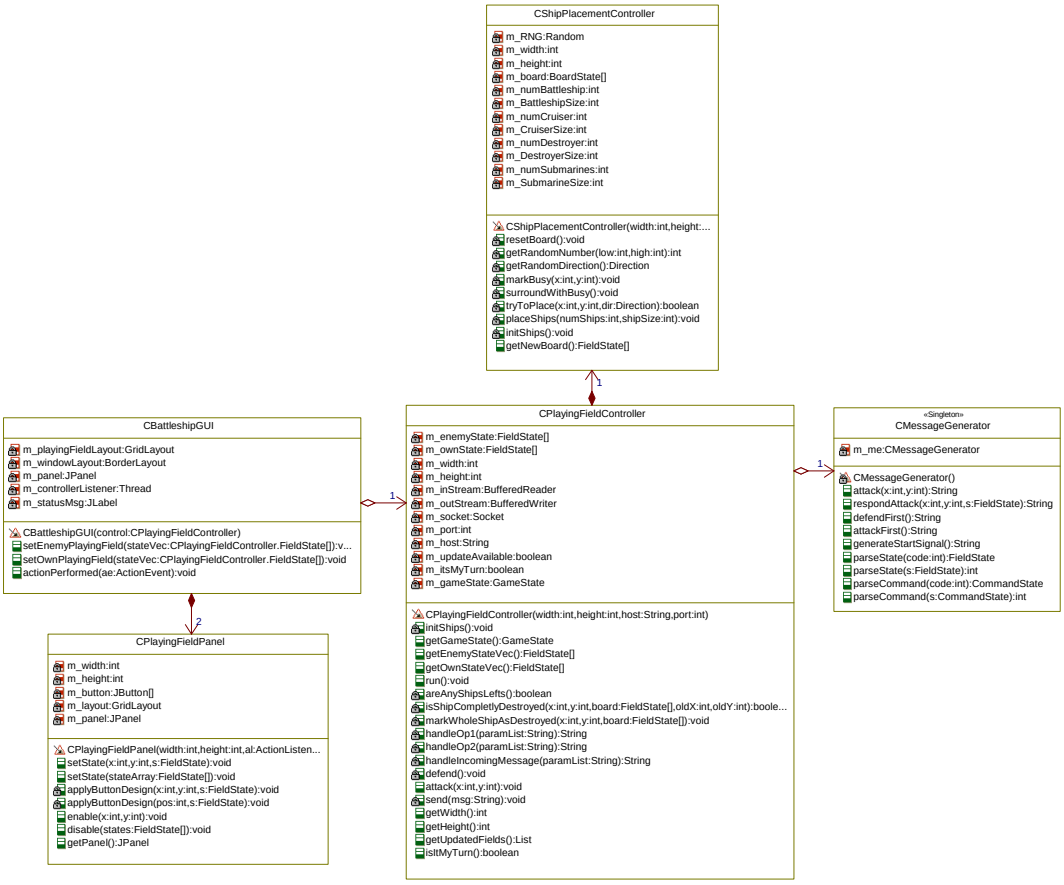

Abbildung 6: Klassendiagramm des Java Clients

Vic - Ich befürchte, dass das Klassendiagramm kaum lesbar ist, macht es dann sinn das reinzunehmen? oder siehst du ne möglichkeit das größer zu machen?

4.2.1 View

Das eigene, sowie das gegnerische Spielfeld werden durch jeweils eine Instanz der Klasse `CPlayingFieldPanel` visualisiert, wie es in Abbildung 7 zu sehen ist. Dieses besteht primär aus einem gitterförmigen Spielfeld, dessen Spielfeldzustände durch Farben codiert sind. Das hier gezeigte Spielfeld stellt das eigene Wissen des gegnerischen Spielfelds dar. Die grauen Felder symbolisieren noch unbekanntes Terrain. Blau bedeutet, dass bei einem vorangegangenen Angriff ein Feld mit Wasser getroffen wurde. Rot symbolisiert ein getroffenes, jedoch noch nicht versenktes Schiff. Versenkte Schiffsfelder sind dunkelgrau

gehalten.

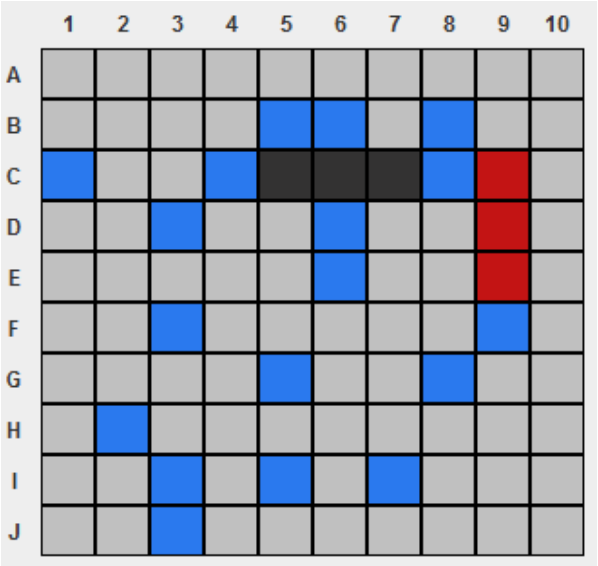

Abbildung 7: Gegnerisches Spielfeld mit einem zerstörten (dunkelgrau) und einem beschädigten Schiff (rot)

Die visuelle Codierung des eigenen Spielfeldes wird analog zum gegnerischen Feld vorgenommen. Hier sind standardmäßig alle Felder aufgedeckt und wie in Abbildung 8 zu sehen ist, hellblau eingefärbt. Die eigenen Schiffe sind ebenfalls sichtbar und mit hellgrauer Farbe hervorgehoben. Angriffe des Gegners, die das Wasser getroffen haben, sind dunkelblau dargestellt. Analog zum gegnerischen Spielfeld sind Treffer rot und versenkte Schiffe dunkelgrau eingefärbt.

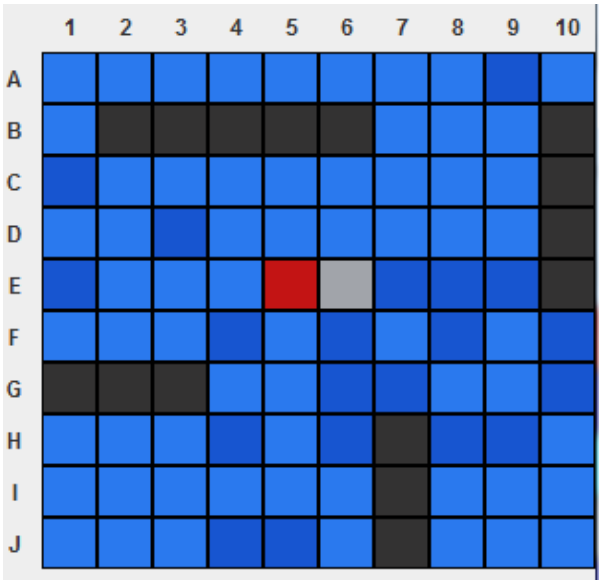

Abbildung 8: Eigenes Spielfeld mit versenkten Schiffen (dunkelgrau), einem beschädigten Schiff (rot) und misglückten Angriffen des Gegners (dunkelblau)

Die GUI-Oberfläche besteht primär aus zwei Instanzen der Klasse `CPlayingFieldPanel`, die jeweils das eigene und gegnerische Spielfeld abbilden. Wie in Abbildung 9 zu erkennen ist befindet sich über jedem Spielfeld eine Überschrift, die die Zugehörigkeit signalisiert. Des Weiteren befindet sich in der unteren linken Ecke ein Statusfeld, das angibt, wer den nächsten Spielzug auszuführen hat bzw. ob man das Spiel verloren bzw. gewonnen hat. Sollte der Gegner am Zug sein, so wird gleichzeitig das gesamte Spielfeld deaktiviert, sodass keine Buttonaktionen ausgeführt werden können.

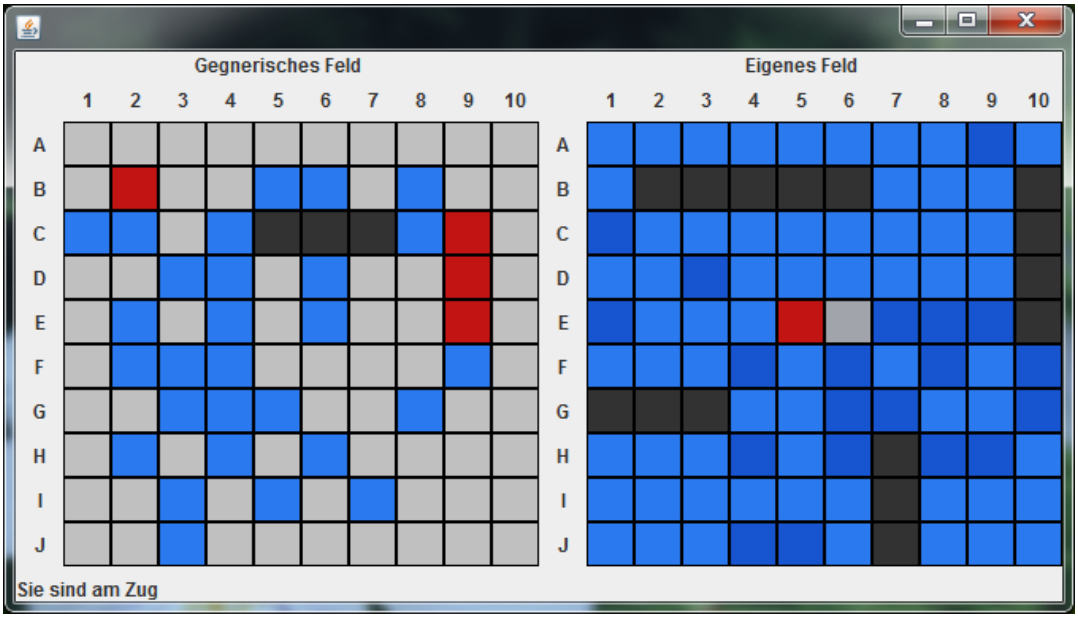

Abbildung 9: Gesamte Benutzeroberfläche

4.2.2 Controller / Model

Die Klasse `CPlayingFieldController` ist das zentrale Element der Spielsteuerung und wertet die eingehenden Nachrichten und Signale aus. Des Weiteren beinhaltet diese Klasse auch die Model-Komponente in Form zweier Arrays, die die Spielfelder repräsentieren. Der Abruf dieser Daten orientiert sich am Observer Entwurfsmuster. Jede Änderung an diesen Arrays wird mittels der javaeigenen Methode `notifyAllObservers()` den Beobachtern mitgeteilt.

Der Controller-Anteil wird durch die Erzeugung eines Threads realisiert, der eingehende Netzwerknachrichten annimmt und auswertet. Zur Generierung ausgehender Nachrichten, die einen Angriff signalisieren, dient die Methode `attack(int x, int y)`. Sie wird durch die Interaktion mit der GUI aufgerufen. Sowohl der Thread, als auch die GUI-Interaktion blockieren sich auf Basis von Monitoren gegenseitig, die mit dem Schlüsselwort `synchronized` signalisiert werden. Mittels dieser Technik wird verhindert, dass die beiden Zustände *ATTACK* und *DEFENCE* unsachgemäß eingenommen werden.

Eingehende Angriffe werden durch den Vergleich mit dem eigenen Spielfeld behandelt und das Ergebnis per Nachricht bekannt gegeben. Gleichzeitig wird der Status des

eigenen Spielfeldes an der entsprechenden Stelle aktualisiert. Im Fall von Wasser wird an dieser Position der Status *Verfehlt* gesetzt. Bei einem Schiff wird der Status auf *Getroffen* geändert. Gleichzeitig wird eine Routine gestartet, die überprüft, ob das das ganze Schiff versenkt wurde. Ist dem so, so wird wiederum eine Routine gestartet, die überprüft, ob alle Schiffe versenkt wurden. Die entsprechenden Ergebnisse werden per Netzwerkantwort übermittelt. Sind keine eigenen Schiffe mehr vorhanden, oder ging die Nachricht ein, dass der Gegner über keine Schiffe mehr verfügt, so wird der Spielstatus auf verloren, bzw. gewonnen gesetzt.

Die Generierung der prologkompatiblen Nachrichten wird in der Klasse `CMessageGenerator` vorgenommen. Diese bietet passende Methoden für jede Aktion und Reaktion und liefert den zu übertragenen Text. Die Klasse ist ebenfalls für die Dekodierung der Opcodes und Ergebnisse in interne Enumerationen zuständig.

In der Initialisierungphase ruft der Controller die Klasse `CShipPlacementController` auf. Diese Klasse ist für die regelkonforme Platzierung der Schiffe auf dem Spielbrett zuständig. Der Vorgang zum Platzieren eines Schiffes ist dabei von seinem Typ und somit seiner Größe unabhängig und kann wie folgt beschrieben werden:

1. Generiere zufällig die x/y Position, sowie die Orientierung, in der das Schiff platziert werden soll.

2. Prüfe ob diese Position belegt ist, oder ob sich in der direkten 4er Nachbarschaft ein Schiff befindet. Es gilt ebenfalls die Spielfeldgrenzen zu berücksichtigen.

3. Wiederhole den vorherigen Schritt rekursiv an der nächsten Position in angegebener Orientierung. Die Schiffsgröße wird mit jedem Rekursionsschritt um 1 verringert, sodass aus dieser Angabe die Abbruchbedingung erzeugt werden kann.

4. Wurde in keinem Feld rekursiv eine Behinderung festgestellt, werden die Felder mit der Schiffskennung versehen.

Diese vier Verarbeitungsschritte werden für jedes zu platzierende Schiff aufgerufen. Jedoch ist nicht gewährleistet, dass eine zufällig generierte Position auch frei ist. Deshalb steht pro Schiffstyp ein maximales Kontingent an Versuchen zur Verfügung, das sich nach jedem erfolgreich platzierten Schiff wieder füllt. Sollte das Kontingent für einen Schiffstypen aufgebraucht worden sein, so wird das Gesamtkontingent für das gesamte

Spielbrett reduziert und das gesamte Spielbrett wird zurückgesetzt. Sollte auch dieses Kontingent aufgebraucht worden sein, wird eine Fehlermeldung per Exception ausgegeben.

4.3 Client - Prolog

Die Aufgaben des Prolog-Clients sind in verschiedene Module unterteilt, um die verschiedenen Teile austauschbar zu halten. Abbildung 4.3 zeigt das Zusammenspiel der Module. Dabei stellt ein Pfeil die Verwendung eines Prädikates aus dem jeweiligen Modul dar. Die im Diagramm dargestellten Pfeile stellen somit die Schnittstellen zwischen den Modulen dar.

Bogi - Bild zum Modulzusammenspiel zusammenklickern ;)

Des Weiteren arbeiten einige der Module gemeinsam auf *globalen* Spielvariablen, die mit Hilfe von dynamischen Prädikaten gespeichert werden. Tabelle 3 zeigt diese Variablen, ihren Zweck sowie ihre Verwendung auf.

Prädikat	Zweck	Verwendet von
`myField/1`	Speichert Zustand des eigenen Spielfeldes	Initialisierungsmodul, Verteidigungsmodul, Ausgabemodul
`enemyField/1`	Speichert Zustand des gegnerischen Spielfeldes	Initialisierungsmodul, Strategiemodul, Angriffsmodul, Ausgabemodul
`openList/1`	Speichert Liste der bevorzugt anzugreifenden, gegnerischen Felder	Initialisierungsmodul, Strategiemodul, Ausgabemodul
`numberOfGames/1`	Speichert Verbleibende Anzahl von Spieldurchgängen (für aufeinander folgende Durchgänge)	Hauptmodul
`numberOfWins/1`	Speichert die Anzahl der bisher gewonnenen Spiele (für aufeinander folgende Durchgänge)	Hauptmodul

Prädikat	Zweck	Verwendet von
`numberOfLosses/1`	Speichert die Anzahl der bisher verlorenen Spiele (für aufeinander folgende Durchgänge)	Hauptmodul
`currentStream/1`	Speichert den aktuell gesetzten Ausgabestrom (Konsole, Datei, keiner)	Hauptmodul, Ausgabemodul

Tabelle 3: Prädikate für globale Spielvariablen

Im Folgenden werden die Aufgaben und die Funktionsweise der verschiedenen Module erläutert.

4.3.1 Hauptmodul

Das Hauptmodul `main.pl` wird beim Starten des Clients aufgerufen. Der Client bietet die Möglichkeit eine konfigurierbare Anzahl von Runden zu spielen. Dies wird voralem bei Spielen verwendet, in denen zwei *Computergegner* gegen einander antreten. Beim Start des Clients werden deshalb zunächst die Anzahl der Spiele, Siege sowie Niederlagen initialisiert und in den Prädikaten `numberOfGames/1`, `numberOfWins/1` sowie `numberOfLooses/1` gespeichert. Außerdem wird der gewünschte Ausgabestream im Prädikat `verbose/1` hinterlegt.

Für jedes neue Spiel initialisiert das Modul zunächst die Verbindung zum Spielserver. Anschließend werden Spielvorbereitungen über das Prädikat `initPrologClient/0` aus dem Initialisierungsmodul getroffen (siehe Abschnitt 4.3.2).

Je nachdem ob der Client im Angriffs- oder Verteidigungsmodus startet, werden die Prädikate `attackFirst/0` oder `defendFirst/0` aufgerufen. Nach Empfang des Startsignals beginnt der Client dann mit einem Angriff oder Verteidigung.

Ein Angriff verwendet die Prädikate `doAttack/2` und `attackResponse/3` des Angriffsmoduls (siehe Abschnitt 4.3.5). Das Prädikat `doAttack/2` liefert den Punkt auf dem gegnerischen Spielfeld der angegriffen werden soll. Die so erhaltenen X-Y-Koordinaten gibt das Hauptmodul an den Server weiter und wartet anschließend auf eine Antwort des Gegners (über den Server). Nach Erhalt dieser Antwort verwendet das Hauptmodul das Prädikat `attackResponse/3`, um die Antwort zu verarbeiten. Mit Abschluss der

Verarbeitung ist der Angriff beendet.

Die Verteidigung verwendet das Prädikat `doDefend/3` des Verteidigungsmoduls (siehe Abschnitt 4.3.4). Zu Beginn der Verteidigung wartet das Hauptmodul zunächst auf den Angriff des Gegners. Die so erhaltenen Koordinaten werden an das Prädikat `doDefend/3` übergeben. Als Resultat liefert dieses Prädikat die entsprechende Antwort für den Gegner. Das Hauptmodul sendet die Antwort gemäß dem Kommunikationsprotokoll an den Server. Damit ist die Verteidigung beendet.

Nach jedem Angriff überprüft der Client, ob er das Spiel gewonnen hat. Gleichermaßen überprüft er nach jeder Verteidigung, ob er das Spiel verloren hat. Tritt einer der beiden Fälle in Kraft, so beendet der Client das Spiel. Anschließend überprüft der Client ob weitere Runden ausstehen. Ist dies der Fall, so wird ein neues Spiel initialisiert. Andernfalls beendet sich der Client mit einer Ausgabe über den Spielverlauf: *End of game. KI won* `n` *times and lost* `m` *times.*.

4.3.2 Initialisierungsmodul

Die Aufgaben des Initialisierungsmoduls `initModule.pl` sind das Initialisieren der Spielfelder, darunter auch das Platzieren der eigenen Schiffe, sowie das Initialisieren der Openlist für primär anzugreifende Felder (siehe Abschnitt 4.3.6).

Zur Initialisierung der Spielfelder verwendet das Modul die Prädikate `initMyField/0` und `initEnemyField/0`. Die Spielfelder werden global in den dynamischen Prädikaten `myField/1` und `enemyField/1` gespeichert, um einen einfachen Zugriff von jedem Modul zu ermöglichen.

Spellcheck from here, please.

Bogi - Schreib bitte erstmal verständliche Sätze ;)

Zur Platzierung der eigenen Schiffe verwendet das Initialisierungsmodul das Prädikat `place/1` aus dem Modul zur Schiffspositionierung (siehe Abschnitt 4.3.3). Die von `place/1` gelieferte Liste beinhaltet (unter anderem) die Koordinaten der Schiffe, welche durch das Prädikat `fillWithShips/3` im globalen, eigenen Spielfeld `myField/1` als belegt gekennzeichnet werden. Hierfür wird die von `place/1` bezogene Liste rekursiv abgearbeitet und die enthaltenen X-Y-Koordinaten in `MyField/1` mit dem Status 6 belegt, der aussagt dass hier ein Teil eines Schiffes steht.

Die Listenelemente enhalten neben den Koordinaten noch eine Nummer zur Identifizie-

rung des Schiffes, sowie die “Teilenummer“ die angibt um den wievielten Teil eines Schiffes es sich handelt. Getrennt sind diese Komponenten durch einen Backslash. Somit stellt sich ein Listenelement wie folgt dar `Schiffs-ID/Teilnummer/X-Koordinate/Y-Koordinate`. Die X- und Y-Koordinate müssen zur korrekten Verwendungen jeweils noch um Eins dekrementiert werden, weil in der `initShips.pl` von einem Koordinatensystem mit den Wertebereichen $1 \leq X/Y \leq 10$ ausgegangen wird, während `myField/1` vom Wertebereich $0 \leq X/Y \leq 9$ ausgeht.

Sequenzdiagramm/Ablaufdiagramm, evtl.

4.3.3 Modul zur Plazierung von Schiffen

Bogi - TODO
ACK! Begonnen, Muse verloren. Später gehts weiter

Die Aufgabe des Positionierungsmoduls `initShips.pl` ist es, die Schiffe der künstlichen Intelligenz auf dem Spielfeld zu platzieren. Hierfür werden in diesem Modul die Regeln zur legalen Positionierung der Schiffe (vgl. 2) implementiert. Neben den Regeln zur Platzierung definiert `initShips.pl` auch die Anzahl und Länge der im Spiel genutzten Schiffe. Des weiteren wurden Prädikate implementiert, welche die Regeln auf die Menge der Schiffe Anwenden um so mögliche Positionierungen dieser auf dem Spielfeld zu erzeugen.

Das Prädikat `place/1` stößt die Erzeugung einer zufälligen Aufstellung aller Schiffe auf dem Spielfeld an und gibt diese als Liste der Form
`Schiffs-ID/Teilnummer/X-Koordinate/Y-Koordinate` an.

4.3.4 Verteidigungsmodul

Die Aufgabe des Verteidigungsmoduls `defendModule.pl` ist es einen Angriff des Gegners zu verarbeiten. Zum einen muss dabei der Status des eigenen Feldes `myField/1` aktualisiert und zum anderen die Antwort für den Gegner bestimmt werden.

Hat der Gegner ins Wasser geschossen, so ist keine Änderung des eigenen Feldes notwendig. Trifft der Gegner jedoch ein Schiff, so muss ermittelt werden, ob dieser Treffer das Schiff lediglich getroffen oder sogar versenkt hat. Außerdem ändert sich die Antwort für den Gegner, wenn das letzte Schiff versenkt wurde.

Ob das letzte Schiff versenkt wurde, erfolgt mit Hilfe einer Abfrage nach verbleibenden Schiffen (Feldstatus 6) im Prädikat `myField/1`.

Die Überprüfung, ob ein Schiff vollständig versenkt wurde, erfolgt über eine rekursive Überprüfung der 4 benachbarten Felder. Dabei erhöht sich die Rekursionstiefe, wenn ein Nachbarfeld ebenfalls als getroffen markiert ist. Da in den Regeln festgelegt ist, dass Schiffe sich nicht berühren dürfen, kann mit diesem Vorgehen festgestellt werden, ob ein Schiff vollständig versenkt wurde, oder sich unter den Nachbarn noch ungetroffene Teile befinden.

Schlägt die zuletzt erläuterte Überprüfung fehl, so wurde nur ein Teil eines Schiffes getroffen. Und die entsprechende Antwort wird zurück gegeben.

4.3.5 Angriffsmodul

Die Aufgabe des Angriffsmoduls `attackModule.pl` ist es, eine Koordinate für den nächsten Angriff zu liefern und außerdem den Rückgabewert des Angriffs zu verarbeiten. Dafür werden die Prädikate `doAttack/2` und `attackResponse/3` verwendet.

Wird das Prädikat `doAttack` aufgerufen, so nutzt das Angriffsmodul zunächst das Prädikat `getPointOfAttack/2` des Strategiemoduls, um den als nächstes zu attackierenden Punkt zu erhalten. Anschließend überprüft `doAttack/2` ob die erhaltene Koordinate im Feld des Gegners `enemyField/1` als unbekannt (Status 0) gilt (Prädikat `doAttackCheck`. Ist dies nicht der Fall, so wird eine neue Koordinate von `getPointOfAttack/2` angefordert. Gilt die Koordinate als unbekannt, so wird dieser Punkt als nächster Angriffspunkt zurück gegeben.

Das Prädikat `attackResponse/3` verarbeitet die Antwort des Gegners auf einen Angriff. Zum einen wird das intern gehaltene, gegnerische Feld aktualisiert (`updateEnemyField/3`), zum anderen wird die Openlist für weitere Angriffe über das Prädikat `updateOpenList/3` aus dem Strategiemodul aktualisiert (siehe Abschnitt 4.3.6).

Zur Aktualisierung des gegnerischen Spielfeldes `enemyField/1` wird der vom Gegener erhaltene Status in das entsprechende Feld eingetragen. Eine Ausnahme besteht für die Antwort *Schiff vollständig versenkt*, in diesem Fall wird das ensprechende Feld lediglich mit dem Status *Treffer* belegt, um die Handhabung des Spielfeldes zu erleichtern.

4.3.6 Strategiemodul

Das Strategiemodul `strategy.pl` stellt das Prädikat zur Bestimmung des nächsten Angriffspunktes `getPointOfAttack/2` zur Verfügung. Außerdem füllt dieses Modul die Liste der priorisiert anzugreifenden Punkte über das Prädikat `updateOpenList/3`.

Beim Aufruf von `getPointOfAttack/2` wird das erste Element der Openlist `openList/2` zurückgegeben. Befinden sich keine Koordinaten in der Openlist `openList/1`, so gibt `getPointOfAttack/2` einen zufälligen Angriffspunkt zurück.

Das Prädikat `updateOpenList/3` erhält die angegriffene Koordinate und den vom Gegner erhaltenen Rückgabewert. Traf der Angriff Wasser oder das letzte Schiff des Gegners, so erfolgt keine Änderung der Openlist.

Wurde das gerade attackierte Schiff vollständig versenkt, so kann die aktuelle Openlist vollständig geleert werden, da stets nur ein Schiff attackiert wird. Außerdem werden die unmittelbar benachbarten Felder des versenkten Schiffes als *Wasser* markiert, denn Aufgrund der Spielregeln darf sich auf diesen Feldern kein weiteres Schiff befinden (Prädikat `surroundWithWater/4`).

Ist die Antwort des Gegeners *Treffer*, so muss die OpenList aktualisiert werden. Dazu werden zunächst alle benachbarten Felder in die Openlist eingetragen, deren Status unbekannt ist (Prädikat `appendFreeFieldToList/2`). Anschließend wird mit dem Prädikat `checkHitDirection/2` überprüft, ob die Orientierung des attackierten Schiffes (horizontal oder vertikal) bereits durch frühere Treffer bekannt ist. Ist dies der Fall, so kann die Openlist entsprechend um auszuschließende Positionen verkürzt werden.

4.3.7 Ausgabemodul

Das Ausgabemodul `outputModule.pl` beinhaltet Prädikate zur Ausgabe der *globalen Variablen* `myField/1`, `enemyField/1` sowie `openList/1`. Die Ausgabe erfolgt normalerweise auf der Standardausgabe und kann während des Spiels oder zu Debugzwecken verwendet werden.

Bogi - ich glaub das 'normalerweise' aus dem vorherigen satz gilt jetzt nicht mehr oder? entweder im Standardcode ändern oder hier im text - was denst du?

Um Eine Überflutung der Ausgabekonsole bei vielen automatisierten Spieldurchläufen (zum Beispiel: KI gegen KI, 1000 spiele) zu vermeiden, kann die Ausgabe über das Prädikat `verbose/1` gesteuert werden (siehe Abschnitt 4.3.8).

4.3.8 Ausgabeverhalten

spellcheck??

Das Modul `verbosity.pl` steuert den Ausgabekanal, auf dem während des Spielens Debug-Informationen dargestellt werden können. Es sind drei verschiedene Ausgabeverhalten implementiert:

1. `verbose(0)` : Keinerlei Ausgabe während des Spiels.
2. `verbose(1)` : Ausgaben während des Spiels werden in eine Textdatei umgeleitet.
3. `verbose(2)` : Ausgaben während des Spiels erscheinen auf der Konsole.

Bei allen Modi wird nach Beendigung aller konfigurierten Spiele, eine Zusammenfassung der Ergebnisse (siehe Abschnitt **??**) auf der Ausgabekonsole Ausgegeben.

Zur Realisierung der verschiedenen Ausgabemodi wird der Standardausgabestrom der SWI-Prolog Umgebung über das Systemprädikat `set_output/1` geändert. Die Implementierung von `verbose/1` findet sich in `verbosity.pl`. Hier wird je nach Argument ein anderer Ausgabestrom definiert und als Standardausgabestrom gesetzt:

- `verbose(0)` : erzeugen eines Null-Stroms per `open_null_stream/1`.
- `verbose(1)` : erzeugen eines Ausgabestroms in die Datei “Output.txt“ im Unterordner “GameLogs“ per `open/3`.
- `verbose(2)` : erzeugen eines Ausgabestroms auf die Ausgabekonsole per `user_output/0`.

Nach dem erzeugen des jeweiligen Ausgabestroms wird dieser über das Systemprädikat `set_output` gesetzt und der aktuelle Ausgabestrom im globalen Prädikat `currentStream/1` gesetzt, sodass er überall im Programm manipuliert werden kann. Vor Ausgabe der Spielzusammenfassung wird der momentane Ausgabestrom über das oben beschriebene Prädikat `currentStream/1` geholt und über das Systemprädikat `close/1` geschlossen. Direkt danach wird `verbose(2)` gesetzt, sodass die Ausgabe der Zusammenfassung in jedem Fall auf der Ausgabekonsole erscheint.

5 Evaluation

5.1 Testkonzept

Entwicklertest, Testbarkeit der Software (Treiber,...), Positiv- und Negativtests, Stresstest (nicht funktional)

5.2 Testfälle

6 Benutzungshinweise für Endbenutzer

7 Ausblick

Ausblick - was fehlt noch? bekannte Fehler, spätere Verbesserungen, Fazit der Teilnehmer

Ausblick: Überprüfung adden z.B. wenn 2er Versenkt, können alle 2er Lücken als Wasser markiert werden, etc.

Ausblick: Schachbrettangriff