*Todo list

- Bogi, Vic - lest euch das bitte durch und guckt ob wir das wirklich so verteidigen können.. thx 29
- zustandsbasierter test - testet auch ungültige zustandsübergänge - möglcih??? . 29
- nicht funktional: portabilitätstest - wurde ja auf 2 plattformen entwickelt??! . . 29
- nicht funktional: Benutzbarkeit - wurde von beate verifiziert ;) 29
- Irgendwie gehts hier zu schnell zur sache - 'Der erste testfall...' vllt einen kleinen vorspann?? 29
- Beschreibung des Tests 30
- Beschreibung des Tests 31
- versteht ihr dass so?! -SB
  (Sobald man ein Schiff am Rand trifft, schließt sich Automatisch eine Richtung der Schiffsorientierung aus!) 34
- Sibille: Halte ich für nicht sinnvoll. würd ich persönlich lieber raus lassen. was sagt victor? 34

Wahlpflichtfach KI

Projektdokumentation: Schiffe Versenken

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

Inhaltsverzeichnis

1 Einleitung

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, dass die künstliche Intelligenz (KI) siegesorientiert spielt. Eine benutzungsfreundliche Oberfläche für menschliche Spieler wird durch ein Java GUI realisiert.

Das zu realisierende Programm soll als Client-Server Anwendung konzipiert werden, die neben einem Duell zwischen Mensch und Maschine, ebenfalls Kombinationen Mensch gegen Mensch und KI gegen KI erlauben soll.

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

Im Folgenden werden die Spielregeln des Spiels *Schiffe-Versenken* beschrieben. Diese, allgemein gültigen Regeln sind als Spezifikation für die Umsetzung des Spiels in Prolog und Java anzusehen.

- **Spielfeld**
  Für das eigene und das gegnerische Spielfeld gelten die folgenden Regeln:
  - Das Spielfeld eines jeden Spielers ist 10x10 Felder groß.
  - Jedes eigene Feld hat einen der folgenden Status: Wasser, Schiff, Getroffen oder Versenkt
  - Jedes gegnerische Feld hat einen der folgenden Status: Unbekannt, Wasser, Treffer oder Versenkt.
- **Schiffe**
  Für die am Spiel beteiligten Schiffe gelten die folgenden Regeln:
  - Folgende Schiffe müssen auf dem Spielfeld platziert werden: 1x 5er, 1x 4er, 2x 3er, 1x 2er.
  - Schiffe dürfen einander nicht berühren.
  - Schiffe dürfen horizontal und vertikal, nicht aber diagonal auf dem Spielfeld platziert werden.

  - Schiffe dürfen diagonal versetzt (aneinander liegende Eckpunkte) platziert werden.
  - Schiffe dürfen nach der ersten Platzierung nicht mehr verschoben werden.

- **Spielablauf**
  Für den Spielablauf gelten die folgenden Regeln und Abläufe:
  - Beide Spieler 'schießen' abwechselnd auf das Spielfeld des anderen.
  - Der Spieler, der sich zuletzt beim Server anmeldet, beginnt.
  - Ein angreifender Spieler teilt dem Gegner die angegriffene Koordinate mit.
  - Der angegriffene Spieler antwortet wahrheitsgemäß mit:
    * *"Wasser"* wenn kein Schiff auf dem angegriffenen Feld steht.
    * *"Treffer"* wenn ein Schiff auf dem angegriffenen Feld steht.
    * *"Versenkt"* wenn der letzte Teil eines Schiffes auf diesem Feld getroffen wurde.
  - **Spielende:** Es gewinnt der Spieler, der zuerst alle fünf Schiffe des Gegners versenkt hat.

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.

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

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 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.

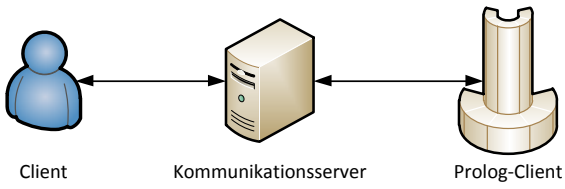


Abbildung 1: Darstellung der möglichen Kommunikationsteilnehmer

Der gesamte Spielverlauf lässt sich anhand der Statusdiagramme in den Abbildungen 2 und 3 beschreiben. Diese sind sowohl für den Java-Client, 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.

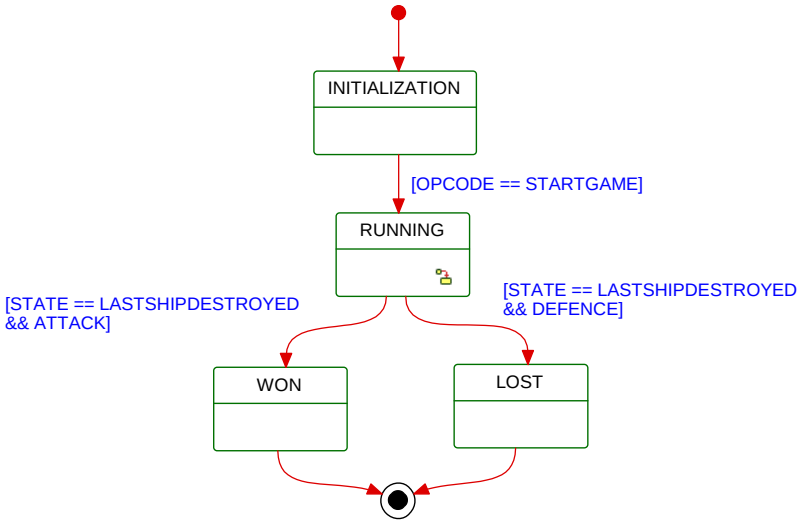


Abbildung 2: Zustandsübergänge des Clients

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 7 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.

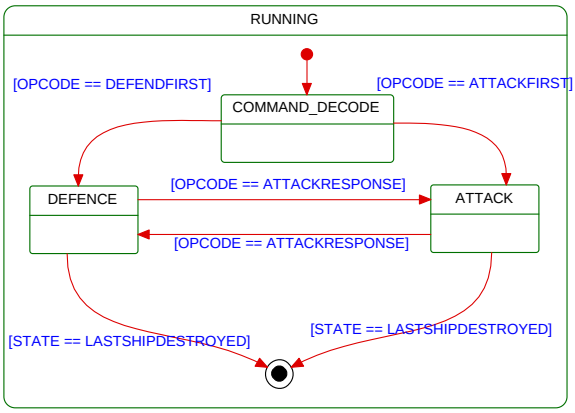


Abbildung 3: Interne Zustandsübergänge während des laufenden Spiels

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.

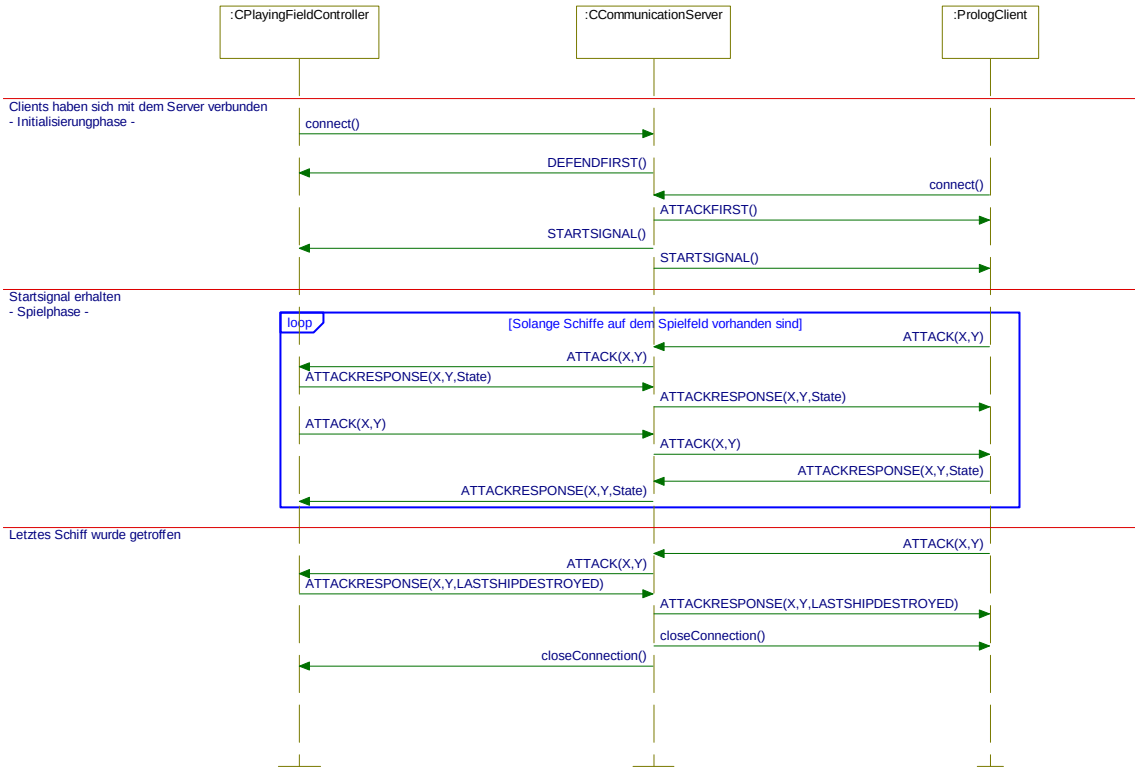


Abbildung 4: Sequenzdiagramm der Kommunikation

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.

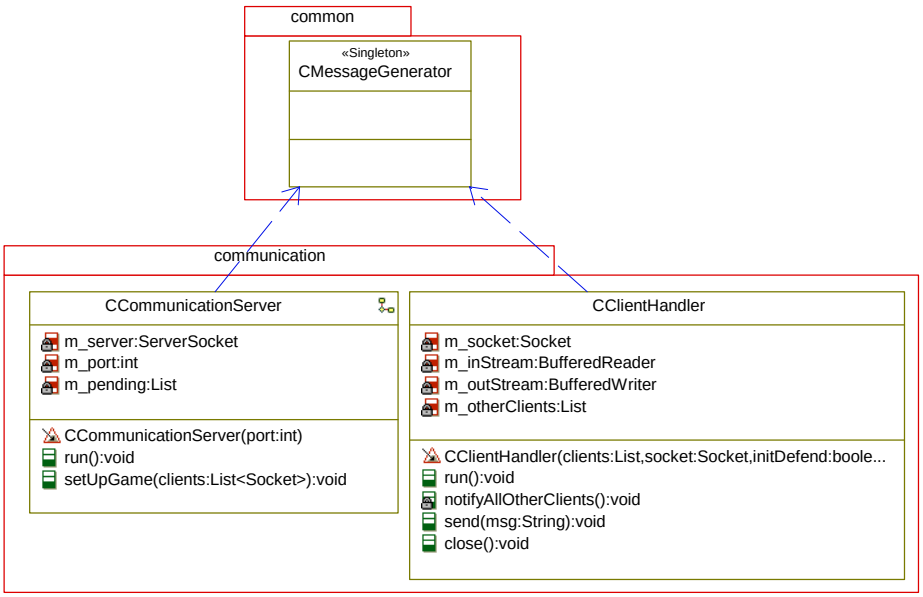


Abbildung 5: Klassendiagramm des Servers

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.

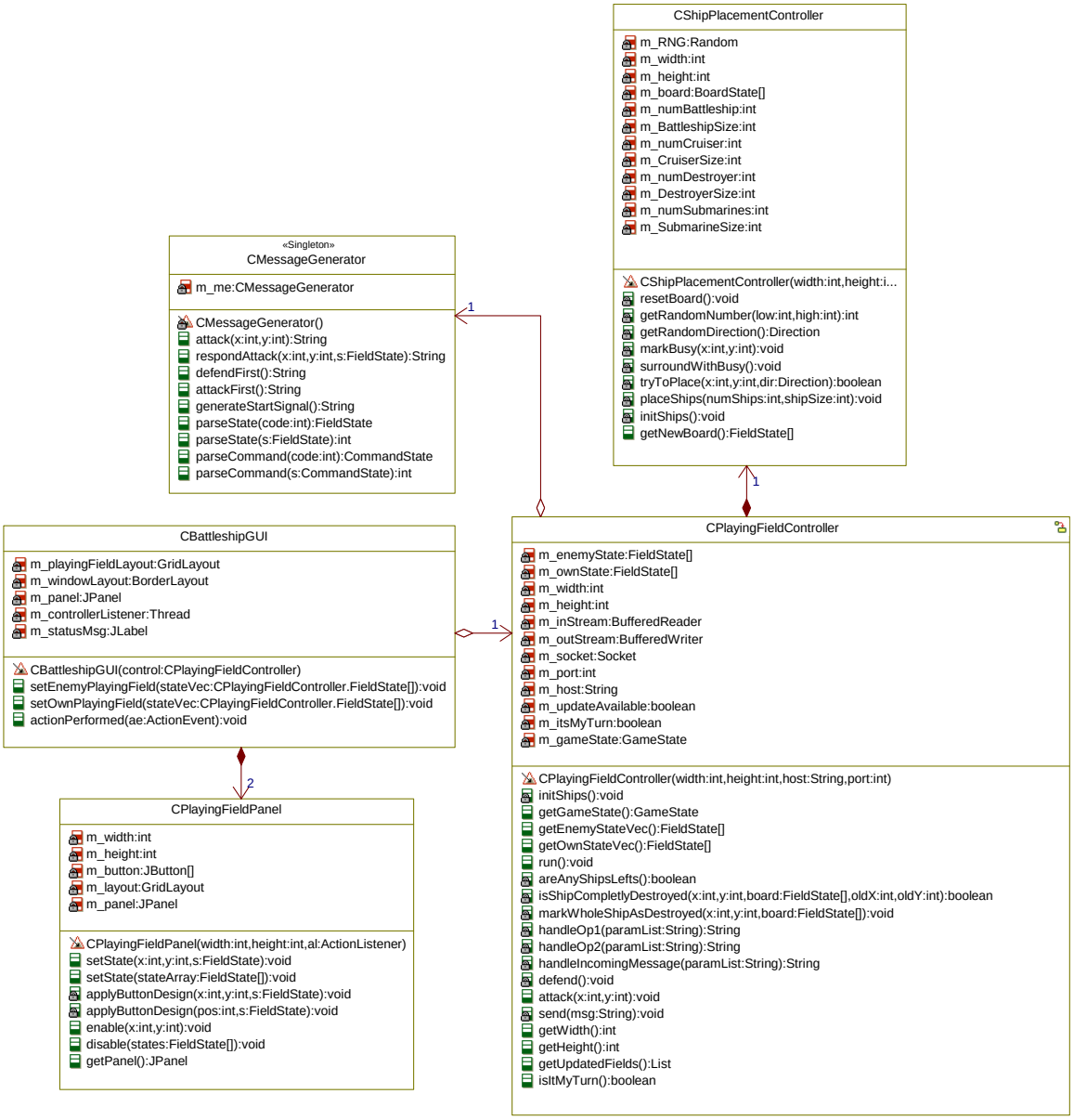


Abbildung 6: Klassendiagramm des Java Clients

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.

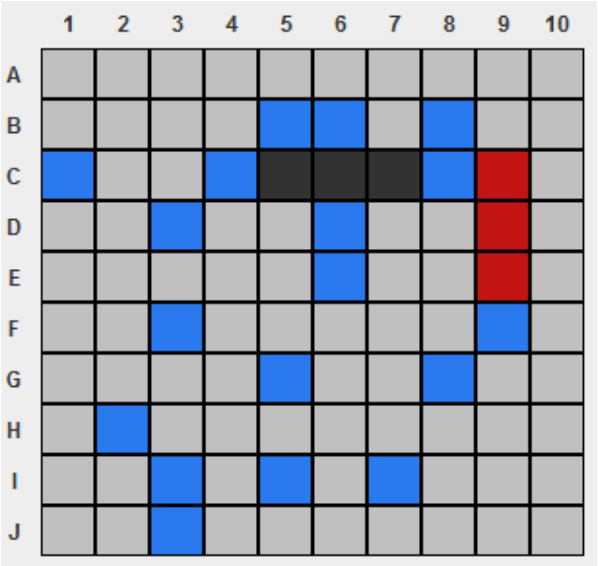


Abbildung 7: Gegnerisches Spielfeld mit einem zerstörten (dunkelgrau) und einem beschädigten Schiff (rot)

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.

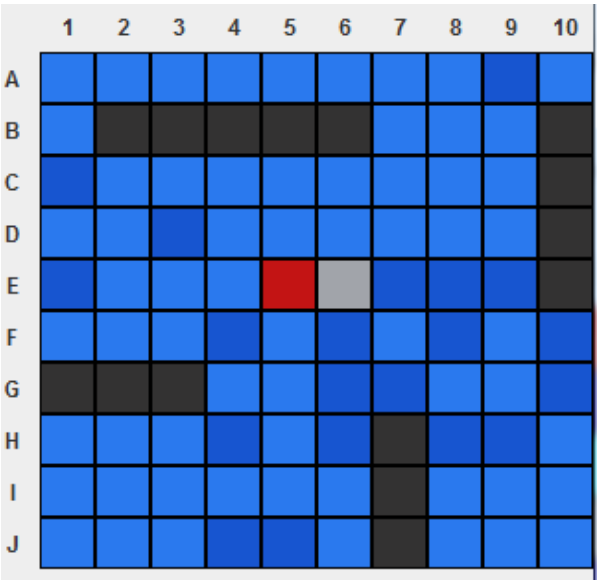


Abbildung 8: Eigenes Spielfeld mit versenkten Schiffen (dunkelgrau), einem beschädigten Schiff (rot) und misglückten Angriffen des Gegners (dunkelblau)

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.

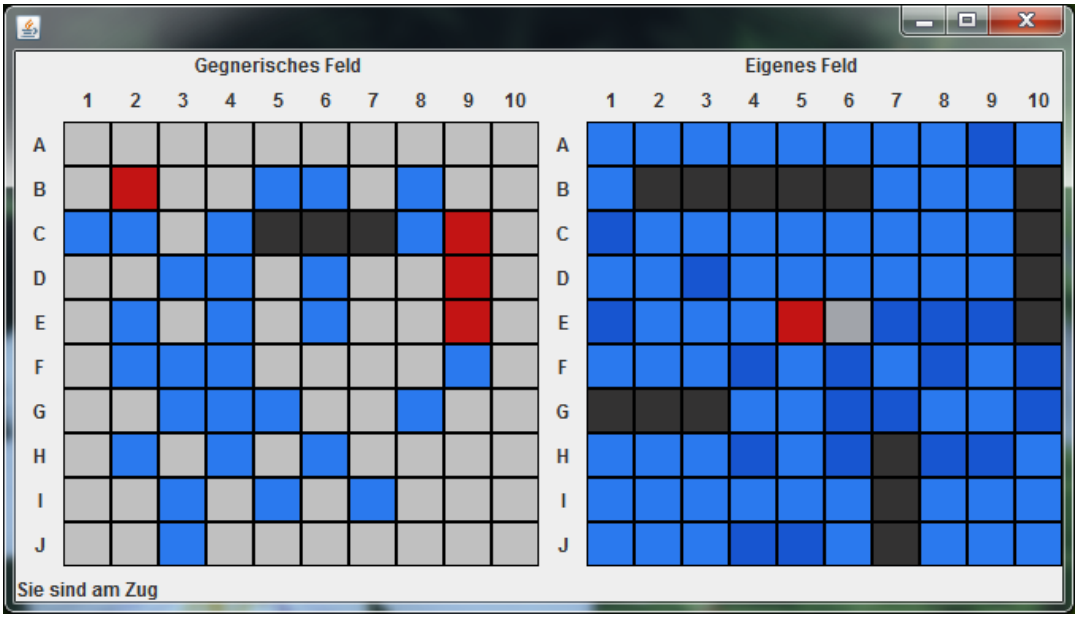


Abbildung 9: Gesamte Benutzeroberfläche

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 10 zeigt das Zusammenspiel der Module. Dabei stellt ein Pfeil die Verwendung eines Prädikates aus dem jeweiligen Modul dar. Die im Diagramm dargestellten Pfeile zeigen somit die Schnittstellen zwischen den Modulen auf.

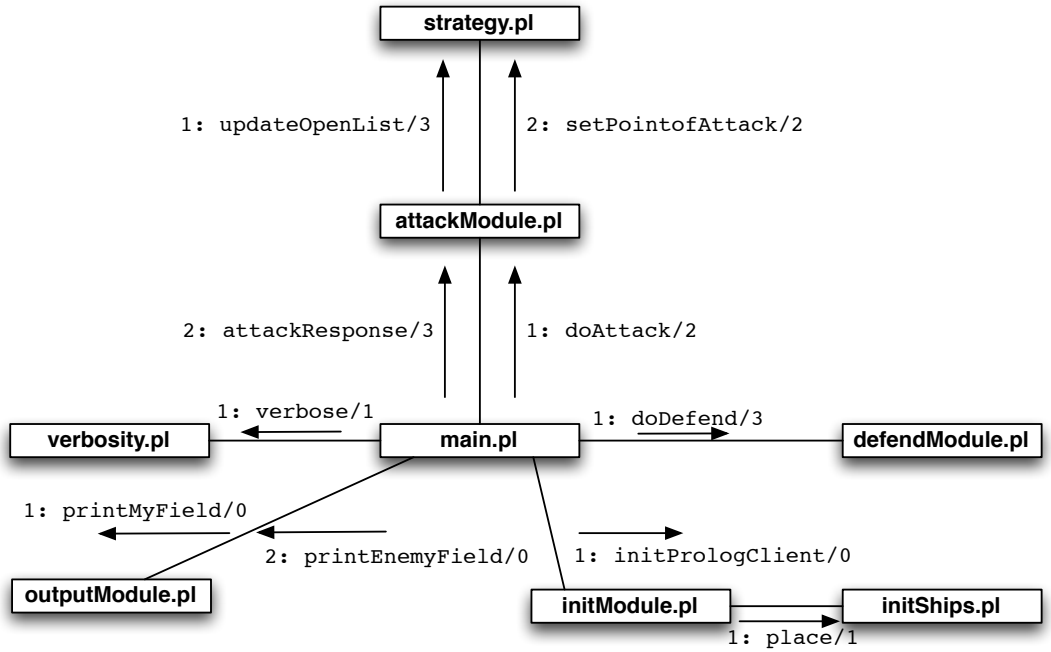


Abbildung 10: UML Kommunikationsdiagramm zur Kommunikation zwischen den Modulen des Prolog-Clients

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
`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 vorallem 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` so-

wie `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.* wobei ***n*** bzw. ***m*** die Anzahl der Siege bzw. Niederlagen ist.

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 Open-

list 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.

Zur Platzierung der eigenen Schiffe verwendet das Initialisierungsmodul das Prädikat `place/1` aus dem Modul zur Schiffspositionierung (`initShips.pl`, 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 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. Dieser Status sagt im Prolog-Client aus, dass sich auf der Koordinate ein Teil eines Schiffes befindet.

Die Listenelemente enhalten neben den Koordinaten noch eine Nummer zur Identifizierung des Schiffes, sowie die "Teilenummer" die angibt um den wievielten Teil eines Schiffes es sich handelt. Getrennt sind diese Komponenten durch einen Schrägstrich. 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 einmal dekrementiert werden, weil im Modul `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.

Abbildung 11 zeigt diesen Ablauf schematisch als UML-Diagramm. Die abgebildete Schleife wird in Prolog durch Rekursion realisiert.

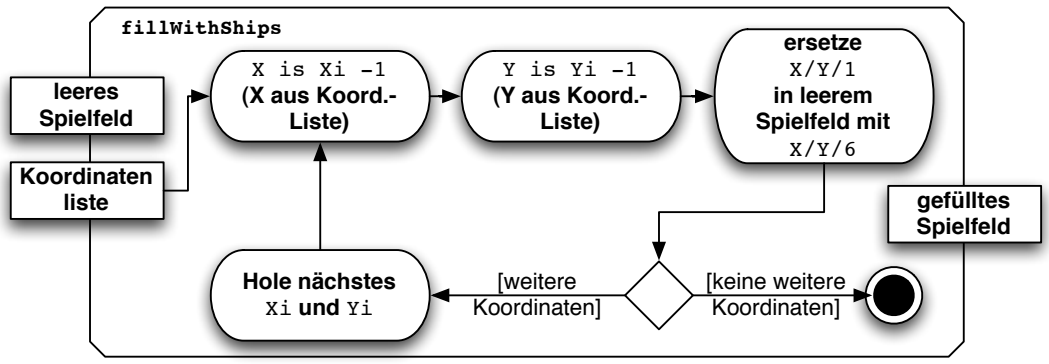


Abbildung 11: UML Aktivitätsdiagramm des Prädikats `fillWithships/3`. *Quelle: Eigene Darstellung.*

4.3.3 Modul zur Plazierung von Schiffen

Die Aufgabe des Positionierungsmoduls `initShips.pl` ist es, die Schiffe der künstlichen Intelligenz auf dem Spielfeld zu platzieren. Hierf¸r werden in diesem Modul die Regeln zur legalen Positionierung der Schiffe (siehe Abschnitt 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.

Um sicherzustellen, dass nur legale Positionierungen verwendet werden, wurden die Spielregeln (siehe Abschnitt 2) wie folgt implementiert:

- Schiffe dürfen horizontal oder vertikal auf dem Spielfeld platziert werden.

```
1  /* Teile des selben Schiffes müssen
   aneinander liegen (Horizontal) */
3  connectedHV(S1,P1,X1,Y1,S2,P2,X2,Y2):-
     S1 == S2,
5    P1 \== P2,
     X1-X2 =:= P1-P2,
7    Y1 == Y2,
     !
9  .
   /* Teile des selben Schiffes müssen
11 aneinander liegen (Vertikal) */
   connectedHV(S1,P1,X1,Y1,S2,P2,X2,Y2):-
13   S1 == S2,
     P1 \== P2,
15   Y1-Y2 =:= P1-P2,
     X1 == X2,
17   !
   .
```

Listing 1: Implementierung der Platzierungsregel,dass Schiffe Horizontal oder Vertikal ausgerichtet sein müssen, in Prolog

- Verschiedene Schiffe dürfen einander nicht berühren, aber diagonal versetzt stehen.

```
/*Ungleiche Schiffe müssen
Ein Feld Abstand halten,
ODER Diagonal versetzt stehen (Vertikal)*/
distance(S1,S2,X1,Y1,X2,Y2):-
  S1 \== S2,
  ((Y1 == Y2,
  abs(X1-X2)>1);
  (Y1 \== Y2,
  abs(X1-X2)>=1)),
  !
.
/*Ungleiche Schiffe müssen
Ein Feld Abstand halten,
ODER Diagonal versetzt stehen (Horizontal)*/
distance(S1,S2,X1,Y1,X2,Y2):-
  S1 \== S2,
  ((X1 == X2,
  abs(Y1-Y2)>1);
  (X1 \== X2,
  abs(Y1-Y2)>=1)),
  !
.
```

Listing 2: Implementierung der Platzierungsregel, dass Schiffe einander nicht berühren dürfen, in Prolog

Die Anwendung dieser Regeln erfolgt durch die Abfolge rekursiver Prädikate, welche eine Liste von legal positionierten Schiffen aufbauen. Es wird also erst ein Schiff auf dem Spielfeld positioniert, dann ein Zweites dazu gesetzt, darauf folgt das dritte Schiff und so weiter, bis alle fünf Schiffe Positioniert sind. Um eine möglichst zufällige und unvorhersehbare Positionierung der Schiffe zu erreichen, werden, da wo es möglich ist, Sequenzen von Zufallszahlen anstatt fest kodierter Listen genutzt. Folgendes UML-Diagramm (Abb. 12) stellt den diesen Ablauf schematisch und vereinfacht dar.

- Das `Template` ist eine Liste, welche die Schiffe und Eigenschaften (die Länge) beschreibt. Für ein fünf Teile langes Schiff, finden sich im Template fünf Elemente der Form `Schiffs-ID: 1-5/ Teilenummer: 1-5/ X-Koordinate: Variabel/ Y-Koordinate: Variabel`.

- Zu Beginn der Positionierung wird die Reihenfolge, in der die Schiffe platziert werden, zufällig festgelegt. Hierfür wird eine zufällige Sequenz der Zahlen 1 bis 5 erzeugt und in der Liste `ships` gespeichert.

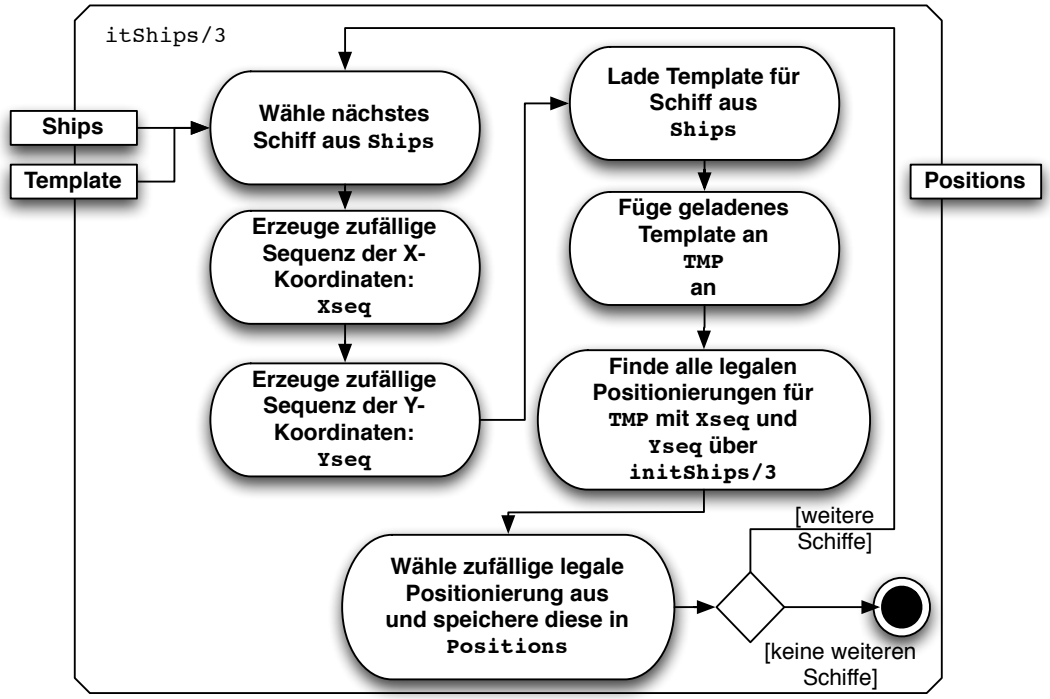


Abbildung 12: UML-Aktivitätsdiagramm zum Prädikat `itShips/3`

Auch in `itShips/3` ist die, in Abbildung 12 dargestellte Schleife, als rekursiver Prädikatsaufruf implementiert (vgl. Abschnitt 4.3.2 und Abbildung 11).

Das Prädikat `initShips/3`, welches in Abbildung 12 referenziert wird, wendet die in Listing 1 und 2 dargestellten Regeln auf die übergebene Liste von Schiffen an. Hierfür werden die übergeben X- und Y-Sequenzen (`Xseq`, `Yseq`; die Wertebereiche des Koordinatensystems, welches das Spielfeld abbildet) der Reihe nach abgearbeitet und somit alle legalen Positionierungen für die übergebene Schiffsliste gefunden. Aufgrund der Tatsache, dass immer nur ein Schiff mit Variablen X- und Y-Koordinaten in der übergebenen Liste vorhanden ist (andere, ggf. vorhandene Schiffe, haben bereits in vorherigen Durchläufen Koordinaten zugewiesen bekommen), hält sich der Rechenaufwand und die Anzahl der möglichen Positionierungen klein und Prolog kann eine Menge an Lösungen finden und zurückgeben.

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 wird lediglich die angegriffene Position aus der Openlist gelöscht.

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`).

Als mögliche Kandidaten der Openlist gehen die Felder der direkten 4er-Nachbarschaft des Treffers ein, wie es in Abbildung 13 zu sehen ist. Die Reihenfolge wurde auf Westen, Osten, Norden, Süden festgelegt.

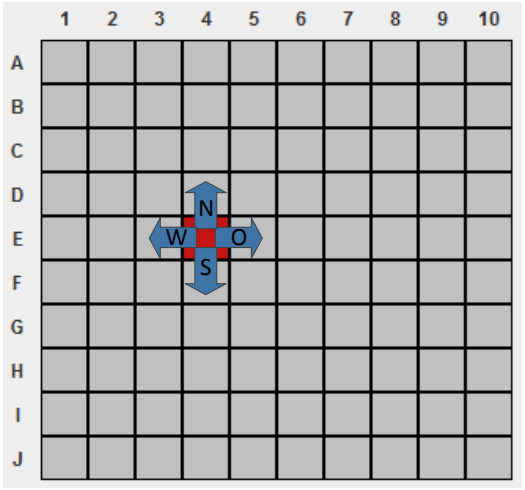
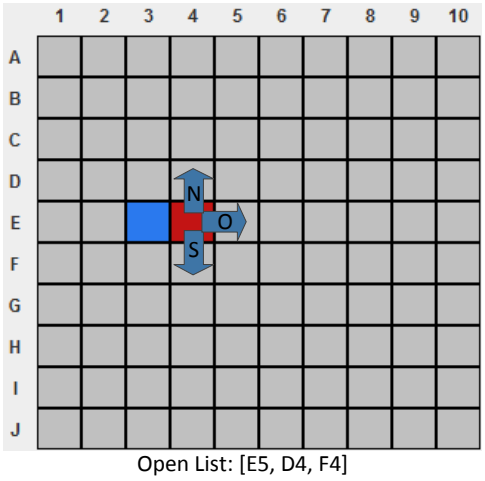
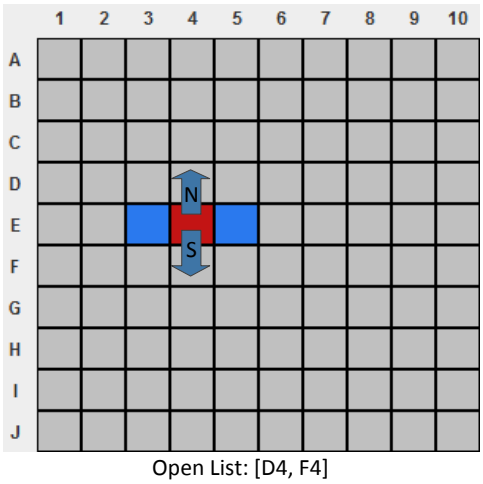


Abbildung 13: Openlist mit benachbarten Feldern initialisieren

In den nachfolgenden Spielzügen werden diese Felder geprüft. Sollten diese sich als “Wasser“ herausstellen, wie es in der Abbildung 14 angedeutet ist, so werden die entsprechenden Einträge ohne weitere Verarbeitung aus der Openlist gelöscht.



(a) Westen (verfehlt)

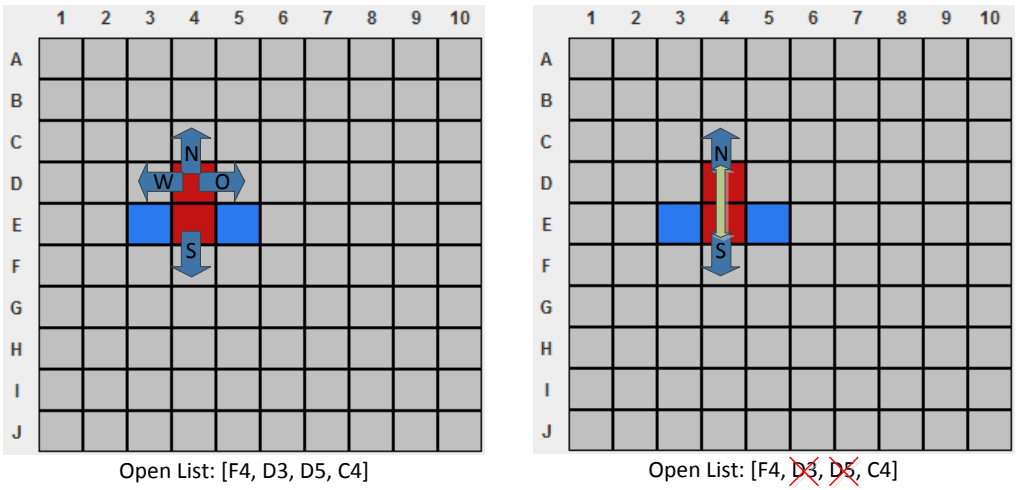


(b) Osten (verfehlt)

Abbildung 14: Openlist nach zwei misglückten Angriffen

Im Falle eines weiteren Treffers wiederholt sich der Algorithmus und nimmt alle unbekannten und benachbarten Felder des Treffers in die Openlist auf (vgl. Abbilidung

15(a)). 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. Im hier gezeigten Beispiel können die horizontalen Felder $D3$ und $D5$ wieder aus der Liste gestrichen werden.



(a) Norden (Treffer)

(b) Bereinigte Openlist

Abbildung 15: Aktualisierung der Openlist nach einem Treffer

Nachdem das Schiff vollständig zerstört wurde, wird die Openlist geleert und die Auswahl der anzugreifenden Koordinaten erfolgt wieder solange zufällig, bis der nächste Treffer erzielt wurde.

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 standardgemäß auf der Standardausgabe und kann während des Spiels oder zu Debugzwecken verwendet werden.

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

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 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

Bogi, Vic - lest euch das bitte durch und guckt ob wir das wirklich so verteidigen können.. thx

Ziel der durchgeführten Tests war es, mögliche Fehlerzustände im implementierten Programm aufzudecken und außerdem zu belegen, dass das System wie erwartet funktioniert.

Da die Verwendung strukturorientierter Testverfahren die Entwicklung eines Testrahmens bedeuten und dies den Rahmen des Projektes sprengen würde, wurde von der Verwendung dieser Verfahren abgesehen. Stattdessen wurde der Einsatz verschiedener Black-Box Methoden vorgesehen.

Um zu überprüfen ob das entwickelte Spiel die gewünschten Funktionen bietet, wurden funktionale Tests durchgeführt. Als Spezifikation hierfür wurden die in Abschnitt 2 beschriebenen Eigenschaften des Spiels *Schiffe-Versenken* verwendet.

zustandsbasierter test - testet auch ungültige zustandsübergänge - möglcih???

Außerdem wurden auch nicht funktionale Testverfahren verwendet. In diesem Kontext wurde auch ein Langzeittest durchgeführt, bei dem zwei Prolog-Clients 1000 Spiele gegeneinander spielen. Ziel dieses Tests war es diese hohe Anzahl von Spielen fehlerfrei zu beenden.

nicht funktional: portabilitätstest - wurde ja auf 2 plattformen entwickelt??!

nicht funktional: Benutzbarkeit - wurde von beate verifiziert ;)

5.2 Testfälle

Irgendwie gehts hier zu schnell zur sache - 'Der erste testfall...' vllt einen kleinen vorspann??

Der erste Testfall untersucht die korrekte Initialisierung des Prologclients. Insbesondere stellt die regelkonforme Platzieung der Schiffe einen wichtigen Bestandteil des Spiels dar.

Im gleichen Rahmen der Prüfung auf die korrekte Platzierung der Schiffe muss ebenfalls validiert werden, ob die geforderte Anzahl an Schiffen und die korrekten Schiffstypen

auf dem Spielfeld zu finden sind.

5.2.1 Testfall 1, Testfall 2

Zur Überprüfung der Korrektheit der Platzierung, der richtigen Gesamtanzahl der Schiffe, sowie die korrekte Anzahl der einzelnen Typen, wurden die Startaufstellungen von 100 KI-gegen-KI spielen gespeichert und ausgewertet. Bei jedem KI-gegen-KI Spiel erzeugt der Prolog Client zwei Aufstellungen (eine je KI-Spieler). Die so resultierenden 200 Aufstellungen wurden vom Entwicklerteam ausgewertet und auf Korrektheit in den angegebenen Aspekten überprüft.

Testergebnis Das Überprüfen der 200 generierten Aufstellungen ergab folgendes Ergebnis:

Geprüfte Aufstellungen	Fehlerhafter Platzierungen	Fehlerhafte Schiffsanzahl	Fehlerhafte Schiffstypen	Korrekte Aufstellungen	Duplikate
200	0	0	0	200	0

Tabelle 4: Testresultat für Testfall 1 und Testfall2

Das Testergebnis zeigt, dass die Routine zum Platzieren der Schiffe wie erwartet arbeitet. Beim Testen wurde gleichzeitig auch überprüft, wie oft Duplikate durch die Routine erzeugt werden (also identische Platzierungen). Das Auftreten keiner Duplikate lässt schlussfolgern, dass der Einsatz der Prolog-Systemprädikate `random/1` und `randseq/3` den gewünschten Effekt liefern (siehe Abschnitt 4.3.3) und wie erwartet arbeiten.

Die Liste der Ausgewerteten Textdaten befindet sich in
`Battleship\MyField_200Testdaten.txt`

5.2.2 subsubsection name

Der dritte Testfall behandelt die Auswahl des nächsten anzugreifenden Spielfeldes. Im besonderen Fokus steht die Aufgabe der Verfolgung eines getroffenen, aber noch nicht vesenkten Schiffes.

Beschreibung des Tests

Der vierte Testfall untersucht die korrekte Behandlung eines versenkten Schiffes. Insbesondere soll geprüft werden, ob die Umgebung in einer 4er-Nachbarschaft mit Wasser

aufgedeckt wird, da an diesen Stellen laut Regelbeschreibung keine Schiffe platziert werden dürfen und somit für die nachfolgenden Züge uninteressant sind.

Beschreibung des Tests

Vorbedignung - KI Ausgabe auf der Konsole

5.2.3 Unittest: CPlayingFieldController

Des Weiteren wurde die Controllerklasse `CPlayingFieldController` des Java-Clients einem Unittest unterzogen. Diese Klasse bildet das zentrale Element der Spiellogik. Das Ziel dieses Tests ist die Sicherstellung der korrekten Statuswechsel, sowie die korrekte Repräsentation der Spielinformationen.

Jedoch ist eine Untersuchung der einzelnen Methoden in diesem Fall nicht zielführend, da die internen Zustände und Variablen nur durch den Nachrichtenaustausch indirekt manipuliert werden können. Außerdem ist zu beachten, dass die Schiffsplatzierung zufällig erfolgt, sodass diese Komponente nicht vorhergesagt und somit geprüft werden kann.

Die Testsequenz entspricht den ersten Zügen eines angehenden Spiels. Sie unterteilt sich in die primär in die Phase der Initialisierung und die des laufenden Spiels.

Im Rahmen des Tests werden zwei Instanzen der Klasse `CPlayingFieldController` erzeugt, die sich mit dem Spieleserver verbinden. Unmittelbar nachdem sich ein Client verbunden hat, geht dieser in die Initialisierungphase über, was anhand des Unittests bestätigt wurde. Außerdem konnte ebenfalls bestätigt werden, dass der erste Spielteilnehmer erwartungemäß im Verteidigungszustand startet.

Im weiteren Verlauf der Testfolge verbindet sich ein weiterer Client. Erwartungsgemäß konnte verifiziert werden, dass dieser zum Einen den ersten Angriffs ausführt und zum Anderen sein Status in die des laufenden Spiels gewechselt ist.

In der zweiten Phase des Tests werden abwechselnd Angriffe gestartet und es wird geprüft, ob sich die internen Spielfelder den eingehenden Informationen entsprechend geändert haben. Die korrekte Darstellung des gegnerischen Spielfelds von *UNKNOWN* zu *WATER* bzw. *HIT* konnte bestätigt werden.

Nach zwei weiteren Spielzügen konnte der korrekte Wechsel vom Angriffs- in den Verteidigungszustand und vice versa der Clients verifiziert werden.

Da die Schiffe zufällig platziert werden, kann eine Evaluation des Spielendes nicht au-

tomatisch erfolgen. Aus diesem Grund wurde händisch ein Spiel JavaClient vs. JavaClient ausgetragen und geprüft, ob nach dem letzten versenkten Schiff der entsprechende Statuswechsel erfolgte. Auch dieser Test konnte erfolgreich bestätigt werden.

6 Benutzungshinweise für Endbenutzer

Um das Spiel *Schiffe-Versenken* zu spielen, muss zunächst der Spielserver gestartet werden. Anschließend können, je nach Spielabsicht zwei der bereitgestellten Clients gestartet werden. Dabei ist es egal, ob zwei Java-Clients, zwei Prolog-Clients oder je ein Client von jedem Typ gestartet wird.

6.1 Systemvoraussetzungen

Um ein fehlerfreies Starten des Servers und der Clients gewährleisten zu können, ist ein installiertes Java-Runtime-Environment der Version 1.6 erforderlich. Dieses kann ggf. unter `http://www.java.com/de/download` (Stand: 16.01.2011) heruntergeladen werden. Des Weiteren muss SWI-Prolog in Version 5.10.1 auf dem System installiert sein, um die KI-Komponenten ausführen zu können. SWI-Prolog kann ggf. unter `http://www.swi-prolog.org/Download.html` bezogen werden.

6.2 Starten des Servers

Der zum Spielen benötigte Server, `Battleship_Server.jar`, kann entweder selbst gestartet werden, oder durch eines der beiliegenden Start-Skripte. Der Server kann ebenfalls durch den nachfolgenden Befehl manuell auf der Kommandozeile gestartet werden:
`java -jar Battleship_Server.jar`

6.3 Starten eines Java-Client

Der Client, welcher das eigene Spielfeld abbildet, `Battleship_Client.jar`, kann analog zum Server, entweder selbst gestartet werden, oder durch eines der beiliegenden Start-Skripte. Ist es erforderlich, den Client eigenhändig zu starten, kann dies über den folgenden Kommandozeilenbefehl getan werden:

```
java -jar Battleship_Client.jar
```

6.4 Starten eines Prolog-Client

Ein Prolog-Client kann über einen Doppelclick auf die Datei `start_prolog_Windows` aus dem Ordner `Battleship` gestartet werden. Es öffnet sich eine Konsole, in der die Ausgaben des Clients (je nach Konfiguration) angezeigt werden.

6.4.1 Konfiguration des Prolog-Client

Standardmäßig ist der Prolog-Client für die Ausführung einer Runde konfiguriert. Außerdem werden die Spielzüge des Prolog-Clients auf der Kommandozeile ausgegeben. Sollen diese Einstellungen geändert werden, so muss die Datei `main.pl` entsprechend angepasst werden. Das Prädikat `numberOfGames/1` kann zur Festlegung der Rundenanzahl und das Prädikat `verbose/1` zur Anpassung der Ausgabe genutzt werden. Dabei steht der Wert `0` für keine Ausgabe, `1` für die Ausgabe in eine Datei und `2` für die Ausgabe auf der Konsole.

6.5 Startskripte

Die beiliegenden Startskripte starten alle zum Spielen erforderlichen Komponenten. Es sind drei Startskripte für verschiedene Spielmodi vorhanden:

- Menschlicher Spieler gegen KI-Spieler: `start_Human-KI-Match_`*`OS`*`.bat`
- KI-Spieler gegen KI-Spieler: `start_KI-KI-Match_`*`OS`*`.bat`
- Menschlicher Spieler gegen Menschlischen Spieler: `start_Human-Human-Match.bat`

Anmerkung: Je nach Betriebssystem ist *OS* durch “Windows“ oder “Unix“ zu ersetzen.

Die Benutzung dieser Startskripts bewirkt, dass alle benötigten Komponenten auf einer Konsole gestartet werden. Dies führt dazu, dass die Konsole auf der das Startskript ausgeführt wird, mit den Ausgaben aller drei Komponenten beschrieben wird. Für Debugging-Zwecke sollte daher auf die Verwendung dieser Skripte verzichtet werden.

7 Fazit und Ausblick

Das entwickelte Programm ermöglicht das Spielen von *Schiffe-Versenken* gegen eine *künstliche Intelligenz*. Landet die entwickelte Intelligenz einen Treffer, so ist sie in der Lage das zugehörige Schiff vollständig zu versenken. Außerdem werden weitere Schüsse auf die unmittelbare Umgebung eines versenkten Schiffes verhindert, da sich hier auf keinen Fall ein weiteres Schiff befinden kann.

Als Erweiterung des Programms könnte man einige Felder für den Beschuss ausschließen, wenn die kleinen Schiffe bereits versenkt wurden. Geht man beispielsweise davon aus, dass das kleinste Schiff (*der 2er*) vollständig versenkt wurde, so kann man alle Lücken, die lediglich Platz für dieses kleinste Schiff bieten, von den weiteren Überlegungen ausnehmen. Somit reduziert sich die Anzahl der möglichen Felder für einen Beschuss und die Wahrscheinlichkeit für einen Treffer erhöht sich.

Als weitere Verbesserung ist es denkbar, die zufälligen Angriffe (bei leerer Openlist) durch ein strukturiertes Angreifen zu ersetzen. Ein mögliches Vorgehen hierbei wäre es, jedes vierte gegnerische Feld anzugreifen und dieses Muster in jeder Zeile um ein Feld zu verschieben. Auf diese Weise ist das Treffen der beiden größten Schiffe garantiert. Trifft dieses Vorgehen durch Zufall nicht auch die anderen Schiffe, so wird beim weiteren Vorgehen jedes zweite gegenerische Feld attackiert. Auf diese Weise werden auch die übrigen Schiffe getroffen.

versteht ihr dass so?! -SB
(Sobald man ein Schiff am Rand trifft, schließt sich Automatisch eine Richtung der Schiffsorientierung aus!)

Sibille: Halte ich für nicht sinnvoll. würd ich persönlich lieber raus lassen. was sagt victor?

Bei der Platzierung von Schiffen kann eine Strategie angewendet werden, welche es vermeidet Schiffe am Rand des Spielfeldes zu platzieren. Hierdurch würden sich die Schussmöglichkeiten für den Gegner nicht noch zusätzlich durch den Spielfeldrand einschränken.

Eine umfangreichere Erweiterung wäre die Entwicklung eines lernenden Spielers, wie es beispielsweise in [1] beschrieben wird. Der Artikel behandelt die Entwicklung einer künstlichen Intelligenz die mit verschiedenen Typen von Gegenspielern trainiert wird. Im tatsächlichen Spiel stellt sich die künstliche Intelligenz dann auf das Verhalten des

Gegners ein, indem es die eigenen Schiffe auf weniger attackierte Felder positioniert und primär Felder angreift auf denen die gegnerischen Schiffe häufig positioniert sind.

8 Literatur

[1] J. G. Bridon, Z. A. Correll, C. R. Dubler, and Z. K. Gotsch. An artificially intelligent battleship player utilizing adaptive firing and placement strategies. `http://battlestar-ai.googlecode.com/svn/ResearchPaper.pdf`. [Online; aufgerufen 26.12.2010].