

---

*Todo list	
■ Bogi - detaillierte Aufgabenbeschreibung . . . . .	3
■ Bogi - Vervollständigen . . . . .	3
■ Zugrunde liegender Entwurf, grafische Darstellung der Softwarearchitektur . .	5
■ Beschreibung der Schnittstellen, z.B. Dateien und ihre Struktur, Benutzte Mo- dule/Bausteine und ihre Einbindung, Installationsvorschriften . . . . .	5
■ Gesamtübersicht - Bild - Modulzusammenspiel? . . . . .	16
■ Bogi - dein initships modul einbringen . . . . .	17
■ Bogi - TODO . . . . .	17
■ Ne schöne Grafik? . . . . .	18
■ Bogi - bitte vervollständigen, du wolltest da noch was adden (13.1.11) . . . .	19
■ Alle - haben wir schon ne Lösung, um den Output komfortabel umzustellen, wenn wir 100x spiele gegen die andere KI machen? . . . . .	19
■ Testkonzept und Testfälle . . . . .	20
■ Benutzungshinweise für den Endbenutzer . . . . .	20
■ Ausblick - was fehlt noch? bekannte Fehler, spätere Verbesserungen, Fazit der Teilnehmer . . . . .	20

# Wahlpflichtfach KI

Projekt: Schiffe Versenken

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

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
<b>2</b>	<b>Spielregeln</b>	<b>3</b>
<b>3</b>	<b>Kommunikationsprotokoll</b>	<b>3</b>
<b>4</b>	<b>Softwarearchitektur</b>	<b>5</b>
4.1	Spielserver . . . . .	9
4.2	Client - Java . . . . .	10
4.2.1	View . . . . .	11
4.2.2	Controller / Model . . . . .	14
4.3	Client - Prolog . . . . .	16
4.3.1	Hauptmodul . . . . .	16
4.3.2	Initialisierungsmodul . . . . .	17
4.3.3	Modul zur Plazierung von Schiffen . . . . .	17
4.3.4	Verteidigungsmodul . . . . .	17
4.3.5	Angriffsmodul . . . . .	18
4.3.6	Strategiemodul . . . . .	18
4.3.7	Ausgabemodul . . . . .	19
<b>5</b>	<b>Evaluation</b>	<b>20</b>
<b>6</b>	<b>Literatur</b>	<b>21</b>

## Abbildungsverzeichnis

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

## Tabellenverzeichnis

1	Kommunikationsprotokoll . . . . .	5
2	Ergebniskodierung . . . . .	5

# 1 Einleitung

Bogi - detaillierte Aufgabenbeschreibung

## 2 Spielregeln

Da die allgemeinen Regeln des Spiels *Schiffe versenken* bekannt sein sollten, werden an dieser Stelle nur die relevanten Eckdaten 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 2er
  - Schiffe dürfen sich nicht berühren
  - „SSchräge“ Platzierung ist gültig: 8er Nachbarschaft. Also nur keine anderen Schiffe in einer 4er Nachbarschaft.
- 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 in eckigen Klammerung umgebene Parameterliste und kann null bis drei durch Kommata getrennte numerische Elemente beinhalten. Anhand des Opcodes richtet sich die Anzahl der zu erwartenden Parameter.

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

$$COMMAND := \backslash(OPCODE, LIST\backslash)\backslash.NEWLINE$$

$$LIST := \backslash[[PARAMS]? \backslash]$$

$$NEWLINE := \backslash\backslash r$$

$$OPCODE := [1 - 5]$$

$$PARAMS := [0 - 9] + [, [0 - 9] + ] \{0, 2\}$$

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 Empfänger dieser Nachricht in “gewonnen“ bzw. “verloren“.

Eine Auflistung der möglichen Ergebnisse eines Angriffs werden in der Tabelle 2 präsentiert.

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

Zugrunde liegender Entwurf, grafische Darstellung der 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 Prologclient gegen einen anderen Prologclient antreten kann.

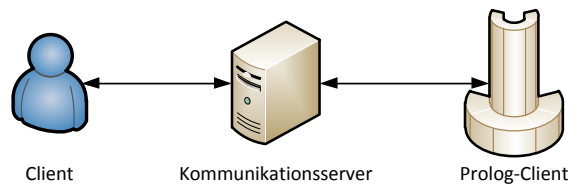


Abbildung 1: Darstellung der möglichen Kommunikationsteilnehmer

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.



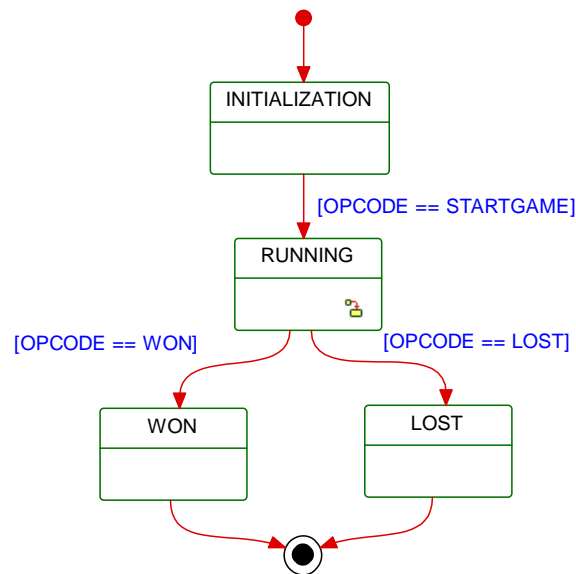


Abbildung 2: Zustandsübergänge des Clients

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, wie in der Antwortnachricht mit dem Opcode = 2 keine Meldung über den Verlust aller Schiffe transferiert wird. Dieses Ereignis wird gemäß Tabelle 2 auf Seite 5 mit dem Ergebniscodex = 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.

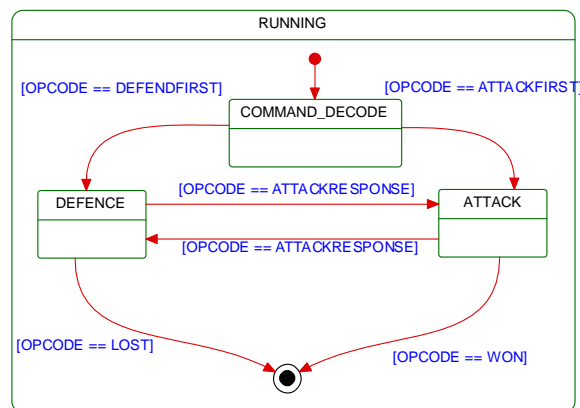


Abbildung 3: Interne Zustandsübergänge während des laufenden Spiels

Die vorgesehene Kommunikationsabfolge wird ebenfalls in Abbildung 4 in Form eines Sequenzdiagramms dargestellt. Der Client, der sich zuerst mit dem Server verbindet, beginnt per Konvention im Verteidigungsmodus 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 zweiten 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.

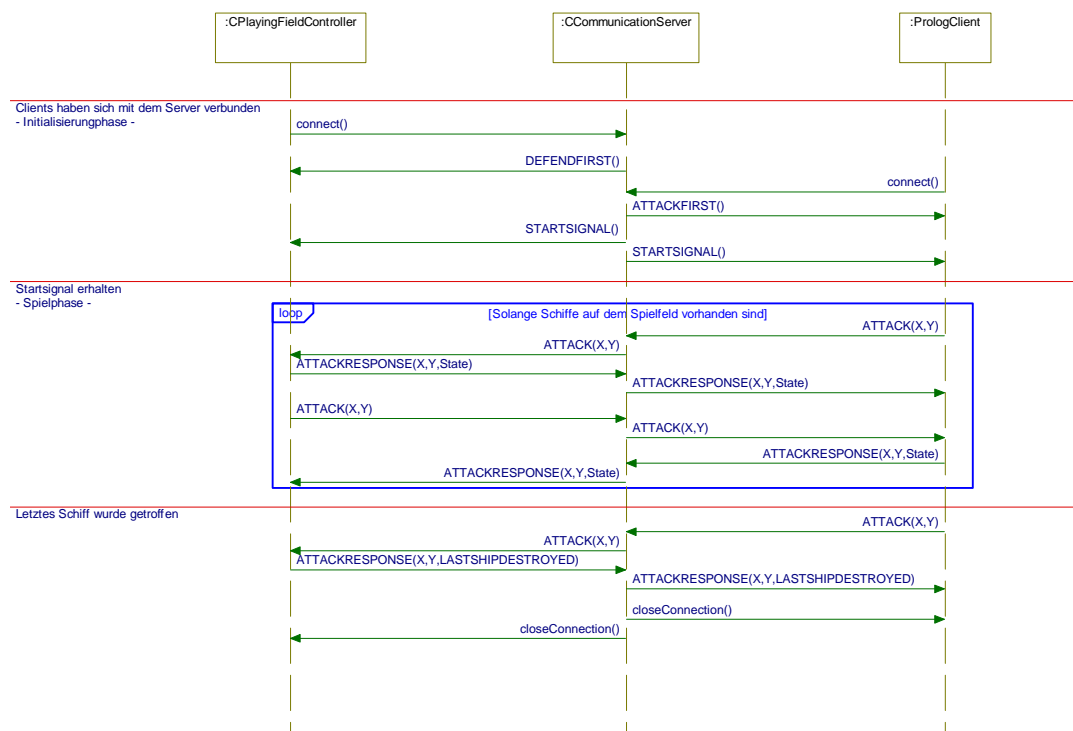


Abbildung 4: Sequenzdiagramm der Kommunikation

## 4.1 Spielserver

Der Server bildet die Schnittstelle zwischen den beiden Kommunikationspartnern, indem er eingehende Nachrichten eines Clients an jeden anderen verbundenen Teilnehmer weiterleitet. Des Weiteren limitiert er die maximale Spielerzahl und generiert das Startsignal, sobald sich zwei Teilnehmer mit ihm verbunden haben. Außerdem legt das Programm 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 eingehender 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 jeden anderen Client weitergeleitet. Für die Überwachung des Eingangsstro-

mes 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.

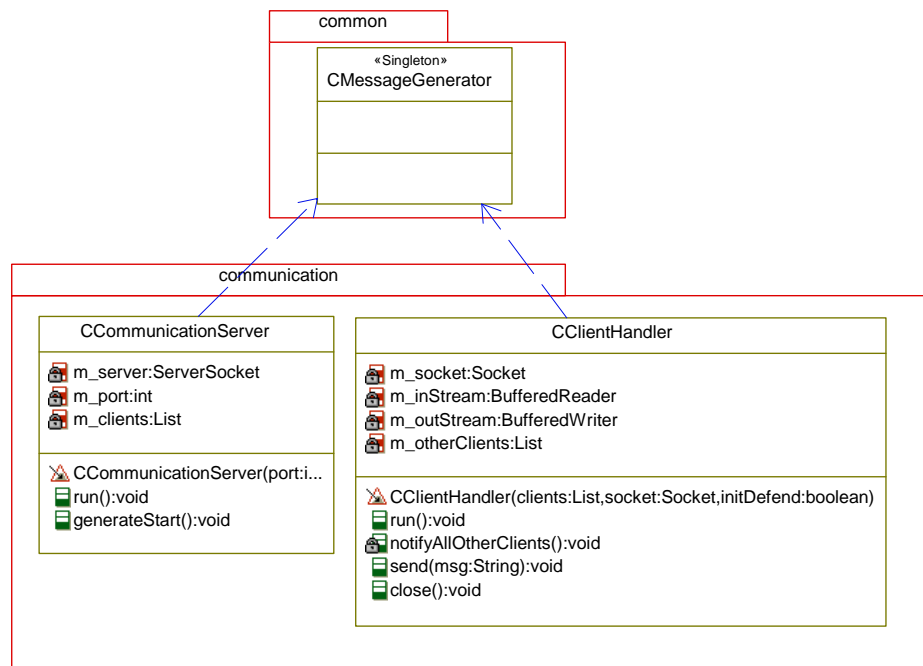


Abbildung 5: Klassendiagramm des Servers

## 4.2 Client - Java

Der hier beschriebene Javaclient 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.

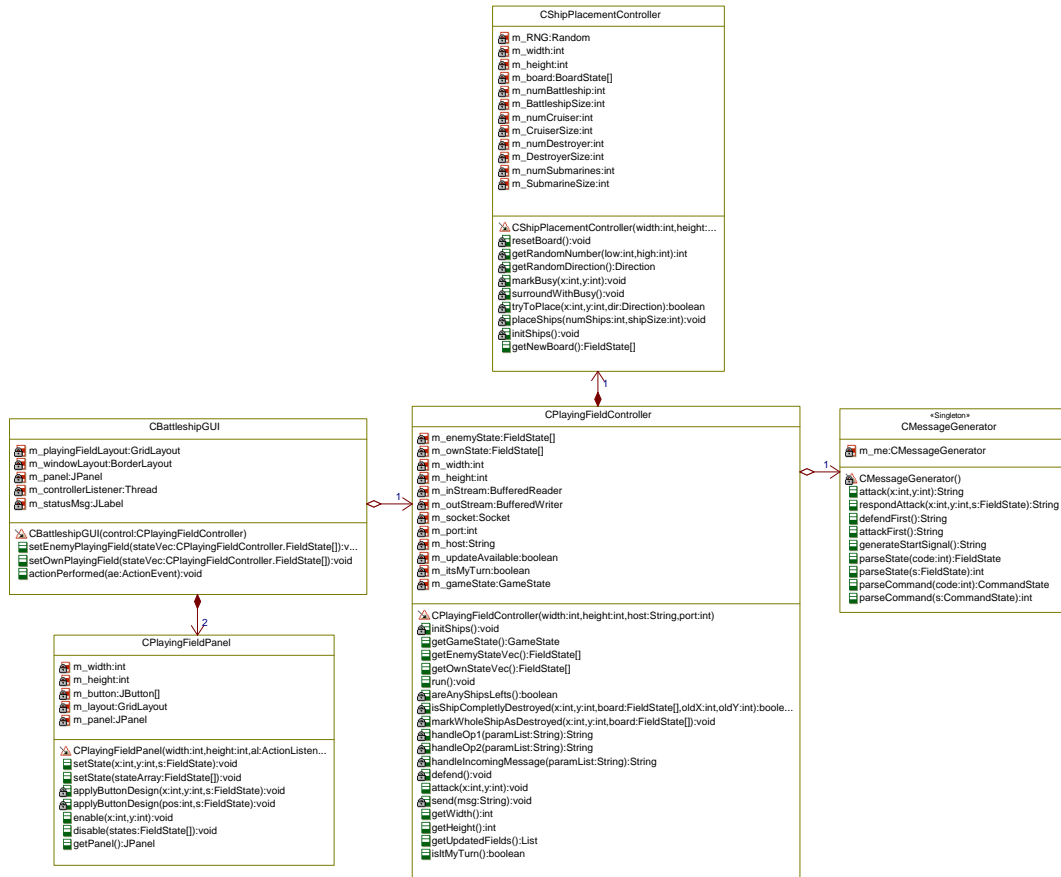


Abbildung 6: Klassendiagramm des Java Clients

#### 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 unbekanntest 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 schwarz gehalten.

	1	2	3	4	5	6	7	8	9	10
A	0,0	1,0	2,0	3,0	4,0	5,0	6,0	7,0	8,0	9,0
B	0,1	1,1	2,1	3,1	4,1	5,1	6,1	7,1	8,1	9,1
C	0,2	1,2	2,2	3,2	4,2	5,2	6,2	7,2	8,2	9,2
D	0,3	~	2,3	3,3	4,3	~	6,3	7,3	8,3	9,3
E	0,4	1,4	2,4	3,4	~	#	~	7,4	8,4	9,4
F	0,5	1,5	2,5	3,5	4,5	#	6,5	7,5	8,5	9,5
G	0,6	1,6	2,6	3,6	4,6	5,6	6,6	7,6	8,6	9,6
H	0,7	1,7	2,7	3,7	4,7	5,7	x	7,7	8,7	9,7
I	0,8	1,8	2,8	3,8	4,8	5,8	x	7,8	8,8	9,8
J	0,9	1,9	2,9	3,9	4,9	5,9	6,9	7,9	8,9	9,9

Abbildung 7: Gegnerisches Spielfeld mit einem zerstörten (schwarz) 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, blau eingefärbt. Die eigenen Schiffe sind ebenfalls sichtbar und mit dunkelgrauer Farbe hervorgehoben. Angriffe des Gegners, die das Wasser getroffen haben, sind hellblau dargestellt. Analog zum gegnerischen Spielfeld sind Treffer rot und versenkte Schiffe schwarz eingefärbt.

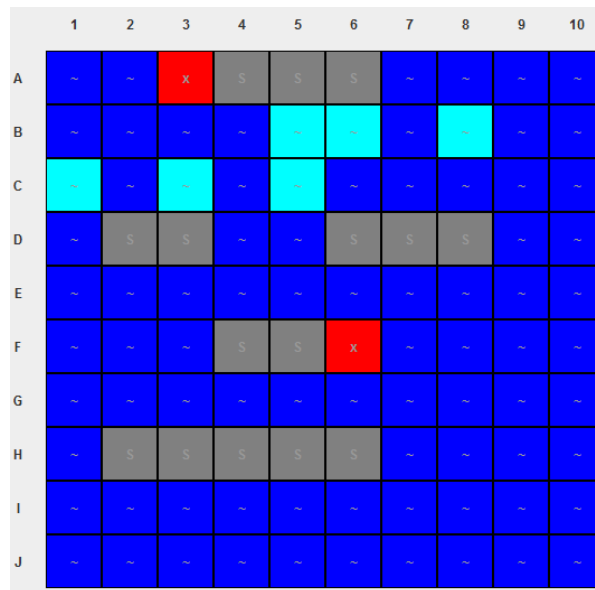


Abbildung 8: Eigenes Spielfeld mit beschädigten Schiffen (rot) und misglückten Angriffen des Gegners (hellblau)

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.

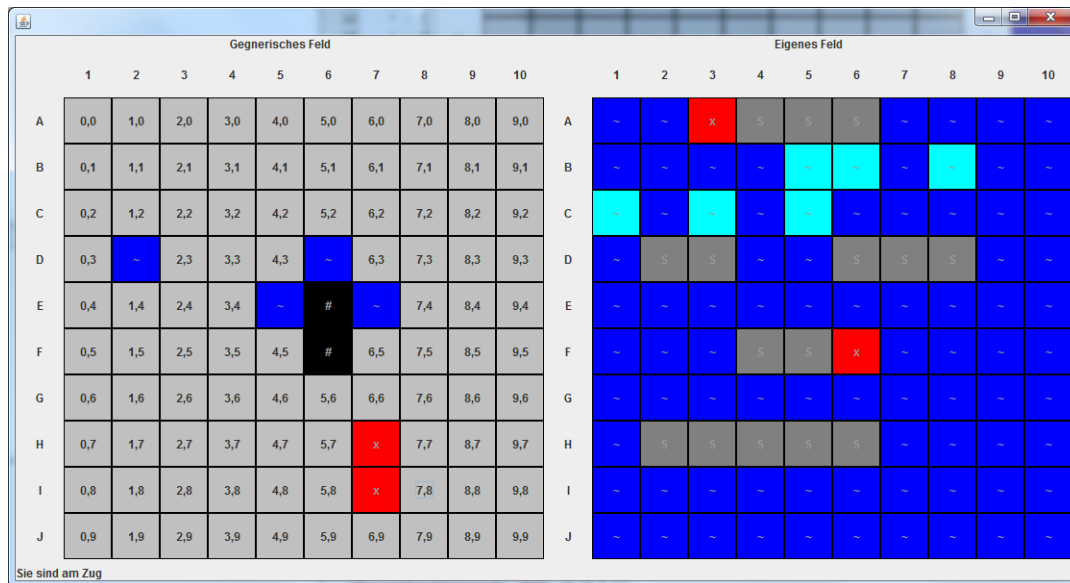


Abbildung 9: Gesamte Benutzeroberfläche

### 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)` und 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 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 für jede Aktion und Reaktion passende Methoden an und liefert den gewünschten zu übertragene Text. Diese Klasse ist ebenfalls für die Dekodierung der Opcodes und Ergebnisse in interne Enumerationen zuständig.

In der Initialisierungsphase 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 8er 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. Im Folgenden werden die Aufgaben und die Funktionsweise dieser Module erläutert.

Gesamtübersicht - Bild - Modulzusammenspiel?

#### 4.3.1 Hauptmodul

Das Hauptmodul `main.pl` wird beim starten des Clients aufgerufen. Das Modul initialisiert 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. Gleichmaßen

überprüft er nach jeder Verteidigung, ob er das Spiel verloren hat. Tritt einer der beiden Fälle in Kraft, so beendet sich der Client mit einer der beiden Ausgaben *KI wins. End of game.* oder *KI loses. End of game..*

#### 4.3.2 Initialisierungsmodul

Die Aufgaben des Initialisierungsmoduls `initModule.pl` sind das initialisieren der Spielfelder, darunter auch das Platzieren der eigenen Schiffe. Hierzu werden die Prädikate `initMyField/0` und `initEnemyField/0` verwendet.

Die verwendeten Spielfelder werden global in den dynamischen Prädikaten `myField/1` und `enemyField/1` gespeichert, um einen einfachen Zugriff von jedem Modul zu ermöglichen.

Das Initialisierungsmodul initialisiert außerdem die ebenfalls global gespeicherte Openlist, die vom Strategiemodul verwendet wird (siehe Abschnitt 4.3.6).

Bogi - dein initships modul einbringen

#### 4.3.3 Modul zur Plazierung von Schiffen

Bogi - TODO

#### 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.

Ne schöne Grafik?

#### 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 Gegner erhaltene Status in das entsprechende Feld eingetragen. Eine Ausnahme besteht für die Antwort *Schiff vollständig versenkt*, in diesem Fall wird das entsprechende Feld lediglich mit dem Status *Treffer* belegt, um die Handhabung des Spielfeldes zu erleichtern.

#### 4.3.6 Strategiemodul

Prädikat	Verwendung
Schnittstellenprädikate	verwendet von

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`).

Ist die Antwort des Gegners *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 auf der Standardausgabe und kann während des Spiels oder zu Debugzwecken verwendet werden.

Bogi - bitte vervollständigen, du wolltest da noch was adden (13.1.11)

Alle - haben wir schon ne Lösung, um den Output komfortabel umzustellen, wenn wir 100x spiele gegen die andere KI machen?

## 5 Evaluation

Testkonzept und Testfälle

Benutzungshinweise für den Endbenutzer

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

## 6 Literatur