Aufgabenstellung des Programmierpraktikums im Wintersemester 2007/08

Vorbemerkung

Willkommen im Programmierpraktikum! Nach Ihrer Zulassung zur Teilnahme erhalten Sie mit diesem Dokument nun die Beschreibung der Aufgabenstellung, mit der Sie sich in den nächsten Monaten auseinandersetzen werden.

Wie schon in den Programmierpraktika der vergangenen Semester, besteht die Aufgabe in der Implementierung eines Strategiespiels – in diesem Fall des traditionellen afrikanischen Spiels **Oware** (dessen Regeln leicht variiert wurden, um die Behandelbarkeit auf dem Rechner zu verbessern). Ihre Aufgabe besteht darin, Oware gemäß dem Pflichtenheft zu implementieren. Dies beinhaltet u.a. die Realisierung einer graphischen Benutzeroberfläche, die Programmierung eines Computergegners auf Grundlage von Techniken des automatischen Problemlösens und die Implementierung einer Netzwerkschnittstelle, mit deren Hilfe Ihr Programm gegen unsere Referenzimplementierung und die Systeme der anderen Praktikumsteilnehmer antreten kann.

Im Vergleich zu den Spielen Kaskade und Hexabang aus den letzten Praktika handelt es sich bei Oware um ein Strategiespiel mit deutlich komplizierteren Regeln. In der Endspielphase kann es zu zyklischen Wiederholungen der Spielsituation kommen, ohne dass das Spiel durch einen regulären Zug beendet werden könnte. In solchen Fällen ist das Spiel abzubrechen. Zur Erkennung zyklischer Spielverläufe muss sich Ihr Programm daher für jeden Spieler die Spielsituationen merken, in denen der Spieler bereits einen Zug gemacht hat, was einen höheren Aufwand für die Implementierung des eigentlichen Spielablaufs bedeutet. Trotz dieser Schwierigkeiten ist Oware ein sehr dankbares Spiel für die Umsetzung auf dem Rechner und Sie werden mit Ihrem Programm eine Spielstärke erreichen, gegen die ein menschlicher Gegner kaum eine Chance hat. Als Ausgleich für die etwas aufwändigere Umsetzung der Spielregeln werden in diesem Praktikum die Anforderungen an die graphische Darstellung des Spielzustands gesenkt. Eine realistisch wirkende Imitation des Spiels wird daher nicht angestrebt. Es reicht völlig aus, wenn Ihr Programm den Spielzustand des Oware-Bretts in schematischer Form darstellt (siehe Kap. 3).

Sie werden für die Implementierung die aktuelle Java Version 6 benutzen. Dabei lernen Sie die Sprache und die Klassenbibliotheken der Standard Edition genauer kennen – so etwa die JFC/Swing-Klassen für die Realisierung der Benutzeroberfläche, die Socket-Klassen für die Realisierung der Netzwerkschnittstelle, die Programmierung mit Threads für die parallele Behandlung mehrerer Clients durch Ihren Zugserver. Sind Ihre Kenntnisse der objektorientierten Programmierung und der Programmiersprache Java bisher mehr theoretischer Natur, so erwerben Sie durch die Realisierung eines größeren Programmierprojekts jetzt auch praktische Programmiererfahrung in Java. Auf freiwilliger Basis können Sie ferner die Darstellung der Programmstruktur in Form von UML-Klassendiagrammen üben und Erfahrungen mit dem automatisierten Test Ihrer Implementierung unter Verwendung von JUnit sammeln.

Sofern Sie nicht eine andere Java-Entwicklungsumgebung gewohnt sind, empfehlen wir Ihnen die Verwendung des Open-Source-Werkzeugs Eclipse (`http://www.eclipse.org/`). Eclipse bietet eine ausgezeichnete Unterstützung für die Entwicklung in Java. Ferner enthält Eclipse bereits eine Umgebung für das Testen Ihrer Klassen mit JUnit und eine komfortable Schnittstelle zur Versionskontrolle mit CVS (damit können Sie jederzeit einen konsistenten früheren Programmzustand wiederherstellen, wenn bei der Überarbeitung der aktuellen Dateien irgendetwas komplett schiefgegangen ist). Sollten Sie zur Benutzung von Eclipse Fragen haben, so bieten wir Ihnen auch hierfür Betreuung an. Sie können auch eine andere Java-Entwicklungsumgebung verwenden, in diesem Fall ist ein Support unsererseits jedoch nicht möglich.

Der Rest dieses Dokuments ist wie folgt gegliedert: Zunächst erhalten Sie die wichtigsten Informationen über den Ablauf des Praktikums und zu den verfügbaren Betreuungsangeboten. Nach einer Erläuterung des Oware-Spiels folgt das Pflichtenheft mit einer Auflistung der Anforderungen, die Sie mit Ihrer Implementierung des Oware-Spiels erfüllen müssen. In gesonderten Abschnitten wird dann der für die automatische Zugberechnung zu verwendende Ansatz und das Kommunikationsprotokoll für die Netzwerkschnittstelle beschrieben. Abschließend geben wir eine Empfehlung für einen sinnvollen Bearbeitungsverlauf. Im Anhang wurden verschiedene Informationen zusammengestellt, die Ihnen beim Verständnis der Aufgabenstellung und bei der Konzeption Ihrer Lösung helfen können. Dies beinhaltet: kommentierte Beispiele für jede Spielregel des Oware-Spiels; kommentierte Beispiele zum Kommunikationsprotokoll für jede mögliche Server-Antwort; Richtlinien zur Gestaltung der Programmdokumentation; Tipps zum Testen Ihres Programms.

Mit diesen Hinweisen können Sie sich direkt an die Arbeit begeben! Bitte fangen Sie frühzeitig mit der Bearbeitung an, insbesondere wenn Sie im Programmieren mit Java noch nicht so routiniert sind. Sollten Sie Fragen zur Aufgabenstellung haben oder bei der Umsetzung auf Schwierigkeiten stoßen, dann nutzen Sie bitte die Betreuungsmöglichkeiten in der Newsgroup oder wenden Sie sich an einen der Ansprechpartner aus dem Betreuungsteam. Wir beantworten gerne Ihre Fragen und werden uns bemühen, Ihnen in allen inhaltlichen oder organisatorischen Belangen weiterzuhelfen.

Ich wünsche Ihnen ein gutes Gelingen bei Ihrem Programmiervorhaben!

Für das ganze Team

— Ingo Glöckner

1 Organisatorisches und Formalia

1.1 Ablauf des Praktikums

- Arbeitsbeginn: 1. Oktober 2007
- Abgabe des fertigen und getesteten Programms: Die Lösungen müssen bis zum 25.1.2008 bei uns eingegangen sein.
- Sichtung der Ergebnisse: ab dem 28.1.2008, Dauer: ca. zwei bis drei Wochen
- Korrekturphase: Sie beginnt, nachdem Sie die vorläufige Bewertung Ihrer Arbeit erhalten haben. Halten Sie sich daher ab Februar noch Zeit frei, denn gegebenenfalls müssen Sie Ihr Programm oder Ihre Programmdokumentation noch nachbessern.
- Abschlusstreffen aller Praktikumsteilnehmer am Samstag, 8.3.2008 oder Sonntag, 9.3.2008 in Hagen. Dort sollen die fertigen Lösungen vorgestellt und das spielstärkste Programm ermittelt werden.

  Hinweise:

  - Das Treffen hat eher informellen Charakter, d.h. Sie brauchen keine Präsentation vorzubereiten, sondern Sie müssen im direkten Gespräch zeigen, dass Sie Ihr Programm genau kennen und bis ins letzte Detail erklären können.
  - Ihre Fachkenntnis müssen Sie auch dadurch beweisen, dass Sie vor Ort noch kleinere Erweiterungen an Ihrem Programm vornehmen können.
  - Ermittlung der Siegerprogramme: Bei den Treffen werden alle Programme der Teilnehmer gegeneinander antreten, um das Programm zu ermitteln, das am stärksten spielt. Zusätzlich wird ein Design-Preis an die aus graphischer Sicht überzeugendste Lösung vergeben.
  - Sie brauchen nur an einer der beiden Abschlussveranstanstaltungen teilzunehmen. Zur Teilnahme werden Sie nur zugelassen, wenn Ihr fertiges Programm und die Programmdokumentation den Anforderungen entspricht. Aus diesem Grund erfolgt die Terminplanung für die Abschlussveranstaltungen erst nach der Sichtung der Oware-Programme. Im Falle absehbarer

Terminprobleme bitten wir jedoch um frühzeitige Rücksprache. In Fragen der Terminplanung wenden Sie sich bitte an Herrn Heinrichmeyer.

1.2 Betreuungsangebote und Ansprechpartner

Ihre Ansprechpartner im Praktikum sind:

- Dr.-Ing. Fritz Heinrichmeyer
- Andrea Frank
- Christoph Doppelbauer
- Dr. Ingo Glöckner

Sie können sich mit persönlichen Fragen an jeden Betreuer wenden, die EMail-Addresse ist wie üblich jeweils `Vorname.Nachname@FernUni-Hagen.DE`. Terminabprachen für die Abschlussveranstaltung sollten jedoch zentral über Herrn Heinrichmeyer erfolgen.
Die Betreuung des Praktikums erfolgt überwiegend über die folgenden zwei Newsgroups.

- In der Betreuungs-Newsgroup `news://feunews.fernuni-hagen.de/feu.informatik.kurs.1580+82+84.betreuung.ws` werden wir Ankündigungen und allgemeine Informationen wie z.B. Änderungen des organisatorischen Ablaufs bekanntgeben. Bitte schauen Sie mindestens einmal pro Woche in diese Newsgroup, um sich über aktuelle Mitteilungen zu informieren.
- Wenn Sie Fragen zum Praktikum haben oder anderen Studierenden eine Hilfestellung geben möchten, so gehen Sie bitte nach `news://feunews.fernuni-hagen.de/feu.informatik.kurs.1580+82+84.diskussion.ws`. In dieser Newsgroup können alle praktikumsbezogenen Probleme diskutiert werden.

Wir werden Fragen, die programmtechnischer Natur sind, nicht per EMail beantworten. Bitte nutzen Sie hierfür die Diskussions-Newsgroup.

1.3 Abgabe der Lösungen

Abgabetermin für Ihr Oware-Programm und die Programmdokumentation ist der 25.1.2008. Die Abgabe erfolgt über das auf `http://sirius.fernuni-hagen.de/propra` bereitgestellte Webformular. Mit diesem können Sie Ihr fertiges Programm und die Programmdokumentation auf unseren Server hochladen. Außerdem ermöglicht das Webformular, die Lösungen der anderen Teilnehmer auszuführen. Ein Upload wird ab Mitte Oktober möglich sein.
Ihre Lösung muss die folgenden Komponenten in den angegebenen Übergabeformaten umfassen:

a. Vorkompiliertes Programm als .jar-Datei. Diese muss sich durch `java -jar IHR_NAME.jar` mit der Java-Version 6 ausführen und testen lassen. Annahmen über den `CLASSPATH` oder das Verzeichnis, aus dem heraus das Programm aufgerufen wird, sind nicht zulässig (siehe auch untenstehende Hinweise zur Erzeugung einer solchen .jar-Datei)

b. Quelltext Ihres Programms samt erstelltem Javadoc als .zip-Datei mit dem Namen `IHR_NAME.zip`. Die Javadoc-Dateien müssen sich in einem gesonderten Verzeichnis `docs` befinden.

c. Programmüberblick als RTF- oder PDF-Datei.

Bitte beachten Sie bei der Erstellung des Programmüberblicks und dem Kommentieren des Quelltexts die in Anhang C beschriebenen Dokumentationsrichtlinien. Unvollständige Lösungen ohne Programmdokumentation werden nicht korrigiert.

Weitere Hinweise:

- Die .jar-Datei mit Ihrer Lösung muss ohne zusätzliche Angabe der Klasse, die die main-Methode enthält, ausführbar sein. Zu diesem Zweck müssen Sie eine sogenannte Manifest-Datei erstellen, in der der Name der Klasse, die die main-Methode enthält, vermerkt ist. Wenn diese Klasse z.B. `Oware` heißt und im Paket `propra.main` liegt, dann lautet der Inhalt der Manifest-Datei wie folgt:
  `Main-Class: propra.main.Oware`
  Davon abgesehen braucht nichts zusätzliches in der Manifest-Datei zu stehen. Die Manifest-Datei muss beim Erstellen der .jar-Datei angegeben werden. Wenn Sie den Standardnamen `manifest.mf` für eine Manifest-Datei verwenden, dann sieht der Aufruf zur Erzeugung eines ausführbaren Archivs wie folgt aus (die Reihenfolge ist zu beachten):
  `jar cmf manifest.mf IHR_NAME.jar *.class`

- Sofern Ihr Programm zusätzlich zu den .class-Dateien noch weitere Ressourcen benötigt (z.B. Icons und andere graphische Elemente), so sollten Sie diese mit in der .jar-Datei ablegen. Sie können dann z.B. die Methode `getResource()` des `ClassLoaders` nutzen, um diese Elemente beim Start Ihres Programms einzulesen.

1.4 Kriterien für den Erhalt des Leistungsnachweises

Für den Erhalt des (unbenoteten) Leistungsnachweises über die erfolgreiche Teilnahme am Programmierpraktikum müssen Sie die folgenden Bedingungen erfüllen:

- Eigenständige Implementierung eines fehlerfreien Programms, das alle im Pflichtenheft genannten funktionalen und nicht-funktionalen Anforderungen erfüllt. Gruppenlösungen sind nicht zulässig.

- Ihr Programm muss die Testfälle durch den auf der Praktikumsseite verfügbaren Test-Client erfolgreich absolvieren (siehe Beschreibung in Anhang D). Sie können dies selbst überprüfen und den Test-Client zum Auffinden von Fehlern in Ihrem Programm nutzen.

- Hochladen Ihrer Lösung einschließlich Programmdokumentation bis zum Abgabetermin 25.1.2008 (siehe Beschreibung der abzugebenden Bestandteile der Lösung und der Abgabemodalitäten im vorigen Abschnitt). Laden Sie keine Lösung hoch, die laut Test-Client noch Fehler besitzt.

- Nach der Abgabe überprüfen wir Ihr Programm zusätzlich mit Variationen der Testfälle des ursprünglichen Test-Clients. Sollten dadurch noch Fehler entdeckt werden, so müssen Sie diese beseitigen.

- Sie müssen bei der abschließenden Präsenzveranstaltung Ihr Programm erläutern und kleinere Änderungen vornehmen können (Einbau eines Zusatzfeatures).

- Es ist Teil Ihrer Verpflichtungen, die Betreuungs-Newsgroup `feu.informatik.kurs.1580+82+84.betreuung.ws` mindestens einmal wöchentlich zu besuchen und dort auf eventuelle Ankündigungen zu achten.

Hinweis zu Manipulationsversuchen und Kopieren von Sourcecode:

> *Manipulationsversuche mit den eingesandten Programmen, z.B. durch Aufspielen von Computerviren oder anderen Programmkomponenten, die nicht der Lösung der Programmieraufgabe dienen, führen – unabhängig von Schadensersatzansprüchen seitens der FernUniversität – zum sofortigen Ausschluss aus dem Praktikum und zur Nichterteilung des Leistungsnachweises.*
>
> *Das Kopieren von Sourcecode aus ähnlichen Projekten aus dem Internet oder aus Lösungen anderer Praktikumsteilnehmer ist unter keinen Umständen gestattet und wird ebenfalls mit einem sofortigen Ausschluss aus dem Praktikum geahndet.*

Abbildung 1: Beispiel für ein Oware-Brett mit Spielsteinen (Quelle: http://de.wikipedia.org/wiki/Bild:Kalaha1.jpg)

2 Das Oware-Spiel

2.1 Vorbemerkung zur gewählten Spielvariante

Vor der Darstellung des Pflichtenhefts soll hier zunächst die gewählte Version des Oware-Spiels erläutert werden. Oware ist ein traditionelles afrikanisches Spiel, von dem unzählige Spielvarianten existieren. Die im folgenden beschriebenen Spielregeln orientieren sich an der *Abapa*-Variante, die bei Wettbewerben verwendet wird. Für die Implementierung auf dem Rechner wurde jedoch ein eindeutiges Abbruchkriterium geschaffen. Ausführliche Beispiele zu den Spielregeln finden Sie in Anhang A.

2.2 Ausstattung des Spiels

Das Oware-Spiel wird auf einem Oware-Brett gespielt, siehe Abbildung 1. Auf dem Brett befinden sich zwei Reihen von jeweils sechs Mulden, die *Häuser* genannt werden. Oware ist ein Zweipersonenspiel; die beiden Spieler sitzen sich gegenüber und kontrollieren jeweils die ihnen zugewandte Reihe als eigenes Territorium. Dabei soll im folgenden stets dem Spieler, der das Spiel beginnt (Spieler A) die untere, 'südliche' Seite des Spielfelds zugeordnet sein und dem zweiten Spieler (Spieler B) die obere bzw. 'nördliche' Seite.[1]

Gespielt wird mit 48 *Steinen* (bzw. Samen, Nüssen o.ä.), die anfangs gleichmäßig auf die Mulden verteilt sind. Im Spielverlauf können die Spieler Steine vom Brett nehmen, die in ihren Besitz übergehen. Zum Aufbewahren dieser Steine gibt es auf manchen Brettern größere Mulden an den Enden des Bretts; in diesem Fall legt jeder Spieler die herausgenommen Steine in das 'Lagerhaus' auf seiner rechten Seite. Auf vielen Brettern wie dem gezeigten fehlen jedoch solche 'Lagerhäuser' und die Steine werden einfach (wie in der Abbildung sichtbar) auf der Seite des Spielers abgelegt, in dessen Besitz sie übergehen.

2.3 Ziel des Spiels

Das Ziel des Oware-Spiels besteht darin, am Ende des Spiels mehr Steine zu besitzen als der Gegner.

[1] Für Spieler wird im folgenden die männliche Form gewählt, was keine Diskriminierung von Oware-Spielerinnen bedeuten soll.

2.4 Spielanfang

Zu Beginn des Spiels befinden sich in jedem Haus vier Steine.

2.5 Spielzüge

Die Spieler machen abwechselnd jeweils einen Zug, wobei jeder Zug in die zwei Phasen *Aussäen* und *Einsammeln* (oder 'Ernten') unterteilt werden kann.

2.5.1 Grundregel für das Aussäen

Der Spieler, der am Zug ist, wählt eines der Häuser auf der eigenen Seite aus, in dem sich noch Steine befinden. Er nimmt alle Steine aus dem gewählten Haus und 'sät' diese im *Gegenuhrzeigersinn* Stein für Stein in die angrenzenden Häuser, also den ersten Stein in das direkt angrenzende Haus, den zweiten in das nächste Haus usw.

Sofern das gewählte Haus mehr als 11 Steine enthält, reichen die Steine beim Aussäen für mehr als eine 'Runde' (eventuell sogar für noch mehr Runden). In diesem Fall ist das Haus, aus dem die Steine genommen wurden, zu überspringen. Nach dem Ende des Säens ist das gewählte Haus also stets leer.

2.5.2 Sonderfall Starvation-Regel

Beim Säen ist die *Starvation Rule* (Verhunger-Regel) zu beachten, durch die jeder Spieler gezwungen wird, den Gegner zugfähig zu halten. Wenn sich im gegnerischen Feld keine Steine mehr befinden, so darf der aktuelle Spieler nur solche Häuser auswählen, die beim Aussäen wieder Steine in die gegnerische Reihe bringen. Nach dem Aussäen müssen sich also immer Steine im Feld des Gegners befinden; andernfalls ist der gewählte Zug nicht zulässig.

2.5.3 Grundregel für das Einsammeln

Auf das Aussäen der Steine folgt das Einsammeln gewonnener Steine. Die Voraussetzung für einen Gewinn ist, dass der letzte Stein beim Aussäen in ein Haus auf der gegnerischen Seite gelegt wurde. Außerdem müssen sich nach dem Ablegen in diesem Haus zwei oder drei Steine befinden. Ist das der Fall, so werden alle Steine in diesem Haus eingesammelt und vom Spieler in Besitz genommen. Dieser Vorgang wird nun rückwärts (d.h. im Uhrzeigersinn) fortgesetzt, d.h. der Spieler darf auch das vorherige Haus leeren, sofern es zwei oder drei Steine enthält usw. Das Einsammeln endet, sobald das betrachtete Haus nicht die geforderten zwei oder drei Steine enthält. Außerdem endet das Einsammeln, wenn die eigene Seite erreicht wird.

2.5.4 Sonderfall Grand-Slam-Regel

Beim Einsammeln ist die sogenannte *Grand-Slam-Regel* zu beachten: Wenn der Gegner nach dem Einsammeln gar keine Steine mehr hätte, dann dürfen gar keine Steine abgeräumt werden. Das Aussäen ist in diesem Fall also ein gültiger Zug, aber es müssen dann alle Steine liegen bleiben.

2.6 Ende des Spiels und Bestimmung des Gewinners

Das Spiel endet, sobald eine der folgenden Situationen eintritt:

(E-1) Ein Spieler hat 25 oder mehr Steine in seinem Besitz. In diesem Fall könnte der Spielausgang durch das weitere Spiel nicht mehr verändert werden.

(E-2) Der Spieler, der am Zug ist, kann keinen erlaubten Zug mehr machen.

(E-3) Ein Zyklus ist aufgetreten, d.h. es kommt zur Wiederholung einer Spielsituation, in der sich der Spieler, der gerade am Zug ist, bereits vorher befand. Bitte beachten Sie, dass es nicht als Wiederholung gewertet wird, wenn der *andere* Spieler in dieser Situation schon einmal am Zug war.

Sofern das Spielende aufgrund eines dieser Kriterien erreicht ist, nimmt jeder Spieler die Steine, die gerade auf seiner Seite des Bretts liegen, zusätzlich in seinen Besitz. Gewinner ist der Spieler, der mindestens 25 Steine hat. Es kann jedoch auch zu einem Gleichstand von 24 zu 24 Steinen kommen. In diesem Fall endet das Spiel mit einem Unentschieden.

3 Pflichtenheft

3.1 Allgemeines

In dem folgenden Pflichtenheft sollen zuerst alle funktionalen Anforderungen (d.h. Anforderungen in Bezug auf den Funktionsumfang der zu entwickelnden Lösung) dargestellt werden. Anschließend wird auf nicht-funktionale Anforderungen wie etwa die verlangte Plattformunabhängigkeit eingegangen. Sofern nicht anders angegeben, handelt es sich um zwingende Anforderungen.

Das gestartete Oware-Programm muss über eine graphische Benutzerschnittstelle zum aktiven Spielen und zum Mitverfolgen von Spielen entfernter Computergegner verfügen. Gleichzeitig muss das Programm eine externe Schnittstelle als 'Zuggeneratorserver' anbieten, durch die es selbst von anderen Oware-Programmen als entfernter Computerspieler genutzt werden kann. Bevor die Anforderungen an die Computerspieler und die Client/Server-Schnittstelle zur Nutzung entfernter Computerspieler spezifiziert werden, sollen im folgenden zunächst der Funktionsumfang und die Grundelemente der graphischen Benutzeroberfläche festgelegt werden.

3.2 Konfiguration eines neuen Spiels

Beim Starten eines neuen Spiels muss der Anwender die Möglichkeit haben, in einem Konfigurationsdialog die Art und ggf. die Spielstärke der Spieler festzulegen. Für Spieler A und Spieler B sind hierbei die folgenden Auswahlmöglichkeiten zu unterstützen, wobei der erstgenannte Spieler A jeweils das Spiel beginnt:

- Mensch (lokal) gegen Mensch (lokal)
- Mensch (lokal) gegen Computerspieler (lokal)
- Mensch (lokal) gegen Computerspieler (entfernt)
- Computerspieler (lokal) gegen Mensch (lokal)
- Computerspieler (lokal) gegen Computerspieler (lokal)
- Computerspieler (lokal) gegen Computerspieler (entfernt)
- Computerspieler (entfernt) gegen Mensch (lokal)
- Computerspieler (entfernt) gegen Computerspieler (lokal)
- Computerspieler (enfernt) gegen Computerspieler (entfernt)

Das Programm muss die folgenden Möglichkeiten zur Konfiguration eines lokalen oder entfernten Computerspielers anbieten:

- Auswahl der Strategie (standardmäßig stehen hier nur Strategie 0 oder Strategie 1 zur Verfügung, wobei die stärkere Strategie 0 voreingestellt ist. Sofern vorhanden, *können* Sie zusätzlich noch weitere Strategien anbieten und deren Namen mit anzeigen).
- Auswahl der Suchtiefe (ganze Zahl zwischen 1 und 9, Voreinstellung 5).

Für entfernte Computerspieler muss das Programm zusätzlich folgende Einstellmöglichkeiten bieten:

- Netzwerkadresse des Rechners, auf dem der entfernte Computerspieler läuft

- Portnummer, unter der der entfernte Computerspieler als Zuggeneratorserver erreichbar ist (hier ist als Voreinstellung 7889 zu wählen; es muss aber möglich sein, diese Voreinstellung im Bereich 0...65535 zu ändern).

3.3 Graphische Darstellung des Spielzustands

Aufgrund der Komplexität der Spielregeln von Oware werden die Mindestanforderungen an die graphische Darstellung bewusst einfach gehalten. Insbesondere wird nicht verlangt, dass Sie ein weitgehend realistisches Oware-Brett darstellen. Die Anzeige des Spielzustands muss aber zumindest den folgenden Anforderungen genügen:

- In der graphischen Darstellung muss das Spielbrett mit zwei horizontalen Reihen aus jeweils sechs Mulden (Häusern) zu erkennen sein.
- Die Mindestanforderung für die Darstellung der Häuser ist eine flächige Darstellung etwa durch eine Kreisscheibe oder ein Rechteck.
- Die Mindestanforderung für die Darstellung der in den Häusern befindlichen Steine ist die Darstellung der Anzahl der Steine (statt der Null kann auch gar nichts angezeigt werden), wobei die Größe des Fonts für die Darstellung der Zahl der Steine zur Größe der Häuser/Mulden passen sollte.
- Auf Wunsch *dürfen* Sie auch die in den Mulden befindlichen Steine selbst zeigen und nicht nur die Anzahl. Auch in diesem Fall muss das Programm jedoch zusätzlich die Zahl der Steine anzeigen (entweder direkt oder als Tooltip für die Häuser).
- Das Programm muss anzeigen, wie viele Steine jeder der Spieler bereits in seinem Besitz hat.
- Aus der Anzeige des Spiels muss hervorgehen, welcher Spieler gerade an der Reihe ist.
- Normalerweise soll der untere Bereich des Bretts zu Spieler A (erster Spieler, beginnt das Spiel) und der obere Bereich zu Spieler B (zweiter Spieler) gehören.
- Für menschliche Spieler ist es evtl. bequemer, auch dann vom unteren Bereich aus zu spielen, wenn der Gegner das Spiel beginnt. Sie *können* die Möglichkeit vorsehen, in diesem Fall die Perspektive zu wechseln und das Brett zu drehen.

3.4 Eingabe von Zügen durch den Benutzer

Eine Eingabemöglichkeit für Züge durch den menschlichen Spieler darf nur bestehen, wenn ein menschlicher Spieler 'an der Reihe ist' und das Spielende noch nicht erreicht ist. In diesem Fall sollen die Mulden/Häuser auf die Mausaktionen des Benutzers wie folgt reagieren:

- **Überstreichen mit der Maus**: Diejenigen Häuser des aktiven Spielers, die einen möglichen Zug darstellen, sollen Ihr Aussehen (z.B. Farbe) beim Überstreichen mit der Maus verändern, um dadurch als gültige Wahlmöglichkeit erkennbar werden. Mit 'Überstreichen' ist hierbei gemeint, dass sich der Mauszeiger im Inneren der Mulde befindet, jedoch noch nicht gedrückt wird. Häuser, die nicht einen gültigen Zug darstellen (Häuser in der gegnerischen Reihe oder solche, die leer sind oder deren Auswahl die Starvation Rule verletzen würde) dürfen nicht auf das Überstreichen mit der Maus reagieren.
- **Anklicken mit der linken Maustaste**: Zur Eingabe eines Zuges klickt der Benutzer das Haus an, aus dem gesät werden soll. (a) Sofern der Benutzer einen gültigen Zug gewählt hat, soll eine optische Rückmeldung erfolgen und der Zug als Abfolge von Einzelschritten ausgeführt werden (siehe Abschnitt 3.8). (b) Es soll eine andere (bzw. gar keine) Rückmeldung auf den Mausklick des Benutzers erfolgen, wenn der entsprechende Zug nicht möglich ist (z.B. Anklicken eines leeren Hauses oder Verletzung der Starvation-Regel). Wenn Sie möchten, *können* Sie in diesem Fall auch den Grund ausgeben, warum der Zug nicht möglich ist.

Hinweise:

- Um wie gefordert zwischen gültigen und ungültigen Zugmöglichkeiten unterscheiden zu können, muss Ihr Programm die noch möglichen Züge vorausberechnen. Außerdem muss das Programm vorab überprüft haben, ob das Spielende bereits erreicht ist.
- Insgesamt muss durch die Eingabelogik sichergestellt werden, dass ein menschlicher Spieler nur tatsächlich mögliche Züge eingeben kann.
- Das dargestellte Spielbrett darf nur innerhalb der dargestellten Häuser bzw. Mulden auf den Mauszeiger reagieren (wenn diese durch Kreisscheiben dargestellt sind, also innerhalb der Kreisscheiben).

3.5 Anfordern eines Zugvorschlags durch den Benutzer

Die Benutzeroberfläche muss ein Bedienelement enthalten, mit dem ein menschlicher Spieler einen Zugvorschlag abrufen kann. Dieses sollte nur aktiv sein (oder gezeigt werden), wenn der Spieler tatsächlich am Zug ist und das Spiel noch nicht beendet ist. Die vorgeschlagene Zugmöglichkeit soll dem Spieler graphisch (z.B. durch Blinken des betreffenden Hauses) angezeigt werden. Der Benutzer kann dem Vorschlag folgen, sich aber auch anders entscheiden und einen anderen Zug wählen.

Zur Erzeugung des Zugvorschlags dient ein lokaler Computerspieler. Dessen Parameter (Strategie und Vorausschautiefe) werden als Teil der Systemeinstellungen festgelegt (siehe Abschnitt 3.12). Die Standard-Einstellungen für die Erzeugung des Zugvorschlags sollen mit der Voreinstellung für den normalen lokalen Computerspieler übereinstimmen.

3.6 Zugberechnung durch lokalen Computerspieler

Damit z.B. der menschliche Benutzer gegen einen Computergegner spielen kann, muss Ihr Programm lokale Computerspieler bereitstellen. Ein solcher lokaler Computerspieler muss ausgehend von der aktuellen Spielsituation stets einen gültigen Zug auswählen. Die Implementierung muss mit Methoden des automatischen Problemlösens bzw. konkret durch das Alpha-Beta-Verfahren erfolgen; orientieren Sie sich dabei an der Beschreibung in Abschnitt 4.

Die Berechnung des nächsten Zugs wird hierbei durch die Vorgabe der maximalen Vorausschautiefe bzw. **Suchtiefe** gesteuert. Bei der Bestimmung des nächsten Zugs darf der Computerspieler die voreingestellte maximale Vorausschautiefe nicht überschreiten.

Ihr Programm muss für den lokalen Computerspieler mindestens zwei **Spielstrategien** mit unterschiedlichem Spielverhalten anbieten, die im wesentlichen von der gewählten Vorschrift zur Stellungsbewertung abhängen werden. Auch hierzu finden Sie Hinweise in Abschnitt 4. Für eine Ihrer Spielstrategien können Sie den dort beschriebenen Grundansatz einer einfachen Stellungsbewertung verwenden, für die stärkere Strategie müssen Sie selbst kreativ werden. Geben Sie jeder Strategie einen Namen (dieser wird für das Oware-Kommunikationsprotokoll benötigt). Ihre stärkere Strategie sollte die Voreinstellung für den Client sein.

3.7 Zugberechnung durch entfernten Computerspieler

Das Programm muss auch ein Spiel gegen einen entfernten Computerspieler ermöglichen. Dessen Adresse, Port, Suchtiefe und Strategie werden bei der Konfiguration eines neuen Spiels eingestellt. Der Zugriff auf den entfernten Computerspieler wird durch socket-basierte Client-Server-Kommunikation realisiert. Dabei kommt das in Abschnitt 5 spezifizierte Oware-Kommunikationsprotokoll zum Einsatz.

Zum Zugriff auf entfernte Computerspieler muss Ihr Programm die Client-Seite dieses Protokolls implementieren. Der Ablauf aus Client-Sicht sieht wie folgt aus:

- Kontaktaufnahme mit dem Server unter der für den entfernten Spieler angegebenen IP-Adresse und Port. Sofern der Verbindungsaufbau scheitert, Abbruch mit Fehlermeldung.
- Senden des bisherigen Spielverlaufs und der Einstellungen für den entfernten Spieler in Form einer `MoveRequestMsg` (siehe Protokoll).

- Einlesen der `ServerResponseMsg` des Servers und Schließen der Verbindung zum Server. Scheitert das Einlesen, so ist ein Fehler auszugeben und das Spiel abzubrechen.
- Falls es sich bei der Nachricht des Servers um eine Fehlermeldung handelt, so ist der vom Server gemeldete Fehler anzuzeigen und das Spiel ebenfalls abzubrechen.
- Korrektheitsüberprüfung des vom Server gewählten Zugs. Sofern der Server einen falschen Zug gewählt hat, Abbruch mit Fehlermeldung. Andernfalls ist der gewählte Zug auszuführen.

Bitte beachten Sie, dass Ihr Programm für jede Zuganfrage an einen entfernten Computerspieler eine neue Verbindung herstellen muss. Die aktuelle Verbindung besteht immer nur für eine Zuganfrage und ist vom Client sofort nach Erhalt der Serverantwort zu schließen.

3.8 Graphische Anzeige des gewählten Zugs als Abfolge von Einzelschritten

Das Programm darf mit der für den Benutzer sichtbaren Ausführung eines Zugs nur beginnen, wenn gewährleistet ist, dass es sich hierbei um einen gültigen Zug handelt. Die sichtbare Umsetzung des Zugs durch Ihr Programm muss dann der Beschreibung in den Spielregeln entsprechen (siehe Abschnitt 2). Insbesondere muss das Programm einen Zug als Abfolge einzelner Schritte des Aussäens und Einsammelns anzeigen können. In der Anzeige des Spielbretts soll man nacheinander sehen:

- Welches Haus der aktuelle Spieler zum Säen ausgewählt hat (ein Schritt)
- Wie die Steine entnommen werden (ein Schritt, in dem die Steinanzahl auf Null gesetzt bzw. das Haus geleert wird)
- Wie die Steine Haus für Haus im Gegenuhrzeigersinn ausgelegt werden (jeweils ein Schritt pro Haus)
- Wie beim Einsammeln die Häuser im Uhrzeigersinn geleert werden und der Spielstand des aktiven Spielers dabei erhöht wird (jeweils ein Schritt pro Haus)

Die Pause zwischen den Schritten muss als Bestandteil der Systemkonfiguration einstellbar sein (siehe Abschnitt 3.12). Bei einer Pause von 0 Sekunden soll die Funktion abgeschaltet werden, d.h. in diesem Fall ist unmittelbar das Ergebnis des Zugs anzuzeigen.

3.9 Verzögerte Zugausführung für die Computerspieler

Bei geringen Suchtiefen berechnen die Computerspieler ihre Züge so schnell, dass ein Betrachter den angezeigten Spielverlauf gar nicht mehr nachvollziehen könnte. In Ihrem Programm muss daher eingestellt werden können, wie lange ein erreichter Spielzustand mindestens zu zeigen ist, bevor der nächste Zug ausgeführt wird (siehe Konfiguration der Systemeinstellungen in Abschnitt 3.12). Beträgt diese Zeitkonstante z.B. 2 Sekunden und hat der Computerspieler den nächsten Zug schon nach 0,5 Sekunden berechnet, so muss das Programm noch 1,5 Sekunden warten, bis es den Zug des Computerspielers tatsächlich ausführt. Diese Verzögerung betrifft nur Züge durch Computerspieler; wählt ein menschlicher Spieler einen Zug, so ist dieser immer sofort auszuführen.

3.10 Erkennung des Spielendes und Darstellung des Spielergebnisses

Nach der Ausführung jedes Zugs muss das Programm testen, ob das Spielende bereits erreicht ist. Wenn eines der drei Endekriterien zutrifft (siehe Darstellung der Spielregeln in Abschnitt 2), so ist das Spiel abzubrechen. Die Spieler erhalten dann die jeweils auf ihrer Brettseite noch befindlichen Steine. Der dadurch erreichte Spielstand und der Gewinner des Spiels (Spieler A, Spieler B, Gleichstand) ist anzuzeigen.

3.11 Nutzung als entfernter Computerspieler (Zuggeneratorserver)

Ihr Oware-Programm muss nicht nur die Bedienung durch einen lokalen Benutzer zulassen, sondern gleichzeitig für andere Oware-Programme als entfernter Computerspieler nutzbar sein. Zur Realisierung dieser Funktionalität muss Ihr Programm die Server-Seite des in Abschnitt 5 beschriebenen Oware-Kommunikationsprotokolls implementieren. Der Ablauf sieht aus Server-Sicht wie folgt aus:

- Der Server muss beim Aufruf des Programms automatisch gestartet werden. Unter Nutzung eines `ServerSockets` wartet er dann auf einem Port auf Zuganfragen von Clients.
- Bei Verbindungsaufbau durch einen Client liest der Server dessen Zuganfrage ein. Diese enthält die vollständige Information über den bisherigen Spielverlauf. Der Server muss sich also nicht selbst den bisherigen Verlauf des Spiels mit diesem Client merken. In der Anfrage des Clients wird ferner auch die vom Server zu verwendende Spielstrategie und Suchtiefe festgelegt.
- Der Server prüft zunächst die Zuganfrage des Clients auf Korrektheit. Ausgehend von dem in der Zugnachricht beschriebenen Spielverlauf rekonstruiert der Server dann die aktuelle Spielsituation und berechnet den nächsten Zug für den aktuellen Spieler. Die eigentliche Zuggenerierung mit der angegebenen Spielstrategie und Suchtiefe muss wie bei einem lokalen Computerspieler mit denselben Einstellungen erfolgen.
- Der Server sendet eine `ServerResponseMsg` mit der Angabe zum gewählten Spielzug (oder mit einer Angabe zum Fehler in der Zuganfrage) zurück an den Client.
- Der Server schließt die Verbindung zum Client sofort nach dem Versand der `ServerResponseMsg`.

Hinweise:

- Die Voreinstellung für den Port des Servers ist 7889; falls belegt, soll das Programm beim Start irgendeinen freien Port wählen und die Portnummer in einer Meldung ausgeben. Der tatsächlich gewählte Port muss auch in den Systemeinstellungen sichtbar sein und zur Laufzeit des Programms geändert werden können (siehe Abschnitt 3.12).
- Die Anzahl der gleichzeitig mit einem Server verbundenen Clients ist nicht begrenzt. Beim Aufbau einer Verbindung durch einen Client muss daher ein neuer Thread gestartet werden, der die Kommunikation mit dem Client abwickelt und die Zugberechnung durchführt.
- Die Spielstrategien des Servers werden vom Client durch fortlaufende Nummern (beginnend mit 0) angesprochen. Die Strategie mit dem Index 0 sollte Ihre stärkste Strategie sein.

3.12 Konfiguration der Systemeinstellungen

Die folgende Liste fasst noch einmal zusammen, welche Einstellungsmöglichkeiten das Oware-Programm neben der Konfiguration neuer Spiele noch bieten muss:

- Parameter für die Berechnung von Zugvorschlägen nach Abschnitt 3.5 (Wahlmöglichkeiten für Strategie und Suchtiefe wie bei der Konfiguration des lokalen Computerspielers)
- Anzeigedauer für die einzelnen Schritte bei der Ausführung von Spielzügen als Folge von Einzelschritten nach Abschnitt 3.8 (ein Wert von 0 soll die Funktion deaktivieren und jeden Zug ohne sichtbare Zwischenschritte ausführen) – Wählen Sie hier eine sinnvolle Abstufung (nicht nur ganze Sekunden!) und Voreinstellung.
- Mindestanzeigedauer für Spielzustände nach Abschnitt 3.9: Hier wird angegeben, wie lange das Ergebnis eines Zugs mindestens zu zeigen ist, bevor mit der graphischen Anzeige eines neuen computerberechneten Zugs begonnen wird. Wählen Sie die Voreinstellung so, dass ein Spiel zweier Computerspieler angenehm zu verfolgen ist. Es soll möglich sein, auch 0 Sekunden (keine Verzögerung) auszuwählen.

- Einstellung des Ports, unter dem das Oware-Programm nach Abschnitt 3.11 als Zuggeneratorserver erreichbar ist – der Port ist hierbei eine ganze Zahl zwischen 0 und 65535, Voreinstellung 7889. Hier soll der vom Server aktuell verwendete Port angezeigt werden und ggf. geändert werden können.

3.13 Nichtfunktionale Anforderungen

3.13.1 Softwaretechnische Rahmenbedingungen

Bitte vergewissen Sie sich, dass Ihr Programm den folgenden Rahmenbedingungen entspricht und in einer entsprechenden Softwareumgebung lauffähig ist:

- Bitte benutzen Sie ausschließlich Klassen aus der Standard-Bibliothek der aktuellen Java-Version JDK 6 (Standard Edition). Diese können Sie ggf. von `http://java.sun.com/` herunterladen.
- Sie dürfen nur eigene Klassen und solche aus der Standard-Bibliothek verwenden. Die Nutzung anderer Klassenbibliotheken ist auch dann untersagt, wenn Sie diese in die .jar-Datei mit Ihrer Lösung aufnehmen.
- Die graphische Benutzeroberfläche ist mit JFC/Swing zu realisieren. Die Realisierung mit AWT, SWT/JFace oder mit GUI-Builder-Werkzeugen, die eigene Klassenbibliotheken erfordern, ist nicht gestattet. Zusätzliche Bibliotheken wie OpenGL für Java und ähnliches dürfen ebenfalls nicht genutzt werden.
- Ihr Programm darf keine betriebssystemspezifischen Annahmen machen (z.B. Makefiles, .bat-Dateien, Shell-Skripte). Es muss plattformunabhängig realisiert sein und in Form einer ausführbaren .jar-Datei vorliegen, die ggf. alle erforderlichen Resourcen (wie zu verwendende Icons) enthält und unabhängig vom Betriebssystem auf diese Resourcen zugreift (siehe auch Hinweise zur Abgabe in Abschnitt 1.3). Unsere Testumgebung besteht aus heterogenen Arbeitsplätzen mit den unterschiedlichsten Betriebssystemen und Ihr Programm muss auf jedem Rechner mit einer Standard-Installation von Java Standard Edition 6 funktionieren.

3.13.2 Spielstärke der Computerstrategie

Ihr Programm muss bei einer Suchtiefe von 5 einen für einen menschlichen Spieler schwer zu schlagenden Computergegner darstellen. Als formale Mindestanforderung für die Spielstärke Ihres Programms wird verlangt, dass es als Spieler A mit den Einstellungen Suchtiefe 5 und Strategie 0 gegen einen von uns bereitgestellten Referenzserver als Spieler B mit Suchtiefe 5 und Strategie 0 gewinnt. Dieser Referenzserver verwendet die in Abschnitt 4.3 beschriebene einfache Stellungsbewertung.

3.13.3 Antwortzeiten

Zur Berechnung eines Spielzugs darf Ihr Programm mit der Strategie 0 und Suchtiefe 6 auf einem durchschnittlichen PC (mit 1 GHz CPU-Takt) nicht länger als 30 Sekunden benötigen.

4 Realisierung der automatischen Zugberechnung

4.1 Grundansatz des automatischen Problemlösens

Zur Implementierung der Suchstrategie benötigen Sie Verfahren des automatischen Problemlösens. Die möglichen Spielverläufe werden hier durch einen Spielbaum dargestellt. Die Knoten des Spielbaums sind Spielsituationen und die Kanten entsprechen den zulässigen Zügen. Die Wurzel des Spielbaums bildet dabei die aktuelle Spielsituation, von der aus der nächste Zug des Computerspielers berechnet werden soll. Die Kinder jedes Knotens sind die Spielsituationen, die durch legale Züge aus der jeweiligen Situation entstehen können. Die Blätter des Baums sind die Situationen, in denen das Spielende erreicht ist.

Bei der Darstellung der Knoten (= Stellungen, Spielzustände) ist im Fall von Oware der Aspekt der Zyklenerkennung zu beachten: Nach Regel (E-3) kommt es zum Spielabbruch, wenn sich eine Spielsituation wiederholt, in der der gerade aktive Spieler bereits vorher am Zug war. Um solche Wiederholungen erkennen zu können, müssen die Knoten des Spielbaums hier nicht nur Informationen über die Stellung der Spielsteine auf dem Brett und den Spielstand enthalten, sondern auch (jeweils getrennt nach den Spielern) die Information darüber, in welchen Spielsituationen sich der jeweilige Spieler bereits vorher befunden hat.

Bei einfachen Spielen ist es möglich, den gesamten Spielbaum zu entwickeln; bei einem komplexen Spiel wie Oware jedoch nicht mehr.[2] Ihr Programm soll daher ausgehend von der aktuellen Spielsituation den Spielbaum nur bis zu einer gegebenen Vorausschautiefe oder Suchtiefe d entwickeln. Das Entwickeln des Spielbaums endet also bei den Knoten des Spielbaums, deren Entfernung von der Wurzel des Spielbaums d Züge beträgt. Damit ist klar, warum die minimale Suchtiefe, die in Ihrem Programm einstellbar sein soll, bei $d = 1$ liegt. In diesem Fall werden nur die mit dem nächsten Zug unmittelbar erreichbaren Spielstellungen betrachtet.

4.2 Das Konzept der Stellungsbewertung

Weil die Entwicklung des Suchbaums bei einer bestimmten Suchtiefe abbricht und in der Regel nicht die Blätter (= Endsituationen) erreicht, ist eine direkte Bewertung der Knoten nach Gewinn- bzw. Verliersituationen nicht möglich. Stattdessen wird eine numerische **Stellungsbewertung** für jeden Knoten auf Tiefe d berechnet. Die Stellungsbewertung soll positiv sein, wenn der aktive Spieler im Vorteil ist (je höher, desto günstiger ist die Situation für den aktiven Spieler), und negativ, wenn der Spieler, der gerade nicht am Zug ist, besser steht (je niedriger, desto günstiger ist die Situation für den Gegner des aktiven Spielers). Bei einer ausgeglichenen Spielsituation ergibt sich demnach die Bewertung 0. Für den Einsatz des im folgenden angenommenen Alpha-Beta-Verfahrens muss die Bewertung b_G des Gegners sogar eindeutig aus der Bewertung b_Z des am Zug befindlichen Spielers hervorgehen, d.h. es muss gelten $b_G = -b_Z$.

4.3 Beispiel einer einfachen Stellungsbewertung für Oware

Im folgenden soll ein Beispiel für eine Funktion zur Stellungsbewertung in Oware erläutert werden. Besitzt der am Zug befindliche bzw. 'maximierende' Spieler Z in der aktuellen Spielsituation z Steine, so fehlen ihm noch $f_Z = \max(25-z, 0)$ Steine bis zum Gewinn des Spiels. Analog muss der Gegner zum Gewinnen noch $f_G = \max(25 - g, 0)$ Steine einsammeln, wobei g der Spielstand von Spieler G ist. Sofern in der aktuellen Spielsituation das Spielende eintritt, sollen sich diese Werte auf den Punktestand der Spieler nach Einsammeln aller Steine beziehen. Betrachten wir nun den Anteil $f_Z/(f_Z + f_G)$ der von Spieler Z noch benötigten Steine im Vergleich zur Gesamtzahl noch benötigter Steine. Ein Wert von 0 ist für Spieler Z offenbar am günstigsten (dann hat Spieler Z bereits gewonnen) und ein Wert von 1 am ungünstigsten (dann hat der Gegner G bereits gewonnen). Durch

$$b_Z = 1 - 2f_Z/(f_Z + f_G)$$

erhalten wir eine Funktion zur Positionsbewertung mit den gewünschten Eigenschaften: Der maximale Wert $b_Z = 1$ wird erreicht, wenn Spieler Z gewinnt und der geringste Wert $b_Z = -1$ wird erreicht, wenn Spieler Z verliert (und damit Gegner G gewinnt). Bei Gleichstand ergibt sich $f_Z = 0$ und für die Stellungsbewertung des Gegners gilt $b_G = -b_Z$.

4.4 Tipps für eine stärkere Stellungsbewertung

Sie können die obige Definition als Basisbewertung in der schwächeren Ihrer beiden Spielstrategien verwenden. Es handelt sich deshalb um keine besonders gute Lösung, weil die eigentliche Stellung (d.h. die Position der Steine in den Mulden) gar nicht in die Bewertung eingeht, sondern nur die Spielstände der beiden Spieler. Für eine stärkere Positionsbewertung in der zweiten zu implementierenden Strategie müssen

[2] Tatsächlich konnte Oware auf einem Cluster aus 72 PCs bereits vollständig durchgerechnet werden, siehe http://www.wikimanqala.org/wiki/Awari_Oracle. Die entsprechende Rechenpower haben Sie im Praktikum jedoch nicht und die Nutzung vorberechneter Stellungsbewertungen ist explizit untersagt.

Sie selbst kreativ werden. Hierzu sollten Sie zunächst selbst ein Gefühl für das Spiel bekommen, indem Sie es eine Zeit lang spielen und typische Spielsituationen kennenlernen. Sobald Sie bewerten können, was gute und was schlechte Spielsituationen sind, können Sie dann beginnen, Ihre Erfahrungen hinsichtlich 'günstiger' Spielkonstellationen durch eine exakte Bewertungsvorschrift auszudrücken. Achten Sie aber darauf, dass diese die geforderten Eigenschaften besitzt, d.h. maximaler Wert bei Gewinn des betrachteten Spielers und $b_G = -b_Z$.

Hier einige Anregungen, welche Faktoren (außer dem erreichten Spielstand) für die Bewertung der Spielsituation aus Sicht des aktiven Spielers noch eine Rolle spielen könnten (es gibt sicher noch viele weitere):

- Wie viele Zugmöglichkeiten hat der aktive Spieler? Wie viele der Gegner?
- Wie viele Häuser auf der eigenen Seite sind insofern 'bedroht', dass ein gegnerischer Zug bei ihnen enden würde? Aus wie vielen dieser Häuser könnte der Gegner durch einen direkten Zug Steine abräumen? Wie viele Steine? Wie steht es umgekehrt beim Gegner?
- Wie viele Züge kann der aktive Spieler theoretisch machen, ohne einen Stein ins Territorium des Gegners zu säen?

  Dieser Faktor ist als Moves in Hand (MIH) bekannt und hat große Bedeutung für die Spielstärke, denn solange ein Spieler keine Steine ins gegnerische Territorium bringt, kontrolliert er das Spiel. Bei der Berechnung des MIH werden die möglichen Züge des Gegners ignoriert: Man wählt das Haus mit der höchsten Position p, das mindestens einen und maximal $5 - p$ Steine enthält. Gibt es kein solches Haus, so ist MIH=0. Gibt es dagegen ein solches Haus, so ist der MIH rekursiv definiert als der um 1 erhöhte Wert des MIH für die Spielkonfiguration, die sich durch Aussäen von Position p ergibt. Neben dem MIH für den aktiven Spieler ist auch der MIH des Gegners bedeutsam, der möglichst gering ausfallen sollte.

Bitte beachten Sie, dass die Methode zur Stellungsbewertung **nur die zu bewertende Spielsituation** analysieren darf. Der Methode ist es ausdrücklich verboten, den Spielbaum weiter zu entwickeln, d.h. die vorgegebene Suchtiefe muss mit der tatsächlichen Vorausschautiefe Ihres Programm übereinstimmen und darf nicht durch die Definition Ihrer Stellungsbewertung manipuliert werden. Eine Untersuchung gültiger Züge (wie bei der Bestimmung bedrohter Positionen oder der Bestimmung des MIH) ist nur erlaubt, wenn sie aus der Sicht jeweils eines Spielers erfolgt und mögliche Gegenzüge (im Sinne einer weiteren Entwicklung des Suchbaums) nicht berücksichtigt.

Indem Sie mehrere Spielstrategien mit unterschiedlichen Stellungsbewertungen in Ihrem Programm einbauen, können Sie im Spiel zweier lokaler Computerspieler mit unterschiedlichen Strategien leicht herausfinden, welche Stellungsbewertung tatsächlich die beste ist. Die so gefundene stärkste Strategie sollten Sie zur Standardeinstellung Ihres Programms machen (bzw. Strategie 0 beim entfernten Zugriff auf Ihr Programm als Zugserver).

4.5 Ermittlung des nächsten Zugs durch das Alpha-Beta-Verfahren

Ausgehend von Ihrer statischen Positionsbewertung b_Z bezieht das Alpha-Beta-Verfahren alle möglichen Spielverläufe bis zur gegebenen Vorausschautiefe d in die Bewertung eines Spielzustands mit ein. Dadurch kann es den unter diesen Bedingungen optimalen Zug für den betrachteten Spieler ermitteln.

Das Alpha-Beta-Verfahren benötigt zusätzlich zur Ausgangssituation und Suchtiefe auch noch eine Angabe darüber, ob der jeweils betrachtete Spieler 'maximierend' oder 'minimierend' ist. Der Spieler Z, für den der neue Zug berechnet werden soll, ist dabei maximierend, denn auf ihn bezieht sich die Stellungsbewertung und hohe Werte von b_Z sind für ihn am besten. Der andere Spieler dagegen ist 'minimierend' und wird versuchen, einen Zug zu einer Spielsituation mit der niedrigsten Bewertung für Z zu spielen.

Im folgenden soll davon ausgegangen werden, dass aus dem aktuellen Spielzustand jeweils auch der aktive Spieler hervorgeht, so dass kein weiterer Parameter für den Spieler erforderlich ist. Ferner muss die Darstellung des Spielzustands auch die zur Endeerkennung benötigten Informationen enthalten. Das Alpha-Beta-Verfahren kann dann schematisch wie folgt beschrieben werden:

```
alphabeta(spielzustand, α, β, maximierend, resttiefe) {
  if (ende(spielzustand) oder resttiefe=0) {
    return bZ(spielzustand);
  } else if (not maximierend) {
    for (jeden Folgezustand fz) {
      wert = alphabeta(fz, α, β, true, resttiefe-1);
      β = min(β, wert);
      if (β ≤ α)
         break;
    }
    return β;
  } else // aktueller Spieler maximiert
    for (jeden Folgezustand fz) {
      wert = alphabeta(fz, α, β, false, resttiefe-1);
      α = max(α, wert);
      if (β ≤ α)
         break;
    }
    return α;
  }
}
```

Die Parameter α und β dienen hierbei zum Abschneiden von irrelevanten Zügen im Suchbaum, von denen aufgrund der bereits betrachteten Züge klar ist, dass sie von den Spielern bei einer optimalen Strategie nie gespielt würden. Die Parameter sind beim ersten Aufruf auf $\alpha = -\infty$ und $\beta = +\infty$ zu setzen (bzw. auf einen passenden praktischen Wert, der für α kleiner sein muss als der minimal mögliche Wert Ihrer Stellungsbewertung und für β größer als der maximal mögliche Wert Ihrer Stellungsbewertung).

Das Alpha-Beta-Verfahren berechnet zunächst nur eine Bewertung des aktuellen Spielzustands. Um den dadurch bestimmten besten Zug zu ermitteln, müssen Sie etwa wie folgt in das Verfahren einsteigen:

```
generiereZug(spielzustand, suchtiefe) {
  if (ende(spielzustand) oder suchtiefe=0) {
    ERROR 'Keine Zugberechnung möglich';
  } else {
    besterZug = egal;
    α = kleinstmöglicher Wert der Stellungsbewertung - 1;
    β = größtmöglicher Wert der Stellungsbewertung + 1;
    for (jeden gültigen nächsten Zug nz) {
      fz = Folgezustand nach Anwendung des Zugs nz;
      wert = alphabeta(fz, α, β, false, suchtiefe-1);
      if (wert > α) {
        α = wert;
        besterZug = nz;
      }
    }
    return besterZug;
  }
}
```

Das beschriebene Verfahren kann weiter verbessert werden, ohne seine Genauigkeit zu beeinflussen. Beispielsweise hat die Reihenfolge, in der die Züge bzw. Nachfolgestellungen durchlaufen werden, Einfluss auf die Effizienz des Verfahrens. Es steht Ihnen frei, als Bestandteil Ihrer Spielstrategien neben der Stellungsbewertung auch die Reihenfolge der Knotentraversierung zu ändern oder mit anderen Varianten des Alpha-Beta-Algorithmus zu experimentieren. Bitte dokumentieren Sie in diesem Fall die von Ihnen gewählte Version des Verfahrens und begründen Sie Ihre Designentscheidungen.

5 Spezifikation des Oware-Kommunikationsprotokolls

5.1 Vorbemerkungen

In den Abschnitten 3.7 und 3.11 des Pflichtenhefts wurde die Vorgehensweise eines Clients beim Zugriff auf entfernte Computerspieler (Zuggeneratorserver) und die Vorgehensweise eines solchen Servers bei der Bearbeitung einer Zuganfrage bereits schematisch beschrieben. Noch nicht geklärt wurden jedoch die Nachrichtenformate für die `MoveRequestMsg` des Clients und die entsprechende `ServerResponseMsg` des Servers. Der Aufbau dieser Nachrichten und die Bedeutung ihrer Bestandteile werden im folgenden spezifiziert. Grundsätzlich handelt es sich um einfache zeilenorientierte Nachrichten aus ASCII-Zeichen. Die Zeilen werden Java-typisch durch `\n` terminiert.
Wichtig:

- An dieser Stelle muss ausdrücklich darauf hingewiesen werden, dass das Kommunikationsprotokoll **bis auf das Byte genau** implementiert werden muss. Dies ist wichtig, damit Programme verschiedener Teilnehmer im Netzwerk miteinander spielen können.
- Benutzen Sie insbesondere auf keinen Fall `println`-Methoden, um den Zeilenvorschub zu generieren. Diese Methoden generieren auf unterschiedlichen Plattformen (Linux, Windows) unterschiedliche Zeichenfolgen. In einem früheren Praktikum war ein falscher Zeilenvorschub für 90% aller Kommunikationsprobleme verantwortlich. Senden Sie daher unbedingt das `\n` als einzelnes Byte.

Neben den Nachrichtenformaten wird im folgenden auch exakt festgelegt, in welchen Schritten der Zugserver die Nachricht des Clients auf Korrektheit prüfen muss und wie die `ServerResponseMsg` bei der Erkennung eines bestimmten Fehlers auszusehen hat. Diese Festlegung ermöglicht automatische Tests, ob Ihr Server das Kommunikationsprotokoll korrekt umsetzt und die Spielregeln beherrscht. Sie können diese Tests durch Aufruf eines Test-Clients selbst durchführen (siehe Anhang D). Dadurch können Sie Fehler in Ihrer Implementierung entdecken und Hinweise zur Art des Fehlers erhalten.
Ausführliche Beispiele zum hier beschriebenen Kommunikationsprotokoll finden Sie in Anhang B.

5.2 Spezifikation der Client-Anfragen als `MoveRequestMsg`

Die `MoveRequestMsg` des Clients besteht aus vier Zeilen der folgenden Form:

```
<strategy>\n
<search-depth>\n
<initial-board>\n
<move-history>\n
```

Hierbei ist

- `<strategy>` die vom Server für die Zugberechnung zu verwendende Strategie. Zulässige Werte für die Strategie sind 0 und 1 (bzw. auch die nächsthöheren Zahlen, sofern Ihr Server noch weitere Strategien unterstützt).
- `<search-depth>` die vom Server für die Zugberechnung zu verwendende Suchtiefe. Zulässige Werte sind die Ziffern 1 bis 9.
- `<initial-board>` die Spielaufstellung zu Beginn des Spiels.
- `<move-history>` eine Folge aller in dem Spiel bisher ausgeführten Spielzüge, d.h. eine Aneinanderreihung beliebig vieler Ziffern zwischen 0 und 5.

Bei `<initial-board>` muss es sich um eine Zeile aus 14 durch einzelne Leerzeichen getrennten ganzen Zahlen zwischen 0 und 48 handeln, deren Summe 48 ergibt. Die Zuordnung der Zahlen zur Spielkonfiguration ergibt sich wie folgt:

$$\texttt{<a>}\ \texttt{<b>}\ \texttt{<}s_{A,0}\texttt{>}\ \ldots\ \texttt{<}s_{A,5}\texttt{>}\ \texttt{<}s_{B,0}\texttt{>}\ \ldots.\ \texttt{<}s_{B,5}\texttt{>}$$

Hier ist <a> der Besitz von Spieler A, <b> der Besitz von Spieler B, $<s_{A,h}>$ die aktuelle Zahl von Spielsteinen im Haus h $\in \{0, 1, 2, 3, 4, 5\}$ von Spieler A, und analog $<s_{B,h}>$ die aktuelle Zahl von Spielsteinen im Haus h von Spieler B. Die Zählung der Häuser eines Spielers erfolgt hier jeweils im Gegenuhrzeigersinn (wie in den Beispielen in Anhang A). Dieselbe Darstellung von Spielzuständen wird auch im folgenden für die `ServerResponseMsg` des Servers angenommen.

Im Normalfall enstpricht `<initial-board>` der durch die Spielregeln vorgegebenen Anfangsbelegung des Oware-Bretts:

```
0 0 4 4 4 4 4 4 4 4 4 4 4 4
```

Für diesen speziellen Spielzustand soll im folgenden die Abkürzung `<standard-initial-board>` verwendet werden. Andere Belegungen für `<initial-board>` werden nicht für normale Spiele, sondern nur für die Korrektheitsprüfung Ihres Programms durch den Test-Client benötigt.

Die `<move-history>` beschreibt den bisherigen Spielverlauf. Ausgehend von der Anfangskonfiguration des Bretts, die durch `<initial-board>` vorgegebenen ist, muss es sich dabei um eine gültige Folge von Zügen handeln, wobei die angegebenen Ziffern sich stets auf das gewählte Haus des jeweils aktiven Spielers beziehen (beim ersten Zug immer Spieler A, beim zweiten Spieler B usw.). Eine Leerzeile für die `<move-history>` ist explizit erlaubt. In diesem Fall wurden noch gar keine Züge gemacht, d.h. die aktuelle Spielsituation entspricht der durch `<initial-board>` beschriebenen Situation und Spieler A ist am Zug.

5.3 Spezifikation der Serverantworten als `ServerResponseMsg`

Eine `ServerResponseMsg` des Servers umfasst stets fünf Zeilen und hat die folgende allgemeine Gestalt:

```
<status>\n
<server-author>\n
<strategy-name>\n
<response-details>\n
<resulting-board>\n
```

Hierbei kann `<status>` einen der folgenden Werte annehmen (je nachdem, ob ein Fehler auftritt und von welcher Art der Fehler ist):

- `ERROR read-failure` – beim Server ist keine aus vier Zeilen bestehende Nachricht angekommen.
- `ERROR strategy` – ungültiger Wert für die Strategie des Servers.
- `ERROR search-depth` – ungültiger Wert für die Suchtiefe des Servers.
- `ERROR initial-board` – ungültige Startkonfiguration des Bretts.
- `ERROR move-history <m>` – gibt an, dass der `m`-te Zug in der `<move-history>` der Client-Nachricht ungültig ist. Die Zählung der Züge beginnt bei 0 für den ersten Zug.
- `OK` – erfolgreiche Analyse und Kontrolle der `MoveRequestMsg`, aus der ein gültiger Zug (einschließlich der Möglichkeit zur Beendigung des Spiels) berechnet wurde.

Die zweite Zeile, `<server-author>`, soll den Namen des Serverautors (also Ihren Namen!) enthalten. Bitte schreiben Sie Ihren Namen nur mit reinen ASCII-Zeichen und verzichten Sie zur Vermeidung eventueller Zeichensatzprobleme auf Umlaute oder Akzente.

Die dritte Zeile, `<strategy-name>`, soll den Namen der aktuellen Strategie des Servers enthalten. Sofern aus der `MoveRequestMsg` des Clients eine korrekte Angabe zur Suchstrategie ausgelesen werden konnte, soll `<strategy-name>` der Name dieser Strategie sein, andernfalls der Name der voreingestellten Standard-Strategie Ihres Servers.

Die möglichen Werte für die vierte Zeile, <response-details>, hängen vom <status> der ServerResponseMsg ab. Für ERROR read-failure steht hier eine Angabe zur ersten Zeile, die nicht mehr gelesen werden konnte. Bei allen anderen Arten von Fehlern wird in <response-details> die fehlerhafte Zeile der MoveRequestMsg wiederholt, auf die sich die Fehlermeldung bezieht. Für den Status OK enthält <response-details> den vom Zugserver berechneten neuen Zug.

Die fünfte Zeile, <resulting-board>, enthält den vom Server ermittelten Spielzustand im selben Format wie bei <initial-board>. Sofern der Server einen korrekten Zug ermitteln konnte (Status OK), ist <resulting-board> der Brettzustand nach Ausführung dieses Zugs. Sofern die Zughistorie einen Fehler enthält (Status ERROR move-history <m>), ist <resulting-board> der beim Nachvollziehen des Spielverlaufs anhand der Startkonfiguration <initial-board> und der <move-history> ermittelte Spielzustand nach Ausführung des Zugs m-1, d.h. direkt vor dem Auftreten des ungültigen Zugs. In allen anderen Fällen ist die Standard-Ausgangsposition <standard-initial-board> für <resulting-board> zu nehmen.

5.4 Vorgehensweise zur Ermittlung der ServerResponseMsg durch den Server

Im folgenden wird die Vorgehensweise des Servers bei der Ermittlung der ServerResponseMsg festgelegt. Es ist wichtig, dass Sie sich exakt an diese Beschreibung halten, weil andernfalls Ihr Zugserver die Tests durch den Test-Client nicht bestehen wird.

5.4.1 Test auf ERROR read-failure

Nach der Kontaktaufnahme durch den Client und dem Start eines neuen Threads zur Abwicklung der Kommunikation mit dem Client und zur Zugberechnung versucht der Server zunächst, eine vollständige MoveRequestMsg bestehend aus vier Zeilen einzulesen.

Der Server soll nun zuerst testen, ob die empfangene Nachricht vier Zeilen umfasst, und im positiven Fall erst anschließend die einzelnen Zeilen auf Korrektheit testen.

Scheitert schon der erste Test und es konnten keine vier Zeilen eingelesen werden, so ist eine read-failure-Nachricht im folgenden Format zu erzeugen:

```
ERROR read-failure\n
<server-author>\n
<default-strategy-name>\n
<where>\n
<standard-initial-board>\n
```

Hierbei gibt <where> die erste Zeile an, die nicht mehr eingelesen werden konnte. Mögliche Werte sind strategy, search-depth, initial-board und move-history.

Ein Fehler der beschriebenen Art wird entweder auftreten, wenn der Client eine Nachricht in einem falschen Format schickt, oder aber im Falle von Störungen der Netzwerkverbindung, wenn der Server den Rest der Nachricht durch einen Verbindungsabbruch nicht mehr einlesen kann.

5.4.2 Test auf ERROR strategy

Im folgenden wird davon ausgegangen, dass der Server vier Zeilen (und damit potentiell eine korrekte MoveRequestMsg) vom Client einlesen konnte. Der Server analysiert in diesem Fall die Zeilen der Reihe nach, beginnend mit der <strategy>-Zeile.

Auf dieser Zeile darf nur eine nicht-negative ganze Zahl stehen. Diese bestimmt die entsprechend numerierte Spielstrategie des Servers. Jeder Server muss mindestens zwei Strategien implementieren. Daher stehen mindestens die Zahlen 0 und 1 für zulässige Angaben zur Strategie. Sofern Sie noch weitere Spielstrategien implementieren und in Ihrem Server anbieten möchten, können Sie fortlaufend höhere Nummern 2, 3 usw. an diese Strategien vergeben und entsprechende Angaben für <strategy> ebenfalls zulassen.

Sofern <strategy> die Nummer einer vom Server unterstützten Strategie ist, so ist die entsprechende Strategie anschließend für die Zuggenerierung zu verwenden. Andernfalls muss der Server eine Fehlernachricht der folgenden Form generieren:

```
ERROR strategy\n
<server-author>\n
<default-strategy-name>\n
<rejected-strategy>\n
<standard-initial-board>\n
```

Hierbei ist `<default-strategy-name>` wieder der Name der voreingestellten Strategie des Servers und `<rejected-strategy>` gerade die als falsch abgelehnte Angabe zur Strategie aus der Nachricht des Clients.

5.4.3 Test auf ERROR search-depth

Sofern die `<strategy>`-Zeile korrekt ist, kontrolliert der Server als nächstes die eingelesene Angabe zur `<search-depth>`. Für diese sind nur die Ziffern von 1 bis 9 zulässig. Andernfalls erzeugt der Server eine Fehlernachricht der folgenden Art:

```
ERROR search-depth\n
<server-author>\n
<strategy-name>\n
<rejected-search-depth>\n
<standard-initial-board>\n
```

Hierbei ist `<strategy-name>` der Name der vom Client verlangten Spielstrategie. Die als falsch zurückgewiesene Angabe zur gewünschten Suchtiefe steht in `<rejected-search-depth>`.

5.4.4 Test auf ERROR initial-board

Sofern auch die Angabe zur `<search-depth>` korrekt ist, prüft der Server als nächstes die Zulässigkeit der Zeile `<initial-board>` mit der Beschreibung des anfänglichen Spielzustands. Die Bedingungen für einen zulässigen Spielzustand sind oben bereits formuliert: `<initial-board>` muss aus 14 mit einfachen Leerzeichen getrennten ganzen Zahlen zwischen 0 und 48 bestehen, die sich zu 48 aufsummieren. Ist diese Bedingung verletzt, so erzeugt der Server eine `ServerResponseMsg` mit dem folgenden Inhalt:

```
ERROR initial-board\n
<server-author>\n
<strategy-name>\n
<rejected-initial-board>\n
<standard-initial-board>\n
```

Hierbei ist `<rejected-initial-board>` gerade die als falsch abgelehnte Anfangsbelegung des Bretts aus der Nachricht des Clients.

5.4.5 Test auf ERROR move-history

Im Falle einer korrekten Angabe zur Startkonfiguration `<initial-board>` testet der Server abschließend die Gültigkeit der Angabe zur `<move-history>`. Hierbei handelt es sich um eine möglicherweise leere Zeichenkette aus den Ziffern `0` bis `5`.

Zur Kontrolle der Korrektheit muss der Server den bisherigen Spielverlauf nachvollziehen. Er beginnt mit dem angegebenen Anfangszustand aus `<initial-board>`, dem aktiven Spieler A und einer leeren Menge besuchter Spielsituationen für beide Spieler (benötigt für die Erkennung von Zyklen im Spielverlauf nach (E-3)).

Dann geht der Server die angegebenen Züge Zeichen für Zeichen von links kommend durch und testet, ob das gerade betrachtete Zeichen für einen gültigen Zug steht. Hierzu testet der Server zunächst, ob das aktuelle Zeichen eine Ziffer zwischen 0 und 5 ist. Wenn ja, muss der Server testen, ob es sich um einen gültigen Zug handelt (d.h. das Spiel ist noch nicht beendet und der Zug ist nach den Spielregeln für den

aktuellen Spieler erlaubt). Im positiven Fall führt der Server den Zug aus und ändert den Zustand des Bretts und die Informationen zur Erkennung wiederholter Spielsituationen dementsprechend. Damit kann der Server nun die Gültigkeit des nächsten Zugs testen, dann des übernächsten Zugs usw. Sofern alle Züge korrekt sind, rekonstruiert der Server auf diese Weise Schritt für Schritt den bisherigen Spielverlauf und hat nach dem letzten Zug die aktuelle Spielsituation ermittelt, aus der er den nächsten Zug berechnen soll. Im negativen Fall eines ungültigen Zugs dagegen erzeugt der Server eine `ServerResponseMsg` der folgenden Form:

```
ERROR move-history <m>\n
<server-author>\n
<strategy-name>\n
<rejected-move-history>\n
<board-before-wrong-move>\n
```

Hierbei ist `<m>` die Position des ungültigen Zugs (Zählung beginnt bei 0 für das erste Zeichen in der `<move-history>`), `<rejected-move-history>` wiederholt die fehlerhafte Zeile aus der Nachricht des Clients, und `<board-before-wrong-move>` gibt den Spielzustand nach dem letzten gültigen Zug an, also die Situation, in der der jetzt zurückgewiesene Zug nicht zulässig ist.

5.4.6 Test auf Erreichen des Spielendes

Sofern die Korrektheitsüberprüfung der gesamten `MoveRequestMsg` des Clients erfolgreich verlaufen ist, hat der Zugserver den aktuellen Spielzustand Schritt für Schritt aus dem angegebenen Startzustand und der übermittelten Zughistorie berechnet. Im Vergleich dazu, einfach die aktuelle Brettsituation zu übermitteln, hat dieses Vorgehen den Vorteil, dem Server auch die vollständige Information über den bisherigen Spielverlauf zu liefern, die dieser zur Erkennung wiederholter Spielsituationen braucht.

Hat der Zugserver nun aus der `MoveRequestMsg` des Clients den aktuellen Spielstand hergestellt und die für die Zyklenerkennung benötigten Informationen ermittelt, so muss er zunächst prüfen, ob das Spielende bereits erreicht ist. Sofern eines der drei Abbruchkriterien erfüllt ist, erfolgt keine Zugberechnung. Stattdessen muss der Zugserver dann eine Spielende-Nachricht der folgenden Form erzeugen:

```
OK\n
<server-author>\n
<strategy-name>\n
finish\n
<final-board>\n
```

Hierbei ist `<final-board>` der entstehende Spielzustand, wenn jeder Spieler die restlichen Steine in seinem Territorium in Besitz nimmt.

Hinweis: Durch diese spezielle Form einer finish-Nachricht kann automatisch geprüft werden, ob Ihr Programm die Kriterien für das Spielende korrekt umsetzt und beim Erreichen des Spielendes den korrekten Spielstand ermittelt, und der in Anhang D beschriebene Test-Client enthält entsprechende Tests. Im üblichen Spielverlauf wird jedoch vermutlich bereits der Client auf das Vorliegen des Spielendes testen und braucht in diesem Fall gar keine `MoveRequestMsg` an den Server zu senden.

5.4.7 Berechung eines gültigen nächsten Zugs

Ist das Endekriterium noch nicht erreicht, so muss der Spielserver mit den angegebenen Sucheinstellungen den nächsten Zug generieren. Der aktuelle Spielzustand und die Informationen für die Zyklenerkennung sind dann rekonstruiert und die Einstellungen für Suchstrategie und Suchtiefe ebenfalls aus der `MoveRequestMsg` des Clients bekannt. Auf Basis dieser Einstellung muss der Zugserver dann den neuen Zug berechnen. Die zu erzeugende `ServerResponseMsg` hat dann die folgende Form:

```
OK\n
<server-author>\n
<strategy-name>\n
```

```
<selected-move>\n
<resulting-board>\n
```

Hierbei ist `<selected-move>` eine Ziffer zwischen 0 und 5, die den vom Server gewählten neuen Zug wiedergibt, und `<resulting-board>` ist der neue Spielzustand nach Anwendung des vom Server gewählten Zugs.

Sofern der Server trotz erfolgreicher Analyse und Kontrolle der `MoveRequestMsg` aufgrund eines internen Fehlers nicht in der Lage ist, eine gültige Antwort zu berechnen, soll er nicht eine Fehlernachricht senden, sondern die Verbindung zum Client abbrechen.

6 Empfohlene Vorgehensweise

Die folgende Auflistung zeigt Ihnen, wie Sie an die Umsetzung des Anforderungskatalogs herangehen können. Durch die beschriebene Vorgehensweise können Sie es vermeiden, sich sofort mit der ganzen Komplexität der Aufgabe befassen zu müssen. Stattdessen bauen Sie Ihre Lösung schrittweise auf.

- Machen Sie sich einen groben Zeitplan.
- Überlegen Sie sich, wie Sie eine Oware-Spielsituation (Steine in den Häusern und im Besitz der Spieler, aktiver Spieler) intern darstellen möchten (noch ohne Berücksichtigung des bisherigen Spielverlaufs für die Zyklenerkennung). Setzen Sie die Regeln zum Säen aus einem Haus, zum Einsammeln der Steine und schließlich zur Ausführung eines vollständigen Zugs um. Implementieren Sie die Regeln zur Erkennung des Spielendes (bis auf die Zyklenerkennung).
- Implementieren Sie ein GUI, mit dem Sie ausgehend von Ihrer Implementierung der Spielsituationen und Spielzüge bereits Mensch gegen Mensch spielen können (eine vollständige Erkennung des Spielendes fehlt noch). Auf Feinheiten wie die Ausführung der Züge in Einzelschritten verzichten Sie und zeigen einfach direkt die entstehende Spielsituation.
- Implementieren Sie die Client-Schnittstelle des Oware-Kommunikationsprotokolls. Damit können Sie bereits ein Spiel Mensch gegen entfernten Computer spielen.
- Entwickeln Sie eine erste (rudimentäre) Strategie für die Zuggenerierung. Spätestens jetzt müssen Sie sich überlegen, wie Sie Informationen für die Erkennung wiederholter Spielsituationen intern darstellen möchten, und ausgehend hiervon das vollständige Kriterium zur Erkennung des Spielendes einschließlich der Zyklenerkennung implementieren.
- Erweitern Sie die eigene Netzwerkkomponente durch die geforderte Serverfunktionalität.
- Testen Sie Ihren Zugserver mit dem von uns bereitgestellten Test-Client und beseitigen Sie ggf. Fehler im Kommunikationsprotokoll und in der Umsetzung der Spielregeln.
- Nutzen Sie den Zugserver zur Realisierung des lokalen Computerspielers, indem Sie mit dem bereits realisierten Client auf Ihren lokalen Zugserver zugreifen.
- Verbessern und vervollständigen Sie die graphische Benutzerschnittstelle. Jetzt können Sie Extras wie das Säen und Einsammeln in Einzelschritten oder die Funktionalität zum Erzeugen von Zugvorschlägen realisieren.
- Verbessern Sie Ihre Computerstrategie und stellen Sie mindestens zwei Varianten Ihrer Spielstrategien lokal und über den Zugserver zur Verfügung.

Andere Bearbeitungsreihenfolgen sind natürlich auch möglich, je nachdem, ob Sie sich zuerst auf die Spiellogik, Zuggenerierung, die graphische Benutzerschnittstelle oder die Netzwerkkommunikation konzentrieren möchten.

Anhang

A Beispiele zu den Oware-Spielregeln

A.1 Schematische Brettdarstellung

Zur Veranschaulichung der Spielregeln soll im folgenden eine schematische Darstellung der Spielsituation dienen. Betrachten Sie hierzu die folgende Abbildung:

Spieler B: b

(5)	(4)	(3)	(2)	(1)	(0)
(0)	(1)	(2)	(3)	(4)	(5)

• Spieler A: a

Der erste Spieler, hier Spieler A genannt, wird unten gezeichnet; dem zweiten Spieler, Spieler B, gehört dementsprechend die obere Reihe des Bretts. Die Häuser jedes Spielers sind im Gegenuhrzeigersinn von 0 bis 5 durchnummeriert. Im Beispiel entspricht die Steinanzahl in jedem Haus gerade dem Index des Hauses. Die schematische Darstellung zeigt ferner zu jedem Spieler den aktuellen Besitz des Spielers (hier als a für Spieler A und b für Spieler B dargestellt). Im Beispiel ist Spieler A am Zug, was durch die •-Markierung ausgedrückt wird.

A.2 Anfängliche Spielsituation

Zum Beginn des Spiels sieht das Brett wie folgt aus:

Spieler B: 0

(4)	(4)	(4)	(4)	(4)	(4)
(4)	(4)	(4)	(4)	(4)	(4)

• Spieler A: 0

In jedem Haus liegen also vier Steine; keiner der Spieler hat schon Steine in seinem Besitz; und Spieler A ist am Zug.

A.3 Einfache Beispiele zum Aussäen

Es wird von der Situation zu Anfang des Spiels ausgegangen. Angenommen Spieler A wählt nun Haus 4 (zweites von rechts) zum Säen aus, so nimmt er zunächst die 4 Steine in diesem Haus an sich:

Spieler B: 0

(4)	(4)	(4)	(4)	(4)	(4)
(4)	(4)	(4)	(4)	**(0)**	(4)

○ Spieler A: 0

(Der offene Ring ○ bei Spieler A soll andeuten, dass es sich hier um eine Zwischensituation beim Ziehen handelt und nicht um das endgültige Ergebnis des Zugs.)

Spieler A legt die vier Steine dann im Gegenuhrzeigersinn der Reihe nach aus:

Spieler B: 0

(4)	(4)	(4)	(4)	(4)	(4)
(4)	(4)	(4)	(4)	(0)	**(5)**

○ Spieler A: 0

Spieler B: 0

(4)	(4)	(4)	(4)	(4)	**(5)**
(4)	(4)	(4)	(4)	(0)	(5)

○ Spieler A: 0

Spieler B: 0

(4)	(4)	(4)	(4)	**(5)**	(5)
(4)	(4)	(4)	(4)	(0)	(5)

◦ Spieler A: 0

Spieler B: 0

(4)	(4)	(4)	**(5)**	(5)	(5)
(4)	(4)	(4)	(4)	(0)	(5)

◦ Spieler A: 0

Der letzte Stein wurde also auf Haus 2 des Gegners gelegt (zur Erinnerung, die Zählung beginnt bei 0). Weil dort am Ende des Zugs 5 Steine liegen und nicht 2 oder 3, kann nichts eingesammelt werden. Das Brett bleibt also, wie es ist, und Spieler B ist jetzt am Zug:

•Spieler B: 0

(4)	(4)	(4)	(5)	(5)	(5)
(4)	(4)	(4)	(4)	(0)	(5)

Spieler A: 0

Spieler B könnte nun seinerseits z.B. das Haus 5 wählen. Nach diesem Zug würde das Brett dann wie folgt aussehen:

Spieler B: 0

(0)	(4)	(4)	(5)	(5)	(5)
(5)	(5)	(5)	(5)	(0)	(5)

• Spieler A: 0

A.4 Aussäen mit Überspringen des gewählten Hauses

Das nächste Beispiel verdeutlicht die Regel zum Überspringen des gewählten Hauses, wenn die Zahl der zu säenden Steine mehr als 11 beträgt. Die Ausgangssituation sei die folgende:

Spieler B: 0

(4)	(6)	(0)	(6)	(4)	(4)
(1)	(2)	(12)	(1)	(4)	(4)

• Spieler A: 0

Wählt nun Spieler A das Haus 2, dann kann er zunächst einmal 11 Steine wie folgt legen (hier mit Hervorhebung des elften gelegten Steins):

Spieler B: 0

(5)	(7)	(1)	(7)	(5)	(5)
(2)	**(3)**	(0)	(2)	(5)	(5)

◦ Spieler A: 0

Der verbleibende 12. Stein wird dann aber nicht auf Haus 2 von Spieler A gelegt, sondern dieses wird übersprungen und der Stein kommt auf Haus 3. Dementsprechend sieht die Spielsituation nach Abschluss des Zugs wie folgt aus:

•Spieler B: 0

(5)	(7)	(1)	(7)	(5)	(5)
(2)	(3)	(0)	(3)	(5)	(5)

Spieler A: 0

A.5 Anwendung der Starvation-Regel

Die Starvation-Regel verbietet einen Zug, der den Gegner nach dem Aussäen ohne Steine lässt. Das nächste Beispiel verdeutlicht diese Regel.

Spieler B: 23

(0)	(0)	(0)	(0)	(0)	(0)
(1)	(0)	(0)	(0)	(0)	(1)

• Spieler A: 23

Hier ist Spieler A am Zug und hat scheinbar zwei Zugmöglichkeiten, Haus 0 oder Haus 5. Weil sich in der gegnerischen Reihe keine Steine mehr befinden, darf Spieler A in dieser Situation jedoch nicht das Haus 0 wählen, denn nach dem Aussäen des Steins aus Haus 0 (der dann in Haus 1 von Spieler A liegen würde) hätte der Gegner weiterhin keine Steine in seiner Hälfte des Bretts. Dagegen ist Haus 5 hier ein gültiger Zug, der nicht die Starvation-Regel verletzt, denn nach dem Aussäen liegt dann ein Stein in Haus 0 des Gegners.

A.6 Einfache Beispiele zum Einsammeln

Die Regeln zum Einsammeln von Steinen sollen zunächst an zwei einfachen Beispielen erläutert werden.

Spieler B: 16

(2)	(2)	(1)	(0)	(1)	(2)
(0)	(0)	(0)	(0)	(3)	(5)

• Spieler A: 16

Wählt hier Spieler A das Haus 5, so sieht das Brett nach dem Aussäen wie folgt aus:

Spieler B: 16

(2)	(**3**)	(2)	(1)	(2)	(3)
(0)	(0)	(0)	(0)	(3)	(0)

◦ Spieler A: 16

Der letzte Stein wurde in Haus 4 des Spielers B gelegt, in dem sich danach 3 Steine befinden. Also darf Spieler A zunächst die Steine in diesem Haus einsammeln:

Spieler B: 16

(2)	(**0**)	(2)	(1)	(2)	(3)
(0)	(0)	(0)	(0)	(3)	(0)

◦ Spieler A: **19**

Daraufhin wird das im Uhrzeigersinn nächstgelegene Haus des Gegners betrachtet, also Haus 3. Dort befinden sich 2 Steine, also darf ebenfalls eingesammelt werden:

Spieler B: 16

(2)	(0)	(**0**)	(1)	(2)	(3)
(0)	(0)	(0)	(0)	(3)	(0)

◦ Spieler A: **21**

Als nächstes wird Haus 2 von Spieler B betrachtet. In diesem befindet sich ein Stein, so dass hier das Einsammeln abbricht. Die neue Spielsituation für Spieler B ist demnach:

•Spieler B: 16

(2)	(0)	(0)	(1)	(2)	(3)
(0)	(0)	(0)	(0)	(3)	(0)

Spieler A: 21

Für ein zweites Beispiel zum Einsammeln wird wieder die folgende Situation betrachtet:

Spieler B: 16

(2)	(2)	(1)	(0)	(1)	(2)
(0)	(0)	(0)	(0)	(3)	(5)

• Spieler A: 16

wobei nun jedoch Haus 4 von Spieler A gewählt wird. Nach dem Aussäen sieht das Brett dann wie folgt aus:

Spieler B: 16

(2)	(2)	(1)	(0)	**(2)**	(3)
(0)	(0)	(0)	(0)	(0)	(6)

◦ Spieler A: 16

Der letzte Stein wurde in Haus 1 von Spieler B gelegt, in dem sich nach dem Legen zwei Steine befinden. Diese können also eingesammelt werden:

Spieler B: 16

(2)	(2)	(1)	(0)	**(0)**	(3)
(0)	(0)	(0)	(0)	(0)	(6)

◦ Spieler A: **18**

Im davor befindlichen Haus 0 befinden sich 3 Steine, die ebenfalls gesammelt werden können:

Spieler B: 16

(2)	(2)	(1)	(0)	(0)	**(0)**
(0)	(0)	(0)	(0)	(0)	(6)

◦ Spieler A: **21**

An dieser Stelle endet dann das Einsammeln, denn das im Uhrzeigersinn gesehen nächste Haus liegt bereits im Territorium von Spieler A. Das Ergebnis des Zugs ist also

•Spieler B: 16

(2)	(2)	(1)	(0)	(0)	(0)
(0)	(0)	(0)	(0)	(0)	(6)

Spieler A: 21

A.7 Anwendung der Grand-Slam-Regel

Das letzte Beispiel zum Einsammeln verdeutlicht die Grand-Slam-Regel. Hierzu wird folgende Situation angenommen, wieder mit Spieler A als dem gerade aktiven Spieler:

Spieler B: 18

(0)	(1)	(2)	(1)	(2)	(1)
(1)	(0)	(0)	(0)	(0)	(5)

• Spieler A: 17

Wählt nun Spieler A das Haus 5 aus, so sieht das Brett nach dem Säen wie folgt aus:

Spieler B: 18

(0)	**(2)**	(3)	(2)	(3)	(2)
(1)	(0)	(0)	(0)	(0)	(0)

◦ Spieler A: 17

wobei der letzte Stein ins Haus 4 von Spieler B gelegt wurde. Nach den normalen Regeln für das Einsammeln könnte hier Spieler A der Reihe nach alle Steine von Spieler B auflesen:

Spieler B: 18

(0)	(0)	(0)	(0)	(0)	(0)
(1)	(0)	(0)	(0)	(0)	(0)

◦ Spieler A: 29

Weil der Gegner dann aber gar keine Steine mehr hätte, ist ein solches Aufsammeln in diesem Fall *unzulässig*. Stattdessen müssen hier nach dem Aussäen alle Steine liegenbleiben. Die neue Situation nach Abschluss des Zugs sieht daher wie folgt aus:

•Spieler B: 18

(0)	(2)	(3)	(2)	(3)	(2)
(1)	(0)	(0)	(0)	(0)	(0)

Spieler A: 17

A.8 Beenden des Spiels durch Erreichen einer Punktemehrheit nach (E-1)

Das folgende Beispiel zeigt das Ende des Spiels durch Erreichen der zum Gewinnen erforderlichen Mindestpunktzahl (E-1). Ausgangsbasis ist die folgende Spielsituation mit Spieler A als aktivem Spieler:

Spieler B: 21

(1)	(0)	(2)	(2)	(2)	(2)
(1)	(0)	(0)	(0)	(0)	(4)

• Spieler A: 13

Wählt nun Spieler A zum Aussäen das Haus 5, so ergibt sich die folgende Spielsituation:

•Spieler B: 21

(1)	(0)	(0)	(0)	(0)	(0)
(1)	(0)	(0)	(0)	(0)	(0)

Spieler A: 25

Damit ist das Endekriterium (E-1) erfüllt (Spieler B kann nicht mehr gewinnen). Zur Ermittlung des Spielstands nimmt jeder Spieler die verbliebenden Steine auf seiner Seite an sich (das Einsammeln der Steine wird hier und im folgenden wie ein abschließender 'Pseudo-Zug' dargestellt):

Spieler B: 22

(0)	(0)	(0)	(0)	(0)	(0)
(0)	(0)	(0)	(0)	(0)	(0)

Spieler A: 26

Der endgültige Spielstand beträgt also 26:22, und Spieler A hat gewonnen.

A.9 Beenden des Spiels durch mangelnde Zugmöglichkeit nach (E-2)

Das folgende Beispiel zeigt die Beendigung des Spiels durch mangelnde Zugmöglichkeit für Spieler B (d.h. durch 'Verhungern' von Spieler A). Die Ausgangssituation für Spieler B ist wie folgt:

•Spieler B: 22

(0)	(1)	(1)	(1)	(1)	(0)
(0)	(0)	(0)	(0)	(0)	(0)

Spieler A: 22

Hier gibt es keinen möglichen Zug für den aktiven Spieler B mehr, durch den ein Stein ins Gebiet von Spieler A kommen würde. Nach der Starvation-Regel ist demnach keines der Häuser 1, 2, 3 oder 4 ein möglicher Zug für Spieler B. Die übrigen Häuser entfallen, da keine Steine auf ihnen liegen. Also gibt es keine gültige Zugmöglichkeit mehr für Spieler B. Das Spiel endet mit Regel (E-2) und die Spieler nehmen die verbliebenden Steine auf ihrer Seite an sich:

Spieler B: 26

(0)	(0)	(0)	(0)	(0)	(0)
(0)	(0)	(0)	(0)	(0)	(0)

Spieler A: 22

Damit ergibt sich ein Spielstand von 22:26 und Spieler B ist der Gewinner.

A.10 Spielende durch Auftreten eines Zyklus nach (E-3)

Das letzte Beispiel verdeutlicht ein Spielende durch wiederholtes Eintreten einer Spielsituation nach (E-3). Hierzu wird von folgender Spielsituation ausgegangen:

Spieler B: 23

(0)	(0)	(0)	(0)	(0)	(1)
(1)	(0)	(0)	(0)	(0)	(0)

• Spieler A: 23

Der gerade aktive Spieler A und Spieler B wählen nun abwechselnd die Häuser 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5 (d.h. sie setzen jeweils den einzigen verfügbaren Stein im Gegenuhrzeigersinn um eine Position weiter). Nach dieser Abfolge von Zügen ist dann wieder Spieler A der aktive Spieler und die Spielsituation ist erneut

Spieler B: 23

(0)	(0)	(0)	(0)	(0)	(1)
(1)	(0)	(0)	(0)	(0)	(0)

• Spieler A: 23

Also gibt es eine Wiederholung einer Spielsituation für Spieler A und das Spiel ist nach (E-3) abzubrechen. Die Spieler nehmen die verbleibenden Steine auf ihrer Seite an sich:

Spieler B: 24

(0)	(0)	(0)	(0)	(0)	(0)
(0)	(0)	(0)	(0)	(0)	(0)

Spieler A: 24

Es ergibt sich also ein Gleichstand von 24:24 und damit ein Unentschieden beider Spieler.

B Beispiele zum Oware-Kommunikationsprotokoll

B.1 Beispiel für ERROR read-failure

Als Beispiel für eine `MoveRequestMsg`, die einen read-failure erzeugt, betrachten wir die folgende dreizeilige Nachricht des Clients:

```
0\n
10 0 4 4 4 4 4 4 4 4 4 4 4 4\n
5\n
```

Hier wurde vonseiten des Clients vergessen, zwischen der `<search-depth> 1` und dem folgenden `<initial-board>` einen Zeilentrenner `\n` auszugeben. Infolgedessen kann der Server nur drei Zeilen einlesen. Weil die vierte Zeile zur `<move-history>` nicht mehr eingelesen werden kann, wird die folgenden `ServerResponseMsg` erzeugt:

```
ERROR read-failure\n
Karl Muster\n
Karl Musters Super-Strategie\n
move-history\n
0 0 4 4 4 4 4 4 4 4 4 4 4 4\n
```

B.2 Beispiel für ERROR strategy

Der Client sende die folgende Nachricht:

```
Teststrategie\n
1\n
0 0 0 5 5 5 5 4 0 5 5 5 5 4\n
5\n
```

Der Client hat fälschlich den Namen einer Strategie übermittelt und nicht die Nummer der Strategie auf den Server. Wegen der ungültigen Angabe `Teststrategie` weist der Server die Anfrage mit der folgenden Fehlernachricht zurück:

```
ERROR strategy\n
Karl Muster\n
Karl Musters Super-Strategie\n
Teststrategie\n
0 0 4 4 4 4 4 4 4 4 4 4 4 4\n
```

B.3 Beispiel für ERROR search-depth

Angenommen, der Client sendet die folgende Nachricht:

```
0\n
0\n
0 0 0 5 5 5 5 4 0 5 5 5 5 4\n
5\n
```

Wegen der nicht zulässigen Angabe `0` zur gewünschten Suchtiefe erzeugt der Server dann die folgende Fehlernachricht:

```
ERROR search-depth\n
Karl Muster\n
Karl Musters Super-Strategie\n
0\n
0 0 4 4 4 4 4 4 4 4 4 4 4 4\n
```

B.4 Beispiel für ERROR initial-board

Der Client sende eine Nachricht der folgenden Form:

```
0\n
1\n
0 0 4 4 4 4 4 4 4 4 4 4 4 5\n
5\n
```

Hier sind insgesamt 49 Steine im Spiel statt der geforderten 48. Der Server lehnt die Anfangskonfiguration daher mit der folgenden `ServerResponseMsg` ab:

```
ERROR initial-board\n
Karl Muster\n
Karl Musters Super-Strategie\n
0 0 4 4 4 4 4 4 4 4 4 4 4 5\n
0 0 4 4 4 4 4 4 4 4 4 4 4 4\n
```

B.5 Beispiel für ERROR move-history

In der folgenden Nachricht verlangt der Client einen unerlaubten Zug (8 ist keine gültige Nummer eines Hauses):

```
0\n
1\n
0 0 4 4 4 4 4 4 4 4 4 4 4 4\n
38\n
```

Der Server weist diesen Zug, der sich an Position 1 befindet, wie folgt zurück:

```
ERROR move-history 1\n
Karl Muster\n
Karl Musters Super-Strategie\n
38\n
0 0 4 4 4 0 5 5 5 5 4 4 4 4\n
```

B.6 Beispiel zum Test auf Erreichen des Spielendes

Während bisher die Reaktion des Servers auf fehlerhafte Nachrichten des Clients erklärt wurde, sollen nun zwei Beispiele die Berechnung neuer Züge zu einer korrekten `MoveRequestMsg` verdeutlichen. Dabei ist zunächst auf den Sonderfall zu prüfen, dass die `MoveRequestMsg` des Clients zwar einen korrekten Spielverlauf beschreibt, das Spielende jedoch bereits erreicht ist.

Betrachten wir hierzu die folgende Nachricht des Clients:

```
0\n
1\n
20 22 0 0 0 0 0 1 1 1 1 1 1 0\n
5\n
```

Die Spielsituation nach Ausführen des Zugs 5 von Spieler A sieht schematisch wie folgt aus:

•Spieler B: 22

(0)	(1)	(1)	(1)	(1)	(0)
(0)	(0)	(0)	(0)	(0)	(0)

Spieler A: 22

Dies ist die Spielsituation, aus der heraus der Server nun für Spieler B einen Zug berechnen soll. Aufgrund der Starvation-Regel gibt es für Spieler B aber gar keinen zulässigen Zug mehr, d.h. gemäß (E-2) ist das Spielende erreicht. Der Server erzeugt daher die folgende Spielende-Nachricht:

```
OK\n
Karl Muster\n
Karl Musters Super-Strategie\n
finish\n
22 26 0 0 0 0 0 0 0 0 0 0 0 0\n
```

B.7 Beispiel zur Berechnung eines regulären Zugs

Das letzte Beispiel soll einen regulären Zug innerhalb des laufenden Spiels verdeutlichen. Hier könnte die `MoveRequestMsg` des Clients wie folgt aussehen:

```
0\n
1\n
0 0 4 4 4 4 4 4 4 4 4 4 4 4\n
33\n
```

Demnach ist Spieler A am Zug und die aktuelle Spielsituation ist:

Spieler B: 0

(5)	(5)	(0)	(4)	(5)	(5)
(5)	(5)	(4)	(0)	(5)	(5)

• Spieler A: 0

Mit seinen aktuellen Einstellungen ermittelt der Zugserver nun das Haus mit der Position 2 als beste Zugmöglichkeit für Spieler A. Dieser Wahl entspricht die folgende `ServerResponseMsg`:

```
OK\n
Karl Muster\n
Karl Musters Super-Strategie\n
2\n
0 0 5 5 0 1 6 6 6 5 4 0 5 5\n
```

C Dokumentationsrichtlinien

Ihre Programmdokumentation sollten Sie mit derselben Sorgfalt erstellen, die Sie sonst etwa für eine schriftliche Seminarausarbeitung ansetzen würden. Beachten Sie bei der Erstellung Ihrer Dokumentation bitte die folgenden Richtlinien:

- Die Programmdokumentation besteht aus einem **Programmüberblick** mit der Darstellung der Lösungskonzeption und dem **kommentierten Quelltext** Ihres Programms zusammen mit der hieraus erzeugten **Javadoc-Dokumentation**.

- In dem Programmüberblick (nicht mehr als 10 Seiten) beschreiben Sie die Lösungskonzeption. Gehen Sie dabei insbesondere auf Entwurf und Bedienung der Benutzeroberfläche und auf die von Ihnen entwickelten Spielstrategien bzw. Ansätze zur Stellungsbewertung ein. Skizzieren Sie außerdem die wichtigsten benutzten Datenstrukturen und die wichtigsten Klassen bzw. Pakete mit ihrer Verantwortlichkeit für die zu lösenden Teilaufgaben (evtl. auch als Java-Klassendiagramm oder UML-Klassendiagram). Gehen Sie an dieser Stelle aber noch nicht auf Implementierungsdetails ein. Der Programmüberblick sollte auch für Leser nachvollziehbar sein, die zwar Programmiererfahrungen haben, aber die Sprache Java nicht beherrschen.
  Der Programmüberblick ist als gesondertes Dokument im ASCII-, RTF- oder PDF-Format abzugeben. Bitte geben Sie auf dem Deckblatt des Programmüberblicks Ihren Namen und Ihre Matrikelnummer an.
  Hinweis: Wenn Sie die Klassen- und Paketstruktur Ihres Programms als UML-Klassendiagramm oder -Paketdiagramm darstellen möchten, so können Sie hierfür z.B. die frei verfügbaren Werkzeuge ArgoUML (siehe `http://argouml.tigris.org/`) oder Umbrello UML Modeller (siehe `http://uml.sourceforge.net/`) benutzen. Die Erstellung von UML-Diagrammen zur Veranschaulichung Ihrer Lösung ist jedoch nicht verpflichtend.

- Überlegen Sie sich einen **einheitlichen Kommentarkopf mit Javadoc-Elementen**, den Sie vor jeder Klasse und Methode einfügen, die kommentiert werden soll. In diesem Kommentarkopf beschreiben Sie den Zweck der verwendeten Parameter, die Aufgabe, die von der Klasse bzw. Methode erfüllt wird, und die Einordnung der Klasse/Methode in den Gesamtkontext des Programmes. Benutzen Sie das Javadoc-Werkzeug, um die Parameter und Rückgabewerte auszuzeichnen. Halten Sie die Beschreibung knapp, zwei bis drei prägnante Sätze zur Aufgabe und Einordnung der Klasse bzw. Methode sollten ausreichen.
  Hinweis: Javadoc wird nur für die nicht-privaten Elemente verlangt.

- **Kommentieren Sie Ihren Programmtext**. Dies bedeutet nicht, dass Sie zu jeder Anweisung einen Kommentar schreiben sollen. Ihr Programm muss aber mit Hilfe der Kommentare soweit verständlich sein, dass der Leser Ihre Lösung gut nachvollziehen kann. Bedenken Sie, dass der Leser mit den von Ihnen eingeführten Datenstrukturen und Methoden nicht vertraut ist, und erläutern Sie daher den Zweck von Wertzuweisungen oder Methodenaufrufen, wenn dieser nicht offensichtlich ist.

Hinweis: Wenn Sie bemerken, dass ein Programmabschnitt so kompliziert ist, dass Sie zur Erläuterung mehrere Sätze brauchen, dann kann dies ein Zeichen für zu geringe Modularisierung Ihrer Lösung sein. In diesem Fall ist es möglicherweise am besten, den betreffenden Programmteil als eigene Methode auszugliedern, deren Name den Zweck des Abschnitts wiedergibt. Keinesfalls sollen Sie einen solchen Abschnitt einfach unkommentiert stehen lassen.

- Beachten Sie schon bei der Erstellung des Programmcodes bitte die unter `http://java.sun.com/docs/codeconv/` publizierten **Code Conventions für die Programmierung in Java**.
- Verwenden Sie darüber hinaus **aussagekräftige Bezeichner** für Ihre Variablen, Klassen und Methoden. Ein gut gewählter Bezeichner ist so kurz wie möglich, aber lang genug, um seine Funktion verständlich zu beschreiben. Abkürzungen sind erlaubt, wenn sie entschlüsselbar und aussprechbar sind (daher wäre z.B. `elem` besser als `elmt`).
- Benutzen Sie ein **einheitliches Vorgehen bei der Gliederung von Deklarationen und der Einrückung von Programmteilen**.
  Anweisungsfolgen zwischen geschweiften Klammern werden mit 2 bis 4 Leerzeichen eingerückt; die schließenden geschweiften Klammern stehen auf der gleichen Ebene wie die übergeordneten Konstrukte, zu denen sie gehören. Also zum Beispiel:

```
if (<Bedingung>) {
     <Anweisungsfolge>
} else {
     <Anweisungsfolge>
}
```

  aber **nicht**:

```
if (<Bedingung>) {
     <Anweisungsfolge>
    } else {
     <Anweisungsfolge>
    }
```

- Vermeiden Sie es, mehr als eine Anweisung in eine Zeile zu schreiben.

D Hinweise zum Testen des Programms

Dieser Abschnitt soll Ihnen einige Denkanstöße zum Testen Ihres Programms geben, die Ihnen das Abliefern einer weitgehend fehlerfreien Lösung erleichtern. Vor dem Testen Ihrer Lösung müssen Sie davon ausgehen, dass (statistisch gesehen) ca. vier bis acht Fehler auf 100 Zeilen Programmcode kommen werden. Beim Testen geht es deshalb darum, ein Programm mit der Absicht auszuführen, Fehler zu finden. Mit einer steigenden Anzahl so gefundener (und beseitigter) Fehler des Programms wird auch das Vertrauen in seine Zuverlässigkeit im praktischen Einsatz erhöht.
Nachfolgend wollen wir Ihnen einige Anregungen geben, wie Sie Ihr Programm effektiv testen können:

- Programmierfehler schleichen sich unvermeidlich in Programmcode ein. Sobald Ihr Programm kompliziert genug ist (und Oware *ist* kompliziert genug), wird das nachträgliche Debugging des Programms und die Suche nach dem Auslösers für einen Fehler des Gesamsystems zu einer zeitraubenden Angelegenheit. Daher empfiehlt es sich, die auftretenden Fehler schon **während der Entwicklung einer Methode oder Klasse** durch gezieltes Testen zu beseitigen. Zu diesem Zeitpunkt ist Ihnen die beabsichtigte Funktionsweise der Klasse/Methode noch genau bewußt und Sie können am leichtesten Testfälle formulieren und schnell den Fehler identifizieren, falls einer der Testfälle noch scheitert. Wenn Sie schon während der Entwicklung testen, kommt das auch der Struktur Ihres Programmcodes zugute ($\Rightarrow$ einfache Methoden mit klar überprüfbarer Aufgabe).

- Testen Sie Ihr Programm **klassenweise**, d.h. rufen Sie die Methoden und Konstruktoren der zu testenden Klasse direkt mit passenden Testwerten auf und überprüfen Sie die zurückgegebenen Werte. Eine in das Gesamtprogramm integrierte Klasse lässt sich nicht mehr zielgerichtet testen, da von der Eingabe des Hauptprogrammes bis zum Methodenaufruf der Zielklasse oft so viele Abhängigkeiten zu berücksichtigen sind, dass man für die Klasse keine individuellen Testwerte mehr erzeugen kann. Gleiches gilt für die Rückgabewerte der Methodenaufrufe, deren Effekt in einem komplexen Gesamtprogramm nicht mehr so leicht festzumachen ist wie bei Betrachtung der isolierten Klasse.

- Mit `JUnit` gibt es ein Framework für Unit-Tests (Test von Einzelkomponenten) in Java, das Ihnen die **Durchführung automatisierter Tests** ermöglicht. Sie brauchen also nicht mehr selbst zu prüfen, ob die Tests erfolgreich ablaufen oder scheitern, und können sie nach Änderung des Programms jederzeit erneut ausführen. JUnit ist in Java-Entwicklungsumgebungen wie Eclipse bereits integriert und sehr einfach nutzbar (weitere Informationen siehe z.B. `http://www.junit.org/`).

- Zum gezielten Auffinden kritischer Testeingaben können Sie sich z.B. der **Grenzwertanalyse** bedienen. Dabei ermitteln Sie für einen Eingabegröße alle gültigen und ungültigen Werte und wählen dann jeweils einen Repräsentanten aus, der gerade noch gültig ist ('auf der Grenze liegt') und je einen Repräsentanten, der knapp außerhalb der Grenze liegt und damit ungültig ist. Wenn Sie z.B. eine Funktion testen, die einen Text mit einer Länge von 1–40 Zeichen als Eingabe bekommt, würden Sie nach der Grenzwertmethode jeweils einen Text der Länge 1 und einen Text der Länge 40 als gültige Eingabe sowie Texte der Längen 0 bzw. 41 als ungültige Eingaben auswählen. Die ungültigen Werte nimmt man in die Testmenge auf, um zu sehen, ob das Programm auch auf fehlerhafte Eingaben sinnvoll reagiert (indem es z.B. eine Warnung ausgibt). Falls solche Testfälle nicht vorgesehen werden, kann es vorkommen, dass zum Beispiel ein Programm in inneren Modulen später aus 'unerklärlichen Gründen' abstürzt (weil im Verlauf der Berechnung intern ein ungültiger Wert erzeugt wurde), oder dass bei bestimmten Eingaben nur unsinnige (weil undefinierte) Ausgaben erscheinen.

- Eine ähnliche Strategie besteht darin, nach **Spezialfällen** (z.B. Definitionslücken wie bei der Division durch Null) zu suchen. Arbeitet ein Sortieralgorithmus z.B. auch korrekt, wenn die Liste der zu sortierenden Werte leer, einelementig oder bereits sortiert ist?

- Für Programmteile mit bedingten Anweisungen sollten Sie idealerweise die Testeingaben so wählen, dass insgesamt alle Programmteile in allen denkbaren Kombinationen durchlaufen werden. Beispiel:

```
if (a=3) {
   if (b=4) {
   <Anweisungsblock>
   } else {
   <Anweisungsblock>
   }
} else {
   <Anweisungsblock>
}
```

  Zum Testen dieses Programmteils sind zumindest die Wertekombinationen

  a=3, b=4
  a=3, b≠4,
  a≠3, b beliebig

  in die Testmenge aufzunehmen.

- Als **Hilfe für die Entwicklung der Netzwerkschnittstelle** Ihres Programms können Sie ab Mitte Oktober von der Propra-Seite (`http://pi7.fernuni-hagen.de/lehre/ProPra.html`) eine ausführbare .jar-Datei `TestClient.jar` herunterladen. Starten Sie zuerst Ihr Programm und danach (auf demselben Rechner!) den Test-Client durch `java -jar TestClient.jar`. Der

Test-Client wird nun mit Ihrem Programm über den Port 7889 Kontakt aufnehmen und durch eine Reihe von Testfällen abprüfen, ob Ihr Programm das Protokoll für den Zuggeneratorserver und die Oware-Spielregeln korrekt implementiert. Da auch hier der Test nur Einzelfälle abprüft, kann er eine sorgfältige Implementierung und gezielte weitere Tests Ihrerseits nicht überflüssig machen. Wenn jedoch einer der Testfälle des Test-Clients scheitert, dann können Sie sicher sein, dass Ihr Programm noch nicht korrekt funktioniert. In diesem Fall gibt der Test-Client eine kurze Beschreibung des Tests aus. Ferner wird die `MoveRequestMsg` des Test-Clients, die erwartete `ServerResponseMsg` des Servers und die tatsächliche Rückgabe Ihres Servers ausgegeben. Wundern Sie sich nicht über das Aussehen der `MoveRequestMsg` und der erwarteten `ServerResponseMsg`– in der Regel versucht der Test-Client, gezielt einen bestimmten Fehler zu provozieren und dadurch die Korrekheit Ihres Servers zu prüfen. 'Übersieht' Ihr Server den ersten Fehler in der `MoveRequestMsg` (oder behauptet er fälschlich einen Fehler noch vor dem ersten tatsächlichen Fehler), so führt dies zu einer Abweichung von der erwarteten `ServerResponseMsg` (bei der es sich wie gesagt meist um eine provozierte Fehlermeldung handelt) und damit zum Fehlschlagen des Tests.