Prof. Dr. Ralf Zimmer
LFE Practical Informatics and Bioinformatics
Department Institut für Informatik
Ludwig-Maximilians-University Munich

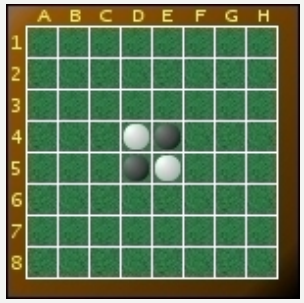
Programmierpraktikum 2020

Regeln, Abgabeserver und Lösungsansätze

TicTacToe GUI – typische Fehler (der letzten Jahre)

- kein Trennung Model + GUI
- 9 einzelne buttons, statt Button[][]
- umständliches Abholen welcher Button geklickt wurde (am besten Button direkt erweitern, so dass Koordinaten gespeichert werden)
- Interaktion zwischen den Klassen nicht durchdacht → statischer Zugriff auf das Spielfeld

Othello Regeln - Übersicht



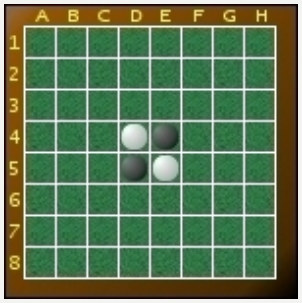
- Spielbrett: 8x8 Felder
- Als Startaufstellung werden vor dem Spielbeginn zwei weiße und zwei schwarze Steine auf die mittleren Felder des Bretts gelegt (Bild)
- Die Zahl der Steine ist praktisch unbegrenzt, denn insgesamt stehen 64 Spielsteine zur Verfügung
- Die Spieler ziehen abwechselnd, Schwarz beginnt. Man setzt entweder einen Stein mit der eigenen Farbe nach oben auf ein leeres Feld, oder man passt. Passen ist aber nur erlaubt, wenn man keine legale Möglichkeit hat, einen Stein zu setzen.
- Man darf nur so setzen, dass ausgehend von dem gesetzten Stein in beliebiger Richtung (senkrecht, waagerecht oder diagonal) ein oder mehrere gegnerische Steine anschließen und danach wieder ein eigener Stein liegt. Es muss also mindestens ein gegnerischer Stein von dem gesetzten Stein und einem anderen eigenen Stein in gerader Linie eingeschlossen werden. Dabei müssen alle Felder zwischen den beiden eigenen Steinen von gegnerischen Steinen besetzt sein.

A: legaler Zug für weiß

- Alle gegnerischen Steine, die so eingeschlossen werden, wechseln die Farbe (indem sie umgedreht werden). Wenn ein gerade umgedrehter Stein weitere gegnerische Steine einschließt, werden diese nicht umgedreht.
- Wenn die Spieler unmittelbar nacheinander passen, wenn also keiner mehr einen Stein setzen kann, ist das Spiel beendet.
- Der Spieler, der am Ende die meisten Steine seiner Farbe auf dem Brett hat, gewinnt. Haben beide die gleiche Zahl, ist das Spiel unentschieden.
- Beispiele für Züge die sich aus diesen Regeln ergeben gibt es hier:

https://www.bernhard-gaul.de/spiele/reversi/reversi-regeln.html

Regeln – Ablauf allgemein



- Spielbrett: 8x8 Felder
- Startaufstellung siehe Bild
- unbegrenzte Anzahl der Steine (64 pro Farbe)
- Die Spieler ziehen abwechselnd, Schwarz beginnt.
- 2 Möglichkeiten:
  - Man macht einen gültigen Zug (siehe nächste Folie)
  - man passt. Passen ist aber nur erlaubt, wenn man keinen gültigen Zug machen kann -> kein taktisches Passen

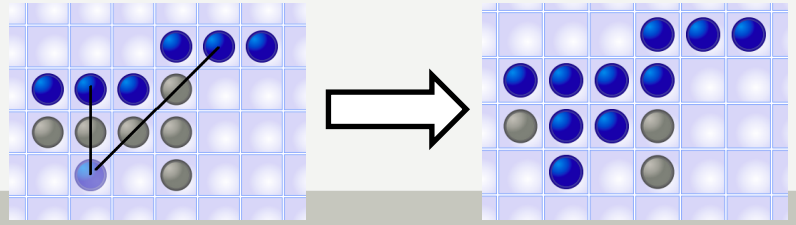
- Spielende: beide Spieler passen unmittelbar nacheinander -> keine gültigen Züge mehr möglich
- Der Spieler, der am Ende die meisten Steine seiner Farbe auf dem Brett hat, gewinnt. Haben beide die gleiche Zahl, ist das Spiel unentschieden.

Regeln - Gültige Züge

A: legaler Zug für weiß

- ausgehend von gesetzten Stein in beliebiger Richtung (senkrecht, waagerecht oder diagonal) müssen ein oder mehrere gegnerische Steine anschließen und danach wieder ein eigener Stein liegen
  - Es muss also mindestens ein gegnerischer Stein von dem gesetzten Stein und einem anderen eigenen Stein in gerader Linie eingeschlossen werden.
  - Dabei müssen alle Felder zwischen den beiden eigenen Steinen von gegnerischen Steinen besetzt sein.
- Alle gegnerischen Steine, die so eingeschlossen werden, wechseln die Farbe.

  - Wenn ein gerade umgedrehter Stein weitere gegnerische Steine einschließt, werden diese nicht umgedreht.



SERVER

Turnierregeln

- **Modus**
  - Jeder spielt gegen Jeden
  - Pro Spieler-Paar (Begegnung) werden mehrere Runden gespielt, die Spieler fangen abwechselnd an. Wer mehr Runden gewinnt, gewinnt auch die Begegnung
  - Die Rangliste wird nach Punkten aufgestellt: Eine gewonnene Begegnung gibt 2 Punkte, Unentschieden 1 Punkt und eine Niederlage 0 Punkte
- **Es gibt zwei Möglichkeiten eine Runde zu gewinnen**
  - Bei Spielende muss man mehr Steine in der eigenen Farbe auf dem Spielbrett haben als der Gegner
  - Der Gegner verliert
- **Es gibt (leider) viele Möglichkeiten zu Verlieren**
  - Der Gegner gewinnt (klar)
  - Man löst eine nicht abgefangene Exception aus...
  - Man liefert einen ungültigen Zug zurück, oder `null`, obwohl es einen gültigen Zug gibt
  - Man benötigt mehr Zeit, als man zur Verfügung hat
- **Unentschieden**
  - Beide Spieler können nicht mehr ziehen und haben gleich viele Steine in ihrer Farbe auf dem Brett

Interface Player

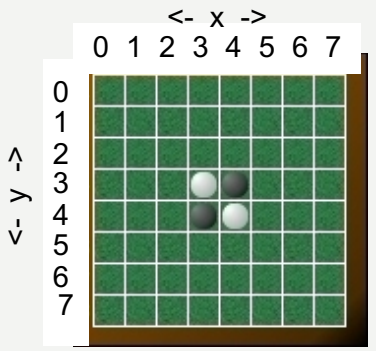
```
public interface Player {
    void init(int order, long t, java.util.Random rnd );
    Move nextMove(Move prevMove, long tOpponent, long t );
}
```

- **`init()`**
  - Wird zu Beginn des Spiels aufgerufen
  - Kann für eine Player-Instanz mehrfach aufgerufen werden -> Player bereit für nächste Partie
  - `order`: 0 bedeutet, dass dieser Spieler zuerst zieht, 1 zieht als zweiter
  - `t`: Gesamte verbleibende Laufzeit, die dem Spieler zur Verfügung steht (nur zur Info)
  - `rnd`: Zufallsgenerator, der für alle Zufallsoperationen benutzt werden muss
- **`nextMove()`**
  - Wird wiederholt aufgerufen, bis ein Gewinner feststeht
  - `prevMove`: vorhergehender Zug des Gegners. Dieser kann `null` sein, falls der Gegner signalisiert, dass er nicht ziehen kann, oder wenn es sich um den Eröffnungszug handelt
  - `tOpponent`: verbleibende Zeit, die dem Gegner zur Verfügung steht (nur zur Info - wird vom Server gemanaged)
  - `t`: verbleibende Zeit, die diesem Spieler zur Verfügung steht (nur zur Info - wird vom Server gemanaged)
  - **Rückgabewert**: Der nächste Zug dieses Spielers

Klasse Move

```
public class Move {
        public final int x;
        public final int y;

        public Move(int x, int y){
            this.x = x;
            this.y = y;
        }
  }
```



- X und Y Koordinaten von 0 bis 7, ausgehend von der oberen linken Ecke des Spielbrettes, x horizontal und y vertikal.

Alle Klassen finden sich zum Herunterladen auf der Homepage

Sourcecode der Klassen NICHT ändern!

Ablauf auf dem Server

```
Player p1 = new KI_1(); //KI muss Konstruktor
Player p2 = new KI_2(); //ohne Parameter haben

for(int i=0; i<rounds; i++){
   p1.init(0, 8000, rnd);
   p2.init(1, 8000, rnd);
   Move last = null;
   while(game_still_running){
      last = p1.nextMove(last, timeOpp, time);
      last = p2.nextMove(last, timeOpp, time);
   }
   //and the same but p2 starts
}
```

Nur zum groben Überblick was passiert, in Wirklichkeit passiert noch mehr, z.B. check ob Spiel beendet und update von timeOpp/time

Voraussetzungen für die Teilnahme am Blockteil

- **Lauffähige GUI**
  - nimmt abwechselnd neue Züge vom Benutzer und vom Computer (Klasse wird zunächst zur Verfügung gestellt) entgegen
  - Prüft die Züge auf Richtigkeit und zeigt sie an
  - Erkennt den Gewinner
- **Lauffähige Player-Klasse**
  - Nimmt Züge des Gegners entgegen und ermittelt nach einer Gewinnstrategie den nächsten optimalen Zug
  - Bildet zusammen mit der GUI ein vollständiges Othello-Spiel
- **Die Player-Klasse kann auf den Server geladen werden und gewinnt dort gegen beide Referenzklassen**

Letze Abgabe: Donnerstag, 14.01.2021

Zwischenstand & Öffnung Server: Donnerstag 17.12. 2020

Server

- AMD athlon 2216
- 64 bit Architektur
- 2 Gb Speicher für die Java-VM
- Java 14
- Nur 1 Thread erlaubt (!!! -> keine GUI mit hochladen)
- Zeit zur Berechnung aller Spielzüge: insgesamt 8 Sekunden (wird für die Endrunde evtl. verlängert)

- Source code aller benötigter Klassen (!) am besten als .jar-Datei bündeln und hochladen
- szte.mi.Player und szte.mi.Move dürfen nicht mit hochgeladen werden (wenn doch werden sie gelöscht)
- Server kompiliert Klassen neu – nicht-UTF8 Zeichen machen Probleme -> keine Umlaute etc. (auch nicht in Kommentaren)

Format

- bei mehreren KI-Varianten:
  - jar enthält Datei player.txt die package.class (z.B. othello.ki.BestKI) enthält
- für machine learning (not recommended) gespeicherte Modelle:
  - müssen auf *.ser enden
  - werden nach /home/proj/public_lehre/propra/dependencies/teamnr.filename gespeichert (z.B. 8.KI.ser) und können von dort geladen werden

Generell gilt

- Entwickle und programmiere die geforderten Klassen alleine. Ihr dürft euch untereinander austauschen, aber Kopien und Plagiate sind nicht erlaubt!
- Beachte in diesem Zusammenhang auch das Infoblatt zu Täuschungsversuchen und Plagiaten. Es ist zum Download auf der Praktikumshomepage
- Bei Problemen oder Fragen wenden Sie sich per Email an **propra@bio.ifi.lmu.de**

LÖSUNGSANSÄTZE

Aufgaben (1)

- **Entwickle eine Klasse, die das zur Verfügung gestellte Player interface implementiert, sich regelgerecht verhält und eine Gewinnstrategie verfolgt.**
  - Die Gewinnstrategie kann einfach gehalten sein (z.B. gültige Züge nach einer festen Bewertungstabelle), es kann aber auch der MiniMax-Algorithmus basierend auf und-oder-Bäumen implementiert werden.
  - Dieser Algorithmus **kann** nun weiter optimiert werden, z.B. durch alpha-beta-pruning und eine Heuristik zur Bewertung der Spielpositionen.
  - Als Hilfestellung dazu gibt es die Pflichtvorlesung “Fortgeschrittene Spielstrategie”
- **Entwickle eine grafische Benutzeroberfläche, die Spielereingaben von menschlichen Spielern und von beliebigen Klassen, die das Player interface implementieren verarbeiten kann.**
  - Zusammen mit der Player-Klasse soll es nun möglich sein komfortabel als Mensch gegen den Computer zu spielen
  - *Optional* ermöglichen Sie das “Beobachten” eines Spiels zweier Player-Klassen

Aufgaben (2)

- **Lade die Player Klasse mit eventuell benötigten weiteren Klassen auf den Abgabeserver.**

  (Es gibt auf der Veranstaltungshomepage die Möglichkeit zum Upload unter Ihrem Namen)

  - Die Klasse muss auf dem Server lauffähig sein
  - Es darf nichts außer Java source-files hochgeladen werden, idealerweise gebündelt als .jar-Datei
  - Die Klassen dürfen eine gewisse Dateigröße (BLOB in MariaDB) nicht überschreiten
  - Beide Referenzspieler müssen besiegt werden
  - Gewinne gegen möglichst viele andere Player Klassen
  - Es gibt eine ewige Bestenliste, d.h. die Klasse tritt gegen die Sieger der letzten Jahre an...

- **Es handelt sich bei dieser Abgabe um eine Prüfungsleistung. Beachte das im Hinblick auf Plagiate und Täuschungsversuche!**

Designüberlegungen

- **Überlege dir, welche Klassen benötigt werden. Möglich sind z.B.**
  - Eine Klasse für die Datenstruktur
  - Eine Klasse für die GUI
  - Eine Klasse pro KI (die Eingabe über die Maus kann auch als KI gesehen werden)
  - Eine Klasse, die Datenstruktur, GUI und KI zusammenbringt
- **Überlege Dir sich, wie diese Klassen kommunizieren müssen:**
  - Die Datenstruktur muss z.B. nichts über die GUI wissen, wohl aber die GUI über die Datenstruktur
  - Die KI muss nichts über die GUI wissen (außer es handelt sich um die Eingabe über die Maus)
  - Die GUI braucht keine Informationen von der KI – diese Informationen gehen direkt an die Datenstruktur
- **Zeichne ein Diagramm der Klassen und überlege dir, wie die Abhängigkeiten am besten implementiert werden (z.B. über setter/ getter, callback-Funktionen oder lokale Variablen)**
- **Die folgenden Milestones gelten als Designvorschläge (Hinweis: verwende möglichst nicht deine TicTacToe-Implementierungen)**

Milestone 1: Datenstrukturen

- **Keine GUI oder KI Programmlogik in den Datenstrukturen!**
- **Finde eine geeignete Repräsentation des 8X8 Spielfeldes, das 3 Zustände pro Feld erlaubt (frei, schwarz, weiß)**
  - Möglich wäre z.B. Ein 2-dimensionales Array, es gibt auch speichereffizientere Möglichkeiten
- **Entwickle Routinen, die für einen gegebenen Spielzug die notwendigen Veränderungen auf dem Spielfeld durchführen**
- **Schreibe Hilfsroutinen, z.B. zum**
  - **schnellen Auslesen der gültigen Spielzüge,**
  - **abfragen, wer der Gewinner ist / ob es einen gibt**
  - **Anzahl der gesetzten Spielsteine**
  - **…**
- **Diese Klasse kann später um Hilfsroutinen erweitert werden, die von der Künstlichen Intelligenz verwendet werden**

Milestone 2: GUI

- Keinerlei Programmlogik in der GUI!
- Schreibe zuerst möglichst sauberen Code, der die Datenstruktur visualisiert.
- Füge einen passenden Listener hinzu
- Überlege dir, wie du sicherstellst, dass nur zum richtigen Zeitpunkt (wenn der menschliche Spieler am Zug ist) Eingaben über die Maus akzeptiert werden
- Überlege, wie du die angeklickten Koordinaten am besten weiterverarbeitest
- Achte darauf, an den entsprechenden Stellen im Code die GUI zu aktualisieren, damit Veränderungen angezeigt werden
- Schreibe eine Runner-Klasse, die deine GUI und Datenstruktur zusammenbringt und ein Spiel Mensch-gegen-Mensch am Computer ermöglicht

Milestone 3: Spielstrategie

- **Schreibe deine Spielstrategie (KI) nicht in die Klasse mit der Datenstruktur**
- **Füge jedoch ggf. Code zur Datenstruktur-Klasse hinzu**
  - Z.B. um einen Spielzug zu “simulieren” ohne die eigentliche Datenstruktur zu verändern
- **Implementiere das Player-Interface**
- **Gib als Zug zunächst zufällige (aber gültige) Züge zurück (hierbei sollten passende Funktionen aus der Datenstruktur helfen)**
- **Erweitere deine Runner-Klasse, so dass Spiele Mensch-gegen-Computer und (optional) Computer-gegen-Computer möglich sind**
- **Implementiere jetzt eine ausgefeiltere Spielstrategie, z.B. mit Hilfe einer guten Heuristik**
- **Implementiere jetzt den MiniMax-Algorithmus um die Spielzüge vorauszuberechnen**

HABT SPASS!!!

Und meldet euch bei den Hiwis bei Problemen