Reversi: Suche und GUI

Plenum Programmierpraktikum

2006-12-21

Probeturnier

- Bis Ende des Jahres: Wir stellen offiziellen Server zur Verfügung.

- Montag, 8.1.2007: Erstes freiwilliges Turnier.
  - Wer teilnehmen will: Bis Freitag davor dem Tutor eine JAR-Datei geben.
  - Zeitlimit: $\leq$ 60 Sekunden.

Forums-Thread

- Änderungen am PlayerInterface stören nicht bei der Aufgabe, sie $sind$ die Aufgabe. Reagieren auf neue Erkenntnisse ist praxisnah und wichtig (und kann auch in Zukunft noch vorkommen).
  - Auch XP setzt auf inkrementelles Entwickeln mit Feedback und nicht auf den einmaligen $großen\ Wurf$.

- Das Interface selbst kann beliebig geändert werden (Exceptions werfen lassen etc.), so lange die Server- und Client-Implementierungen das Protokoll einhalten.

- Lösung in anderen Programmiersprachen als in Java: gerne, dann wird aber ausser Konkurrenz mitgespielt. Zudem müsst ihr mit Eurem Tutor abklären, wie und ob er Eueren Spieler starten kann.

Zeitbeschränkung: Iterative Deepening

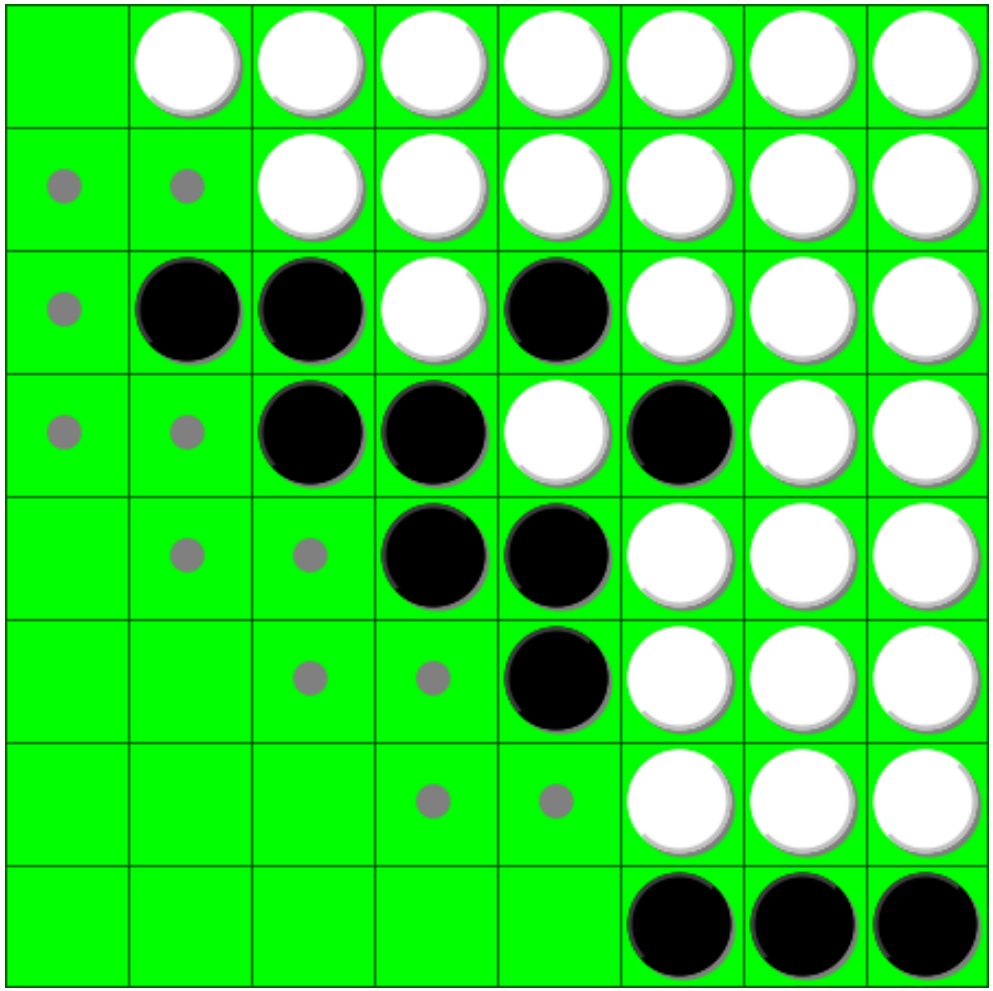
- Pro Durchlauf wird der bisherige Baum um eine Ebene verlängert.
- Lege momentan bestes Ergebnis an einer geteilten Stelle ab.
- Ein separater Thread wacht nach dem Zeitlimit auf und liefert dieses Ergebnis an den Server zurück.

Bewertungsfunktion

Mehr Kunst als Wissenschaft. Nutze Beobachtungen über gutes Spielen:

- Stabilität: $Stabil$ sind Steine, die nicht mehr umgedreht werden können.
- $Mobilität$ sagt aus, wie viele Zugmöglichkeiten ein Spieler hat.

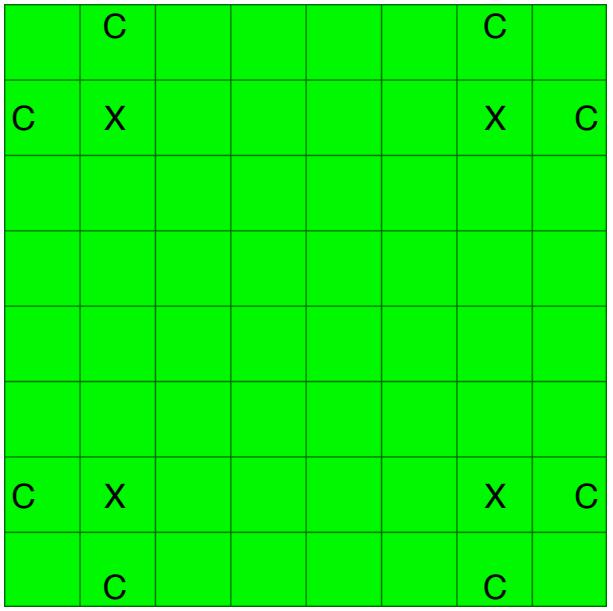
Stabile Steine und Mobilität



- Weiss hat viele stabile Steine.
- Schwarz hat geringe Mobilität.

X- und C-Felder

Stabile Steine wachsen von der Ecke her.
⇒ X- und C-Felder sind wichtig.
⇒ Kanten sind wichtig.



Bewertungsfunktion

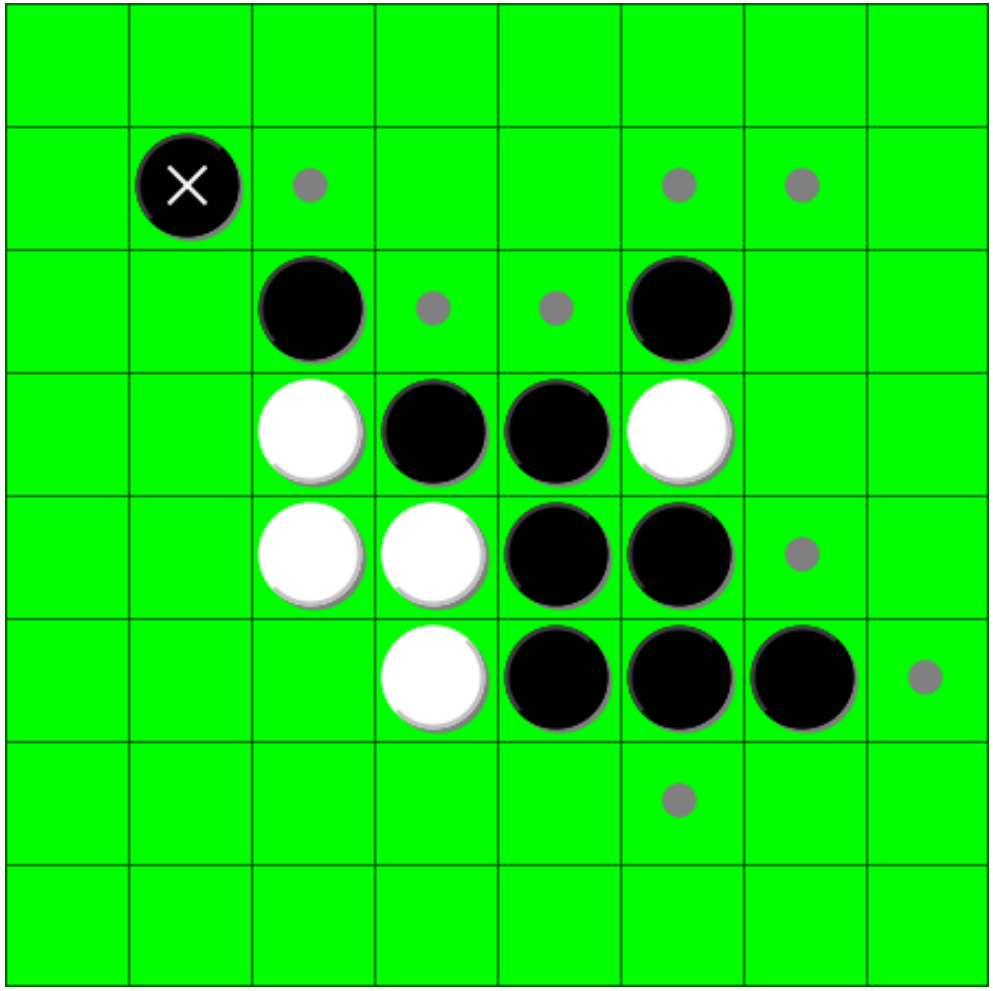
Ideen:

- Kantenbewertung durch Tabellen.
- Mobilität bewerten $\Rightarrow$ Front mitberechnen.

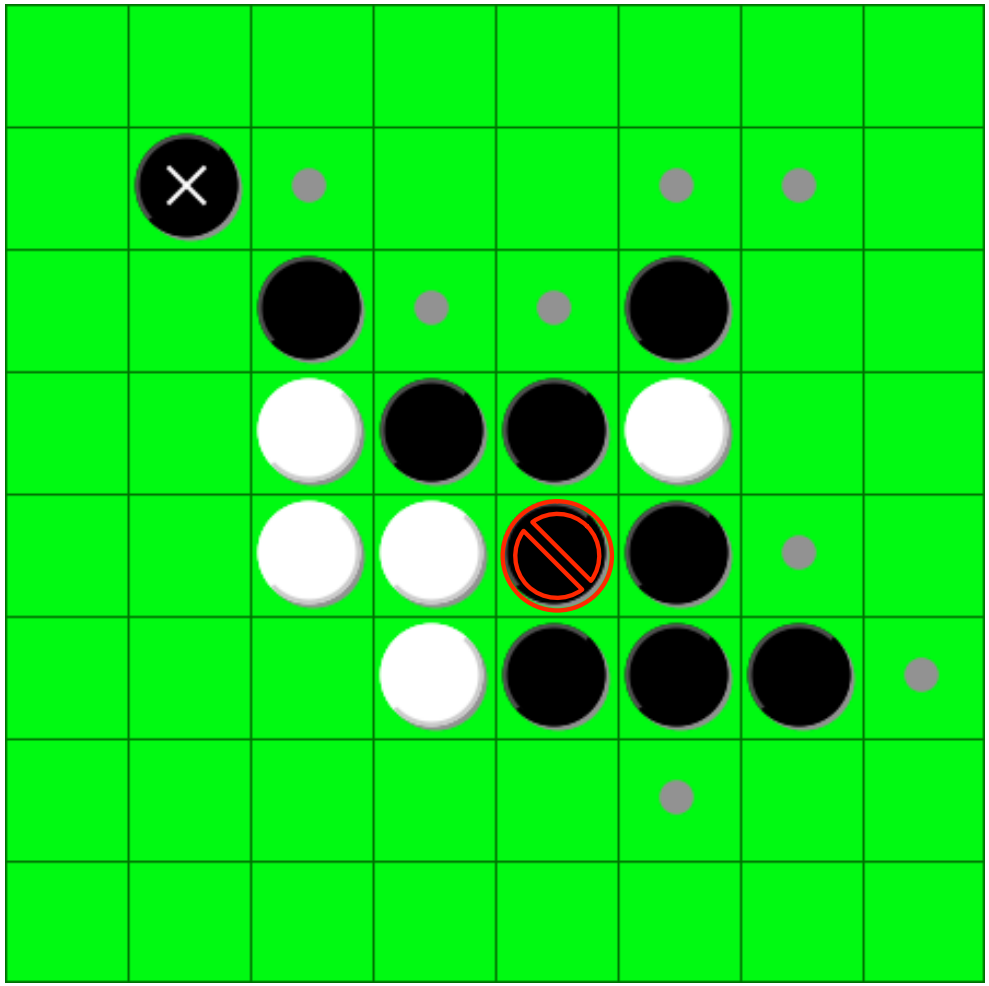
Front

- Man kann neue Felder nur über Steine hinweg erreichen.
- Front-Stein: Einer der 8 Nachbarsteine ist leer.
- Zweck:
  - Zur schnellen Berechnung legaler Züge: Nur Nachbarn $gegnerischer$ Steine müssen überhaupt angesehen werden.
  - Approximation für Mobilität.

Was ist die Front hier?



Front: alle bis auf einen Stein



Design

- Klassen verkörpern (*reifizieren*) konzeptuelle „Dinge“.
- Data-driven versus Responsibility-driven Design: Klassen können als Komponenten (Dienst-Erbringer) oder als Datenbehälter gesehen werden.
- Übliche Methode: Problem als Text beschreiben. Nomen werden Klassen, Adjektive werden Attribute, Verben werden Methoden.
- Mit der Zeit bekommt an ein inutitives Gefühl, was eine Klasse werden sollte und was nicht. Design-Patterns helfen manchmal als Vorbild.

CRC-Cards

- Responsibility-Driven Design-Methode.
- Classes, Responsibilities, Collaborations
- Pro Klasse: CRC auf eine Karteikarte schreiben.
- Im Team Szenarien durchspielen (jeder übernimmt eine Klasse).

Klasse	
Responsibilities	Collaborations

Coding

Enums

`java07`: package `enums`

- Mit Enums bekommt man statisch überprüfte Indizes und Werte-Sammlungen.

- Standard-Konvertierung
  - Von String: `Enum.valueOf(str)`
  - Nach String: `enumValue.getName()`, `enumValue.toString()` [Unterschied?]
  - Von int: `Enum.values()[myInt]`
  - Nach int: `enumValue.ordinal()`

- Kompaktes Speichern von Enums: `java.util.EnumSet`

Menüs

- Cross-platform Menü-Taste: Control unter Windows und Linux, Command auf dem Mac. Lösung:

```
Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()
```

- Sehr wiederholungsreicher Code.

`java07`: menus.Menus

Klassen versus Instanzen

Klassen als Recheneinheit sind $unter\ Java$ eher unpraktisch.

- Man kann keine Interfaces für Klassen definieren.
- Die Daten sind nur einmal vorhanden (je nach Anwendung ist das positiv oder negativ).
- Deshalb wird das Singleton-Pattern gerne verwendet.

Klassen versus Instanzen

`java07`: package `class_instance`

Eine Public-Variable ist *an sich* nicht schlecht! Grund für die Private-Regel: Überwachung beim Setzen des Werts (JavaBeans!), leichte Änderung der Wahl zwischen berechneten und gespeicherten Werten. Letzterer Grund ist mit Refactoring sehr viel weniger wichtig.

Abstrakte Oberklassen

Abstrakte Oberklassen sind wie ein Template, das man ausfüllen muss. Durch Schlüsselwörter wie `abstract`, `final` und `protected` kann man festlegen, was die Ausfüllenden dürfen und was nicht.

`java07`: package `inheritance`

Sonstige Hilfsmethoden

- Wichtig für Unit-Tests: Methode `equals()`.
- Wer `equals()` überschreibt, muss auch `hashCode()` überschreiben: `http://www-128.ibm.com/developerworks/java/library/j-jtp05273.html`

`java07`: package `util`

Aufgaben

User-Stories

- Ant-Script erzeugt eine JAR-Datei `reversi-XX.jar` (XX = Gruppennummer), Hauptklasse ist der automatische Spieler.
- Server
  - Erzwingt wahlweise(!) eine Zeitbeschränkung in Millisekunden.
- Grafische Benutzeroberfläche für den manuellen Spieler.
- Automatischer Spieler.
  - Hält sich an Zeitbeschränkung.
- Wer will: Screencast mit der GUI (bis 16. Januar)

Was ist erlaubt?

- Im Zweifelsfall: nachfragen.
- Erlaubt: Vorausberechnen und ablegen in Dateien (die bei Spielstart gelesen werden).
- Verboten: alles, was den aktuellen Rechner verlässt (abgesehen von Server-Antworten). Beispiele:
  - Verteiltes Rechnen.
  - Denial of Service Attacken.

Links

- Man kann Java-Programme auf dem Mac sehr einfach wie eine normale Anwendung aussehen lassen (inkl. Verstecken der JAR-Datei hinter einer echten Anwendung): http://developer.apple.com/documentation/Java/Conceptual/Java14Development/07-NativePlatformIntegration/NativePlatformIntegration.html

- Englischer Wikipedia-Eintrag: Verweis auf kleine Reversi-Bücher.

- Deutscher Wikipedia-Eintrag: Genauere taktische Ausführungen.

- CRC-Cards: http://www.dfpug.de/konf/konf_1998/02_oop/d_crc/d_crc.htm