Lobby-Modus

Plenum Programmierpraktikum

2007-01-09

Anmerkungen zum Turnier

- Gruppennummer ($\neq 00$!) im Namen.
- Immer den gleichen Namen wählen (sonst kann man sich nicht mehr rückmelden).
- Namen vor dem Einloggen vom Benutzer anfragen.
- Eine GUI nimmt dem automatischen Spieler Zeit weg!
- Wenn man Ant zum Erstellen von JAR-Dateien verwendet haben die Exceptions manchmal keine Zeilennummer. Lösung:

```
<javac debug="true" debuglevel="lines,source" .../>
```

Monitor

- Weiterer Betriebsmodus des Reversi-JARs (bis heute abend zum herunterladen).
- Online: beobachten des Turniers.
- Offline: „nachspielen“ eines alten Turniers anhand eines sogenannten „Line-Logs“.

Turnierergebnisse

Lobby-Modus: Motivation

- Bisher: Menge automatischer Spieler wird von einem Server kontrolliert. Nach den Anmeldung kann man sich nicht mehr umentscheiden (Fokus auf Reconnect).

- Wunsch: Mit anderen, vor allem menschlichen, Spielern spielen können.
  $\Rightarrow$ Player hat viel mehr Kontrolle: kann seinen Gegner wählen, die Lobby verlassen etc.

- Feedback erwünscht. Bis nächste Woche Donnerstag sind noch Änderungen möglich.

Lobby-Modus: Protokoll

Überarbeitete Regel-Kennzeichner:

```
public static final String RULES_REVERSI_TOURNAMENT = "Reversi-Tournament";
public static final String RULES_REVERSI_LOBBY = "Reversi-Lobby";
// ...
```

Liste der Spieler übertragen:

```
public void waitingPlayersChanged(String[] playerNames);
```

Lobby-Modus: Protokoll

```
/** Do nothing */
public static final String ACTION_WAIT = "!wait";

/** Leave the lobby. Afterwards the socket will be closed. */
public static final String ACTION_QUIT = "!quit";

/** Signature: play(playerName, [timeout, forceTimeout])  */
public static final String ACTION_PLAY = "!play";

public LobbyAction getLobbyAction();
```

- Spieler kann selbst den Timeout bestimmen oder die vom Server gewählten Werte übernehmen.
- Implementierungs-Varianten: Für jede Action eine extra Klasse oder eine einzige Klasse für alle Actions.

Eingesendete Fragen

Schleifen: Was ist besser?

```
do {
    playerName = getName();
    matcher = LEGAL_NAME.matcher(playerName);
} while (!matcher.matches());
```

.....

```
while(true){
    playerName = getName();
    matcher = LEGAL_NAME.matcher(playerName);
    if (matcher.matches()) {
        break; // decision=true
    } else {
        System.out.println("Name ungültig");
    }
} // while (!decision)
```

Schleifen: Kriterien

- Hauptkriterium: Ist der Code möglichst nahe an meinem mentalen Modell? Ist er einfach zu verstehen?
  - Beispiel: `break` innerhalb einer Schleife, wenn man denkt: „Wenn das nicht klappt, dann muss ich abbrechen.“. Aber: je größer der Abbruchblock in der Schleife wird, desto mehr leidet die Übersichtlichkeit.
- Manchmal erhöhen zusätzliche Variablen das Verständnis.
  - Beispiel: tief geschachtelte Ausdrücke.
- Manchmal erschweren zusätzliche Variablen das Verständnis.
  - Beispiel: Schleifen-Variante mit Abbruchvariable `decision`.
  - Beispiel (nächste Folie): Wenn die Kontrollfluss komplizierter wird.

Komplizierterer Kontrollfluss *mit* Variablen

Besser lesbar trotz Duplizierung. Ähnliche Szenarien gibt es bei `return`.

```
if (hasX) then {
    label.setText("x");
} else {
    label.setText("");
}
```

```
String text; // oder: Default "", unten nur then-Fall
if (hasX) then {
    text="x";
} else {
    text="";
}
label.setText(text);
```

Schleife bis es keine Fehler mehr gibt

```
int number;
while (true) {
    try {
        String s = JOptionPane.showInputDialog("Zahl:");
        number = Integer.parseInt(s);
        break;
    }
    catch (NumberFormatException e) {
        System.out.println("Das war keine Zahl!");
    }
}
System.out.println("Eingabe: "+number);
```

- Cancel wird nicht behandelt!
- Kann man das besser machen?

Schleife bis es keine Fehler mehr gibt

- Lösung ist OK, auch wenn hier ungewöhnlich gearbeitet wird: Bei Fehlern wird weitergemacht, sonst abgebrochen.

- Problem: bei einem Ergebnis soll nichts weiteres mehr ausgeführt werden. Die Lösung ist ähnlich „unkonventionell“ wie das vorhergehende Beispiel.

```
try {
    performFirst();
    performSecond();
    performThird();
} catch(ResultException re) {
    // verarbeite Ergebnis
}
```

Anweisungen bei Fehlern nicht ausführen

```
try {
    closeGame(field.createPieceColor(params[0]));
    writeToCom(VOID_RESPONSE); // nicht bei Exception
    askForReplay(); // nicht bei Exception
} catch (InvalidColorException e) {
    System.out.println(e.getMessage());
}
```

- Einzig sinnvolle Lösung.
- Wenn der Block zu lange wird: Methoden extrahieren.
- Bei Exceptions: Denken in Transaktionen (Einheiten unteilbarer, zusammengehöriger Funktionalität).

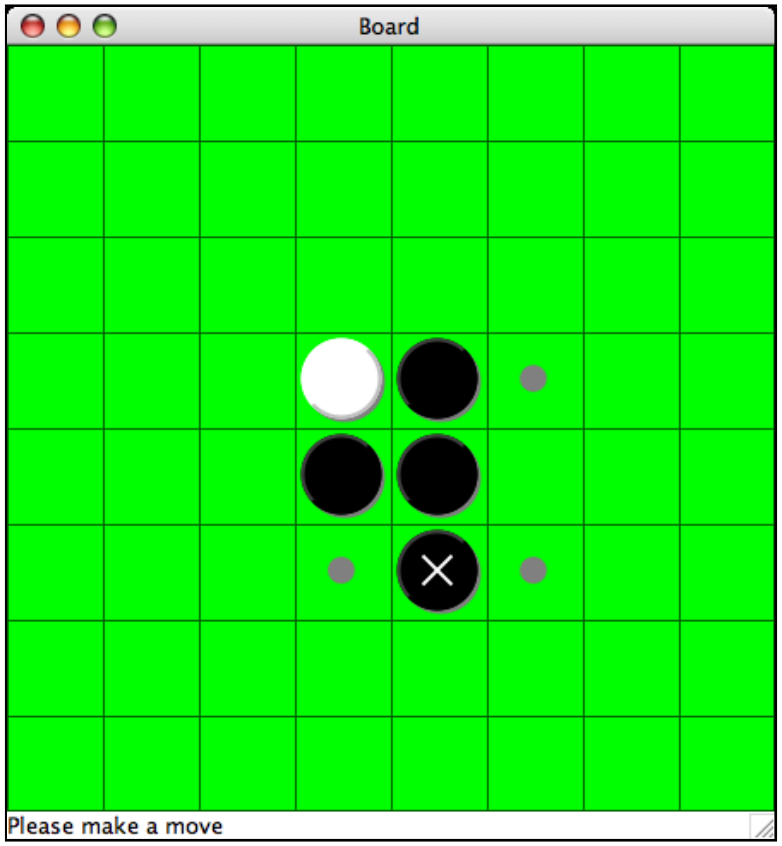
Code-Formattierung

- Code formattieren in Eclipse. Damit: einheitlich, lesbar. Aber: nicht immer passt, was Eclipse macht.
  - Auswirkung beschränken: Ist Text ausgewählt wird nur dieser umformattiert.
  - Light-Version: Control-I nur zum richtigen Einrücken.

- Whitespace: nicht zu viel, nicht zu wenig.

Tipps und Ideen

GUI-Ideen



Zeige an: letzter Zug des Gegners, Zugmöglichkeiten.

Multithreading

Der Zugriff auf geteilte (*shared*) Daten muss synchronisiert werden.

- Datencontainer, in dem jede öffentliche Methode `synchronized` ist.
- Oder: bei jedem Zugriff auf eine Datenstruktur `list` ein `synchronized(list)` um die entsprechenden Anweisungen wickeln.
- Elegante und robuste Alternative zum shared State: Daten kopieren (z. B. über eine Queue).

Multithreading

Vermeide `catch(Exception)`: Man sollte nur auf Dinge reagieren, die man vorhergesehen hat.

- `finally` hilft, dass selbst bei geworfenen Exceptions kritische Anweisungen aufgeführt werden (siehe unten).

- Ausnahme: Bei fremdem und unkontrollierbarem Code kann man mit einem allgemeinen Abfangen von Exceptions für höhere Robustheit sorgen.

```
writeLock();
try {
    // ...
} finally {
    writeUnlock();
}
```

Multithreading und Swing

- Bei Netzwerkanwendungen wird es offensichtlich, dass man eine Swing-Awendung explizit als parallele Anwendung planen muss.

- Swing-Code komplett als separaten Thread betrachten.

Anwendungsthread läuft parallel zum Swing-Thread:

```
firstStep();
signal.receive();
// Menükommando "Next Step" => signal.send()
nextStep();
```

Varianten: Eine BlockingQueue verwenden, Menükommando vor dem receive() disablen.

Continuations

- Bei Continuations setzt der Aufgerufene aktiv die Tätigkeit fort, der Aufrufer übergibt ihm, was zu tun ist.

- Continuations sind in funktionalen Sprachen weit verbreitet, oft als Konstrukt: `doSomething(arg, successBlock, failureBlock)`.

- Das `invokeLater`-Argument ist eine Continuation.
  - Es ist nicht einfach, dass man der Continuation die richtigen Daten mitgibt und für korrekte Synchronisierung sorgt.

Beispiel: Continuations in JavaScript

Synchrone Lösung (warten):

```
result = invokeServer(5);
doSomethingWith(result);
doSomethingElse();
```

Asynchrone Lösung (weitermachen, wenn fertig):

```
result = invokeServer(5, function(result) {
    doSomethingWith(result);
    doSomethingElse(); // !!!
});
```

Tipp: Swing und HTML

```
JLabel label = new JLabel();
label.setText("<html>This is <b>bold</b> text</html>");

pane = new JEditorPane();
pane.setContentType("text/html");
pane.setEditable(false);
pane.setText("This is <b>bold</b> text");
```

Was klappt alles? Fast das komplette HTML 3.2! Beispiele:

- Bold, italics, underlined.
- Schriftfarbe (`<font>`-Tag).
- JEditorPane: Links, reagieren auf Klick.

Abgabe

- Termine: Siehe Programmierpraktikums-Homepage.