Praktikum „SEP: Java-Programmierung“ SS 2019

Game of Life: Typische Fehler

Thomas Bunk und Karlheinz Friedberger

Probleme

- Kompilierung
- Interface
- JUnit-Test

Stil: Zeilenbreite und Tabs

- Erlaubte Zeilenbreite: Maximal 100 Zeichen
- Ab und zu mal 110 Zeichen ist noch OK
- Alle Abgaben erfüllen diese Regel :-)

- **Aber:** Tabulator immer noch verwendet
- **Tabs sind verboten**

Performance: String-Konkatentation

- Schlecht: Ständig umkopieren

```
String s = "";
for (int i = 0; i < getColumns(); i++) {
  for (int j = 0; j < getRows(); j++) {
    s += getCell(i,j);
  }
  s += "\n";
}
return s;
```

- Besser: (mutable) StringBuilder statt (immutable) String:

```
StringBuilder sb = new StringBuilder();
for (int i = 0; i < getColumns(); i++) {
  for (int j = 0; j < getRows(); j++) {
    sb.append(getCell(i,j));
  }
  sb.append("\n");
}
return sb.toString();
```

Verletzung der Zuständigkeiten

- Verwendung der Grid-Implementierung direkt im Modell:

```
// Anmerkung: Board implements Grid
private static Board grid = new Board();
```

- Besser: nur Grid verwenden

```
private static Grid grid = new Board();
```

- Warum soll das besser sein?

Verletzung der Zuständigkeiten

- Verwendung der Grid-Implementierung direkt im Modell:

```
// Anmerkung: Board implements Grid
private static Board grid = new Board();
```

- Besser: nur Grid verwenden

```
private static Grid grid = new Board();
```

- Warum soll das besser sein?
  $\rightarrow$ Verwendung des Interfaces führt zu
  - leichter Austausch der Implementierung
  - Modularität
  - keine Verletzung der gegebenen Interfaces durch Compiletime-Check

Lesbarkeit

Warum kompliziert...

```
if (xsize < xnew) {
  xval = xsize;
} else {
  xval = xnew;
}
```

...wenn es auch einfache geht:

```
xval = Math.min(xsize, xnew);
```

→ kürzerer Code bzw. Nutzung von vorhandenen Hilfmethoden

- führt meistens zu weniger Fehler
- ist oftmals einfacher zu debuggen und warten
- ist oftmals besser strukturiert und zu lesen

Lange Methoden

```
public static void main(String[] args) {
  InputStreamReader input = new InputStreamReader(System.in
  BufferedReader stdin = new BufferedReader(input);
  execute(stdin);
}

private static void execute(BufferedReader stdin) {
  try {
    boolean running = true;
    Grid grid = new Board();
    while (running) {
      System.out.print("gol ");
      String input = stdin.readLine();
      ...
      // 400 Zeilen:
      // keine Hilfsmethoden, Fehlerbehandlung redundant,..
  }
```

Probleme mit der Eingabe

- Leerzeichen in beliebiger Kombination
  - Testfall: Eingabe eines einzelnen Leerzeichens + Enter (Beachte Verhalten von String.split)
- EOF (Schließen der Eingabe) behandeln
  - Testfall: Strg+d
  - einzige sinnvolle Behandlung: Beenden des Programs
- Ziffern zum Parsen:
  - Regex `[0-9]+` ungenügend!
  - Besser `Integer.parseInt` verwenden.
  - NumberFormatException behandeln!

Beispiel mit diversen Problemen

```
/**
 * Durchläuft Array und sobald es eine living cell findet cout +1 .
 */

public void countNeighbor() {
```

Probleme:

- Unnötige Leerzeile zwischen JavaDoc und Signatur
- Dokumentation veröffentlich Interna (Array!)
- Bitte entweder Deutsch oder Englisch, nicht beides!
- Mehrere Rechtschreibfehler
- Methode im Board public sichtbar $\rightarrow$ unnötige Verletzung des Interfaces Grid

Konstanten

```
private boolean cellStatusNextRound(int col, int row) {
  int numAlive = numAliveNeighbours(col, row);
  Cell cell = getCell(col, row);
  if (cell.isAlive() && (numAlive == 2 || numAlive == 3)) {
    return true;
  }
  if (!cell.isAlive() && (numAlive == 3)) {
    return true;
  }
  return false;
}
```

In dieser Lösung sind alle Konstanten nur in einer Methode verwendet. Das ist in Ordnung. Ansonsten wären Konstanten eine gute Wahl.

True/False

```
if (cells[i][j].getAlive() == true) {...}
```

→ Vergleiche mit TRUE/FALSE sind unnötig!

```
if (cells[i][j].getAlive()) {...}
```

Komplexität

```
private static final Shape[] SHAPES = new Shape[] {...};
public Shape getShapeByName(String name) {
  for (int i = 0; i < SHAPES.length; i++) {
    if (SHAPES[i].getName().equals(name)) {
      return SHAPES[i];
    }
  }
  return null;
}
```

- besser: for-each-Loop
- noch besser: `Map<String, Shape>` mit Zugriff in O(1)