Praktikum „SEP: Java-Programmierung“ SS 2019

Trie: Typische Fehler

Thomas Bunk und Karlheinz Friedberger

Stil: Fallunterscheidung

- Tief geschachtelte `if-else`-Kaskade:

```
if (command.startsWith("a")) {
    ...
} else {
  if (command.startsWith("b")) {
    ...
  } else {
    if (command.startsWith("c")) {
      ...
    } else {
      ...
    }
  }
}
```

Stil: Fallunterscheidung

- Gute `if-else`-Kaskade:

```
if (command.startsWith("a")) {
  ...
} else if (command.startsWith("b")) {
  ...
} else if (command.startsWith("c")) {
  ...
} else {
  ...
}
```

Stil: Fallunterscheidung

- Noch besser:

```
switch (command.toLowerCase().charAt(0)) {
  case 'a':
    ...
    break;
  case 'b':
    ...
    break;
  ...
  default:
    ...
    break;
}
```

Stil: Magic Numbers

- Überall magische Zahl 26:

```
Node[] children = new Node[26];
...
for (int i = 0; i < 26; ++i) {
  ...
}
```

- Besser Konstanten verwenden:

```
private static final int MAX_CHILDREN
    = 'z' - 'a' + 1;
Node[] children = new Node[MAX_CHILDREN];
...
for (int i = 0; i < children.length; ++i) {
  ...
}
```

- Weitere Verbesserung: Iteration mit `for-each`

Stil: Zeilenbreite und Tabs

- Erlaubte Zeilenbreite: Maximal 100 Zeichen
- Ab und zu mal 110 Zeichen ist noch OK
- Praktomat-Rekord bei Trie 2019:
  `Shell.java: Maximale Zeilenbreite: 174 Zeichen`
- Praktomat-Rekord bei Trie 2018:
  `Shell.java: Maximale Zeilenbreite: 536 Zeichen`
- Achtung: CheckStyle zählt Tabulator als 8 Zeichen
- **Tabs sind aber ohnehin verboten**

Stil: Ungarische Notation (Systems Hungarian)

- Java ist stark und statisch typisiert
- Typinformation im Bezeichner (sogenannte Systems-Hungarian Notation) ist daher
  - im besten Fall redundant
  - im schlimmsten Fall falsch und irreführend
  - fast immer schlechter Stil
- Maximal verwirrendes Beispiel:

  ```
  int charInt;
  ```

- Sinnlose Bezeichnung:

  ```
  StringBuilder strBuilder = new StringBuilder();
  ```

- Gutes Beispiel: Prefix 'p' bei finalem Parameterwert:

  ```
  public Integer points(final String pKey) {...}
  ```

Performance: String-Konkatentation

- Schlecht: Ständig umkopieren

```
String result = "";
for (Node child : children) {
  if (child != null) {
    result += child;
  }
}
```

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

```
StringBuilder result = new StringBuilder();
for (Node child : children) {
  if (child != null) {
    result.append(child);
  }
}
```

- Noch besser: StringBuilder als Parameter für rekursive (private!) Hilfsmethode

Kontrollfluss: Exceptions

- Schlecht: Ablaufsteuerung durch Exceptions

```
StringBuilder result = new StringBuilder();
for (Node child : children) {
  try {
    result.append(child.toString());
  } catch (NullPointerException e) {
    // child does not exist
  }
}
```

- Stattdessen: **vorher** auf `null` prüfen (siehe vorherige Folie)

Verletzung der Zuständigkeiten

- Ausgabe direkt im Modell:

```
class Trie {
  ...
  public void add(String key, int value) {
    ...
    if (...) {
      System.out.println("Error! "
          + key + " already exists!");
    }
  }
  ...
}
```

  - **Massive Verletzung der Zuständigkeit**
  - Geheimnis der `Shell` verletzt
  - Model dadurch weder wiederverwendbar noch austauschbar

Warum einfach...

Einfach	Kompliziert
`int x = 5;`	`int x = new int[] { 5 }[0];`
`char a = 'a';`	`char a = "a".charAt(0);`

Lange Methoden

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

private static void execute(BufferedReader stdin) {
  try {
    boolean running = true;
    Trie trie = new Trie();
    while (running) {
      System.out.print("trie ");
      String input = stdin.readLine();
      ...
      // 250 Zeilen:
      // keine Hilfsmethoden, Fehlerbehandlung redundar
  }
```

Probleme mit der Eingabe

- deutsche Umlaute, Satzzeichen und Klammern
- Leerzeichen in beliebiger Kombination
  - Testfall: Eingabe eines einzelnen Leerzeichens + Enter (Beachte Verhalten von String.split)
- Ziffern zum Parsen:
  - Regex `[0-9]+` ungenügend!
  - Besser `Integer.parseInt` verwenden.

Beispiel mit diversen Problemen

```
/**
 * Search for the last node representing teh end of a string
 * and change the pionts.
 *
 * @param key String
 * @param newPoints int
 * @return boolean. True if the search and replacement is successfull
 */

public boolean change(String key, int newPoints) {
```

Probleme:

- Parameterdokumentation nutzlos
- Unnötige Leerzeile zwischen JavaDoc und Signatur
- Besser {`@code true`} statt `True` oder `TRUE`
- Rechtschreibfehler

Problem: Unnötiger Code bzw. StringBuilder

```
public String toString() {
  String output = root.toString();
  return output;
}
```

```
String out = "";
for (Node child : children)
  if (child != null) out += child.toString();
return out;
```

Redundanz

```
public Integer points(String key) {
  if (trie.find(key) == null) {
    return null;
  }
  if (trie.find(key).getPoints() == null) {
    return null;
  }
  return trie.find(key).getPoints();
}
```

---

besser:

```
public Integer points(String key) {
  Node child = trie.find(key);
  return (child == null) ? null : child.getPoints();
}
```