Praktikum „SEP: Java-Programmierung“ WS 2018/19

Merkle Tree: Typische Fehler

Thomas Lemberger und Martin Spießl

Style

- Codestyle einhalten!
- u.a.:
  - Reihenfolge von Konstanten und Attributen
  - Konstruktoren vor anderen Klassenmethoden
  - innere Klassen nicht verstreuen

Aussagekräftiges Javadoc

```
/**
 * Create a mutable MerkleTree. // Javadoc of constructor
 * @param <V> for several values // Better: <V> type of values
 */
public class MutableMerkleTree<V> implements HashTree<V> {

/**
* Returns the mutable tree.
*
* @return // don't leave tag empty! (here: remove it)
*/
public MerkleTree<V> returnTree() { .. }

/**
* Build new MerkleTree.
*
* @return Return HashTree // Better: returns a HashTree with
                          // the info given to this builder
*/
public HashTree<V> build() {
```

SuppressWarnings

```
@SuppressWarnings({"unchecked", "rawtypes"})
public final class Shell {
  //...
  // Raw type
  MutableMerkleTree tree = new MutableMerkleTree(..);
  //...
}
```

- SuppressWarnings wollte gezielt eingesetzt werden, nicht auf Klassen-level
- Kommentar mit Erklärung, warum das nötig ist, ist hilfreich
- Hier: Fehleranfälliges Programmieren, vor dem nicht gewarnt wird

catch (NullPointerException)

```
try {
  //...
  input = std.readLine();
  if (input == null) {
      throw new NullPointerException();
  }
} catch (NullPointerException e) {
  printError("You have to enter something");
}
```

- `input == null` wenn Nutzer abbricht, nicht wenn Eingabe leer (dann leerer String)
- Echte NullPointerExceptions von anderen Stellen werden geschluckt

$\Rightarrow$ Schwer zu debuggen

e.printStackTrace()

```
try {
  //...
} catch (SomeException e) {
  e.printStackTrace();
}
```

- Das wollen wir **nie** sehen
- Verwirrende Informationen für Programmnutzer
- Keine sinnvolle Behandlung—kann nach außen geworfen werden

Falsches Verhalten schlucken

```
class UnmodifiableMerkleTree<V> implements MerkleTree<V> {
  public void setHash(int idx, long hash) {
    // do nothing and never tell anybody about it
  }
}
```

- Wenn der Programmierer diese Funktion aufruft, erwartet er, dass ein Hash gesetzt wird (oder will es zumindest)
- Er sollte darauf hingewiesen werden, dass das nicht möglich ist/etwas an der Programmierung falsch sein muss

```
class UnmodifiableMerkleTree<V> implements MerkleTree<V> {
  public void setHash(int idx, long hash) {
    throw new UnsupportedOperationException(
        "Unmodifiable tree doesn't support this");
  }
}
```

Booleans vs. Enums

```
public final class Shell {
  boolean start = true;
  boolean build = false;
  boolean check = false;
  // ...
  private void changeToBuildMode() {
    start = false;
    build = true;
    check = false;
  }

  private void processInput() {
    if (start) {
      processBaseInput();
    } else if (build) {
      processBuildInput();
    } else if (check) {
      processCheckInput();
    } else {
      throw new AssertionError("Unhandled mode");
    }
  }
```

- Theoretisch möglich: Zwei Werte auf true.

→ Welcher Modus hat Vorrang?

- Schlecht erweiterbar um neue Modi
- Besser: Enums oder *State Pattern* (siehe Folienende)

Integer vs. Enums

```
private static final int STATE_START = 0;
private static final int STATE_BUILD = 1;
private static final int STATE_CHECK = 2;

private int state = START_STATE;

private void processInput() {
switch(state) {
    case STATE_START:
    /...
  }
}
```

- Nur ein Modus auf einmal, aber viele ungültige Werte möglich
- Bei vielen Zuständen ist Prüfen von „Doppelbelegung“ aufwendig und fehleranfällig
- Besser: Enums oder *State Pattern* (siehe Folienende)

Integer vs. Boolean

```
private int isOperationSuccess(..) {
  if (success) {
    return 0;
  } else {
    return -1;
  }
}
```

- Unsicherheit: `0 == false` oder `0 == true`?
- `boolean` trägt automatisch mehr Semantik
- Kein Verlass darauf, dass bei Erweiterungen nicht zusätzliche Werte eingeführt werden, die dann im Frontend nicht behandelt werden

While(true)

```
while(true) {
    providePromptAndProcessInput();
}
```

- Schleife kann nur durch Exception oder Programmabbruch verlassen werden
- Besser: quit-Variable, die Bedingung trägt

Funktionalität im Konstruktor

```
public Shell() {
  boolean quit = false;
  while (!quit) {
    quit = providePromptAndProcessInput();
  }
}
```

- Semantik des Konstruktors: „Erstelle ein Shell-Objekt“
- Objekt wird nie vollständig erstellt
- Beinhaltet zusätzliche Logik
- Verhindert andere Wiederverwendung: Weder Konstruktoren/Vererbung noch Komposition möglich

Redundante Ausführung von Methoden

```
private void ready() {
  if (tree.getMissing().isEmpty()) {
    //...
  } else {
    String result = tree.getMissing().toString();
    //...
  }
}
```

- Returnwert von `tree.getMissing()` könnte sich zwischen Abfrage und Benutzung ändern
- Unnötig teuer. Besser:

```
private void ready() {
  T nodesThatMissInfo = tree.getMissing();
  if (nodesThatMissInfo.isEmpty()) {
    //...
  } else {
    String result = nodesThatMissInfo.toString();
    //...
  }
}
```

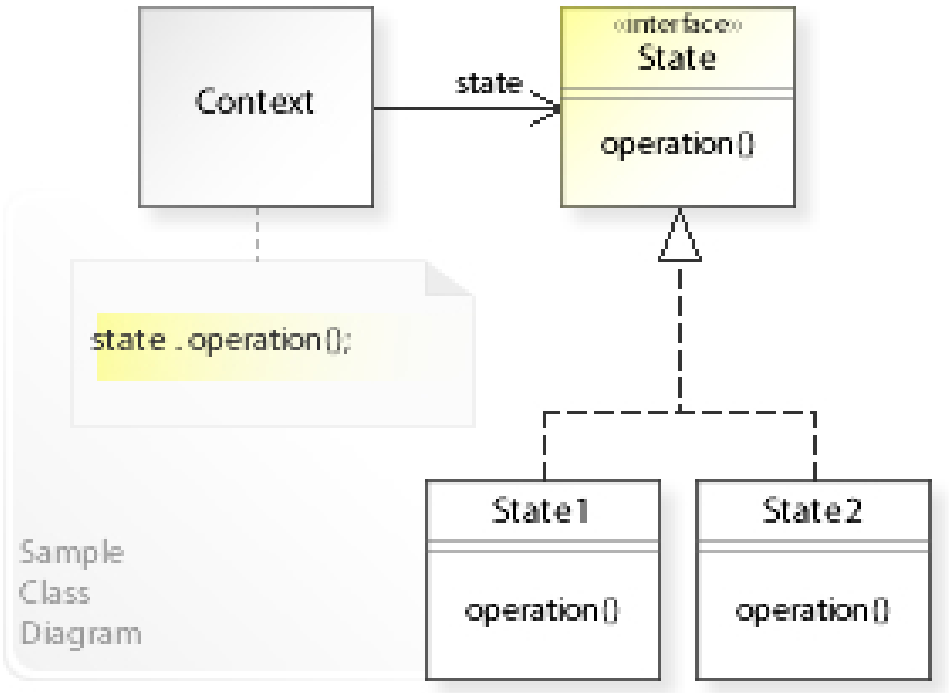
`equals(Object o)` und `hashCode()`

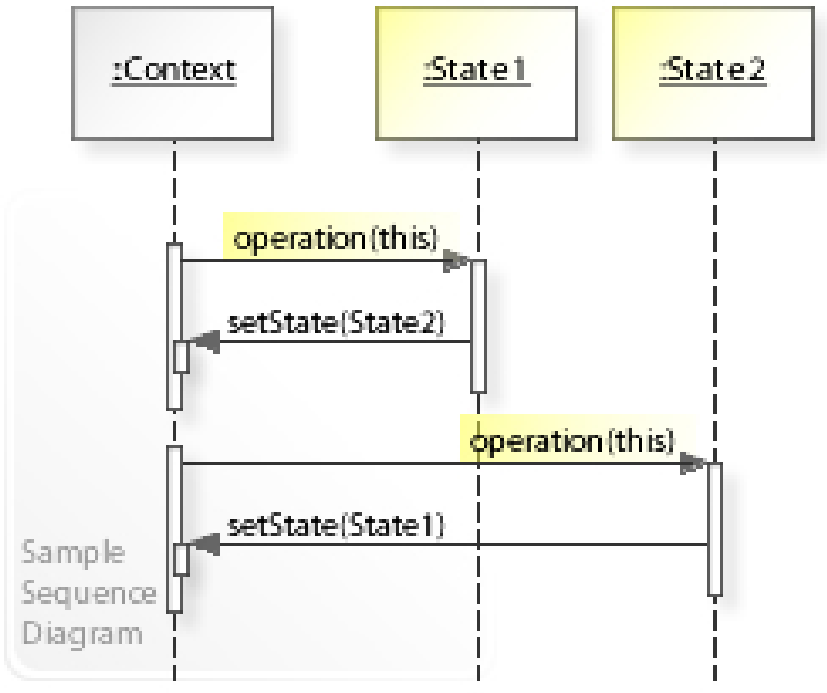
- Zu beachten bei `equals(Object o)`
  1. Für `b == null` gilt, dass `a.equals(b)` niemals wahr ist
  2. `equals(Object o)` muss reflexiv sein,
     d.h. `a.equals(a)` ist immer wahr
     (außer wenn `a == null` wegen `NullPointerException`)
  3. `equals(Object o)` muss symmetrisch sein,
     d.h. `a.equals(b) == b.equals(a)` ist immer wahr,
     außer `a == null` oder `b == null`
     (wegen `NullPointerException` und Regel 1)
  4. Außerdem transitiv und konsistent
- `hashCode()` muss **immer** konsistent zu `equals(Object o)` sein
  - wenn `a.equals(b)`,
    dann gilt `a.hashCode() == b.hashCode()`
  - Die Umkehrung ist aus Effizienzgründen erstrebenswert,
    kann aber nicht immer gelten
    (Taubenschlag-/Schubfachprinzip)

Sinnvolle Variablennamen ;-)

```
private BodyBuilder arnold = new BodyBuilder();
```

Ergänzung: State Pattern





- Spezialform des *Strategy*-Pattern

Ergänzung: State Pattern I

```
public class Shell {
  private ShellState currentState = new StartState(this);
  //...
  private void providePromptAndRead() {
    while (!quit) {
      // String[] tokens, ...
      try {
        quit = currentState.handleInput(tokens);
      } catch (UnknownCommandException e) {
        printError(...);
      }
    }
  }
}

interface ShellState {
  boolean handleInput(String[] pInputTokens)
    throws UnknownCommandException;
}
```

Ergänzung: State Pattern II

```
class StartState implements ShellState {
  private Shell shell;

  public StartState(Shell pShell) { shell = pShell; }

  @Override
  public boolean handleInput(String[] pInputTokens)
      throws UnknownCommandException {
    switch (pInputTokens[0]) {
      case 'new': // necessary commands ...
        shell.setState(new BuildState(shell));
        return false;
      case 'new_check': // necessary commands ...
        shell.setState(new CheckState(shell));
        return false;
      case 'quit':
        return true;
      default:
        throw new UnknownCommandException();
  }} // } for method and class
```

Ergänzung: State Pattern III

```
class BuildState implements ShellState { // ... }
class CheckState implements ShellState { // ... }
class UnknownCommandException extends Exception { // ... }
```

- Wenn Methode void-return: Neuer State kann in Methode returned werden, anstatt Instanz auf Kontext (Shell) zu halten
- Doppelter Code für gemeinsame Kommandos kann noch durch Vererbung, Komposition oder *Decorator*-Pattern gelöst werden

```
public boolean handleInput(String[] pInputTokens)
    throws UnknownCommandException {
  try {
    return startState.handleInput(pInputTokens)
  } catch (UnknownCommandException e) {
    return handleInputsSpecificToThisState(pInputTokens);
  }
}
```