Ein Griff in die Scala-Trickkiste

Heiko Seeberger, Roman Roelofsen

WeigleWilczek

JAX, Mainz, 2. Mai 2011

Builder Pattern

Builder Pattern

- API Design zum Setzen mehrere Parameter
- setter-Methode-Ansatz

```
val config = new Config
config.setA(1)
config.enableLog()
config.setB(2)
val created = config.create()
```

- Builder-Pattern-Ansatz

```
val created =
  new Config().setA(1).setB(2).enableLog().create()
```

Builder Pattern

- Methodenaufrufe geben i.d.R. **this** zurück...
- ... oder ein Objekt welches sich die Konfiguration "merkt"
- Manche Konfiguration stehen eventuell erst nach dem Setzen bestimmer Werte zur Verfügung

```
val created = new Config().enableLog().withDbLog().create()
```

Builder Pattern - Implementierung

- Definition

```
class ABad {
  var number = 1

  def double(): ABad = {
    number *= 2
    this
  }
  def triple(): ABad = {
    number = number * 3
    this
  }
}
```

- Verwendung

```
println(new ABad().double().triple().number)
```

Builder Pattern - Implementierung

- Definition

```
class BBad1 extends ABad {
  def random(): BBad1 = {
    number *= Random.nextInt
    this
  }
}
```

- Verwendung

```
println(new BBad1().random().number)
println(new BBad1().random().triple().number)
```

- Problem

```
println(new BBad1().double().triple().random().number)
```

Builder Pattern - Implementierung

- Lösung (wie in Java)

```
class BBad2 extends ABad {
  override def double(): BBad2 = {
    super.double()
    this
  }

  override def triple(): BBad2 = {
    super.triple()
    this
  }

  def random(): BBad2 = {
    number *= Random.nextInt
    this
  }
}
```

Builder Pattern - Implementierung

- Lösung (wie in Java)
  - Alle Methodendeklarationen müssen wiederholt werden
  - Lösung ist sehr fragil, da Änderungen in der Superklasse (z.B. neue Methode) die Kette "unterbrechen" können
  - Alle Subklassen müssen informiert und nachgezogen werden

Builder Pattern - Implementierung

- Lösung mit Scala this.type

```
class AGood {
  var number = 1
  def double(): this.type = {
    number *= 2
    this
  }
  def triple(): this.type = {
    number = number * 3
    this
  }
}
```

Builder Pattern - Implementierung

- Lösung mit Scala this.type

```
class BGood extends AGood {
  def random(): this.type = {
    number *= Random.nextInt
    this
  }
}
```

- Verwendung

```
println(new BGood().double().triple().random().number)
```

Stackable Modifications

- Logging Klassen
  - Basis-Interface: `Log`
  - Implementierungen: `NormalLog`, `DBLog`

Stackable Modifications

- Logging Klassen
  - Basis-Interface: `Log`
  - Implementierungen: `NormalLog`, `DBLog`
- Modifizierungen
  - `ImportantLog` (Text + '!!!')
  - `SuperImportantLog` (Großbuchstaben)

Stackable Modifications

- Logging Klassen
  - Basis-Interface: `Log`
  - Implementierungen: `NormalLog`, `DBLog`
- Modifizierungen
  - `ImportantLog` (Text + '!!!')
  - `SuperImportantLog` (Großbuchstaben)
- Ziel
  - Log-Implementierungen sollen beliebig mit den Modifizierungen erweitert werden können
  - Modifizierungen müssen sich kombinieren lassen
  - Je nach Komplexität kann die Reihenfolge wichtig sein

Stackable Modifications - Log Klassen

```
abstract class Log {
  def log(msg: String): Unit
}

class ConsoleLog extends Log {
  def log(msg: String) {
    println("LOG: " + msg)
  }
}

class DBLog extends Log {
  def log(msg: String) {
    Database.save(msg)
  }
}
```

Stackable Modifications - Modifizierungen

```
trait ImportantLog extends Log {
  abstract override def log(msg: String) {
    super.log(msg + "!!!")
  }
}

trait SuperImportantLog extends Log {
  abstract override def log(msg: String) {
    super.log(msg.toUpperCase)
  }
}
```

Stackable Modifications - Verwendung

```
class AppLog extends ConsoleLog with ImportantLog with
    SuperImportantLog
val log1 = new AppLog
log1.log("A log message")

val log2 = new ConsoleLog with ImportantLog
log2.log("Another message")

val log3 = new DBLog with SuperImportantLog
log3.log("Another message")

val log4 = new DBLog with ImportantLog with SuperImportantLog
log4.log("Another message")
```

Kontrollstrukturen abstrahieren

Aufgabe

(1) Resource öffnen (2) Resource verwenden
(3) Resource schließen (4) Fehler abfangen

Java

```
InputStream is = null;
try {
  is = new FileInputStream(new File("..."));
  is.read();
} catch (Exception e) {
  // ...
} finally {
  try {
    is.close();
  } catch (Exception e1) {
    // ...
  }
}
```

Kontrollstrukturen abstrahieren

Aufgabe

(1) Resource öffnen (2) Resource verwenden
(3) Resource schließen (4) Fehler abfangen

Scala

```
withResource(new FileInputStream(new File("...")))) { is =>
  // ...
}
```

Kontrollstrukturen abstrahieren

Aufgabe

(1) Resource öffnen (2) Resource verwenden
(3) Resource schließen (4) Fehler abfangen

Scala Implementierung

```
def withResource[A <: { def close() }](res: A)(block: A => Any) {
  try {
    block(res)
  } catch {
    case e: Throwable => throw e
  } finally {
    try {
      res.close()
    } catch {
      case e: Throwable => throw e
    }
  }
}
```

Kontrollstrukturen abstrahieren

- Java-Code verwendet oft Exceptions als gültigen Rückgabewert
- Automatische Exception / Option Konvertierung im Scala-Kontext daher sinnvoll

```
import scala.util.control.Exception._

val saveParse = catching(classOf[NumberFormatException])
val numberOption = saveParse opt { "123b".toInt }

numberOption match {
  case Some(n) => n
  case None => 0
}
```