www.novatec-gmbh.de

NOVATEC

Make IT happen!

Effektive Scala Entwurfsmuster

Java Forum Stuttgart 2011

Andreas Tönne, Director R&D, NovaTec GmbH

Übersicht

NOVATEC
Make IT happen!

Einführung in typische Scala Pattern

- Basierend auf ausgewählten Schlüsselfeatures von Scala
- „Abrechnung“ mit traditionellen GoF Pattern
- Beispiele für idiomatisch korrekte Pattern
- Einschließlich Pimp My Library, Cake und Type Classes

Implementierungsskizzen und Erläuterungen

Umfangreiche Beschreibungen und Codebeispiele unter

scala.novatec-gmbh.de

(Ab 7.7.2011, moderierter deutschsprachiger Blog)

Direktor R&D der NovaTec GmbH

Standort Frankfurt am Main


NOVATEC
make IT happen!

NOVATEC // Ingenieure für neue Informationstechnologien GmbH

Friedrich-Ebert-Anlage 36
D-60325 Frankfurt am Main

Diplom-Informatiker
Andreas Tönne
Managing Consultant

phone +49 (0)69 244 333-0
fax +49 (0)711 220 40-899
mobile +49 (0)151 1216 7504
andreas.toenne@novatec-gmbh.de

Entwickler, Berater und Projektmanager seit 1987

25 Jahre Smalltalk

15 Jahre Java

1 Jahr Scala ☺

Design Pattern

NOVATEC
Make IT happen!

Wir alle wissen was Pattern sind?

GoF: Gamma, Helm, Johnson, Vlissides: „Entwurfsmuster“ (Addison-Wesley, 1996)

„Keines der Entwurfsmuster … beschreibt neuartige Entwürfe. (…) Trotzdem wurden viele dieser Entwürfe noch nie dokumentiert. Sie gehören entweder zum Allgemeinwissen … oder sie sind Teil erfolgreicher … Systeme.“

„Die Entwurfsmuster … sind Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidert sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.“

Design Pattern entstehen im Kontext der Möglichkeiten einer Programmiersprache

- Umgehen unerwünschte Eigenarten der Sprache
- Bilden zusammengesetzte wünschenswerte neue Sprachkonstrukte
- *Dokumentieren idiomatisch guten Einsatz der Sprache*

GoF Pattern – Bewertung für Scala

NOVATEC
Make IT happen!

Welche GoF Pattern „passen“ zu Scala?

- Manche Pattern werden „unsichtbar“ durch Scala Sprachmittel ausdrückbar
- Manche Pattern werden zu Anti-Pattern da man das Ursprungsproblem in Scala anders löst

Unsichtbare Pattern

- Besucher, Brücke, Iterator, Prototyp, Singleton, Strategie

Vereinfachte Pattern

- Abstrakte Fabrik, Beobachter, Dekorierer, Fabrik, Interpreter, Proxy, Zustand, Zuständigkeitskette

Anti-Pattern

- Befehl, Schablonenmethode

Sonstige Pattern

- Adapter, Erbauer, Fassade, Fliegengewicht, Kompositum, Memento, Vermittler

Grundlagen neuer Scala Pattern

NOVATEC
Make IT happen!

Jedes neue Scala Feature kann Basis neuer Pattern sein

- Komplexe Designs kompakter als Pattern beschreibbar (anstelle eines Frameworks)
- Höherwertige Abstraktionen über der Programmstruktur und Typen erlauben neuartige Compilezeit Pattern

Scala läuft auf der JVM aber …

- Nicht alle Pattern können 1-1 nach Java übersetzt werden
- Bsp. Cake-Pattern bringt seinen Nutzen zur Compilezeit
- Nicht übersetzbar: Ausnutzen von Modulstruktur und besserem Typsystem

Wir konzentrieren uns auf

- object
- Funktionsobjekte
- Traits
- Implicit Konvertierungen
- Selftype

Scala Feature - Object Definition

NOVATEC
Make IT happen!

Ein object ist ein Modul mit Typen, Werten und Methoden

```
object MyObject {
        val constant:Int = 2
        var variable:String = "Hello World"
        def method(param:Int) = variable * param
}
```

Ähnlich einer Klasse aber mit impliziter Singleton-Instanz

- MyObject.constant, MyObject.variable, MyObject.method(MyObject.constant)
- Ersetzen statische Definitionen in Java Klassen
- Konzept: Companion Object

Singleton Pattern

NOVATEC
Make IT happen!

```
object Singleton {
        val instance= Initialisierungsausdruck
}
```

- Lazy initialisiert
- Threadsicher

Idiomatisch besser: Das im Singleton gespeicherte Objekt direkt als object implementieren

object wird häufig zur Modularisierung eingesetzt

- Scope für Definitionen

Typsichere Parametrische Factory

NOVATEC
Make IT happen!

```
sealed abstract class FactoryItem
case object Type1 extends FactoryItem
case object Type2 extends FactoryItem

object Factory {
        def createItem ( which:FactoryItem):OType = which match {
                case Type1 => new ObjectType1()
                case Type2 => new ObjectType2()
        }
}
```

Factory.createItem(Type2)

- Vermeidung von Vererbung zur Factory-Differenzierung
- Typsicher
- Nachteil: Außerhalb des definierenden Scopes nicht erweiterbar

Scala Feature - First-class Funktionen

NOVATEC
Make IT happen!

Funktionen sind Objekte (First-Class Werte)

- Können in Variablen gespeichert werden
- Als Parameter übergeben werden
- Zur Laufzeit erzeugt werden

Literalsyntax: (arg1,… argN) => Ausdruck

Intern Objekte mit einer Methode apply(parm1, … parmN)

Methoden können als Funktionsobjekte extrahiert werden

Idiomatische Nutzung:

- Siehe andere funktionale Programmiersprachen (und den Scala Blog)
- Ersetzung von Hilfsklassen als Träger von Methoden in Java Pattern
- Definition von neuen Kontrollstrukturen

Iterator Pattern

NOVATEC
Make IT happen!

Iteratoren werden „internalisiert“ und mit dem Iteratorbody parametrisiert

Java:

```
List<String> result = new ArrayList<String>;
for (String elem: list) {
        if (elem.indexOf("World") >= 0)
                result.add(elem)
}
```

Scala:

```
list.filter(x:String =>x.contains("World"))
```

Vorteile:

- Multiple Iteratoren gekapselt in Collections
- Collection-Typ des Resultats wird durch die Collection bestimmt und nicht durch den Aufrufer

Visitor Pattern

NOVATEC
Make IT happen!

Visitor = Iteration über Objektstruktur + Methode für jeden iterierten Elementtyp

Scala Lösung #1: Definiere internen Iterator wie bei einer Collection

- Bsp.: aTree.inorderDo(node:TreeNode => ….)

Scala Lösung #2: Rekursion über Objektstruktur in Pattern-Matching

```
abstract class Expr
case class Num(n: Int) extends Expr
case class Sum(l: Expr, r: Expr) extends Expr
case class Prod(l: Expr, r: Expr) extends Expr

def evalExpr(e: Expr): Int = e match {
        case Num(n) => n
        case Sum(l, r) => evalExpr(l) + evalExpr(r)
        case Prod(l, r) => evalExpr(l) * evalExpr(r)
}
```

Observer Pattern

NOVATEC
Make IT happen!

Ein weiteres Beispiel für die Ersetzung einer Hilfsklasse Observer durch eine Funktion

```
class Event()

trait Subject[S, E<: Event] {
        this:S =>
        type Observer =  (E, S) => Unit
        private var observers:List[Observer] = Nil
        def observe(observer: Observer) =
                observers = observer :: observers
        def notify(evt:E) = observers.foreach((o:Observer) => o(evt, this))
}


class MyEvent(val param:String) extends Eventclass MySubject extends
Subject[MySubject, MyEvent] mySubject.observe((event, subject) =>
println(subject + "notified" + event))
mySubject.notify(new MyEvent("Parameter")
```

Cell Pattern

NOVATEC
Make IT happen!

Idiomatisch: Objekte erhalten Funktionscharakter wenn es eine primäre Methode gibt

Problem: Wir müssen eine Variable per Referenz übergeben

Lösung: Wir definieren ein Value-Objekt mit Funktionsverhalten

```
class Cell[T](var value:T) {
        def apply() = value
        def update(v:T) = value=v
}

val cell = new Cell ("")
cell() = "Hello World"
println(cell())
```

Scala Feature - Traits

NOVATEC
Make IT happen!

Traits sind eine Mischung aus abstrakter Klasse und Interface

- Können Typen, Variablen und abstrakte sowie konkrete Methoden enthalten

Traits sind dazu gedacht, Verhalten zu einer Klassen „hinzu zu mischen“ (Mixin)

class MyClass extends Trait1 with Trait2 with Trait3 {….

Traits überschreiben vorhandene Definitionen in der Reihenfolge Rechts nach Links

- Trait3 überschreibt Trait2 überschreibt Trait1 überschreibt MyClass Superclass

Idiomatisch:

- Traits dienen der statischen Parametrisierung von konkreten Objekten
  Siehe das kommende Decorator Pattern
- Traits dienen der Modularisierung
  Siehe das Cake Pattern

Decorator Pattern

NOVATEC
Make IT happen!

Idiomatisch: Verwendung von Trait Mixins anstelle von Delegation

new A with T1 ergibt "T1 A"
new A with T1 with T2 ergibt "T2 T1 A"

Vorteil: Compilezeit-Operation, gut lesbar, Decorator enthalten nur geänderte Methoden
Nachteil: Decorator-Chain nicht dynamisch änderbar

```
class A {
        override def toString = "A"
}
trait T1 {
        override def toString = "T1" + super.toString()
}
trait T2 {
        override def toString = "T2" + super.toString()
}
```

Scala Feature - Implicit Definitionen

NOVATEC
Make IT happen!

Val-Definitionen und Methodenparameter können als implizit gekennzeichnet werden

- Der Compiler löst Typkonflikte auf, indem er nach passenden impliziten Definitionen sucht
- Für die folgenden Pattern genügt etwas Intuition → Mehr Präzision im Scala Blog

Schwieriges Thema und Kandidat für „Gefährlichstes Feature“ in Scala

Sehr wichtiges Feature für besonders fortgeschrittene Scala-Pattern

Dringende Empfehlung: Nutzung auf festgeschriebene Pattern beschränken!

Idiomatische Nutzung:

- Automatische Typkonvertierung (siehe Adapter Pattern)
- Erweiterung geschlossener Klassen (siehe Pimp My Library Pattern)
- Statisch aufgelöste Dependency Injection (siehe Type Classes Pattern)

Adapter Pattern

NOVATEC
Make IT happen!

Automatische Konvertierung in eine anonyme Adapter-Unterklasse

Keine Veränderung an den Originalklassen

```
class Adapter {
        def someMethod() = ...
}

class Adaptee {
        def differentMethod() = ...
}

val adapter: Adapter = new Adaptee
```

Compiler sucht hier nach impliziter Methode Adaptee=>Adapter

```
implicit def targetAdapteeConv(x:Adaptee):Adapter = new Adapter {
        override def someMethod() = x. differentMethod()}
```

→ val adapter: Adapter = targetAdapteeConv(new Adaptee)

Pimp My Library Pattern

NOVATEC
Make IT happen!

Problem: Wie erweitere ich eine bestehende API ohne explizite Konvertierungen in Extension-Unterklassen, ohne Wrapper und ohne Casts?

Lösung: Eine Variante des Adapter-Pattern auf Methodenebene.

- Eine anonyme Klasse mit der gewünschten Methode X über dem vorhandenen Typ T
- Eine implizite Konvertierung von T zur anonymen Klasse für den Aufruf von X

Kriterien:

- Verwende anonyme Klassen weil es idiomatischer ist und nicht den Namensraum zu müllt

Es folgt ein Beispiel „3 :: 52“ als Ausdruck für eine Calendar Instanz mit Uhrzeit
Dazu müssen wir eine :: Methode der Klasse Integer „unterschieben“

Pimp My Integer

NOVATEC
Make IT happen!

3::52 → Compiler sucht nach einer impliziten Konvertierung von Int zu einem Typ mit ::
Ergebnis: pimp(3)::(52)

```
implicit def pimp(hours: Int) = new {
        def ::(minutes:Int) = {
                var cal = java.util.Calendar.getInstance()
                cal.set(java.util.Calendar.HOUR_OF_DAY, hours)
                cal.set(java.util.Calendar.MINUTE, minutes)
                cal
        }
}
```

Vorteile:

- Einfache, lokale Erweiterung
- Leichtgewichtig: wir erzeugen kleine Instanzen mit einer Methode
- Erweiterung wird durch den Scope kontrolliert

Scala Feature - Selftype (Anhand eines Beispiels)

NOVATEC
Make IT happen!

Selftype ist eine Art Cast von this auf einen gewünschten Zieltyp der konkreten Objekte

```
trait Hello { def hello = "Hello" }

class Test { self: Test with Hello =>          // das ist der Selftype
        val helloWorld = hello + "World"       // hello kommt aus Trait Hello
}

new Test
error: class Test cannot be instantiated because it does not conform to its self-type Test with Hello

val ok = new Test with Hello
ok.helloWorld  → HelloWorld
```

Schlechtere Alternative:

```
class Test extends Hello {  val helloWorld = hello + "World"}
```

Cake Pattern

NOVATEC
Make IT happen!

Problem:

- Dekomposition in Module bei kreuzweiser Abhängigkeit der Module
- Deklarative Komposition des Ergebnis ohne Rückgriff auf Delegation, DI o.ä. Techniken
- Austauschbare Implementierungen der Module

```
var aVariable:Int
def methodWithDep() = dependentMethod(bValue)
```

```
val bValue:Int
def dependentMethod(i:Int) = i*aVariable
```

Dekomposition 1. Versuch

NOVATEC
Make IT happen!

```
trait ModuleA {
        var aVariable:Int
        def methodWithDep() = dependentMethod(bValue)
}

trait ModuleB {
        val bValue:Int
        def dependentMethod(i:Int) = i*aVariable
}
```

Dies ist nicht compilierbar!

Java: Explizite Referenz auf benutzte Module, DI und Delegation

Scala: Composite selftypes und Mixins

Cake Pattern

NOVATEC
Make IT happen!

Cake bildet das modulare Design des Scala Compilers (Komposition verschiedener Module zur Konstruktion des Compiler, der REPL und scaladoc)

Konzept:

- Dekomposition in Traits
- Scope-Abhängigkeit der Module ermöglicht durch zusammengesetzte selftypes
- Komposition in einer Klasse
- Statisch typbar wenn in der Klasse alle Abhängigkeiten aufgelöst sind

→ Als ob alle Traits manuell in ein großes Modul zusammen kopiert wurden

Vorteile:

- Komposition auch bei zyklischen Abhängigkeiten kein Problem
- Keinerlei Overhead oder erhöhte Komplexität durch die Modularisierung
- Einfacher Zusammenbau von Implementierungsvarianten

Cake Pattern Skizze

NOVATEC
Make IT happen!

```
trait ModuleA {
        self:ModuleA with ModuleB =>
        var aVariable:Int
        def methodWithDep() = dependentMethod(bValue)
}

trait ModuleB {
        self:ModuleB with ModuleA =>
        val bValue:Int
        def dependentMethod(i:Int) = i*aVariable
}

class Composite extends ModuleA with ModuleB {
        ....
}
```

Type Class Pattern

NOVATEC
Make IT happen!

Wichtiges Pattern im Scala 2.8 Collection-Design

Problemstellung: Wir müssen verschiedene Klassen mit polymorph nutzbaren Methoden (einem neuen Konzept) ausstatten, ohne die Klassen zu erweitern.

Lösung: Generalisierte Form des Pimp My Library Pattern.

Idee:

- Implementierungen des Konzepts für die verschiedenen Klassen
- Mache diese als implizite neue Klassendefinition verfügbar
- Methoden die das Konzept polymorph nutzen wollen, empfangen dieses durch einen impliziten Methodenparameter

Vorteile:

- Vermeidet Vererbung und Modifikation von Klassen
- Eine Methode für verschiedene Parametertypen
- Einfache Benutzung durch implizites DI der benötigten Konzeptimplementierung
- Reichweitensteuerung über Scoping einfach

Type Class Pattern Beispiel

NOVATEC
Make IT happen!

```
trait InvertableConcept[CType] { def inverse (obj:CType) : CType }

implicit val intInvertable = new InvertableConcept[Int] {
        def inverse (i:Int) = -i
}
implicit val seqInvertable = new InvertableConcept[Seq] {
        def inverse (s:Seq) = s.reverse
}

def inverted[CType](obj:CType)(implicit concept:InvertableConcept[CType])
        = concept.inversed(obj)
```

inverted(42) → inverted[Int](42)(inInvertable) → -42

inverted(1::2::3::Nil) → inverted[Seq](1::2::3::Nil)(seqInvertable) → List(3,2,1)

inverted("HelloWorld") → Compiler Fehler

Weitere neue Scala Pattern

NOVATEC
Make IT happen!

Duck Typing

Swiss Army Knife

Typesafe Builder

Virtual Class

scala.novatec-gmbh.de

(Ab 7.7.2011, moderierter deutschsprachiger Blog)

Vielen Dank!

Exkurs: Typausdrücke für Instanzen

NOVATEC
Make IT happen!

new A with T1 with T2 *Compound Typ*

new A { def toString = "Anonymes A" } *Refinement*

var duck:{ def toString:String } = new A *Strukturtyp (Duck Typing)*

Idiomatisch:

- Sehr häufige Verwendung von Typausdrücken anstelle von Klassendefinitionen
  - Führen intern zu Java anonymen Klassen
- Reduktion der Zahl der konkreten Klassendefinitionen im Vergleich zu Java
- Expliziter Ausdruck der Kompositions-Struktur einer Instanz