Webarchitekturen

Vierschichtenarchitektur für Webanwendungen:

<table>
<tr><td>Visualisierung
(Client Tier)</td><td>Web Browser</td></tr>
<tr><td>Präsentation
(Presentation Tier)
(JSP, Groovy & Grails, ...)</td><td rowspan="2">Web Server,
Application Server
(Tomcat, Java EE, ...)</td></tr>
<tr><td>Anwendungslogik
(Business Tier)
(Java, Groovy, ...)</td></tr>
<tr><td>Datenhaltung
(Resource Tier)</td><td>Database Server
(MySQL, Oracle, ...)</td></tr>
</table>

- Skriptsprachen (Perl, PHP, Python, Groovy)
- Server pages (ASP, JSF, JSP, ASP.net)

Einbindung Skriptsprachen in Web Server:

- **CGI**-Schnittstelle (Common Gateway Interface)
  - \+ Standardisiert und ausgereift und zuverlässig
  - \- Höherer Ressourcenverbrauch (Neue Interpreter-Instanz bei jedem dynamischen HTML-Seitenabruf)
    - Beschleunigung durch **FastCGI** (Interpreter wird einmal in Speicher geladen und bleibt dort)
- **Modulschnittstelle**
  - \+ Interpreter-Instanz bleibt im Speicher; Schneller als CGI
  - \- Nicht so ausgereift und stabil wie CGI
  - \- Fehlende Standardisierung (ISAPI für IIS, Apache Module z.B. mod_php)

Java Servlets

- Servlet: Java Programmcode, der zur Laufzeit mit *println* HTML Seite erzeugt
- Servlet Container: Ablaufumgebung für Servlets, JSP und JSF

Groovy

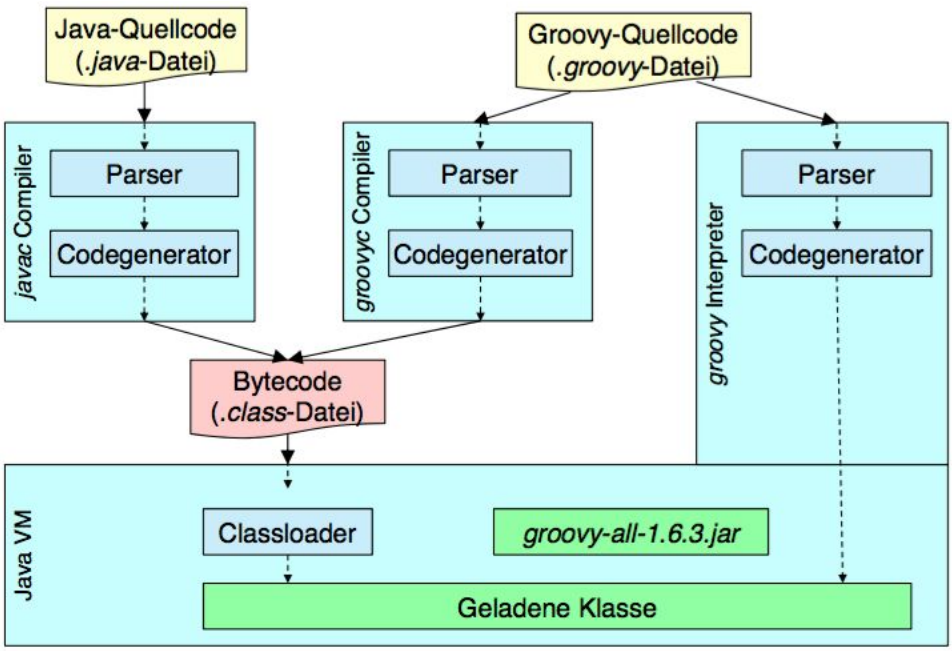
1. Allgemeines

Anforderungen an Sprachen für dynamische Webseiten:

- Gute Integration mit HTML, CSS, JS
- Leicht erlernbar, leichte Datenbankprogrammierung

- Leicht handhabbare Listen, assoziative Arrays (Maps)
- Skriptfähig und Programmiereffizienz

Groovy, Java und JVM:



Java vs Groovy

Vereinfachungen und Erweiterungen von Groovy:

- Alle Datentypen sind Objekte
- Dynamische Typisierung
- Operatoren überladen und überschreiben
- vereinfachte Nutzung von Listen, assoziative Arrays (Maps) und Ranges
- Closures
- Verzicht auf Main-Funktion, Semikolons, ...
- beide laufen in JVM
    - Plattformunabhängig
    - integriertbar mit Java, Byte-Code kompatibel
    - Java Quellcode muss **vor Ausführung** compiliert werden
    - Groovy kann wahlweise kompiliert oder **direkt als Skript** ausgeführt werden (implizite Compilierung beim ersten Aufruf)

2. Syntax und Kontrollkonstrukte

Syntax:

- Semikolons können generell entfallen, außen gewollte Endlosschliefe, *while(true);*

Variabledeklaration

- ***def*** statt explizitem Datentyp

- Bei Initialisierung einer Variable, braucht nicht unbedingt ***def***, z.B. str = 'hi'
- **Deklaration lokaler Variablen** benötigen ***def*** oder Datentypangabe
- Datentyp ergibt sich aus Initialisierungswert (dyn. Typisierung)

Bedingte Abfrage

if-Bedingung (Besonderheit)

- Der Ausdruck muss nicht boolean sein. Alles was nicht Null oder leerer String ist gilt als true

Switch

- Switch Variable darf String sein
- Erlaubte Case-Zweigen:
  - Range, 0..9
  - Listen, [1, 2, 3]
  - Datentyp, float
  - Pattern Matching, ~/[A-Z]*/
  - Closure, {i % 2 == 0}

Schleife

for (element in iterable)

- iteriert durch Ranges, Arrays, Collections, Maps
- def name="bla"; for (i in *0..<name.size()*) println name[i]

Exception Handing

try {} catch() {} finally {}

- geprüfte Exceptions brauchen nicht abgefangen werden
- {} darf nicht weggelassen werden
- Typ der Exception muss in Catch angegeben werden

3. Funktionen

Formale Positionsparameter

- Parameterdatentyp durch ***def*** ersetzbar und darf auch fehlen

Opt. formale Parameter

- letzter Parameter vom Typ ***Object[]***, *def max(x, Object[] y)*

Benannte Parameter (Es gibt in Java nicht)

Sprache		Java		Groovy	
Parameter-zuordnung		*Position*	*Name*	*Position*	*Name*
Funktionskopf	mit Typangabe	*int max(**int** x, **int** y)*	-	*int max(**int** x, **int** y)*	*def max(**Map** map)*
	ohne Typangabe	-		*def max(def x, def y)*	*def max(**def** map)*
				def max(x, y)	*def max(map)*
	mit Defaultwert	-		*def max(x, y=**0**)*	-
	optionale Parameter	*int max(**int** x, **int...** yArray)*	-	*def max(**Object[]** o)*	implizit
Rumpf		*return x>y ? x : y;*	-	*x>y ? x : y;*	*map.x>map.y ? map.x : map.y;*
Aufruf		*int z = max(**5,3**);*	-	*def z = max(**5,3**)*	*def z = max(**x:5, y:3**)*
Vorteile		+ bequeme Syntax + opt. Parameter	-	+ bequeme Syntax + opt. Parameter + Defaultwerte	• Parameter + Reihenfolge bel. + bel. weglassbar
Nachteile		• Parameter - Reihenfolge fest - kein Defaultwert	-	• Parameter - Reihenfolge fest	- unbequeme Syntax

- genau ein formaler Parameter vom Typ ***Map***
- nimmt sämtliche benannten Parameter als assoziatives Array auf

*def greet(**Map** param) param.firstName; ⟶ greet(firstName:"jo")*

- Funktions- und Methodendefinition
  - implizit public, wenn kein private oder protected angegeben
  - globale Funktion können außerhalb von Klassen definiert werden
- Funktions- und Methodenaufruf
  - Beim parameterlosen Aufruf dürfen Klammern **nicht** entfallen
  - Beim verschachtelten Aufrufen darf nur äußerster Klammer entfallen

4. Closures

Warum sollen Funktionen/Methoden auch Objekt sein?

- Als Parameter an Funktions- und Methodenaufrufe übergeben
- **Einsatz** für
  - Callback-Methoden asynchroner Aufrufe
  - Event Handling
  - Vergleichsfunktion in Sortieralgorithmen
  - Aufruf einer als Parameter übergebenen Funktion für jedes Element einer Collection

Array-Iterator

*array.each { print "$**it**, "; }* ⟶ "$it": GString, { … }: Funktionsobjekt (Closure)
new File("file.txt").eachLine { print $it }

Vorteil: Trennung zwischen Iterationssteuerung und Aktion pro Collection-Element

Definition: Anonymer Codeblock zwischen { }
Objekt vom Typ Closure, das:

- einer Variable zugewiesen werden kann, *def closure = { print "hi" }*
- ausgeführt werden kann, *closure()*
- einem Methodenaufruf als Parameter übergeben werden kann, *array.each { … }*
- aus einer Funktion zurückgegeben werden kann, *return { … }*

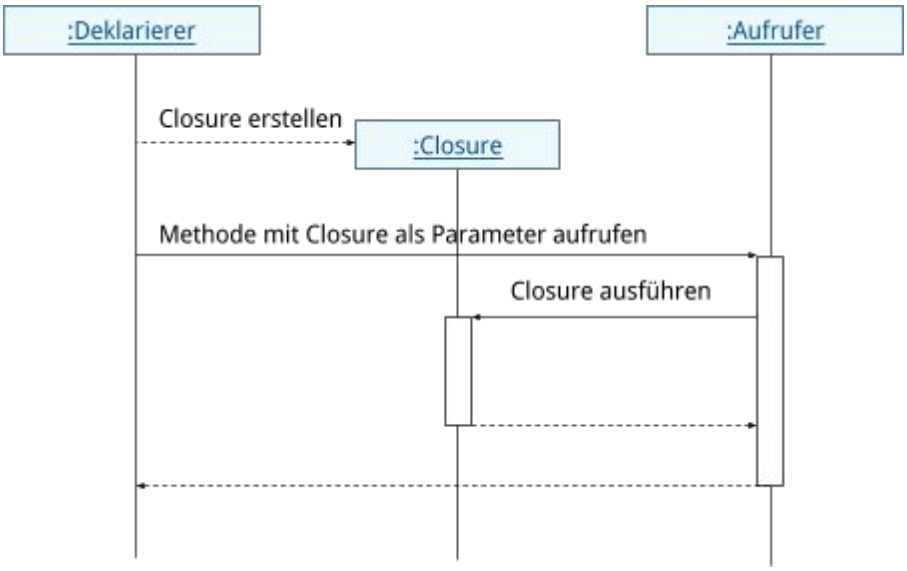
Parameterübergabe

{param1, param2 -> Closurerumpf }, z.B. *int max = {x, y -> x > y ? x : y }*

Umwandlung Funktion/Methode ⟶ Closure

- globale Funktionen: ***this.&**funktionsname*
- Methode eines Objekts: ***ref.&**methodenname*
- Klassenmethode einer Klasse: ***K.&**methodenname*

Ablauf beim Aufruf einer Closure



Sichtbarkeit von Variablen: Deklarationsort entscheidet

Closure zum **Filtern von Collections**

- Collection grep (Closure closure): liefert Collection mit Elementen, für die die Closure true liefert; *def p = [1, 2, 5]* ⟶ *p.grep {it < 5 }* ⟶ [1, 2]
- für assoziative Arrays; *{it.key / it.value}*

Überladen von Closure ⟶ nur über Multimethoden (Bsp. ?)

5. Datentypen und Operationen

Rechenoperationen erfolgen auf Wrapper-Klassen, z.B. 1 + 2 steht für 1.plus(2)

Vergleichsoperationen (==, !=, <==>, <, <=, >, >=)

- <==> entspricht compare-Methode, ergibt -1, 0, 1, gilt insbesondere für Strings

Vergleich von Referenzen

- mit der Methode ***is****(Object o)*, true, wenn beide Objekte an der selben Speicheradresse liegen, sonst false

Überladen von Operatoren (F.72 - 74)

Zahloperatoren
a.intdiv(b) liefert eine Ganzzahl zurück, *3.intdiv(2) // 1*

Ganzzahlmethoden

- n.upto(limit), 3.upto(7) // 3 4 5 6 7
- n.downto(limit), 7.downto(3) // 7 6 5 4 3
- n.step(limit, s), 7.step(3, -2) // 7 5
- n.times(), 3.times() { print “*”) // ***

Stringliterale

Besonderheit beim Apostroph

- Als String verwendet (Stringverkettung). ‘kann “Anführungszeichen” beinhalten, kann auch innerhalb “ein ‘String’” verwendet werden. Wenn allein verwendet wird, muss aber escapet werden, z.B. ‘ein \’String\’ escapen’

GStrings

- ermöglichen Ausgabe von Variablen und berechneten Werten in Strings, z.B. def name = “Peter”, print “Hey **$name**”,
- **${ausdruck}** wird ersetzt durch den Wert des Ausdrucks ausdruck
- Jeder “String” und /String/ ist GString, solange er ein $-Zeichen enthält, das nicht escapet und nicht das letzte Zeichen ist
- ‘Apostrophierte Strings’ sind grundsätzlich kein GString

Stringoperatoren

- plus (+): Hintereinanderhängen
- Minus (-): Erstes Vorkommen des 2. Strings aus 1. String löschen
- Multiplikation mit Ganzzahl: Mehrfaches Hintereinanderhängen
- Inkrement (++): Ersetzt letztes Zeichen durch seinen Unicode-Nachfolger
- Dekrement (--): Ersetzt letztes Zeichen durch seinen Unicode-Vorgänger
- Zugriff auf Zeichen mit Arrayindizes
    - Vorwärts: index > 0
    - Rückwärts: index < 0
    - Indexbereiche: [3..5], [3..<5]
    - Ausdrücke: [i/3..i/2]

Stringmethoden: center(n), center(n,c), contains(substr), reverse(), size()

StringBuffer-Operatoren

- StringBuffer überlädet [ ]-Operator zum Lesen und Ändern von Zeichen
- Sonderfälle:
    - Reines Einfügen: Indexbereich Länge 0, [0..<0]
    - Reines Löschen: Ersatzstring Länge 0, [0..1] = “”

6. Collections

Groovy-Collections

- Bauen auf Java-Collections auf
- **Vereinfachung**
    - statische Initialisierung mit Array-Notation, z.B. *def list = [“a”, “b”, “c”]*
    - Zugriff auf einzelnes Element mit [ ], z.B. *print list[-1] // c*
    - Zugriff auf Bereich mit Range-Indizierung, [0..1], [0..<2], [0..-2]
    - Element hinzufügen mit + und +=, z.B. list += “d”
    - Element entfernen mit - und -=, z.B. list -= “d”
    - Positionen ersetzen, list[0..1] = [“d”, “e”], ⟶ [“d”, “e”, “c”]

        - wenn Ersatz kürzer als Original: überzählige Elemente werden gelöscht, list[0..1] = [“f”], → [“f”, “c”]
        - Spezialfall: Ersatz hat Länge 0 → Elemente werden gelöscht, list[0..0] = [], → [“c”]
        - wenn Ersatz länger als Original: zusätzliche Elemente werden hingefügt, list[0..0] = [“f”, “g”], → [“f”, “g”], muss ein Range sein, sonst entsteht verschachtelte Liste
    - Iteration per for-Schleife, *for (elem in list) print “$elem ”*
    - Iteration per each-Closure, *list.each {elem -> print “$elem ”}*

Listen durchsuchen

- Alle Elemente liefern, list.**findAll** { it < xx }
- Erstes Element liefern, list.**find** { it < xx }

Liste nach Sortierkriterium sortieren, z.B. *list.sort { it[-1] }*

Listenquantoren

- Alle Elemente prüfen, list.**every** { .. }
- Mindestens ein Element prüfen, list.**any** { .. }

Mengen

- können jedes Element nur einmal enthalten
- unterstützen alle bei Listen besprochenen Operatoren und Zugriffe
- Definition
    - unsortiert: def list = […] **as Set**
    - sortiert: def list = […] **as SortedSet**

Bereche (Ranges)

- benötigen aufzählbare Datentyp, der
    - Interface Comparable implementiert
    - Operatoren ++ und --, d.h. Methoden *next()* und *previous()* definiert
- **Anwendung**
    - Indizierung von Arrays, Listen und Mengen
    - Schleifen, *def n = 8; for (i in 1..n/2) { … }*
    - Switch-Befehle, *c = ‘A’; switch(c) { case ‘a’..’z’: …; break; }*

Assoziative Arrays (Maps)

- bestehen aus Key-Value pair (key: alphanumerischer Schlüssel)
- Beispiel: def user = [name: “Hase”]
- Anwendung (F.102-105)
    - schneller Zugriff auf Datensätze über Schlüsselattribut
    - einfache Verzeichnisse (z.B. Zuordnung von Raumnummern zu Namen)
    - Zählen der Häufigkeit von Wörtern
    - Zugriff auf GET- und Post-Parameter von HTML-Formularen

Menge aller Schlüssel: map.**keySet()**, Collection aller Werte: map.**values()**

7. Klassen und Groovy Beans

Klassendefinition (wie in Java)

- implizit public, wenn nicht protected oder private angegeben
- meherere Klassen und Skriptcodes dürfen in einer Datei definiert werden - Ersatz für fehlende Inner Classes
- keine Inner Classes
  - Hilfsklassen in selbe Datei auf obere Ebene als Ersatz
  - Closures ersetzen anonyme Inner Classes

Groovy Beans

sind Groovy Klassen mit

- **parameterlosem Konstruktor**
  - implizit, wenn keine eigene Konstruktoren definiert
  - ansonsten selbst definieren
- (implizite) Getter- und Setter-Methoden für alle Attribute (Groovy Properties)
  - implizit (public) für Attribute ohne public/protected/private
- serialisierbar

Konstruktoren (wie in Java)

- implizit public (F.110)
- Wenn **kein** anderer Konstruktor definiert ist
  - impliziter Konstruktor mit **benannten Parameter** für alle Attribute
    - Syntax: ***Klassenname(attr1: val1, attr2: val2)***
    - Attribute ohne Initialisierungsparameter werden *null*
    - class Prof { def name }; def p = new Prof(name:"bla")

Sicheres Dereferenzieren mit ?.

- Motivation: Objektreferenzen können *null* sein
  - umständlicher Code bei jedem Attributabruf / Methodenaufruf objRef.attrName bzw. methode() auf objRef != null prüfen
- **Lösung: sicheres Dereferenzieren mit ?. statt .**
  - Attributabruf / Methodenaufruf über Nullreferenz wird ignoriert und liefert null statt Exception, z.B. *Professor prof, if (prof**?.**vorname)*

Expandos: Dynamisches Objekt

- instanziert aus der Groovy-Klasse Expando
- Anwendung: **beliebige Attribute nachträglich hinzufügen durch**
  - Angabe als benannter Parameter im Konstruktoraufruf
  - bloßes Beschreiben eines noch nicht vorhandenen Attributs
  - z.B. *c = new Expando(re:9, im:3);* ***c.name*** *= "Expando"*

8. Reguläre Ausdrücke

Match-Operator (==~)

Testet kompletten String auf Einhaltung eines regulären Ausdrucks

Syntax: booleanResult = testString **==~** Pattern (Pattern: Regulärer Ausdruck als /Slashy String/, um Probleme mit Escape-Zeichen \ zu vermeiden)

z.B. 'abc' ==~ /[abc]/ → false (Pattern deckt nur ein Zeichen ab)

'abc' ==~ /[abc]*/ → true, 'abc' ==~ /^a**.***/ → true (alles, was mit a beginnt)

Find-Operator (=~)

- **Analysiert und zerlegt String anhand eines regulären Ausdrucks**
- Syntax: ergArray = testString **=~** Pattern
- Ergebnis: 1-dim Array **ohne** Capture Groups **()**, 2-dim **mit** Capture Groups

Capture Groups

- innerhalb eines regulären Ausdrucks, gebildet mit (...)
- (X), X ist ein regulärer Teilausdruck
- Ausnahme: (?:X) definiert eine Gruppe, die keine Capture Group ist

Quantifizierer-Strategie: Greedy, Reluctant, Possesive

Strategie	Symbol	Bedeutung
Greedy (gierig)		Erstmal maximal viele passende Zeichen aus String nehmen, notfalls wieder welche zurückgeben
Reluctant (zurückhaltend)	**?**	Erst mal minimal viele passende Zeichen aus String nehmen, notfalls noch welche dazu nehmen
Possesive (besitzergreifend)	**+**	Maximal viele passende Zeichen aus String nehmen und eisern verteidigen, selbst wenn Pattern scheitert

Anwendungen des Find-Operators

- Benutzereingaben analysieren, auf syntaktische Korrektheit prüfen, in einzelne Bestandteile zur Weiterverarbeitung zerlegen
- liefert in Wirklichkeit kein 2-dim. Array, aber Instanz der Klasse *java.util.regex.Matcher*
- Methoden des java.util.regex.Matcher (F. 142-143)

Pattern-Operator (~)

- matcher = **~**/Pattern/
- parst und compiliert Pattern, um künftige Zugriffe zu beschleunigen (F.144)
- **Anwendung**
  - Collection grep(Pattern), *list.grep(~/Patter/)*
  - Switch-Case, *case ~Pattern: ... // pattern.isCase(ausdruck)*

Weitere Anwendungen von regulären Ausdrücken:

- String replaceAll(Pattern, replacement)
- String[] split(String splitPattern)
- String eachMatch(Pattern) { Closure }

9. Metaprogrammierung und Builder

Metaprogrammierung

Fähigkeit von Programmen,

- **Programmcode als Daten** zu behandeln (einlesen, analysieren, generieren, transformieren, ändern und ergänzen)
- sich selbst zu **analysieren** (Attribute und Methoden einer Klasse zur Laufzeit herausfinden)
- sich selbst zu **erweitern**, d.h. sein eigenes Verhalten zu ändern, ggf. neuen Programmcode zu erzeugen

Realisierungsvariante

1. Zugriff auf Interna der **Laufzeitumgebung** per API (erfolgt zur **Laufzeit**)
    a. Reflection, Werte abfragen und ändern, ohne die Struktur zu ändern, z.B. Java Reflection Interface
    b. echte Metaprogrammierung, Beispiel: Groovy Metaprogrammierung
2. Ausführung von Befehlen, die in Strings oder anderen Datentypen vorliegen (konstruriert zur **Laufzeit**)
3. außerhalb des eigentlichen Programms (erfolgt zur **Compilezeit**: Präprozessoren, Präcompiler, Codetransformation, z.B. IDL-Präcompiler in CORBA)

Reflection

- Programm kennt seine Struktur zur Laufzeit
- Dynamischer Attributzugriff (Name des abzufragenden Attributs liegt als String vor)
- Dynamischer Methodenaufruf (Name aufzurufender Methode liegt als String vor)
- Vorteile
    - höhere Flexibilität
    - Parametrisierung von Attribut- und Methodennamen
- Nachteile
    - keine Typprüfung durch Compiler mehr möglich
    - geringere Performance

Echte Metaprogrammierung

- ermöglicht strukturelle Änderungen am Programmcode zur Laufzeit
    - Hinzufügen neuer Attribute und Methoden
    - ohne den Quellcode der Klasse zu besitzen oder zu ändern
    - funktioniert auch für alle mitgelieferten Klassen der Klassenbibilothek
    - Vorteile
        - Ergänzung vorhandener Klassen um neue Methoden
        - Konverter und Rechenoperationen zu selbst definierten Datentypen
        - Selbstmodifizierender Code möglich
    - Nachteile: geringere Performance

Vergleich Java Reflection / Groovy Metaprogrammierung

Kriterium	Java Reflection	Groovy Metaprogrammierung
Zugriff	objRef.**class** Klasse.**class**	objRef.**metaClass** Klasse.**metaClass**

Dynamisch Attribut zur Klasse hinzufügen

- ExpandoMetaClass.enableGlobally()
  - damit auch bereits bestehende Instanzen neue Attribute / Methoden sehen
  - sonst nur die Instanzen, die nach der Erweiterung noch erzeugt werden
- Class.metaClass.newAttribute = initialValue
  - Initialwert muss vorhanden sein, legt Datentyp fest
- static-Attribute lassen sich nicht nachträglich zu Klassen hinzufügen

Dynamisch (static) Methode zur Klasse hinzufügen

- Class.metaClass.newMethod = { ... }
- Class.metaClass.static.newMethod = { ... }

Dynamisch Konstruktor zur Klasse hinzufügen

- Class.metaClass.constructor = { ... }, Rückgabewert **muss** neu erzeugte Instanz sein

Fehlende Methoden abfangen: **methodMissing**

- Falls aufgerufene Methode nicht existiert (weder als Methode noch als Closure in der Klasse, auch nicht als Closure in der Metaklasse)
- ruft Groovy *methodMissing(String methodName, args)* auf, als Methoden in der Klasse und als Closure in der Metaklasse
- **Anwendung**: Rumpf für fehlende Methode nachliefern
  - oft erst dynamisch zur Laufzeit generiert
  - Anwendung z.B. in Grails für findXXX der Domänenklassen
- **Vorteile**
  - es brauchen nicht Unmengen von Methoden auf Vorrat programmiert werden
  - automatische Generierung zur Laufzeit
- **Nachteile**
  - geringere Performance und Gefahr der Endloserekursion

Intercept -Cache-Invoke-Pattern

- Problem: schlechte Performance mit *methodMissing()*
- Lösungsansatz: **Cache für dynamisch erzeugte Methoden**
  - bei erstmaligem Aufruf
    - fängt *methodMissing()* den Aufruf der fehlenden Methode ab (**Intercept**)
    - erzeugt dynamisch die neue Methode als Closure
    - fügt die Methode dynamisch der Metaklasse als Closure hinzu (**Cache**)
    - ruft die neue Methode auf, um das Ergebnis zu berechnen (**Invoke**)
  - bei allen weiteren Aufrufen
    - ist gewünschte Methode in der Metaklasse vorhanden und direkt aufrufbar
- **Anwendung**: Verwendung in Grails für findXXX in Domänenklassen
- **Vorteile**: gute Performance trotz automatischer Erzeugung dynamischer Methoden

Aspektorientierte Programmierung

- **Ziel**: generische Funktionalitäten (Cross Cutting Concerns)
  - von der eigentlichen Anwendungslogik trennen
  - über Klassen / Methoden hinweg verwenden
- Anwendungsbeispiele: Protokollierung aller Methodenaufrufe, Fehlerbehandlung, ...

- Verwendung mit Groovy
  - Jede Klasse kann Interface ***GroovyInterceptable*** implementieren
  - reines Marker-Interface ohne Methoden
  - alle Methodenaufrufe laufen über *invokeMethod()*-Aufruf der Klasse, dort kann
    - **vor** und **nach** dem Methodenaufruf eigener Code ausgeführt werden
    - die urprüngliche Methode aufgerufen werden, ggf. unter Bedingungen
  - Verbesserung: indirekter Aufruf der abgefangenen Methode über Metamethode:
    - delegate.metaClass.**getMetaMethod**(name, args).**invoke**(delegate, args)

Builder

- Design Pattern zum Erzeugen komplexer Objekte, bekommt Beschreibung als Parameter
- Anwendung: Verarbeitung domänenspefizischer Sprachen, z.B. XML

10. Testen (Nicht relevant)

Einfache skriptbasierte Tests
assert boolescheBedingung [: “Fehlerbeschreibung”]

- boolesche Bedingung muss **true** sein, sonst Programmabbruch, [:] optional

JUnit-Tests

- alle Methoden
  - parameterlos, Rückgabetyp *void*, Methodenname beliebig
  - entscheidend ist die Annotation
  - Annotation darf an mehreren Methoden stehen

Grails

1. Einführung

Prinzipien

1. **Convention over Configuration**
   a. fest vorgegebene Ordner für Domänenklassen (Model), Controller, Views, Internationalisierung, Skriptcode, Testcode
   b. minimaler Konfigurationsaufwand durch
      i. sinnvolle Standardwerte für alle Konfigurationsparameter
      ii. Konfiguration über Groovy basierte Domänenspezifische Sprachen
      iii. weitgehenden Verzicht auf XML-Konfigurationsdateien

2. **Don't Repeat Yourself**
   a. Vermeidung unnötiger Code-Redundanzen
   b. Automatische Generierung aus Domänenklassen
      i. Datenbank
      ii. Controller und Views für CRUD-Operationen per **Scaffolding**

3. **Agility**
   a. wahlweise statische oder dynamische Typisierung
   b. kurze Entwicklungszyklen, agile Webentwicklung, Rapid Prototyping, leichtes Testen

4. **Open Source**

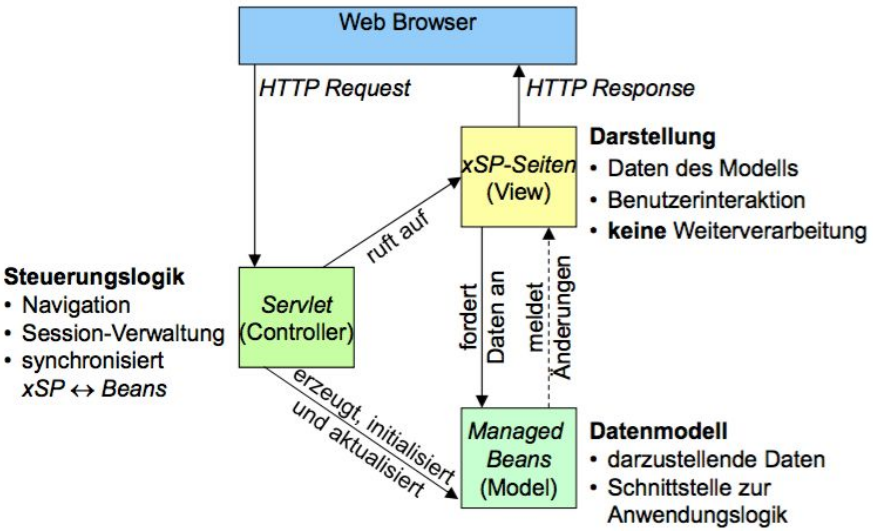
Design-Patterns

1. **Model View Controller (MVC)**
   a. Model: Domänenklassen mit automatischer Persistenz (GORM / Hibernate)
   b. View: Groovy Server Pages (GSP), HTML basiert
   c. Controller: Eigene Instanz für jeden Request

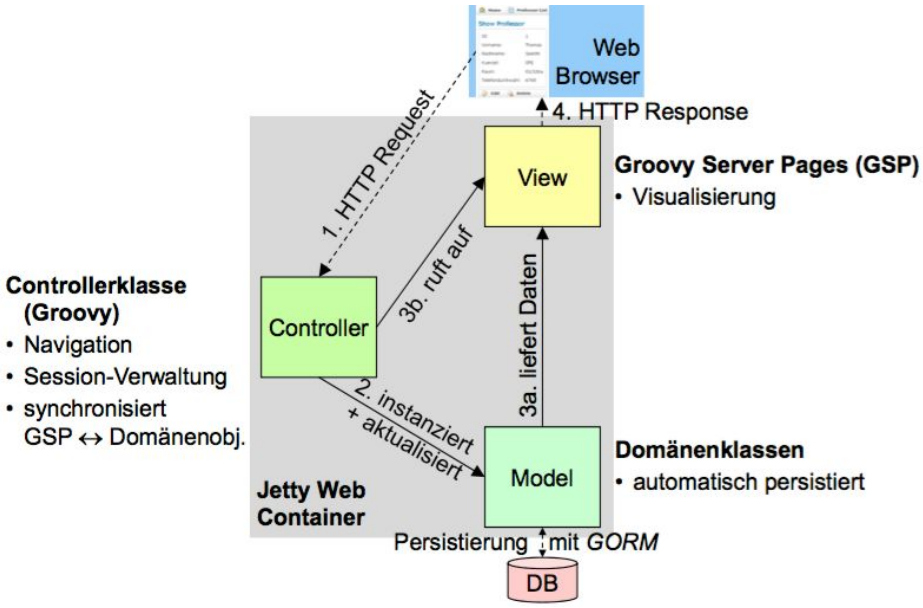
2. **Scaffolding**: CRUD-Funktionalität automatisch generiert aus Domänenklassen
   a. **statisches** Scaffolding
      i. Groovy-Quellcode für Controller aus Domänenklassen generieren
      ii. Groovy Server Pages für Views aus Domänenklassen generieren

   b. **dynamisches** Scaffolding
      i. Controller und Views dynamisch zur Laufzeit erzeugen, ohne Quellcode

Spezialisierung MVC für Webanwendung



MVC in Grails



2. Controller

Aufgaben: steuert die Webseiten zu **einer** Domänenklasse

- Navigationslogik
- enthält Controlleraktionen
- Schnittstelle zur Anwendungslogik
- steuert Persistierung

Realisierung als **Groovy Klasse**

- Kein Verbrauch der Vererbung
- Kein Konstruktor

- scaffold-Attribut (nur für dynmisches Scaffolding, ***def scaffod=Domänenklassenname***)

3 Möglichkeiten um Controller zu erzeugen:

- **dynamisches Scaffolding**
  - dynamische Generierung zur Laufzeit, **kein Quellcode erzeugt**, **nicht** änderbar
  - Anwendung: typische CRUD-Oberflächen, z.B. Admin-Oberflächen
- **statisches Scaffolding ("Gerüstbau")**
  - statische Generierung als Quellcode aus der Domänenklasse
  - beliebig änder- und erweiterbar
  - Anwendung: CRUD-basierte, komplexere Oberflächen
- manuelle Programmierung
  - höherer Programmieraufwand
  - Anwendung: Oberflächen abseits der üblichen CRUD-Funktionalität

Dynamisches Scaffolding

- Controller verwaltet genau **ein** Domänenklasse
  - gleicher Name wie Domänenklasse, Endung Controller
  - Quellcode beinhaltet nur **Zuordnung** zur **Domänenklasse**, sonst nichts, alles andere wird dynamisch zur Laufzeit erzeugt
- Controller erstellen
  - **create-controller** Controllername
  - generiert Controllerklasse als Quellcodegrüst, ***def scaffold=****Domänenklasse* ergänzen

Statisches Scaffolding

- Controller als **Quellcode** generieren
  - enthält Methoden für Aktionen ***index, show, delete, edit, update, create, save***
  - nachbearbeitbar
- zugehörige Views (GSPs) als Quellcode generieren (nachbearbeitbar)
- Grails-Kommandos
  - **generate-controller** Domänenklasse
  - **generate-views** Domänenklasse
  - **generate-all** Domänenklasse
- **Controller Actions** (F.247 - 256)
  - **Alle Aktionen benötigen readOnly-Transaktion**, sofern nicht anders annotiert
    - @transactional (readOnly = true) class XXXController { … }
  - respond Professor.list(params), model: [professorCount: Professor.count()]
  - respond: View darstellen, Content Negotiation, list-Methode der Domänenklasse liefert Datensätze, key im View

Erlaubte HTTP-Methoden

- **Default**: alle erlaubt
- **sinnvoll**: **GET für Änderungsaktionen** (*save, update, delete*) verbieten, vgl. F.258
  - erschwert URL-Manipulation
  - verhindert Mitlesen von Parameterwerten auf der URL
- Konfiguration in der Controllerklasse (assoziatives Array mit key: value)
  - Default für durch dynamisches Scaffolding generierte Controller:

- *static allowedMethods = [save: "POST", update: "PUT", delete: "DELETE"]*

Aufruf von Controller-Aktionen aus GSPs, z.B. über form action="xxx" oder <g:link action="xxx"/>

Transaktionssteuerung mit @Transactional
- Alle Controlleraktionen laufen **in einer GORM/Hibernate-Transaktion** ab
- Transaktionsarten
  - read/write (default), annotiert mit @Transactional
  - read only, @Transactional(readOnly=true)
- @Transactional für Controllerklasse und -methoden - höhere Priorität

Parameterübergabe View (GSP) → Controlleraktion
- alle HTTP-Parameter in Umgebungsvariable *params* - assoziatives Array, z.B.
  - params = "[vorname: 'xxx']"
- automatische Bereitstellung der Domäneninstanz für *id*-Parameter

Controlleraktion auslösen
- Link in GSP
  - <**g:link** controller="Professor" action="list">Alle auflisten</**g:link**>
  - url: <**g:link** url="[controller:'professor', action:'list']">Alle auflisten</**g:link**>
- Form in GSP
  - <**g:form** method="..."><**g:actionSubmit action="update"** /></**g:form**>
  - actionSubmit wirkt wie ein Methodenaufruf (=Controlleraktion)

URL des Projekts:
http://host:port/(Projektname?)/Controllername/Controlleraction/Instanz-ID?key=value
Servlets verbergen sich hinter
- Projektpfad (?) - Grails Standardservlet mit Liste der Controller
- jedem Controller
  - eigenes Servlets, das den Controller implementiert
  - Parameter für dieses Servlet: action name, id der Domäneninstanz, weitere Params

Methodensignatur für parameterlose Aktion
- Zugriff (GET/POST): über Umgebungsvariable *params* (assoziatives Array)
- ***def create()*** *{ respond new Professor(params) }*
- Aufruf: http://host:port/(Projektname?)/Controllername/**create**

Methodensignatur für Aktion mit HTTP-Parametern
- Datentyp **muss** in Signatur angegeben sein (elementare + deren Wrapper)
- formale Parameter dürfen weggelassen werden, Zugriff kann über *params* erfolgen
- ***def index*** *(Integer max) { params.max = ... }*
- Aufruf: http://host:port/(Projektname?)/Controllername/**index**?max=xxx

Methodensignatur für Aktion mit Domäneninstanz
- HTTP-Parameter **id** wird übergeben; **erster Parameter** muss vom Typ Domänenklasse sein

- ***def edit*** *(Professor professorInstance) { respond professorInstance }*
- Aufruf: http://host:port/(Projektname?)/Controllername/**edit**/id(?key=value)

Siehe **Umgebungsvariablen im Controller** (F.275)

Zugriff auf Sessionvariablen mit *session*

- Session automatisch erzeugt
- session wie assoziatives Array mit alle Sessionvariablen
- Add: session.id = params.id; Auslesen: session.id; Löschen: session.removeAttribute("id")

Fehlermeldung an nächste GSP liefern mit flash

- Flash-Objekt automatisch im Controller erzeugt (F.278)
- Flash.message wird im Controller als String abgelegt, z.B. flash.message = message(code: ..., args: [...])
- In nächster GSP abfragen mit *<g:if test="${**flash.message**}">${flash.message}</g:if>*
- Danach wird *flash.message* automatisch gelöscht

Logging in Grails und Appender (?)

Logging

- Log Meldung auslösen mit Umgebungsvariable **log**
  - Zwei Methoden in allen Controller: log.lev(Object msg) und log.lev(..., Throwable)
- Syntax: ***logger*** *"zuLoggendeKomponente", LogginLevel*
- Zu loggende Komponente: z.B. professorverwaltung(.ProfessorController)
- Logging Level: *FATAL, ERROR, WARN, INFO, DEBUG, TRACE, OFF, ALL*

Appender

Typ: DBAppender, ConsoleAppender, FileAppender, RollingFileAppender

Parameterübergabe Controller

Model: Controller liefert Model

- Aktions-Methode gibt assoziatives Array mit Variablenwerten für GSP zurück
- def xxx() { **[key: value]** }, name des Views: xxx.gsp
- In View mit **${key}** zugreifen

Model + View

- Controller löst Rendern eines **Views**(GSP) aus und liefert **Model** dazu
- **render** view: "viewName", model:"myModel"

falls View-Name=Action-Name → Model, sonst Model +View

Rendern ohne View

- render "viewName"
- kein Model und View benötig
- ins XML/JSON-Format
  - **contentType**:"text/xml"/ "application/json"
  - import grails.converter → render domainInstance **as XML/JSON**

Content Negotiation (CN)

- **Ziel**: Optimales Format für jedes Endgerät

- Mitteilung des Wunschformats: im **HTTP-Header-Attribut Accept** als MIME Content-Type
- Automatische CN per respond: **respond domainInstance**, **model**: [key: value]
- View wird automatisch anhand der Endung (z.B. .xml) im entsprechenden Format angezeigt

Fehlermeldung durch Content Negotiation

Anwendung:

- nur bei Browser Clients im Klartext zur Einbettung in GSP mitliefern, alle anderen Clients bekommen HTTP-Status
- Anpassung des Ausgabeformats an Endgeräte (automatisch über respond-Aufruf)
- endgerätespezifische Eingabeparameter (z.B. xml, json)

Umleitung auf andere Aktion

- des eigenen Controllers: ***redirect****(****action:show****, id: instance.id)*
- eines anderen Controllers: ***redirect****(controller: "Professor", action: show, id: instance.id)*
- mit Parametern: ***redirect****(controller: "Professor", action:show, params:[id: params.id])*

Interceptors

- unterbrechen Ausführung von Controlleraktionen: bool *before()*, bool *after()*, void *afterView*
- *after()*: kann anderen View zum Render auswählen und Model verändern
  - Anwendung:
    - before: Überprüfen der eingegebenen Parameter, Login-Handling
    - after: Überprüfen und Manipulation der Antwort der Aktion
- erstellen durch **create-interceptor** Domänenklasse
- erzeugter Interceptor mit Dateinamen *DomänenklassennamenInterceptor.groovy*
- **Unbedingter Interzeptor**
  - **kein** Konstruktor im Interzeptor
  - Name des Interzeptors: ProfessorInterceptor() { … }
- **Bedingter Interzeptor**
  - selbstdefinierter parameterloser Konstruktor
  - Konfiguration durch einen oder mehrere match-Aufrufe
    - parameter: controller, action, method(http), uri
    - match controller: (eigenen Controller enthalten?) und andere Controller

manuelle Programmierung

- unabhängig von Domänklassen und Datenbank
- kann mehrere Domänenklassen repräsentieren
- volle Flexibilität, aber höher Programmierungsaufwand
- View:
  - Einstiegsseite immer views/index.html
    - nicht mit Interzeptoren abfangen, nicht umleiten
  - jede GSP muss auf "main-layout" beinhalten

3. Domänenklassen

Definition: Grundlage der **Persistierung** in Grails

- pro Domänenklasse **eine Datenbanktabelle**

- **Instanzen** der Domänenklasse entsprechen **Zeilen** der DB-Tabelle
- **Attribute ≙ Spalten** der DB-Tabelle
- **Instanzen** werden durch Grails automatisch persistiert
  - technische Basis: **GORM** (Grails Object Relational Mapping)
  - basiert auf Hibernate

Realisierung als Groovy Bean

- normale Groovy-Klasse (Kein Konstruktor, kein Verbrauch der Vererbung)
- implizite Getter und Setter
- serialisierbar

Attribute

- **statisch typisiert** (mit **Datentypangabe**) - Typsicherheit
- automatisch in DB persistiert mit GORM
- Beziehungsattribute zu anderen Domänenklassen möglich (1:1, 1:n, m:n)

Constrains

- **Integritätsregeln** für Attribute
  - Muss- / Kann-Attribute (nullable, blank)
  - Größenbeschränkungen (maxSize)
- Klassenvariable **constraints**
  - definiert Integritätsregeln für Attribute mit DSL (Domain Specific Language)
  - statisch, weil für alle Instanzen gültig
  - Name constraints fest vorgegeben (Convention Over Configuration)
  - Realisierung als **Closure** (kein Parameter, kein Rückgabewert) - Attribute mit benannten Parametern als Map definieren
    - ***static constraints = {*** *age(nullable: false, range: 20..50* ***}***

toString()

- Überladen **muss** als Rückgabetyp String sein
- **GORM-Methoden** für Domänenklassen (autom. dyn. bereitgestellt, ohne Quellcode), z.B. list(), save(), get(id), count(), delete(), findByXXX()
- Weitere Methoden nicht sinnvoll
  - Anwendungslogik gehört in Controller / wird von dort aufgerufen
  - Domänenklassen als reine Datencontroller + Persistenzschnittstelle
- Anwendung: “Instanzname” ausgeben, z.B. für Debugging

Beziehung in Domänenklassen

- über spezielle **static** Attribute
- 1:0..1 → **nullable**:true
- oder **Einbetten** von Objekten
  - **static embedded** = [‘adresse’]
  - vermeidet:
    - Henne-Ei-Problem
    - unnötige Datenbank-Joins beim Abruf der Daten
    - eigenen Controller und eingene GSP für Adresse
  - Auswahl

        - als **eigene Domänenklasse**
            - bekommt (überflüssige) Datenbanktabelle
        - als **weitere Klasse** in Quellcodedatei der Klasse Student
            - Groovy erlaubt Definition mehrerer Klassen in einer Quellcodedatei
            - keine eigene Datenbanktabelle
        - als **normale Groovy-Klasse** im Ordner src/main/groovy
            - keine eigene Datenbanktabelle
            - beste Lösung
- 1:n ⟶ hasMany
- n:1 ⟶ belongsTo
- n:0..1 ⟶ belongsTo + **nullable**:true
- bsp.: static **belongsTo**=[hochschule:Hochschule], static **hasMany**=[fakultaeten:Fakultaet]
- m:n

Tabellenstruktur in GORM

- Attribute
    - id: künstlicher Primärschlüssel, automatisch generiert
    - version: Versionsnummer des Datensatzes für optimistisches Sperren
    - xxx_id: Fremdschlüsselattribute nach Bedarf
    - -... Attribute der Domänenklasse, **alphabetisch** sortiert
- Constraints
    - PRIMARY: Attribut id eindeutiger (Primär-)schlüssel
    - FK_xxx: Index für jeden Fremdschlüssel
- Primärschlüsselerzeugung
    - abhängig von zugrunde liegender Datenbank
    - in MySQL: id-Attribut mit Extra AUTO_INCREMENT

4. RESTful Web Services

5. GSP