Fachbereich Informatik und Medien -
Wissenschaftliches Arbeiten und Schreiben - Master Inf. Prof.
Loose WS 2011/12

Untersuchung von sprachorientierten Programmierparadigmen, deren Ausprägungen und Akzeptanz bei Domänenexperten.

Vorgelegt von: Nils Petersohn

am: 8.11.2011.

Inhaltsverzeichnis

1 Einleitung

Die Sektionen würden bei der echten Arbeit wegfallen.

1.1 Motivation (Heranführen an das Thema)

Das Wort “Abstraktion” bezeichnet meist den induktiven Denkprozess des Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres [...] alsö[...] jenen Prozess, der Informationen söweit auf ihre wesentlichen Eigenschaften herab setzt, dass sie psychisch überhaupt weiter verarbeitet werden können. (nach [wika]) Die grundlegenden Abstraktionsstufen in der Informatik sind wie folgt aufgeteilt: Die unterste Ebene ohne Abstraktion ist die der elektronischen Schaltkreise, die elektrische Signale erzeugen, kombinieren und speichern. Darauf aufbauend existiert die Schaltungslogik. Die dritte Abstraktionsschicht ist die der Rechnerarchitektur. Danach kommt eine der obersten Abstraktionsschichten: “Die Sicht des Programmierers”, der den Rechner nur noch als Gerät ansieht, das Daten speichert und Befehle ausführt, dessen technischen Aufbau er aber nicht mehr im Einzelnen zu berücksichtigen braucht. (nach S. 67 [Rec00]). Diese Beschreibung von Abstraktion lässt sich auch auf Programmiersprachen übertragen. Nur wenige programmieren heute direkt Maschinencode, weil die Programmiersprachen der dritten Generation (3GL) soviel Abstraktionsgrad bieten, dass zwar kein bestimmtes Problem aber dessen Lösung beschrieben werden kann. Die Lösung des Problems muss genau in der Sprache beschrieben werden und setzt das Verständnis und die Erfahrung in der Programmiersprache und deren Eigenheiten zur Problemlösung voraus. Das Verständnis des eigentlichen Problems, dass es mit Hilfe von Software zu lösen gilt, liegt nicht immer zu 100% bei dem Programmierer, der es mit Java oder C# bzw. einer 3GL lösen soll. Komplexe Probleme z.B. in der Medizin, der Architektur oder im Versicherungswesen sind oft söumfangreich, dass die Aufgabe des “Requirements Engeneering” hauptsächlich darin besteht, zwischen dem Auftragnehmer und Auftraggeber eine Verständnisbrücke zu bauen. Diese Brücke ist auf der einen Seite mit Implementierungsdetails belastet und auf der anderen mit domänenspezifischem Wissen (Fach- oder Expertenwissen). Die Kommunikation der beiden Seiten kann langfristig durch eine DSL

begünstigt werden, da eine Abstrahierung des domänenspezifischen Problems angestrebt wird. Die Isolation der eigentlichen Businesslogik und eine intuitiv verständliche Darstellung in textueller Form kann sogar soweit gehen, dass der Domänenexperte die Logik in hohem Maße selbst implementieren kann, weil er nicht mit den Implementierungsdetails derselben und den syntaktischen Gegebenheiten einer turingvollständigen General-Purpose-Language wie Java oder C# abgelenkt wird. Im Idealfall kann er die gewünschten Anforderungen besser abbilden.(vlg. [hei11]). Beispiele für DSL sind z.B.: Musiknoten auf einem Notenblatt, Morsecode oder Schachfigurbewegungen ("Bauer e2-e4") bis hin zu folgendem Satz: "wenn (Kunde Vorzugsstatus hat) und (Bestellung Anzahl ist größer als 1000 oder Bestellung ist neu) dann ... ". "Eine domänenspezifische Sprache ist nichts anderes als eine Programmiersprache für eine Domäne. Sie besteht im Kern aus einem Metamodell einer abstrakten Syntax, Constraints (Statischer Semantik) und einer konkreten Syntax. Eine festgelegte Semantik weist den Sprachelementen eine Bedeutung zu." (vlg. S.30 [mda]).

1.2 Begriffserklärung

Eine DSL beinhaltet ein Metamodell. In einer Domänengrammatik gibt es das Konzept an sich, das beschrieben werden soll. Konzepte können Daten, Strukturen oder Anweisungen bzw. Verhalten und Bedingungen sein. Das Metamodell oder auch das Semantische Modell besteht aus einem Netzwerk vieler Konzepte. Das Paradigma "Language Orientated Programming" (LOP) identifiziert ein Vorgehen in der Programmierung, bei dem ein Problem nicht mit einer GPL (general purpose language) angegangen wird, sondern bei dem zuerst domänenspezifische Sprachen entworfen werden, um dann durch diese das Problem zu lösen. Zu diesem Paradigma gehört auch die Entwicklung von domänenorientierten Sprachen und intuitive Programmierung (intentional programming). "Intentional Programmierung ist ein Programmierparadigma. Sie bezeichnet den Ansatz, vom herkömmlichen Quelltext als alleinige Spezifikation eines Programms abzurücken, um die Intentionen des Programmierers durch eine Vielfalt von jeweils geeigneten Spezifikationsmöglichkeiten in besserer Weise auszudrücken." (vlg. [wikb]) Es werden zwei Arten von DSLs unterschieden [FP11].

1.3 Aufgabenstellung (kurz, knapp, präzise) und Erwartungen

Das Paradigma der sprachorientierten Programmierung und die damit verbundenen Konzepte der domänenorientierten Programmierung sollen in dieser Arbeit analysiert und geprüft werden.

Eine einfache DSLs soll mit Hilfe von Groovy-Metaprogramming als interne DSLs entworfen werden.

Dabei stellen sich folgende Fragen: Wie effizient ist die Erstellung einer DSL mittels der Programmiersprache Groovy und wie reagiert der Domänenexperte auf die DSL?

Der Domänenexperte, die keine Erfahrung mit Programmierung haben, sollen an interne DSLs für deren bekannte Domäne herangeführt werden, um diese anschließend nach Lesbarkeit, intuitivem Verständnis und Flexibilität zu bewerten. Dabei bekommt der Proband mehrere Aufgaben, die mit den gegebenen DSLs zu lösen sind.

1.4 Gliederung

Zuerst werden theoretische Grundlagen zum Thema “Sprachorientierte Programmierung” und Metamodellierung erläutert. Der praktische Teil beginnt mit der Vorstellung der Problemdomäne und deren verschiedenen Anforderungen. Der nächste Schritt in dieser Arbeit beschäftigt sich mit der konzptionellen Implementierung der DSL. Weiterhin soll die Testumgebung mit den Probanden spezifiziert werden, um danach die Durchführung und die Ergebnisse auszuwerten und zu beschreiben.

1.5 Abgrenzung

Der Fokus dieser Arbeit soll auf die textuelle und nicht auf die graphische Repräsentation von DSLs abzielen. “Natural language processing” soll nur oberflächlich betrachtet werden. Der praktische Teil berichtet ausschließlich über die Groovy-Metaprogrammierung.

2 Theorie

2.1 Domain Specific Languages

Eine domänenspezifische Sprache (engl. domain-specific language, DSL) ist, im Gegensatz zu gängigen Programmiersprachen, auf ein ausgewähltes Problemfeld (die Domäne) spezialisiert. Sie besitzt hoch spezialisierte Sprachelemente mit meist natürlichen Begriffe aus der Anwendungsdomäne. Das Gegenteil einer domänenspezifischen Sprache ist eine universell einsetzbare Programmiersprache (engl. general purpose language, GPL), wie C und Java, oder eine universell einsetzbare Modellierungssprache, wie UML.

Mit Hilfe einer solchen Sprache können ausschliesslich Problembeschreibungen innerhalb des jeweiligen Problemgebiets beschriebne werden. Andere Problembereiche sollen ausgeblendet werden, damit der Domänenexperte sich nur auf das für Ihn wichtigste in dem jeweiligen Bereich konzentrieren kann.

Der Domänenspezialist (z.B. ein Betriebswirt) ist mit dem Problembereich (z.B. Produktpreisbildung) sehr vertraut. Die Domänensprache, z.B. zur Beschreibung von Preisbildungskomponenten und deren Zusammenhänge, gibt dem Betriebswirt ein mögliches Werkzeug, um die Preise für Produkte (z.B. Computerhardware) dynamisch anzupassen. Diese DSL ist dann aber für andere Bereiche, wie z.B. der Aufstellung des Personalschichtplans nicht einsetzbar.

Die Charakteristiken einer DSL sind vorzugsweise minimale Syntax die nur die nötigsten Mittel zur Strukturierung benötigt um die Lesbarkeit zu erhöhen und keine Ablenkung vom Problem zu schaffen. Was genau eine minimale Syntax ausmacht ist schwer messbar zu machen. Vorzugsweise sollte die Syntax keine Redundanzen aufweisen, wie das z.B. bei XML der Fall ist indem das offene und geschlossen element nochmals den selben namen tragen. Als Faustregel sollte ein Zeichen bzw. eine minimale Kombination aus mehreren Zeichen und deren Position in der DSL für eine Informationseinheit verwendet werden. Als Zeichen sind hier insbesondere leerzeichen und Zeilenumbrüche gemeint. Solche die auch eine angelehnte Bedeutung zu der natürlichen Sprache haben. Z.B. Klammern, Semikolen, Kommas, Punkte,

Slashes ...

Fowler ist der begründeten Ansicht, dass zu einer DSL immer ein semantisches Modell [FP11, p. 159] exsistieren sollte, das unabhängig von der eigentlichen DSL ist aber direkt dazugehört. Es ist das Modell zu der DSL oder auch das Schema. Die DSL instanziiert das semantische Modell und ist damit eine gut lesbare Form der Modellinstanziierung. Das semantische Modell hat somit große Ähnlichkeit mir einem Domänen Modell [Fow03]. Vorteilhaft ist damit die klare Trennung von Angelegenheiten [**?**], auf der einen Seite das Parsen der DSL durch den Parser und auf der anderen Seite die daraus resultierenden Semantiken. Die DSL ist damit eine Abbildung des Modells. Veränderungen können an diesem formalen Modell separat zur DSL durchgeführt werden. Z.B. kann das semantische Model das von einer Zustandsmaschine sein wie in Abbildung 2.1 graphisch dargestellt.

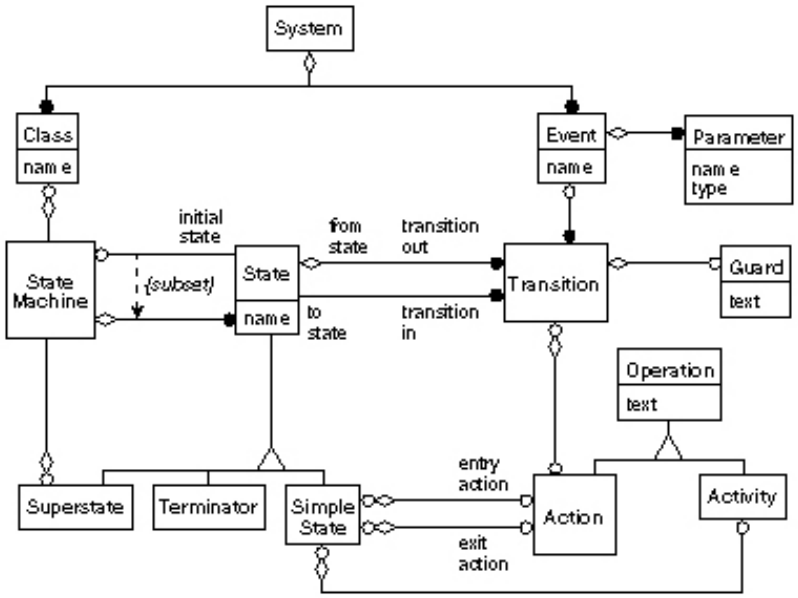

Abbildung 2.1: Semantisches Modell einer Zustandsmaschine - neilvandyke.org/smores/

z.B. ist das Metamodell von Zuständen eines Geldautomaten das Modell einer Zustandsmaschine (Abb. 2.1). Das Zustandsdiagramm des Geldautomaten ist damit das in Abbilung 2.2 dargestellte. Es beschreibt alle Zustaende und Zustandsübergänge bzw. deren vorausgehende Ereignisse bzw. Aktionen. Eine DSL könnte dieses Semantische Modell also Metamodell der Zustandsmaschine mit den Gegebenheiten des Geldautomaten instanziieren. (Code Listing 2.1).

Listing 2.1: DSL Ausschnitt für ein Geldautomat

```
1 event :KarteEinschieben
2 event :karteOk
```

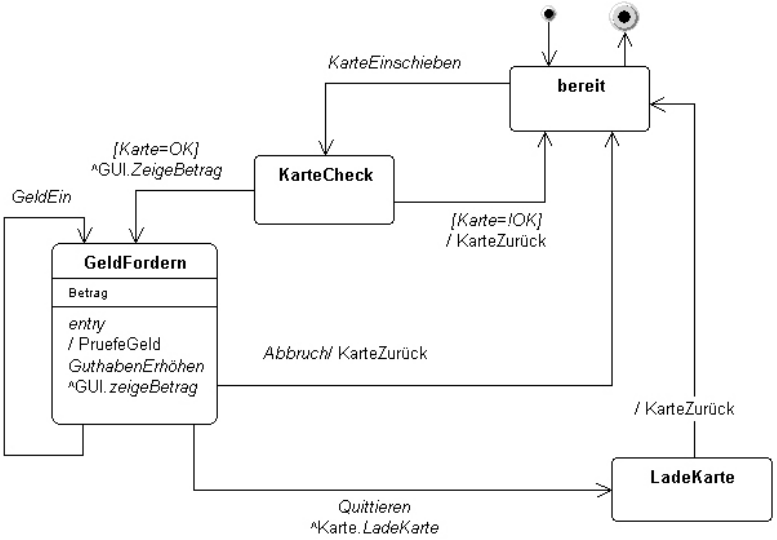

Abbildung 2.2: Zustandsdiagramm Geldautomat `https://www.fbi.h-da.de/uploads/RTEmagicC_f2da95d8df.gif.gif`

```
3  event :karteFalse
4  event :GeldEin
5  event :Abbruch
6  event :Quittieren
7  event :KarteZurueck
8
9  condition :karteOk
10 condition :karteFalse
11
12 state :bereit do
13         transitions :KarteEinschieben => :KarteCheck
14 end
15
16 state :KarteCheck do
17         actions:  GUI.ZeigeBetrag wenn :karteOK,
18                           null wenn :karteFalse,
19         transitions :karteOk => :GeldFordern
20                                 :karteFalse => :bereit
21 end
22
23 state :Geld_Fordern do
24         actions :KarteZurueck wenn :Abbruch
25         transitions :GeldEin => :Geld_Fordern,
26                                 :Abbruch => :bereit,
27                                 :Quittieren => :LadeKarte
28
29 end
30
31 state :LadeKarte do
32         actions :Karte.LadeKarte
33         transitions :KarteZurueck => :bereit
34 end
```

Im Sprachsektor des Techonologieradars (Juli 2011 von Thoughtworks)[1] sind sind die domänenspezifischen Sprachen unverändert nahe dem Zentrum angesiedelt. Thoughtworks ist der Meinung, dass DSLs eine alte Technologie ist und bei Entwicklern einen unterschätzten Stellenwert hat. (nach [Boa11]) Diese Quelle steht aber unter Vorbehalt in Hinblick auf den Verkauf des Buches von Martin Fowler (Chief Scientist von Thoughtworks) und Rebecca Parsons (CTO von Thoughtworks) über DSLs[2]

2.1.1 Unterscheidungen

Martin Fowler unterscheidet zwischen ausprägungen solcher DSLs, indem er deren Beziehung zu einer GPL benennt. Externe DSLs sind eigenständige und unabhängige Sprachen die einen eigenen sepziell angefertigten Parser besitzen.

"Sowohl die konkrete Syntax als auch die Semantik können frei definiert werden. SQL oder reguläre Ausdrücke sind Vertreter von externen DSLs. Wenn eine DSL innerhalb bzw." [unk12] mit einer GPL definiert wurde, nennt er diese interne DSL. Solche eingebettete Spracherweiterungen sind mit den gegebenene Mittel der "Wirtssprache", oft deren Möglichkeit zur Metaprogrammierung (Kapitel 2.4), erstellt. Vorzugsweise sind solche Wirtssprachen dynamisch typisiert wie z.B. Ruby, Groovy oder Scala. "Dadurch sinkt der Implementierungsaufwand. Eine interne DSL ist immer eine echte Untermenge einer generelleren Sprache." [unk12]

DSLs die nicht in der Hostsprache implementiert sind werden trotzdem von einer anderen Programmiersprache geparst und weiterverarbeitet. Zusätzlich ist man bei einer separaten Sprachdefinition nicht syntaktisch eingeschränkt. Damit sind vor allem Szenarien gemeint bei denen z.B. Methodennamen hinter die Argumentenklammer geschrieben ist: "(1,1)addiere" oder sonstige nur denkbare syntaktische Abwandlugnen.

2.2 Sprachorientierte Programmierung

Ein Programmierer einer Anwendung auf der Internet-Platform benutzt viele Sprachen die alle für einen bestimmten Zweck eingesetzt werden. HTML für die Struktur und CSS für die visuelle Repräsentation dieser Stuktur und SQL um Daten aus der Datenbank abzufragen bzw. aufzuwerten. Der Entwickler muss sich dann für ein Web-Framework entscheiden, das in einer GPL wie Java, Ruby oder PHP geschrieben ist. Nicht zuletzt ist Javascript noch

[1] http://www.thoughtworks.com/radar
[2] Domain Specific Languages, Addison Wesley 2011

eine zusätzliche Sprache die Client oder auch Serverseitig[3] eingesetzt wird. Das Paradigma sprachorientierte Programmierung ermuntert den Programmier dazu eigene Sprachen zu entwerfen die den jeweiligen Problembereich explizit beschreiben.

"Language oriented programming (LOP) is about describing a system through multiple DSLs[...]LOP is a style of development which operates about the idea of building software around a set of DSLs".(vlg. [Fow05])

Sprachorientierte Programmierung ist nach Fowlers Beschreibung als ein Vorgehen bei dem eine Programmiersprache sich auch mehreren Unter-Programmiersprachen zusammensetzt, deren Funktionen mit definierten Sprachmodellen beschrieben werden. Diese Sprachmodelle sollen die fachlichen Probleme auf eine natürliche Art beschreiben können, ohne dabei mehr zu beschreiben als das Fachgebiet benötigt. Dabei liegt der Fokus auf der Problembeschreibung und nicht auf der Lösungsbeschreibung.

In dem oft zitierten Paper beschreibt Fowler ein kleines Programm, das dazu dient Textdateien, die eine bestimmte Struktur haben in Objektinstanzen zu überführen. Eine EDI-Nachricht [4] (Electronic Data Interchange) ist ein Beispiel dafür (Listing 2.2).

Listing 2.2: EDIFACT Nachricht für einen Verfügbarkeitsanfrage

```
1  UNA:+.? '
2  UNB+IATB:1+6XPPC+LHPPC+940101:0950+1'
3  UNH+1+PAORES:93:1:IA'
4  MSG+1:45'
5  IFT+3+XYZCOMPANY AVAILABILITY'
6  ERC+A7V:1:AMD'
7  IFT+3+NO MORE FLIGHTS'
8  ODI'
9  TVL+240493:1000::1220+FRA+JFK+DL+400+C'
10 PDI++C:3+Y::3+F::1'
11 APD+74C:0:::6++++++6X'
12 TVL+240493:1740::2030+JFK+MIA+DL+081+C'
13 PDI++C:4'
14 APD+EM2:0:1630::6+++++++DA'
15 UNT+13+1'
16 UNZ+1+1'
```

Je nachdem wie z.B. die ersten drei Zeichen einer Zeile sind, wird beim Parser eine bestimmte "Parsing-Strategie" angewedet. [Gam95] Die angewendete Strategie wird dann dazu verwendet um den Rest der Zeile auszulesen. Diese Strategie kann zusätzlich konfiguriert werden. Folgendes ist nun konfigurierbar: Die ersten drei Zeichen an sich. Weithin die Zielklasse die je nach den ersten drei Zeichen mit den restlichen Daten der Zeile als Klassenvariablenwerte

[3] nodejs.org
[4] http://en.wikipedia.org/wiki/EDIFACT

instanziiert werden soll und vor allem an welcher Stelle der Zeile der Wert einer bestimmten Klassenvariable vorkommt.

Mit diesem kleinen Programm wurde eine Abstraktion gebaut, die durch Konfiguration der Strategien spezifiziert werden kann. Also um die Abstraktion zu benutzen müssen die Strategien konfiguriert werden und deren Instanzen an den Reader-Treiber übergeben werden (Listing 2.3).

Listing 2.3: Instanzen der verschiedenen Strategien

```
1  public void Configure(Reader target) {
2    target.AddStrategy(ConfigureServiceCall());
3    target.AddStrategy(ConfigureUsage());
4  }
5  private ReaderStrategy ConfigureServiceCall() {
6    ReaderStrategy result = new ReaderStrategy("IFT", typeof (
       ServiceCall));
7    result.AddFieldExtractor(7, 30, "RequestType");
8    result.AddFieldExtractor(19, 23, "CustomerID");
9    result.AddFieldExtractor(24, 27, "CallTypeCode");
10   result.AddFieldExtractor(28, 35, "DateOfCallString");
11   return result;
12 }
13 private ReaderStrategy ConfigureUsage() {
14   ReaderStrategy result = new ReaderStrategy("TVL", typeof (Usage
       ));
15   result.AddFieldExtractor(4, 8, "CustomerID");
16   result.AddFieldExtractor(9, 22, "CustomerName");
17   result.AddFieldExtractor(30, 30, "Cycle");
18   result.AddFieldExtractor(31, 36, "ReadDate");
19   return result;
20 }
```

Da diese Konfigurationen besser konfigurierbar machen ohne immer neuen Bytecode generieren zu müssen könnte man eine YAML[5]-Datei schreiben und diese als Input für die Strategiekonfiguration benutzen (Listing 2.4).

Listing 2.4: Instanzen der verschiedenen Strategien - YAML

```
1  mapping IFT dsl.ServiceCall
2    4-18: RequestType
3    19-23: CustomerID
4    24-27 : CallTypeCode
5    28-35 : DateOfCallString
6
7  mapping  TVL dsl.Usage
8    4-8 : CustomerID
9    9-22: CustomerName
10   30-30: Cycle
11   31-36: ReadDate
```

[5]http://de.wikipedia.org/wiki/YAML

Jemand, der den Parser und dessen Anwendung versteht, kann in kurzer Zeit etwas mit der YAML Datei anfangen. Genau der Inhalt dieser YAML-Datei ist nun schon eine kleine Sprache. Die konkrete Syntax ist genau das YAML Format. Eine andere konkrete Syntax könnte das XML Format sein. Da dieses aber zu “verbos” ist dient es nicht der leserlichkeit. Die Abstrakte Syntax ist nun die Basisstruktur: “Mehrere Abbildungen von Zeilentypen auf Klassen. Jeweils mit den drei Buchstaben, der Zielklasse und deren Felder bzw. deren Position”. Egal ob in XML oder in YAML, die abstrakte Syntax bleibt immer gleich.

Wenn eine minimalistische Syntax eine Vorraussetzung für eine DSL ist, kann man die YAML-Datei auch in Ruby darstellen (Listing 2.5). Das hat zur Folge, dass der Inhalt der DSL mit einem Ruby Interpreter gelesen und verarbeitet werden kann. Wenn auch der andere Code in Ruby geschrieben sein wuerde (Stragie Implementation, AddFieldExtractor, AddStrategy, ...) dann wäre Code Listing 2.5 eine interne DSL und Ruby die Wirtssprache. Also ein Untermenge von Ruby und eine konkrete- zu unseren abstrakten Syntax.

Listing 2.5: Instanzen der verschiedenen Strategien - RUBY

```ruby
mapping('SVCL', ServiceCall) do
        extract 4..18, 'customer_name'
        extract 19..23, 'customer_ID'
        extract 24..27, 'call_type_code'
        extract 28..35, 'date_of_call_string'
end
mapping('USGE', Usage) do
        extract 9..22, 'customer_name'
        extract 4..8, 'customer_ID'
        extract 30..30, 'cycle'
        extract 31..36, 'read_date'
end
```

2.3 Groovy

Diese Programmiersprache wurde entwurfen, um auf der JVM (Java Virtual Machine) ausgeführt zu werden. Ruby, Python, Dylan und Smalltalk dienten als Inspiration für die Entwickler von Groovy.

Eine Maxime für den Entwurf von Groovy war die hochgradige Kompatibilität zu Java. Die Sprache ist auch syntaktisch stark an Java angelehnt. Wenn eine .java Datei in eine .groovy Datei umbenannt wird, dann ist diese genau so ausführbar. Groovy Klassen können auch von Java Klassen erben.

Groovy besitzt Eigenschaften, die sich besser als die von Java eignen, um eine DSL zu entwerfen. Dazu zählt die wahlweise dynamische Typisierung, Closures, native Syntax für

Maps, Listen und Reguläre Ausdrücke, ein einfaches Templatesystem, eine XQuery-ähnliche Syntax zum Ablaufen von Objektbäumen, Operatorüberladung und eine native Darstellung für BigDecimal und BigInteger.

2.4 Metaprogrammierung

2.4.1 groovy syntaxeigentschaften

Omitting parentheses Groovy allows you to omit the parentheses for top-level expressions, like with the println command:

println "Hello"method a, b vs:

println("Hello") method(a, b) When a closure is the last parameter of a method call, like when using Groovy's 'each' iteration mechanism, you can put the closure outside the closing parens, and even omit the parentheses:

list.each(println it) list.each() println it list.each println it Always prefer the third form, which is more natural, as an empty pair of parentheses is just useless syntactical noise!

There are some cases where Groovy doesn't allow you to remove parentheses. As I said, top-level expressions can omit them, but for nested method calls or on the right-hand side of an assignment, you can't omit them there.

def foo(n) n

println foo 1 // won't work def m = foo 1

2.5 Metaprogrammierung in Groovy

2.5.1 Closures

2.5.2 Kategorien

3 MDA / MDSD und DSM Unterschiede

4 Praktischer Teil

Im praktischen Teil soll beschrieben werden, wie eine interne DSL mit Hilfe der Groovy-Metaprogrammierung erstellt wurde und wie der Domänenexporte darauf reagiert hat.

4.1 Die Fachliche Domäne

Der fachliche Bereich im gesamten Kontext ist die Hotellerie. Diese Arbeit betrachtet die Betriebswirtschaftliche Unterdomäne und darin, noch spezieller die tagesabhängige Preisbildung für Hotelzimmer.

4.2 Vorgehen

Nach der Zieldefinierung soll eine Bestandsaufnahme gemacht werden um die Rahmenbedingungen für das Experiment offenzulegen. Danach werden die Vorüberlegungen zur Zielerreichung dargestellt und anschließend die Implementierung der Lösung beschrieben. Daraus entstehen jeweils Unterziele die in den einzelenen Abschnitten näher beschrieben sind. Anschliessend bietet diese Arbeit jeweils eine Ausgewertung und Beurteilung.

4.3 Zieldefinierung

Das Ziel war es eine Sprache zu erstellen, die sich ausschließlich durch Preisbestimmung für jeden möglichen Tag in der Zukunft bzw. für jedes Apartment im Hotel definiert.

4.4 Bestandsaufnahme

Zuerst wurde unverbindlich eine Bestandsaufnahme gemacht. Das Hotel “Schoenhouse Apartments” besteht aus 50 Apartments in Berlin Mitte. Der Geschäftsführer ist Dipl.-Ing. Immanuel Lutz (Domänenexperte). Dieser bestimmt auch hauptsächlich die Preisbildung der

Apartments. Weiterhin besitzt das Hotel ein in Java geschriebenes Property-Management-System (PMS), das zur Verwaltung folgender Hauptkomponenten dient: Zimmer-, Gäste-, Apartment-, Sonderleitungs- und Preisverwaltung. Derzeit wird ein neuartiges PMS erstellt, dass auf Groovy-und-Grails basiert.

4.5 Vorüberlegungen

Die Vorüberlegung erfolgte ohne den Domänenexperten lediglich die Zustimmung für das Experiment "Textuelle-Preisberechnung" war gegeben, Da der Domänenexperte nur wenig Zeit dafür preisgeben wollte. Die Vorüberlegungen bestanden hauptsächlich aus der Grammatikstruktur und deren Semantik.

Die DSL soll die Geschäftslogik für die dynamische Bildung der Zimmerpreise beschreiben. In einem Hotel sind die Preise abhängig von Faktoren wie "Angebot und Nachfrage auf dem Markt", Investitionskosten, Zimmerkategorie, Nebenkosten, Rabattaktionen, Provisionen der Geschäftspartner für eine Zimmervermittlung, Zeitraum und überschneidende Ereignisse in der Umgebung [HK93, S. 44]. Mit Ereignissen sind Veranstaltungen oder Feiertage sowie Saisons gemeint, die die Angebot und Nachfragen beeinflussen. Bewertungen, die das Hotel auf Buchungsportalen bekommen hat sind auch Preisentscheident. Wenn z.B. mehrere schlechte Bewertungen abgegeben wurden und darauf nur noch wenige Gäste Buchen muss überlegt werden, ob das mit dem Preis zu regulieren geht. Nicht zuletzt beeinflusst die Auslastung einer Zimmerkategorie oder die Gesamtauslastung des Hotels den Zimmerpreis. Das bedeutet, wenn nur noch ein Zimmer im Hotel verfügbar ist, dann kann es entsprechend teuer verkauft werden. Die Auslastung, Ereignise und die Tage bis zu den Ereignissen zusammen kombiniert beinflussen den Preis weiter. Auch die aktuelle Liquidität des Unternehmens kann einfluss darauf haben. Der Zimmerpreis ist auch sensibel gegenüber den Preisen der direkten Konkurenz in der Umgebung. Der Faktor Markt ist der wohl am schwersten zu determinierende, da er sich aus vielen anderen Faktoren zusammensetzt. Dazu gehört z.B. die Beziehung zwischen angebotenen Hotelzimmern und nachgefragten. Wenn die Auslastung steigt und die Nachfrage gleich bleibt, dann resultiert das in steigende Preise. Bei sozialen, kulturellen oder positischen ereignissen weichen die Zimmerpreise erheblich von der "Rac-Rate" (Grundpreisrate) ab. Es stellt sich als schweirig heraus, alle Faktoren deterministisch zu modellieren, da vor allem der subjektive Geschmack oder persönliche Motivationen der potentiellen Gäste nur über statistische Werte berechenbar sind. Genau so ist es mit der Faktorenauswahl bei ökonomischen bzw. volkswirtschaftlichen Werten, um die Kaufkraft der internationalen Gäste zu bestimmen. Formal kann mit diskreten Werten

modelliert werden, die in direkter Beziehung zu dem Hotel stehen. Indirekte Beziehungen werden hier aus den oben genannten Gründen nicht betrachtet.

Der Geschäftsführer muss genau diese Preislogik für sein Unternehmen individuell, unabhängig und zeitnah regeln können.

4.6 Erstellung der DSL

Begonnen hat die Erstellung der DSL mit der Vorstellung, dass es im PMS ein Textbereich gibt in dem die der Text eingefügt und editiert werden kann. Der Texteditor sollte mindestens ein “Rich-Text-Editor” sein, damit der Domänenexpoerte den Text formatieren kann. Unter dem Textbereich ist es notwenig, zwei Buttons bereitzustellen. Einen um die DSL Live anzuwenden und einen um die DSL zu simulieren also zu testen.

Da das zukünftige PMS in Groovy und Grails erstellt wird und Groovy viele Möglichkeiten der Metaprogrammierung und der DSL erstellung explizit hat, ist es naheliegend, das Experiment in einer internen DSL umzusetzen.

Aus der Zieldefinition geht hervor, dass das Resultat der Preisberechnungslogik eine Tabelle sein muss, die für jeden Tag und jeden Zimmertyp eine Gleitkommazahl als Preis beinhaltet (Tabelle 4.1).

Datum	Zimmerkategorie	Tagespreis
1.1.2013	Zimmerkategorie1	95.00
1.1.2013	Zimmerkategorie2	105.00
2.1.2013	Zimmerkategorie1	95.00
2.1.2013	Zimmerkategorie2	105.00
3.1.2013	Zimmerkategorie1	95.00
3.1.2013	Zimmerkategorie2	105.00

Tabelle 4.1: Zielstruktur

Perspektivisch war der Gedanke, dass man von einer Menge alle Elemente dazu benutzen muss, um den Preis zu bilden. Damit ist gemeint, dass sich die Tage in einer Menge befinden und auch die Zimmerkategorien Mengenbasierend sind. Weiterhin besteht eine Berechnungslogik zum größten Teil aus mathematischen Ausdrücken bzw. Formeln. Bemerkenswert ist, das die Erstellung der Domänenlogik nicht zuerst auf der Grundlage des semantischen Modells erstellt wurde sondern rein Intuitiv auf Basis von bekanntem Domänenwissen. Da sich das semantische Modell als das einer Bash-Script-Sequenz mit Schleifen herausstellte, ist es nicht verwunderlich, dass ein Programmierer mit langjähriger Erfahrung das auch ohne Schema erstellen konnte. Begonnen wurde mit einem TestTreiber, der eine Textdatei einliest und

diese Interpretiert bzw. evaluiert. Cliff James hatt das in einem Tutorial[1] bewerkstelligt und folgenden Trick angewendet: Die DSL befindet sich innerhalb einer separaten Datei und ist nach Ausführung des Einlesecodes in einen interpolierten String umgewandelt. Anschließend wird dieser String in einen Closure-Block eingefügt. Da dieser eingefügte String, innerhlab einer “run” Methode liegt ist der Aufruf immer der selbe. Die DSL wird letztendlich von der Groovy Shell Instanz evaluiert ([Koe07, S. 368]). Doch ohne den Kontex um die DSL, speziell ausenstehende Werte ist die DSL nutzlos. Daher wird eine Instanz von Binding ([Koe07, S. 368]) dazu benutzt um Variablen an das Script zu übergeben. die Binding Instanz wird dazu benutz um die Variable run eine Closure zuzuweisen die die loadDSL Methode im runner aufruft. Die Binding instanz wird dann an die GroovyShell instanz übergeben um die Assoziationen zu gewährleisten (Code Listing 4.1) . Mit Groovy-Metaprogrammierung ist es möglich den Kontext einer Instanz zu wechseln, also die Instanz einer Klasse. Delegate wechselt also zu “this” und damit ist dann die DSL Bestandteil des DSLRunners. Das bedeutet, das alles was in der DSL-Datei geschrieben wurde jetzt Methoden und Variablen der DSLRunner-Klasse referenzieren kann.

Listing 4.1: DSL-Runner

```
1  class DSLRunner {
2
3      void loadDSL(Closure cl) {
4        println "loading DSL ..."
5            cl.delegate = this
6            cl()
7      }
8
9      void usage() {
10         println "usage: DSLRunner <scriptFile>\n"
11         System.exit(1)
12     }
13
14     static void main(String [] args) {
15       DSLRunner runner = new DSLRunner()
16       if(args.length < 1) { runner.usage() }
17
18       def script = new File(args[0]).text
19       def dsl = """run {  ${script} }"""
20
21       def binding = new Binding()
22       binding.run = { Closure cl -> runner.loadDSL(cl) }
23       GroovyShell shell = new GroovyShell(binding)
24       shell.evaluate(dsl)
25    }
26 }
```

[1] http://www.nextinstruction.com/blog/2012/01/08/creating-dsls-with-groovy/

Da nun jedmögliche Textdatei an den DSLRunner übergeben werden konnte, um Groovy-Script-Code auszuführen ist es dementsprechend auch möglich den Inhalt des besagten Textfeldes als Input zu benutzen. Diese triviale Implementierung wurde übersprungen. Eine Schleifeniteration besteht aus einer List oder einem Abschnitt (Range) gefolgt von der each Methode und der auszuführenden Closure als Parameter für diese “each” Methode 4.2.

Listing 4.2: Orgniale Groovy Schleifenbeispiel http://groovy.codehaus.org/Collections

```
1 (1..10).each({ i ->
2   println "Hello ${i}"
3 })
```

Die Versionkontrollhistorie zeigt, dass der erste Eintrag in der DSL aus einer Schleife über einem Zeitraum von zwei Jahren erstellt wurde. Denn eine finale Liste kann nur durch eine Schleifenähnliche Funktion erstellt werden 4.3.

Listing 4.3: erster DSL Enturf

```
1 (heute.bis(heute + 2.years)).each({ day ->
2         println day
3 });
```

Durch die im theoretischen Teil vorgestellten Kategorien war es nun möglich anstatt der speziellen Notation für zwei Jahre, bzw. die Instanziierung einer Dauer (Duration) die Notation 2.years zu verwenden.

Weiterhin wurde die Instanz der ExpandoMetaClass der Date-Klasse (Date.metaClass) dazu verwendet, um eine Methode namens “bis” für die Date-Klasse zu definieren, die wieder ein Date Objekt als Argument entgegennimmt und daraus einen (Zeit)Abschnitt (engl. Range, Notation: “start..stop”) daraus ableitet.

Durch das Binding Objekt konnte die vordefinierte instanz (new Date()) mit dem Variablennamen “heute” übergeben werden. Diese Variable konnte somit in der DSL also als solche verwendet werden.

Die erste Spalte der Zieltabelle ist somit darstellbar, aber die lesbarkeit war erhelblich durch Sonderzeichen beeinträchtigt. Ziel war nun die Lesbarkeit erheblich zu steigern indem Sonderzeichen weitestgehend eliminiert werden und englische Begriffe durch deutsche zu ersetzen. Zuerst wurde das wort “each” durch “alle” ersetzt. Das gelang dadurch, dass die die Bedeutung an sich “jedes Element” in einer bestimmten Menge (engl. Collection) ihren Ursprung hat. Hier wurde wiederrum das ExpandoMetaObject dafür Benutzt um der Überklasse “java.util.Collection” eine Closure für die neu definierte Eigenschaft “alle” zu

übergeben. Die Closure sollte nun eine Iteration über alle Elemente in der Menge leisten und dabei nochmals eine Closure entgegennehmen in der dann die Operation auf das Element definiert wird. Ausserdem muss der Deligierte wieder auf die Mengeninstanz gewechselt werden. Abbilung 4.4 zeigt den Codeabschnitt im DSLRunner.

Listing 4.4: .alle({..})]Definition von [1,2].alle({..})

```
1 Collection.metaClass.alle = { Closure closure -> // eine Closure
    als Argument der Methode alle also [1,2].alle({...})
2     delegate.each { // alle Elemente in der Collection
3         closure.delegate = closure.owner //neuzuweisung des
              delegierten
4         closure(it) //closureaufruf
5     }
6 }
```

Weiterhin wurde aus 2.years eine neue Kategorie definiert, die die deutsche Bezeichnung von Jahren benutzt. Also 2.jarhe oder 1.jahr. Dazu wurde das metaClass ExpandoMetaObject von der Klasse Number dahingehend verändert, dass solche Konstruktionen möglich werden (Code Listing 4.5).

Listing 4.5: Expando Metaclass Jahre

```
1 Number.metaClass {
2   use(TimeCategory) { // *.years. *.days, *.months ...
3     getJahre { delegate.years } // 10.jahre = 10.years
4     getJahr { delegate.jahre }  // 1.jahr = 1.years
5   }
6 }
```

Durch die vorgestellten Syntaxeigenschaften ist es möglich die Klammern wegzulassen und damit den in Code Listing 4.6 dargestellten DSL-Code zu erzeugen.

Listing 4.6: Iterationsnotation auf Basis von Kategorien

```
1 heute bis 2.jahre alle { tag ->
2         ...
3 }
```

Um diesen gut lesbaren Code noch mehr an die deutsche Ausdrucksweise anzulehnen ist die Verwendung von Command Expressions hilfreich um eine Fluid DSL zu erstellen. Das bedeutet, dass in der deutschen Sprache eigentlich folgender Ausdruck der natürlichste waer: “alle Tage von heute bis in zwei Jahren einzeln auflisten und jeden Tag immer als Tag bezeichnen.” Nun diese ehr lange Ausdrucksweise ist zwar prezise aber enthält gegenüber einer mit minimalen Sonderzeichen geschriebenen Notation noch zu viele Begriffe. Ein valider

Kompromiss ist folgender: “von heute bis 2.jahre alleTage { tag -> }”. Dieser Kompromiss wurde ausgehend von der vorhandenen Programmiersprache in der die DSL “eingebettet” sein soll und der subjektiven Empfindungsweise des Erstellers gemacht. Die Präposition “von” ist der Name einer Methode, die als Argument ein Datum akzeptiert und eine Methode als Rückgabewert hat. “Von” ist somit eine Methode höherer Ordnung[2] und in Code Listing 4.7 dargestellt.

Listing 4.7: Fluent Interface Implementierung

```
1 def von(Date start) {
2       [bis: { end ->
3               use(groovy.time.TimeCategory) {
4                   (start..end)
5               }
6           }
7       }]
8 }
```

Der Rückgabetyp ist eine HashMap mit “keys” als Methodennamen und Closures als “values” bzw. dazugehörige “Methodenkörper”. Wenn genau das der Fall ist, ist so ein Listeneintrag widerrum ein Objekt an dem Methoden aufgerufen werden können. Wenn eine Map zurückgegeben wird, dann identifiziert sich der eintrag der Map anhand des Schlüssels (“bis”). Bis referenziert somit einen Closurekörper, der ein Argument entgegennimmt das vom Typ Date ist. Letztendlich gibt diese Closure ein Instanz von Range zurück. Der Vorteil dabei ist, dass das Argument vom ersten Methodenaufruf (“von(datum)”) in der Closure des zweiten Methodenaufrufs benutzt werden kann und somit dieses “Fluent Interface” eine abgekapselte Einheit darstellt. Daraus folgt nun folgende Notation neue Notation für die DSL (Code Listing 4.8). Durch triviales kopieren der “alle” zu “alleTage” ExpandoMetaObejkt Instanz wurde nach dem DRY[3] Prinzip die Closure wiederverwendet und alleTage steht für eine Menge, genauer für eine ObjectRange zur Verfügung.

Listing 4.8: Fluent Interface Anwendung

```
1 von heute bis 2.jahre alleTage { tag ->
2         ...
3 }
```

Wie in der Vorbetrachung angemerkt beziehen sich die Preise nicht nur auf den Zeitraum sondern werden nicht jedem einzelnen Zimmer sondern einer Kategorie zugeordnet. Die Information aus der Hostdomäne (Abb. 4.1) bzw. alle Zimmerkategorien, muss nun in Verbindung mit der DSL gebracht werden.

[2] de.wikipedia.org/wiki/Funktion_höherer_Ordnung
[3] Dave Thomas, interviewed by Bill Venners (2003-10-10) http://www.artima.com/intv/dry.html

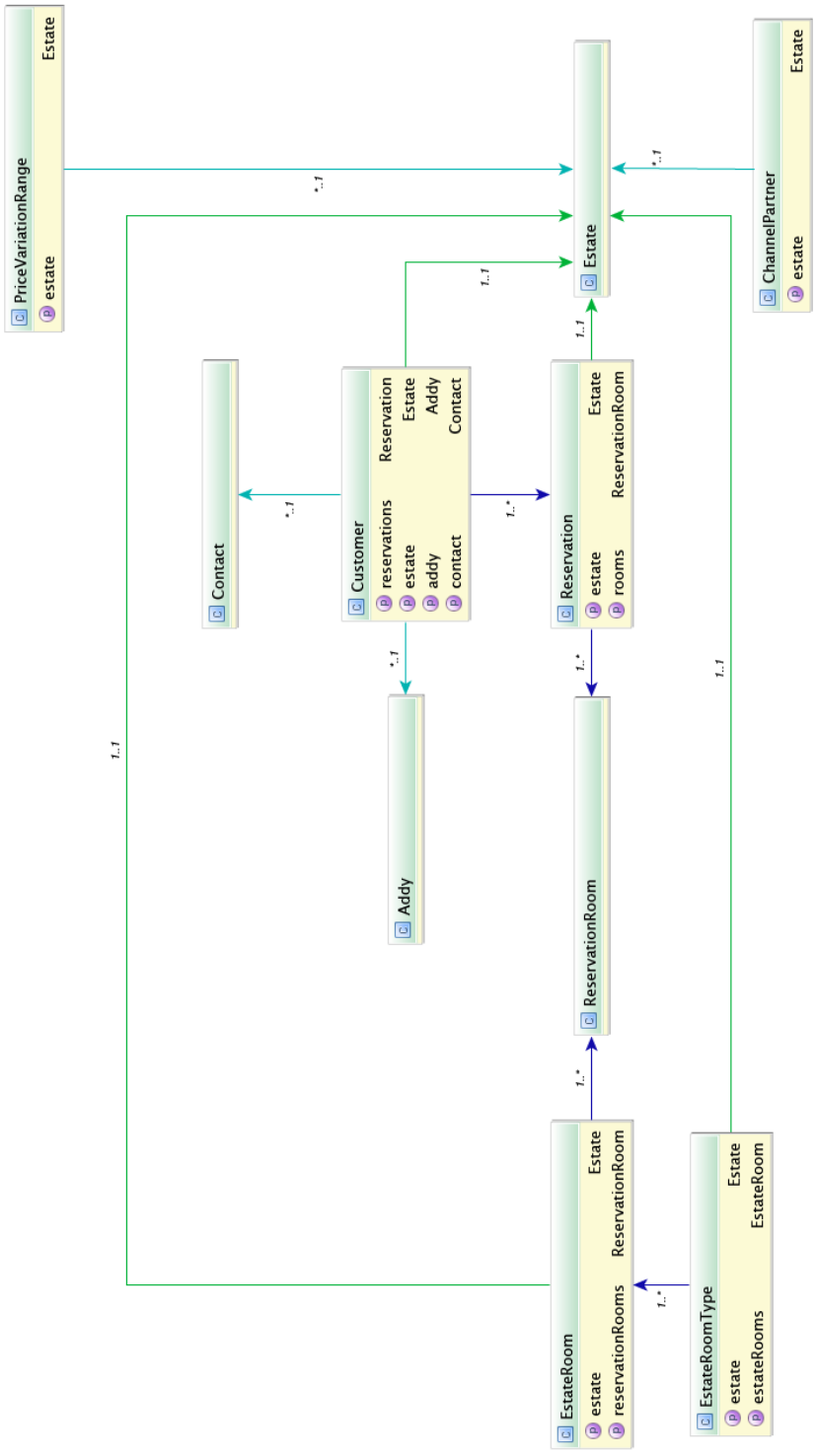

Abbildung 4.1: PMS - Domänen Modell Diagramausschnitt

Bestenfalls sollte folgende Semantik vollkommen ausreichen: “Hotel.Zimmerkategorien” um die Menge abzubilden. Das Binding Objekt erlaubt nun eine Referenzierung der Domänenmodellinstanzen mittels dem Grails Framework. Problematisch ist die unterschiedliche

Benennung der DSL komponenten und der Domänenmodelle. Beispielsweise heisst das Hotel im Domänenmodell “Estate” und in der DSL nur “Hotel”. Wiederrum heissen die Zimmerkategorien nicht so sondern “EstateRoomType”. Es ist also notwendig ein Mapping zu erstellen, dass genau diese Fälle abdeckt. Der Binding Schlüsselwert für das Estate Objekt ist dann Hotel. Da aber die Zimmerkategorien auf kein Feld innerhalb des Domänenmodells referenziert, muss ein erneutes Mapping erfolgen. Trivial wäre es in dem Domänenobjekt eine Kopie auf “estateRoomTypes” zu machen. Da aber so keine Kapselung erreichen wird ist es notwendig ein WrapperObjekt zu erstellen und das an die DSL zu binden (Code Listing 4.9).

Listing 4.9: EstateDSLWrapper.groovy

```
class EstateDSLWrapper {
    Estate estate
    EstateDSLWrapper(Estate estate) {
        this.estate = estate
    }
    def Zimmertypen = estate.roomTypes;
}
```

Das Binding ist in Listing 4.10 dargestellt.

Listing 4.10: estateBinding.groovy

```
binding."Hotel" =
new EstateDSLWrapper(
        new Estate(
                name: "Schoenhouse",
                estateRoomTypes: [
                new EstateRoomType(name: "typ1", grundpreis: 95),
                new EstateRoomType(name: "typ2", grundpreis: 105)
                ]
        )
);
```

Analog dazu ist dieses Vorgehen auch mit den definierten Ereignissen “PriceVariationRange” durchführbar, welche aus dem Domänenmodell an die DSL gebunden werden. Da eine Iterationsnotation (“alle”) eingeführt wurde ist es insgesamt nun möglich **Schleife**n zu schachteln Code Listing 4.11.

Listing 4.11: multipleLoops.dsl

```
Hotel.Zimmertypen.alle { ZimmerTyp ->
  von heute bis 3.months alleTage { Tag ->

    TagesPreis = ZimmerTyp.Grundpreis

    Ereignisse.alle { Ereignis ->
        ...
```

```
8       }
9     }
10  }
```

In Code Listing 4.11 ist Zusätzlich auch schon der erste **Ausdruck** in Zeile 4 dargestellt. Trivialer weise handelt es sich um eine Variablendefinition inklusive **Zuweisung**. Dieser Greift auf die Iterationsvariable “ZimmerTyp” zu und referenziert die in dem Wrapper festgelegte Eigenschaft Grundpreis. Im DomänenModell Estate heisst diese Klassenvariable “racRate”. Die **Variable** Tagespreis ist letztendlich die die modifiziert werden soll und anschließend in die Ergebnistabelle dem Tag und der Kategorie zugewiesen werden soll. Die Endtabelle soll in Form einer Liste definiert werden um dann mit dem Listenoperator («) diese zu füllen. Die Listennotation ist trivialerweise folgende: “listenname = []”.

Bisher wurden alle Informationen beschrieben um eine finale implementierung durchzuführen. Code Listing 4.12 zeigt, das die Zielstellung dahingehen erreicht ist, das eine Liste wie in Tabelle 4.1 durch die DSL berechnet wird.

Listing 4.12: Triviale Lösung des Problems

```
1
2  liste = []
3
4  Hotel.Zimmertypen.alle { ZimmerTyp ->
5    von heute bis 3.months alleTage { Tag ->
6
7      TagesPreis = ZimmerTyp.Grundpreis
8
9      liste << [ ZimmerTyp.name, Tag, TagesPreis ]
10   }
11 }
12
13 /*
14 Ausgabe:
15 ...
16 typ1, Fri Apr 06 12:14:52 CEST 2012, 95.00
17 typ1, Sat Apr 07 12:14:52 CEST 2012, 95.00
18 typ1, Sun Apr 08 12:14:52 CEST 2012, 95.00
19 ...
20 typ2, Thu May 17 12:14:52 CEST 2012, 105.00
21 typ2, Fri May 18 12:14:52 CEST 2012, 105.00
22 typ2, Sat May 19 12:14:52 CEST 2012, 105.00
23 typ2, Sun May 20 12:14:52 CEST 2012, 105.00
24 typ2, Mon May 21 12:14:52 CEST 2012, 105.00
25 typ2, Tue May 22 12:14:52 CEST 2012, 105.00
26 typ2, Wed May 23 12:14:52 CEST 2012, 105.00
27 typ2, Thu May 24 12:14:52 CEST 2012, 105.00
28 ...
29 */
```

Das Resultat kann automatisch und transparent gegenüber dem Domänenexperten durch ein XML oder JSON Mapping an die “PartnerChannels” (Booking.com oder HRS) geschickt werden. Das zu implementieren ist nicht Bestandteil dieser Arbeit.

Da das Grundgerüst der DSL damit geschaffen ist erfolt nun die Anpassung des Tagespreises durch **Formel**n und **Bedingung**en.

Da der Domänenexperte höchstwahrscheinlich mit Prozenten arbeiten will sollte es für denjenigen möglich sein diese Zahlen**funktion** einfach benutzen zu können. Mit Hilfe von Kategorien ist es möglich das zu bewerkstelligen um letztendlich folgendes DSL Wort[4] zu erstellen: “10 prozent Tagespreis” die Kategorie dazu lautet ist in Code Listing 4.13 dargestellt.

Listing 4.13: Kategoriedefinition für Prozent

```
1  //Definition
2  class EnhancedNumber {
3          static Number prozent(Number self, Number other) {
4              other * self / 100
5          }
6  }
7
8  //Anwendung
9  use(EnhancedNumber) {
10         assert 10 prozent 100 == 10
11 }
```

Wie in der Vorüberlegung schon angedeutet gibt es z.B. eine Preiserhöhung wenn ein bestimmtes Ereignis eingetroffen ist. So ein wiederkehrendes Ereignis ist z.B. ein Wochenende. Wenn also der Domänenexperte sich dazu entscheidet den Preis am Wochenende um 10% anzuheben sollte er folgendes in der DSL schreiben können: “wochenendaufschlag = wenn tag.wochenende dann 10 prozent tagesPreis”. Wieder wurde hier die Methode eines Fluent Interfaces benutzt wie bei der Zeitabschnittbestimmung (von(x).bis(y)). In Listing 4.14

Listing 4.14: DSL - if else Ausdruck

```
1  def wenn(bedingung) {
2      [dann: { statement ->
3                  bedingung ? statement : 0
4      }]
5  }
```

Diese Wenn dann Kombination ist wie an der 0 zu erkennen nur für Formeln einsetztbar. Alle Zusätzlichen Erweiterungen für die Date-Klasse sind in Code Listing 4.15 dargestellt. Darunter befindet sich auch die Erweiterung “getWochenende bzw. wochenende”.

[4] http://de.wikipedia.org/wiki/Wort_$Theoretische_Informatik$

Listing 4.15: Erweiterungen für die Date-Klasse

```
use(TimeCategory) {
    Date.metaClass {
        getWochenende = {
                        date = delegate
            date[Calendar.DAY_OF_WEEK] == Calendar.SATURDAY ||
            date[Calendar.DAY_OF_WEEK] == Calendar.SUNDAY
        }
        bis = { Date bis ->
                        def von = delegate
            (von..bis)
        }
    }
}
```

Dem Domänenexperten wird nun unterstellt, dass er x += 1 als Summierung für x = x + 1 erlerenen kann. Letztendlich wäre er nun in der Lage folgenden Ausdruck zu schreiben: “TagesPreis += wenn Tag.wochenende dann 10 prozent TagesPreis”.

Weiter könnte sich der Hotelbetreiber dazu entscheiden folgende modifikation an dem Tagespreis durchzuführen: Je nach dem wie das Hotel prozentual ausgelastet ist, wird der Tagespreis, um diesen prozentualen Anteil von einem drittel des Grundpreises, erhöht oder verringert. Wieder durch die Binding-Möglichkeit können weitere vordefinierte Variablen übergeben werden. Z.B. “binding.gesamtzimmer = Estate.estateRoomTypes*.count()” und weiterhin die Anzahl der freien Zimmer als Methode (Code Listing 4.16)

Listing 4.16: Vordefinierte Variablen

```
//freie Zimmer Funktion
def freieZimmer(tag) {
    BerechnungsService.freieZimmer(tag)
}
...
//  bergabe   der gesamtzimmer an die DSL
binding.gesamtzimmer = Estate.get("schoenhouse").estateRoomTypes
   *.size()

//-----------------------------------------------------------------
//Benutzung dieser DSL Spracherweiterung
verf gbareZimmer = freieZimmer tag
// abh ngig von der Auslastung wird ein Teil von einem Drittel
   der Grundkosten aufaddiert.
tagesPreis += verf gbareZimmer / gesamtzimmer * (typ.grundpreis
   / 3)
```

Abschließend soll hier noch weiter die Möglichkeit vorgestellt werden wie auf die vorher erwähnten bzw. vordefinierten Ereignisse zugegriffen werden kann um eine Tagespreismanipulation durchzuführen. Code Listing 4.17 zeigt eine mögliche Form der DSL in der ca. 90%

der Konzepte Beispielhaft dargestellt sind.

Listing 4.17: DSL Beispiel

```
liste = []

Hotel.Zimmertypen.alle { typ ->

  von heute bis 2.jahre alleTage { tag ->  //oder: heute bis 2. months

    tagesPreis = typ.grundpreis

    ereignisse.alle { ereignis ->

        TagInnerhalbEreignis = tag.innerhalb ereignis

        tagesPreis += wenn TagInnerhalbEreignis dann 10 prozent tagesPreis   // oder auch:  10 / tagesPreis * 100

        tageEntfernt = tage von: heute, bis: ereignis.start // oder: abstand { von heute bis ereignis.von }

        nichtvorbei = tageEntfernt > 0
        bald = tageEntfernt < 10

        lastMinuteRabatt = (tageEntfernt * 0.5).prozent tagesPreis

        tagesPreis += wenn bald und nichtvorbei dann lastMinuteRabatt
    }

    freieZimmer = freieZimmer tag    // 0.5 = die h lfte aller zimmer ist belegt.

    tagesPreis += freieZimmer / gesamtzimmer * (typ.grundpreis / 3)

    wochenendaufschlag = wenn tag.wochenende dann 10 prozent tagesPreis

    tagesPreis += wochenendaufschlag

    liste << [typ.name, tag, tagesPreis]


   }

}
```

Dabei sei nochmal auf die besondere Konstruktion hingewiesen “tage von: heute, bis: ereignis.start”. Das ist eine spezielle Notation in Groovy namens “named parameters”. “tage” ist eine Methode, die zwei Parameter entgegennimmt. “von” und “bis”, die auch so benannt

werden müssen. Durch das Entfernen der klammern ist jetzt die dahinterliegenede Struktur zu erkennen. Die Alternative hinter dem Ausdrück im Kommentar (Zeile 15) ist anders aufgebaut. “abstand” ist nun eine Methode, die eine Closure als Argument entgegennimmt, in diesem Fall einen Abschnitt (Range).

4.7 Das semantischen Modell

Im Kapitel 4.6 wurde beschrieben wie die DSL erstellt wurde. Dabei wurden einige Begriff in bold markiert um die wichtigsten Meta-Bestandteile unterschwellig zu verdeutlichen. In Abbilung 4.2 ist deren Zusammensetztung bzw. das semantische Modell der Preisberechnung dargestellt. Dieses ist sehr stark an das einer Skriptsprache angeleht. In diesem Modell sind nur Berechnungsbestandteile durch Asdrücke und Schleifen definiert. In einer Scriptsprache ist darüber hinaus noch mehr möglich.

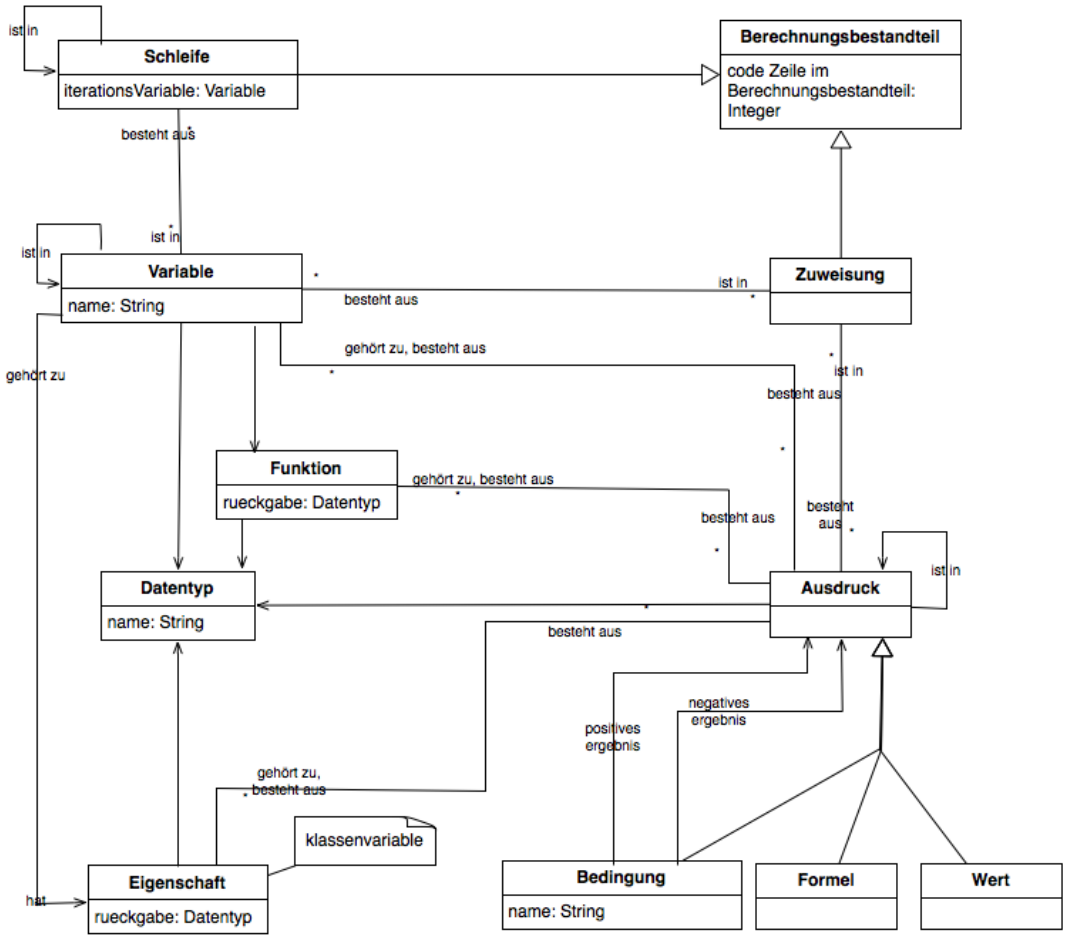

Abbildung 4.2: Preisberechnung semantisches Modell

Die Erstellung dieses Modells, wurde nach der Erstellung der DSL gefertigt um daraus z.B. eine bessere Dokumentation zu erstellen, die dem Domänenexperten hilft die DSL zu erstellen.

5 Auswertung

5.0.1 Wahl der DSL Variante

Bewertung des Probands: Lesbarkeit, intuitivem Verständnis und Flexibilität

5.0.2 Beurteilung des Domänenexperten

5.0.3 Beurteilung der Meta-Programmierungstools von Grovvy

einfacher da kein aufwand für gui zu betreiben ist.

6 Zusammenfassung und Schlussbetrachtung

[PT06]

Literaturverzeichnis

[Boa11] Board, ThoughtWorks Technology A.: Technology Radar. (Jan. 2011), Januar. http://www.thoughtworks.com/sites/www.thoughtworks.com/files/files/thoughtworks-tech-radar-january-2011-US-color.pdf

[Fow03] Fowler, M.: *Patterns of enterprise application architecture.* Addison-Wesley Professional, 2003

[Fow05] Fowler, M.: Language workbenches: The killer-app for domain specific languages. In: *Accessed online from: http://www. martinfowler. com/articles/languageWorkbench. html* (2005)

[FP11] Fowler, M. ; Parsons, R.: *Domain-specific languages.* Addison-Wesley Professional, 2011

[Gam95] Gamma, E.: *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995

[hei11] heise: *JetBrains verffentlicht Meta Programming System 2.0.* http://www.heise.de/developer/meldung/JetBrains-veroeffentlicht-Meta-Programming-System-2-0-1327137.html. Version: 22.08. 2011

[HK93] Hahn, H. ; Kagelmann, H.J.: *Tourismuspsychologie und Tourismussoziologie: Ein Handbuch zur Tourismuswissenschaft.* Quintessenz Verlags-GmbH, 1993

[Koe07] Koenig, et a. D.: *Groovy in action.* Manning Publications Co., 2007

[mda] *modelgetriebene softwareentwicklung (techniken, engeneering, management)*

[PT06] Pekka-Tolvanen, J.: Domnenspezifische Model- lierung fr vollstndige Code-Generierung,. In: *Javaspektrum* (2006), S. 9–12

[Rec00] Rechenberg, P.: *Was ist Informatik?: eine allgemeinverständliche Einführung.* Hanser Verlag, 2000

[unk12] unknown: *Domnenspezifische Sprache.* http://de.wikipedia.org/wiki/Domaenenspezifische_Sprache. Version: 2 2012

[wika] *Abstraktion*. `http://de.wikipedia.org/wiki/Abstraktion`

[wikb] *Intentional Programming*. `http://de.wikipedia.org/wiki/Intentional_Programming`