Fachbereich Informatik und Medien
Masterstudiengang Informatik - Projekt III
Prof. Dr. rer. nat. Gabriele Schmidt, Prof. Dr.-Ing. Susanne Busse

Untersuchung des sprachorientierten Programmierparadigmas anhand von Metaprogrammierung und einer damit erstellten DSL für Preispolitik in einem Apartmenthaus.

Vorgelegt von: Nils Petersohn B.Sc.
am: 29.02.2012.

Zusammenfassung

Sprachorientierte Programmierung ist in dieser Arbeit betrachtet worden. Dazu wurde eine DSL mit Hilfe der Groovy-MOP erstellt, um daran das Paradigma zusammen mit dem Domänenexperten unter folgenden Gesichtspunkten zu bewerten: Lesbarkeit, intuitivem Verständnis und Flexibilität. Der Autor beschreibt intensiv die Erstellung der DSL und damit die Metaprogrammierung der gewählten Programmiersprache.

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Das Wort “Abstraktion” bezeichnet meist den induktiven Denkprozess des Weglassens von Einzelheiten und des Überführens auf etwas Allgemeineres oder Einfacheres [...] also [...] jenen Prozess, der Informationen soweit 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 so umfangreich, 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 [SVEH07]).

Im Sprachsektor des Technologieradars (Juli 2011 von Thoughtworks)[1] sind die domänenspezifischen Sprachen unverändert nahe dem Zentrum angesiedelt. Thoughtworks ist der Meinung, dass DSL 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]

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 bestehen 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

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

auszudrücken." (vlg. [wikb]) Es werden zwei Arten von DSLs unterschieden [FP11].

1.3 Aufgabenstellung

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

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

Dabei stellen sich folgende Fragen: Wie effizient ist die Erstellung einer DSL mittels der Programmiersprache Groovy (Kapitel 5.2) und wie reagiert der Domänenexperte auf die DSL (Kapitel 5.1)?

Domänenexperten, 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 bekommen die Probanden mehrere Aufgaben, die mit den gegebenen DSLs zu lösen sind.

1.4 Gliederung

Im theoretischen Teil stellt diese Arbeit eine Abhandlung über die Grundlagen des "sprachorientierten Programmierparadigmas" (LOP) dar und die damit verbundene Metaprogrammierung speziell für die Programmiersprache "Groovy".

Der praktische Teil befasst sich mit der Anwendung dieses Paradigmas. Der Autor entwirft dazu eine interne DSL in Groovy und erklärt die Vorgehensweise von der Idee bis zur Implementierung. Anschließend findet eine Auswertung bzw. Bewertung des Paradigmas und dessen Nutzen statt.

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. Es werden nur interne DSLs betrachtet und es findet kein Vergleich zu externen DSLs statt.

2 Sprachorientierte Programmierung

Ein Programmierer einer Anwendung für die Browser-Platform benutzt viele Sprachen, die alle für einen bestimmten Zweck eingesetzt werden: HTML für die Struktur der Anwendung, CSS für die visuelle Repräsentation dieser Struktur 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 eine zusätzliche Sprache, die Client oder auch Serverseitig[1] 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. [Fow05b])

Sprachorientierte Programmierung ist nach Fowlers Beschreibung ein Vorgehen, bei dem eine Programmiersprache sich aus 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 betrifft. 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 [2] (Electronic Data Interchange) ist ein Beispiel dafür (Listing 2.1).

Listing 2.1: 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'
```

[1] nodejs.org
[2] http://en.wikipedia.org/wiki/EDIFACT

```
IFT+3+NO MORE FLIGHTS'
ODI'
TVL+240493:1000::1220+FRA+JFK+DL+400+C'
PDI++C:3+Y::3+F::1'
APD+74C:0:::6++++++6X'
TVL+240493:1740::2030+JFK+MIA+DL+081+C'
PDI++C:4'
APD+EM2:0:1630::6+++++++DA'
UNT+13+1'
UNZ+1+1'
```

Je nachdem wie z.B. die ersten drei Zeichen einer Zeile sind, wird beim Parser eine bestimmte "Parsing-Strategie" angewendet. Die angewendete Strategie[GHJV94] wird dann dazu verwendet, um den Rest der Zeile auszulesen. Diese Strategie kann zusätzlich folgendermaßen konfiguriert werden: Die ersten drei Zeichen an sich, die Zielklasse die je nach den ersten drei Zeichen mit den restlichen Daten der Zeile als Klassenvariablenwerte 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 2.2).

Listing 2.2: Instanzen der verschiedenen Strategien

```
public void Configure(Reader target) {
  target.AddStrategy(ConfigureServiceCall());
  target.AddStrategy(ConfigureUsage());
}
private ReaderStrategy ConfigureServiceCall() {
  ReaderStrategy result = new ReaderStrategy("IFT", typeof (
      ServiceCall));
  result.AddFieldExtractor(7, 30, "RequestType");
  result.AddFieldExtractor(19, 23, "CustomerID");
  result.AddFieldExtractor(24, 27, "CallTypeCode");
  result.AddFieldExtractor(28, 35, "DateOfCallString");
  return result;
}
private ReaderStrategy ConfigureUsage() {
  ReaderStrategy result = new ReaderStrategy("TVL", typeof (Usage));
  result.AddFieldExtractor(4, 8, "CustomerID");
  result.AddFieldExtractor(9, 22, "CustomerName");
  result.AddFieldExtractor(30, 30, "Cycle");
  result.AddFieldExtractor(31, 36, "ReadDate");
  return result;
}
```

Um diese Konfigurationen besser konfigurierbar zu machen, ohne immer neuen Bytecode

generieren zu müssen könnte man eine YAML[3]-Datei schreiben und diese als Input für die Strategiekonfiguration benutzen (Listing 2.3).

Listing 2.3: Instanzen der verschiedenen Strategien - YAML

```
mapping IFT dsl.ServiceCall
  4-18: RequestType
  19-23: CustomerID
  24-27 : CallTypeCode
  28-35 : DateOfCallString

mapping  TVL dsl.Usage
  4-8 : CustomerID
  9-22: CustomerName
  30-30: Cycle
  31-36: ReadDate
```

Ein Domänenexperte, der etwas vom Edifact-Parsen versteht, kann ohne fundierte Programmierkentnisse die YAML Datei interpretieren. Der Inhalt dieser YAML-Datei ist eine kleine Sprachinstanz. Die konkrete Syntax ist das YAML Format. Eine andere konkrete Syntax könnte das XML Format sein. Die abstrakte Syntax ist die Basisstruktur: “Mehrere Abbildungen von Zeilentypen auf Klassen, jeweils identifiziert durch drei Buchstaben und eine Postitionsbestimmung der Klassenvariablen innerhalb der Nachricht”. 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 2.4). 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 würde (Stragie Implementation, AddFieldExtractor, AddStrategy, . . .), dann wäre Code Listing 2.4 eine interne DSL und Ruby die Wirtssprache, also eine Untermenge von Ruby und eine konkrete- zu der abstrakten Syntax.

Listing 2.4: Instanzen der verschiedenen Strategien - 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
```

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

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 Sprachelemente mit meist natürlichen Begriffen 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 beschrieben werden. Andere Problembereiche sind ausgeblendet, 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 Charakteristik einer DSL ist vorzugsweise eine 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 zu messen. Vorzugsweise sollte die Syntax keine Redundanzen aufweisen, wie das z.B. bei XML der Fall ist, indem das offene und geschlossene Element den selben Tagnamen tragen. Es sollte ein Zeichen bzw. eine minimale Kombination aus mehreren Zeichen und deren Position in der DSL für eine Informationseinheit verwendet werden. Damit sind Zeichen gemeint, die auch eine angelehnte Bedeutung zu der natürlichen Sprache haben, z.B. Klammern, Semikola, Kommas, Punkte und Querstriche insbesondere auch Leerzeichen und Zeilenumbrüche.

Fowler ist der begründeten Ansicht, dass zu einer DSL immer ein semantisches Modell [FP11, p. 159] existieren sollte, das unabhängig von der eigentlichen DSL ist, aber direkt dazugehört. Es ist das Modell zu der DSL oder auch ihr Schema. Die Anwendung der DSL instanziiert das semantische Modell und ist damit eine gut lesbare Form der Modellinstanziierung. Das semantische Modell hat somit große Ähnlichkeit mit einem Domänen Modell [FRF02]. Vorteilhaft ist damit die klare “Trennung der Angelegenheiten” (separation of concerns) [HL95], 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 Modell das von einer Zustandsmaschine sein, wie in Abbildung 2.1 graphisch dargestellt.

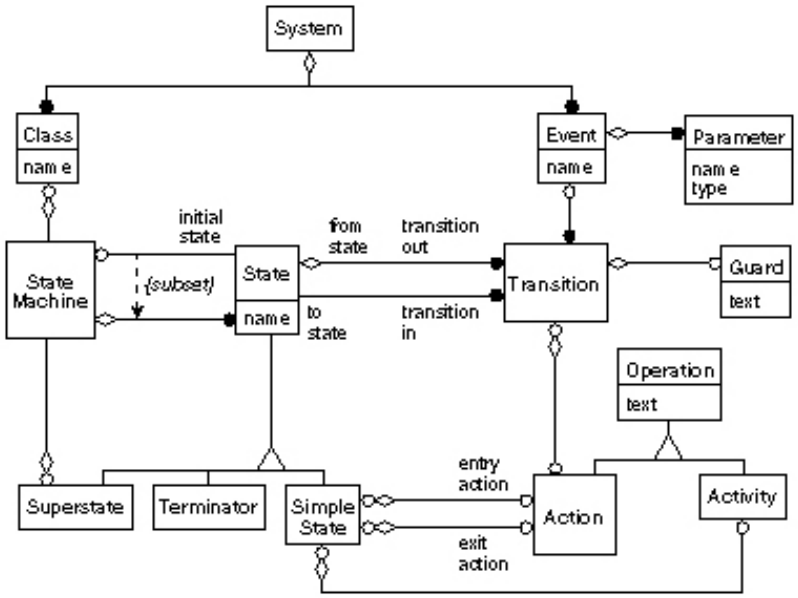

Abbildung 2.1: Semantisches Modell einer Zustandsmaschine - neilvandyke.org/smores/

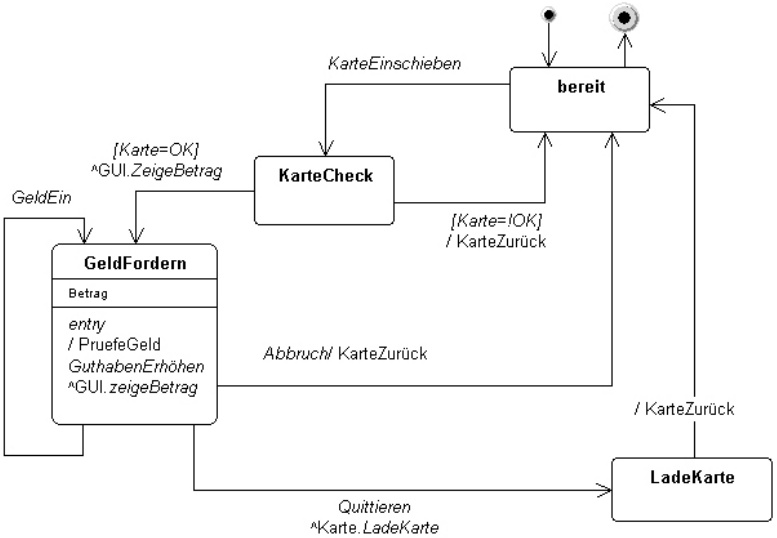

Abbildung 2.2: Zustandsdiagramm Geldautomat `https://www.fbi.h-da.de/uploads/RTEmagicC_f2da95d8df.gif.gif`

Z.B. ist das Metamodell von Zuständen eines Geldautomaten das Modell einer Zustandsmaschine (Abb. 2.1). Das Zustandsdiagramm des Geldautomaten ist damit das in Abbildung 2.2

dargestellte. Es beschreibt alle Zustände und Zustandsübergänge bzw. deren vorausgehende Ereignisse bzw. Aktionen. Eine DSL könnte dieses semantische Modell, das Metamodell der Zustandsmaschine, mit den Gegebenheiten des Geldautomaten instanziieren. (Code Listing 2.5).

Listing 2.5: DSL Ausschnitt für einen Geldautomat

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

2.1.1 Unterscheidungen

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

“Sowohl die konkrete Syntax als auch die Semantik können frei definiert werden. SQL oder reguläre Ausdrücke sind Vertreter von externen DSLs.” [unk12] Wenn eine DSL innerhalb

bzw. mit einer GPL definiert wurde, nennt er diese interne DSL. Solche eingebettete Spracherweiterungen sind mit den gegebenen Mitteln der “Wirtssprache”, genauer die Möglichkeit zur Metaprogrammierung (Kapitel 2.3), 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 von einer anderen Programmiersprache geparst und weiterverarbeitet. Eine eigenständige Sprachdefinition unterliegt keiner Einschränkung durch die Wirtssprache. Damit sind absurde Szenarien gemeint, bei denen z.B. Methodennamen hinter die Argumentenklammer geschrieben sind: “(1,1)addiere” oder sonstige denkbare syntaktische Abwandlungen.

2.1.2 Vergleich zu MDA / MDD

Model Driven Architecture (MDA) bzw. Model Driven Development (MDD) und Metaprogrammierung bzw. DSLs haben eine vergleichbare Problemstellung.

Mit der MDA richtet die OMG ihren Fokus nicht auf die Nutzung von DSM-Sprachen, sondern auf generisches UML in graphischer Form, ihre eigene Standard-Modellierungssprache.

Auch graphische Modelle müssen eine formale Sprache haben, damit sie weiter verarbeitet werden können. Eine DSL ist eine textuelle Repräsentation des semantischen Modells.

Die Komplexität und das Abstraktionsniveau sind abhängig von dem verwendeten semantischen Modell, nicht seiner Repräsentation.

Eine DSL hat den Vorteil, dass ihre Darstellung simpler ist. Man kann sie beispielsweise mit einem einfachen Text Editor bearbeiten oder mit trivialen Mitteln ein “diff” zweier Versionen erstellen (vgl. [CM07]. [Spi08], nach [Bie08]).

“Mit der MDA richtet die OMG ihren Fokus nicht auf die Nutzung von DSM-Sprachen, sondern auf generisches UML, ihre eigene Standard-Modellierungssprache. Sie versucht nicht, das möglicherweise existierende Domänen-Fachwissen einer Firma einzukapseln, sondern nimmt an, dass dieses Wissen nicht vorhanden oder irrelevant ist. ” [PT06]

2.2 Interne DSLs mit Groovy

Groovy wurde entworfen, 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.

In den nächsten Abschnitten handelt diese Arbeit von den wichtigsten Features dieser Programmiersprache, die sich für die Erstellung einer internen DSL eignen.

2.2.1 Syntaxeigentschaften

Bei "top-level" Ausdrücken wie z.B. `println("x")` ist es erlaubt, die Klammern wegzulassen: `println "x"`

Wenn der letzte Parameter einer Methode eine Closure ist, dann kann die Closure auch ausserhalb des Methodenaufrufs geschrieben werden: "`list.each(){ println it }`". Die Klammern können auch weggelassen werden: "`list.each { println it }`".

Bei Klassenmethoden können bei deren Aufruf die Klammern nicht weggelassen werden. Bei Zuweisungen ebenfalls nicht "`def x = methodenName y`".

2.3 Metaprogrammierung

Mittels Metaprogrammierung ist ein Programmierer in der Lage Quelltext zur Laufzeit zu ändern. Ein Meta-Objekt-Protokoll (MOP) stellt diese dafür benötigten Mittel bereit.

2.4 Die MOP Leistungsträger

Das MOP besteht aus den Haupt-Leistungsträgern: "Metaclass", "Kategorien" und "Expandos".

2.4.1 Expando

Expandos sind dynamische Repräsentationen von Groovy-Beans (Java-Beans mit einfacherer Syntax). Wenn eine Variable von einem Expando angefragt wird und diese nicht existiert, dann wird keine Exception geworfen sondern null zurückgegeben. Auch wenn eine Variable belegt wird und noch nicht existiert, dann wird diese im Expando erzeugt. Es ist auch möglich, eine Closure einer Klassenvariable zuzuweisen. Diese ist dann genau wie eine Methode aufrufbar. Die Notation von einem Expando ist "def x = new Expando()"

2.4.2 Kategorien

Wenn eine Methode, einer bereits existierenden Klasse, zur Laufzeit hinzugefügt werden soll, dann erfolgt das mittels "Kategorien". Um eine Methode x zu der vorhandenen Klasse java.lang.Number hinzuzufügen, ist das Erstellen einer neuen Klasse mit einer statischen Methode x erforderlich. Das erste Argument der Methode ist eine Referenz auf die Instanz. Angewendet wird dann eine Kategorieklasse durch die folgende Notation: `use(KlasseMitStatischemX){zahl.x}`

2.4.3 Metaclass

Jede Groovy Klasse implementiert die "groovy.lang.GroovyObject" Schnittstelle. Dadurch ist die Methode `getMetaClass` auf jedem Groovy-Objekt vorhanden. Die `MetaClass` Klasse wird dadurch referenziert.

Verwendeten Java-Klassen implementieren dieses Interface nicht. Die `MetaClass` wird den Klassen auch zugewiesen und in einer `MetaClass`-Registrierung (Registry Muster [FRF02]) verwaltet. In der `MetaClass` werden alle Metainformationen der eigentlichen Klasse verwaltet. Die `MetaClass` erlaubt es, nach dem Expando (Abschnitt 2.4.1) Prinzip, neue Klassenvariablen und Methoden einem Objekt hinzuzufügen. Javaklassen beschreiben das Verhalten zur Zeit der Übersetzung, Metaklassen das zur Ausführungszeit. Die Kardinaliät zwischen Objekt und Metaklasse ist immer 1:1. In Abbildung 2.3 sind die Beziehungen zw. Groovy Metaklassen, Klassen und Instanzen dargestellt.

2.4.4 Closures

Closures sind abgeschlossene Code-Abschnitte, die an sich einen eigenen Datentyp darstellen. Closures werden von geschweiften Klammern abgegrenzt. Innerhalb einer Closure sind Parameter bzw. Argumente vom Ausführungscode durch einen Pfeil getrennt ->. Parameter

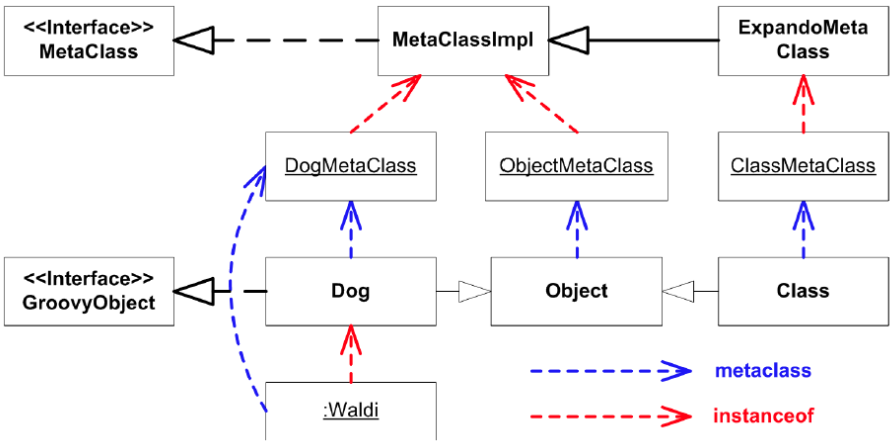

Abbildung 2.3: Beziehung von Metaklassen, Klassen und Instanzen [LB])

vor dem Pfeil sind durch Kommas getrennt. Closures besitzen das Keywort `this` aber auch noch `owner` und `delegate`

owner Das Attribut Owner ist eine Referenz auf die Elternklasse. Diese Eigenschaft kann nicht geändert werden. `Owner` ist in den meisten Fällen gleich der Referenz von `this`, außer wenn die umschließende Closure in einer andern Closure liegt.

delegate Diese Eigenschaft ist genau wie die `owner` Eigenschaft zu verstehen, jedoch mit dem bedeutenden Unterschied, dass dem `delegate` ein anderes Objekt zugeordnet werden kann. Das ist mit der `owner` Eigenschaft nicht möglich. So kann z.B. eine Closure einem fremden Objekt zugeordnet werden, jedoch ist das Elternobjekt immer noch das in dem die Closure definiert wurde. Mit der Neuzuweisung von `delegate` zum Fremdobjekt ist die Elternreferenz veränderbar.

2.4.5 ExpandoMetaClass

Die Metaclass Eigenschaft aus Abschnitt 2.4.3 ist ein Expando (Abschnitt 2.4.1). Diese ExpandoMetaClass kann auch dazu verwendet werden um z.B. Methoden zu überschreiben. Eine Statische Methode kann mittels folgendem Ausdruck hinzugefügt oder überschrieben werden: `Klasse.metaClass.static.x = { Closure-Inhalt}`. Methoden können auch überladen werden. Konstruktoren können hinzugefügt oder überschrieben werden.

3 Implementierung einer DSL

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

3.1 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.

3.2 Vorgehen

Nach der Zieldefinierung soll eine Bestandsaufnahme gemacht werden, um die Rahmenbedingungen 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.

3.3 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.

3.4 Bestandsaufnahme

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 Haupt-

komponenten dient: Zimmer-, Gäste-, Apartment-, Sonderleistungs- und Preisverwaltung. Derzeit wird ein neuartiges PMS erstellt, dass auf Groovy-und-Grails basiert.

3.5 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 der 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, diese beeinflussen Angebot und Nachfragen Bewertungen, die das Hotel auf Buchungsportalen bekommen hat, sind ebenfalls entscheidend für den Preis. Wenn z.B. mehrere schlechte Bewertungen abgegeben wurden und darauf hin nur noch wenige Gäste buchen, muss überlegt werden, ob das mit dem Preis zu regulieren ist. 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, Ereignisse und die Tage bis zu den Ereignissen zusammen kombiniert, beeinflussen den Preis weiter. Auch die aktuelle Liquidität des Unternehmens kann Einfluss darauf haben. Der Zimmerpreis ist auch sensibel gegenüber den Preisen der direkten Konkurrenz 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 und nachgefragten Hotelzimmern. Wenn die Auslastung steigt und die Nachfrage gleich bleibt, dann resultiert das in steigende Preise. Bei sozialen, kulturellen oder politischen Ereignissen weichen die Zimmerpreise erheblich von der "Rac-Rate" (Grundpreisrate) ab. Es stellt sich als schwierig 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.

3.6 Erstellung der DSL

Begonnen hat die Erstellung der DSL mit der Vorstellung, dass es im PMS einen Textbereich gibt, in dem die DSL eingefügt und editiert werden kann. Der Texteditor sollte mindestens ein "Rich-Text-Editor" sein, damit der Domänenexperte 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 explizit Groovy viele Möglichkeiten der Metaprogrammierung und der DSL-Erstellung hat, ist es naheliegend, die Preislogik 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 3.1).

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 3.1: 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 die Berechnungslogik zum größten Teil aus mathematischen Ausdrücken bzw. Formeln. Bemerkenswert ist, dass 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 dies auch ohne Schema erstellen konnte. Markus Völter beschreibt dazu in seinem Artikel[Vol11], dass die Schwelle zwischen Programmieren und Modellieren immer kleiner wird, was bei dieser Arbeit der Fall

ist. Begonnen wurde mit einem TestTreiber, der eine Textdatei einliest und diese interpretiert bzw. evaluiert. Cliff James hat 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, innerhalb einer "run" Methode liegt, ist der Aufruf immer derselbe. Die DSL wird letztendlich von der Groovy Shell Instanz evaluiert ([Koe07, S. 368]). Doch ohne den Kontext, in dem die DSL "ausgeführt" werden soll, ist die DSL nutzlos. Daher wird eine Instanz von Binding ([Koe07, S. 368]) dazu benutzt, um vordefinierte Variablen an das Script zu übergeben. Z.B. um der Variable `run` eine Closure zuzuweisen, die die loadDSL Methode im Runner aufruft wie im Code Listing 3.1).

Die Binding Instanz wird dann an die GroovyShell Instanz übergeben, um die Assoziationen zu gewährleisten.

Mit Groovy-Metaprogrammierung ist es möglich, den Kontext von einer in eine adere Instanz zu wechseln. `Delegate` wechselt also zu `this` und damit ist dann die DSL Bestandteil des DSLRunners. Das bedeutet, dass alles was in der DSL-Datei geschrieben wurde jetzt Methoden und Variablen der DSLRunner-Klasse referenzieren kann.

Listing 3.1: 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 Implementierung wurde übersprungen, da die DSL durch testgetriebene Verfahren [Bec02] direkt in der IDE[2] ausgeführt werden kann. Eine Schleifeniteration besteht aus einer Liste oder einem Abschnitt (Range), gefolgt von der `each` Methode und der auszuführenden Closure als Parameter dafür. Siehe Code Listing 3.2.

Listing 3.2: 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 3.3.

Listing 3.3: 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)`) ableitet.

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

Die erste Spalte der Zieltabelle ist somit darstellbar, aber die Lesbarkeit war erheblich durch Sonderzeichen beeinträchtigt. Ziel war nun die Lesbarkeit zu steigern, indem Sonderzeichen weitestgehend eliminiert werden und englische Begriffe durch deutsche ersetzen werden. Zuerst wurde das Wort `each` durch `alle` ersetzt. Hier wurde das ExpandoMetaObject

[2] www.jetbrains.com/idea/

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. Abbildung 3.4 zeigt den Codeabschnitt im DSLRunner.

Listing 3.4: .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.jahre` oder `1.jahr`. Dazu wurde das metaClass Expando-MetaObject von der Klasse Number dahingehend verändert, dass solche Konstruktionen möglich werden (Code Listing 3.5).

Listing 3.5: 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 3.6 dargestellten DSL-Code zu erzeugen.

Listing 3.6: Iterationsnotation auf Basis von Kategorien

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

Um diesen gut lesbaren Code an die deutsche Ausdrucksweise anzulehnen, ist die Verwendung von Command Expressions hilfreich, um eine Fluent Intercace [Fow05a] DSL zu erstellen. Das bedeutet, dass in der deutschen Sprache eigentlich folgender Ausdruck der natürlichste wäre: "alle Tage von heute bis in zwei Jahren einzeln auflisten und jeden Tag immer als Tag bezeichnen." Diese umständliche Ausdrucksweise ist zwar präzise, enthält aber 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 Autors 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[3] und in Code Listing 3.7 dargestellt.

Listing 3.7: Fluent Interface Implementierung

```
def von(Date start) {
     [bis: { end ->
             use(groovy.time.TimeCategory) {
                 (start..end)
             }
         }
     }]
}
```

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 wiederum 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, welches vom Typ `Date` ist. Letztendlich gibt diese Closure eine 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 ergibt sich nun folgende neue Notation für die DSL (Code Listing 3.8). Durch triviales Kopieren der `alle` zu `alleTage` ExpandoMetaObjekt Instanz wurde nach dem DRY Prinzip[4] die Closure wiederverwendet und `alleTage` steht für eine Menge, genauer für eine ObjectRange zur Verfügung.

Listing 3.8: Fluent Interface Anwendung

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

Wie in der Vorbetrachung angemerkt, beziehen sich die Preise nicht nur auf den Zeitraum, sondern auch auf die jeweilige Zimmerkategorie. Die Information aus der Hostdomäne (Abb. 3.1) bzw. aller Zimmerkategorien, muss nun in Verbindung mit der DSL gebracht werden.

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

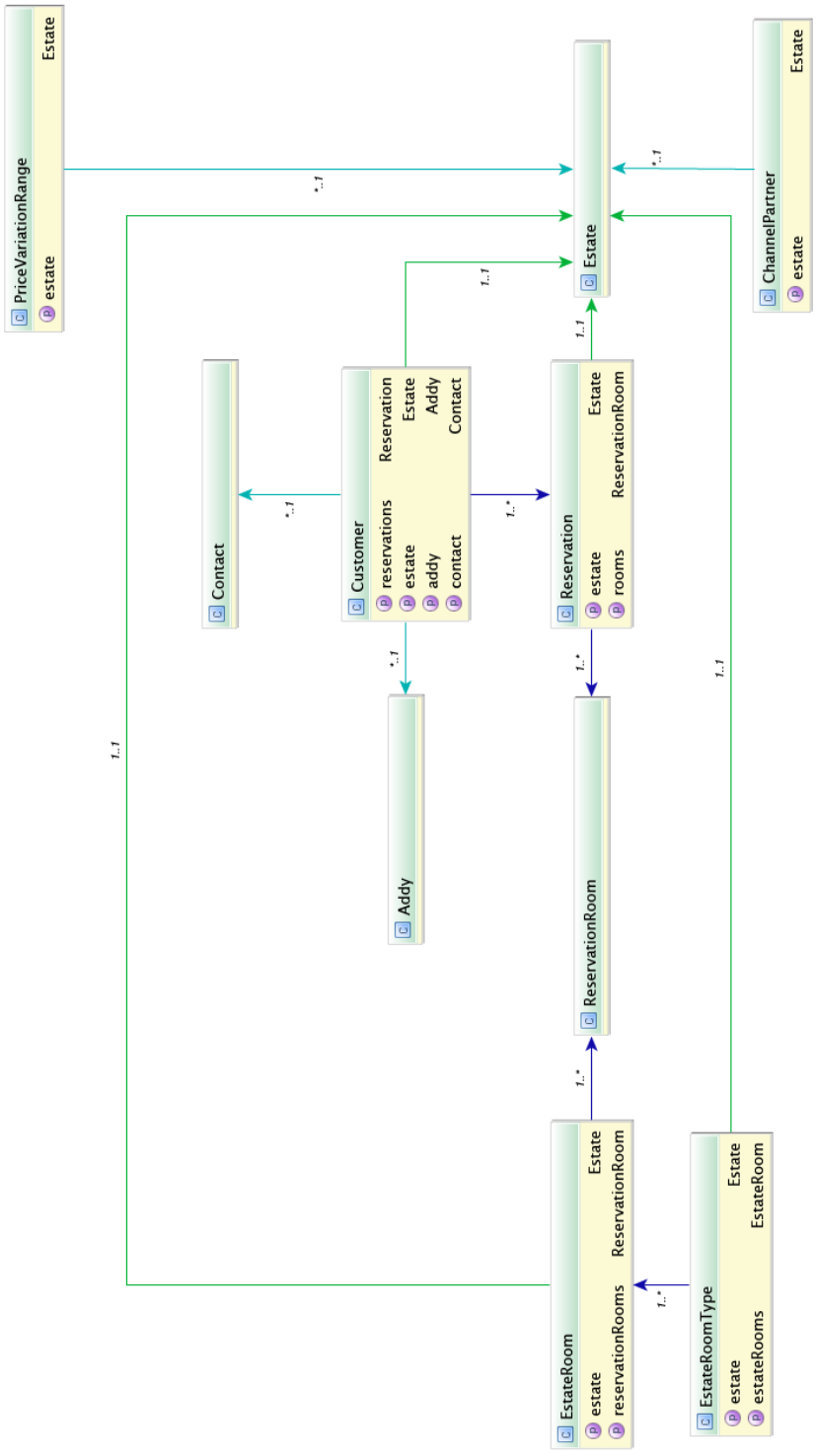

Abbildung 3.1: PMS - Domänen Modell Diagrammausschnitt

Bestenfalls sollte folgende Semantik nützlich sein: `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 Notation und der Domänenmodelle. Beispielsweise heisst das Hotel im Domänenmodell `Estate` und in der DSL nur `Hotel`. Wiederum heissen die Zimmerkategorien nicht so, sondern `EstateRoomType`. Es ist notwendig ein Mapping zu erstellen, das 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 sind, 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 Wrapper-Objekt (Adapter-Muster) [GHJV94] zu erstellen und das an die DSL zu binden (Code Listing 3.9).

Listing 3.9: EstateDSLWrapper.groovy

```
1 class EstateDSLWrapper {
2     Estate estate
3     EstateDSLWrapper(Estate estate) {
4         this.estate = estate
5     }
6     def Zimmertypen = estate.roomTypes;
7 }
```

Das Binding ist in Listing 3.10 dargestellt.

Listing 3.10: estateBinding.groovy

```
1  binding."Hotel" =
2  new EstateDSLWrapper(
3          new Estate(
4                  name: "Schoenhouse",
5                  estateRoomTypes: [
6                  new EstateRoomType(name: "typ1", grundpreis: 95),
7                  new EstateRoomType(name: "typ2", grundpreis: 105)
8                  ]
9          )
10 );
```

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 nun insgesamt möglich, **Schleife**n zu schachteln Code Listing 3.11.

Listing 3.11: multipleLoops.dsl

```
1 Hotel.Zimmertypen.alle { ZimmerTyp ->
2   von heute bis 3.months alleTage { Tag ->
3
4     TagesPreis = ZimmerTyp.Grundpreis
5
```

```
6       Ereignisse.alle { Ereignis ->
7           ...
8       }
9     }
10  }
```

In Code Listing 3.11 ist zusätzlich auch schon der erste **Ausdruck** in Zeile 4 dargestellt. Es handelt 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 und anschließend in der 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 3.12 zeigt, dass die Zielstellung dahingehend erreicht ist, dass eine Liste wie in Tabelle 3.1 durch die DSL berechnet wird.

Listing 3.12: 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, erfolgt 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[5] zu erstellen: "10 prozent Tagespreis". Die Kategorie dazu ist in Code Listing 3.13 dargestellt.

Listing 3.13: 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. Ein wiederkehrendes Ereignis ist z.B. ein Wochenende. Wenn sich also der Domänenexperte 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". Wie bei der Zeitabschnittbestimmung (von(x).bis(y)) wurde hier wieder die Methode eines Fluent Interfaces benutzt (Code Listing 3.14).

Listing 3.14: 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 einsetzbar. Alle zusätzlichen Erweiterungen für die Date-Klasse sind in Code Listing 3.15 dargestellt. Darunter befindet sich auch die Erweiterung "getWochenende bzw. wochenende".

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

Listing 3.15: 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 erlernen 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 3.16).

Listing 3.16: Vordefinierte Variablen

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

//-------------------------------------------------------------
//Benutzung dieser DSL Spracherweiterung
verfgbareZimmer = freieZimmer tag
// abhaengig von der Auslastung wird ein Teil von einem Drittel der
    Grundkosten aufaddiert.
tagesPreis += verfgbareZimmer / 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 Tagespreis-Manipulation durchzuführen. Code Listing 3.17 zeigt eine mögliche Form der DSL, in der ca.

90% der Konzepte beispielhaft dargestellt sind.

Listing 3.17: DSL für Preispolitik

```
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 nochmals auf folgende besondere Konstruktion hingewiesen: `tage von: heute, bis: ereignis.start` (Ziele 15). 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

dahinterliegende Struktur zu erkennen. Die Alternative hinter dem Ausdruck im Kommentar (Zeile 15) ist anders aufgebaut. `abstand` ist eine Methode, die eine Closure als Argument entgegennimmt, in diesem Fall einen Abschnitt (Range).

3.7 Das semantische Modell

Im Kapitel 3.6 wurde beschrieben wie die DSL erstellt wurde. Dabei wurden einige Begriffe mit bold markiert, um die wichtigsten Meta-Bestandteile zu verdeutlichen. In Abbildung 3.2 ist deren Zusammensetzung bzw. das semantische Modell der Preisberechnung dargestellt. Dieses ist sehr stark an das einer Skriptsprache angelehnt. In diesem Modell sind nur Berechnungsbestandteile durch Ausdrücke und Schleifen definiert. In einer Scriptsprache ist darüber hinaus noch mehr möglich.

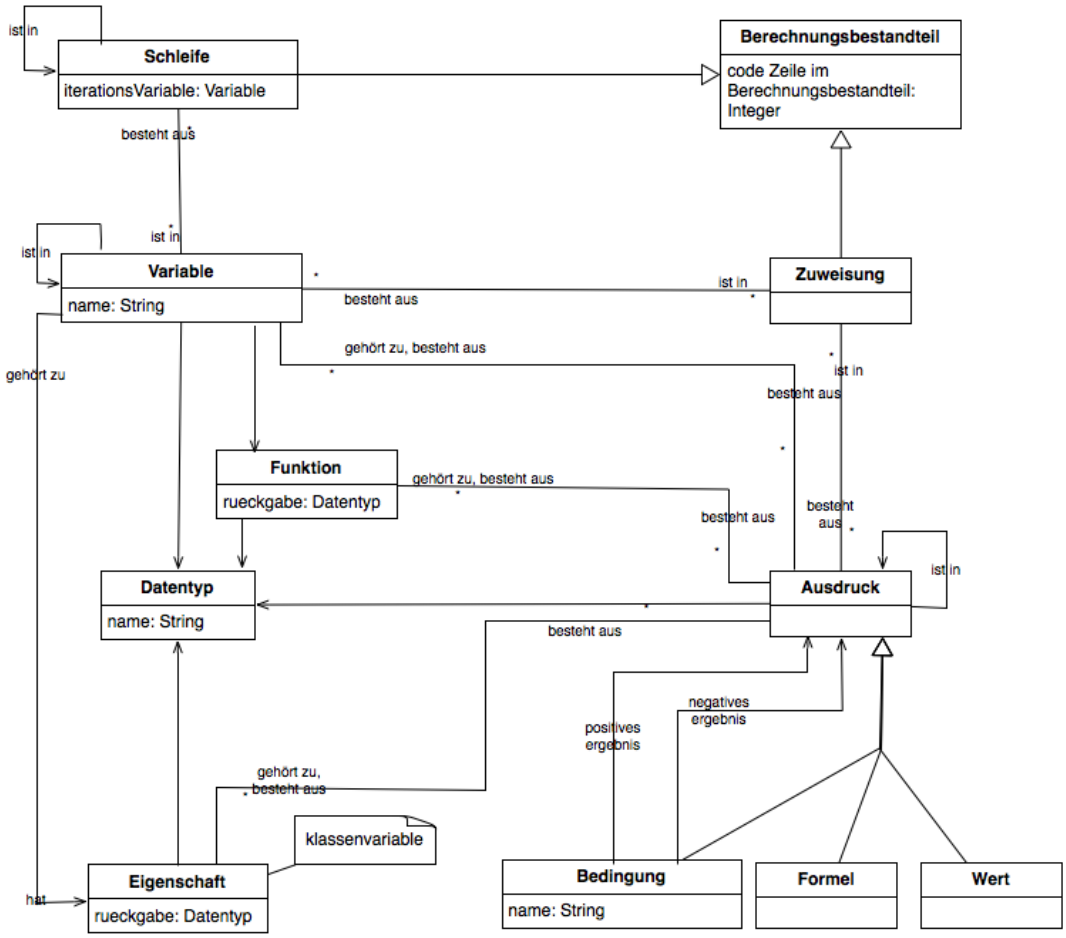

Abbildung 3.2: Preisberechnung semantisches Modell

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.

In Abbildung 3.3 ist ein Instanzdiagramm, der ersten DSL Zeilen von Code Listing 3.17, um das semantische Modell zu verifizieren. Durch die Codezeilen-Eigenschaft eines Berechnungsbestandteils kann die Abarbeitungssequenz dargestellt werden. Die drei Schleifen werden abgebildet und die einzelnen Ausdrücke innerhalb der Schleifen, wie z.B. `wenn TagInnerhalbEreignis dann 10 prozent tagesPreis` oder `Tagespreis = Zimmertyp.Grundpreis`. Die grau markierten Instanzen sollen farblich darstellen, dass es sich um ein und dasselbe handelt.

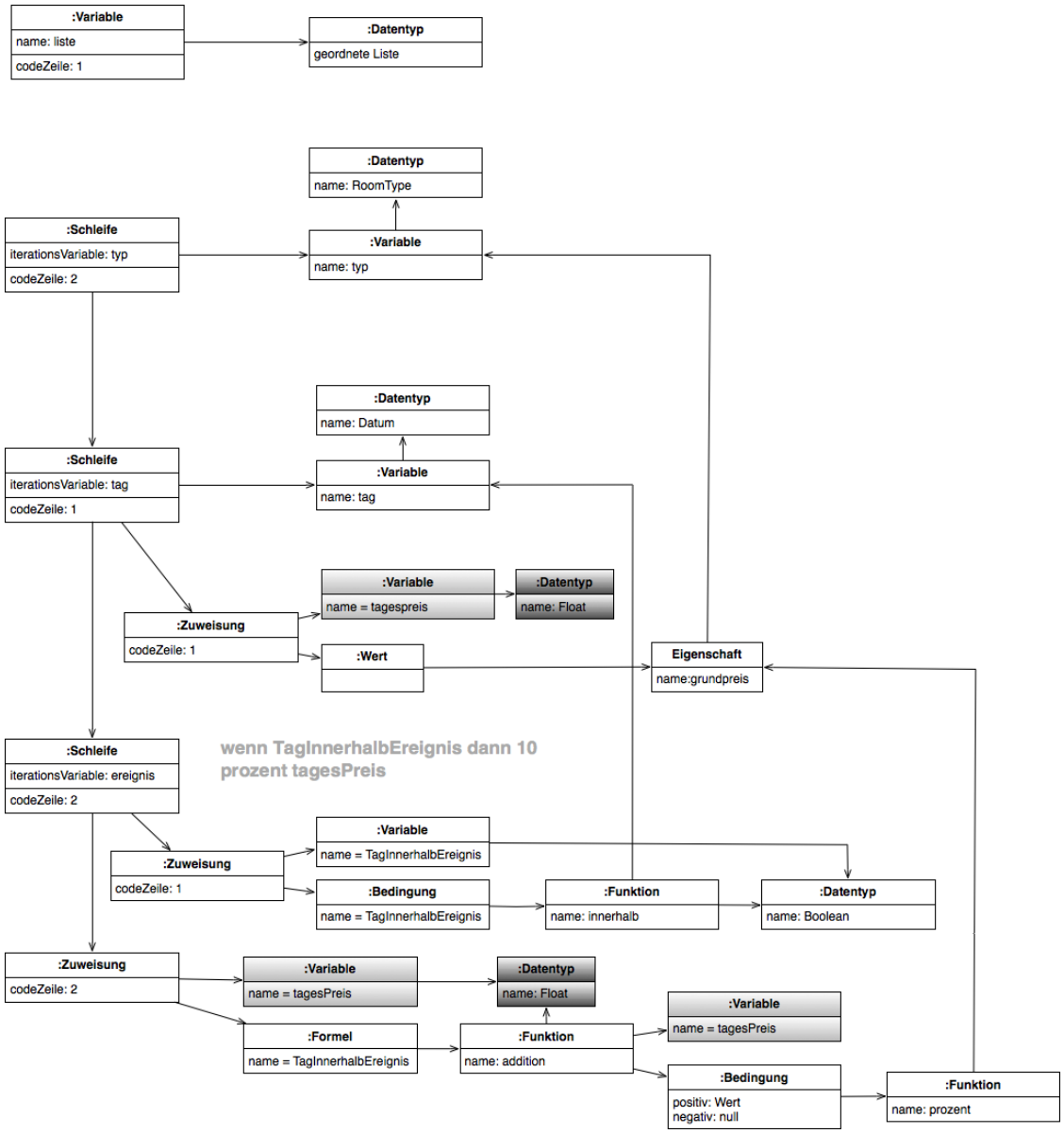

Abbildung 3.3: Instanzdiagramm der ersten Codezeilen

4 Einbezug der Domänenexperten

Die Vorstellung der DSL ist hier in einem Interviewmitschnitt[1] aufgezeichnet worden.

4.1 Aufgaben zur DSL für den Domänenexperten

Der Entwurf der DSL wurde dem Domänenexperten vorgestellt und es wurden Aufgaben bezüglich der in Code Listing 3.17 gezeigten DSL gemacht. Die Aufgaben sind:

- Erhöhen sie den Grundpreis aller Kategorien um 10 Euro.
- Der Zeitraum soll anstatt von heute von morgen beginnen.
- Die Ereignisse sollen bei der Berechnung keine Rolle mehr spielen.
- Die Gesamtzimmeranzahl ist um eins gestiegen.
- Der Lastminuterabatt soll in Lastminutezuschlag umgewandelt werden.
- Definieren sie neu: ExtraLastminuterabatt bei dem 5 Tage vorher das Zimmer 20 Euro weniger kosten soll.
- Die Liste soll mit einem Tabellenkopf versehen werden, der die Spaltenbezeichung beinhaltet: “Kategorie”, “Datum”, “Tagespreis”.

4.2 Lösungen zu den Aufgaben

Einer anderen Domänenexpertin, ohne betriebswirtschaftlichen Hintergrund und viel Erfahrung im Umgang mit Excel, wurde die vorhandene DSL (Code Listing 3.17) gezeigt und erklärt. Dabei sollten die Fragen beantwortet werden. Die Expertin war nur mit Hilfe dazu in der Lage, die Fragen zu beantworten. Die Befragung fand im Büro ohne jegliche Störungen statt. Der Expertin wurde die Preislogik-DSL vorgestellt und es wurden Erläuterungen zum Denkprozess bei der Erstellung der DSL kommuniziert (ca. 20 Minuten). 20 Minuten wurden für die Aufgabenlösung und 2 Minuten zur Auswertung verwendet. Durch die Abwesenheit

[1] 51 min - http://srvme.de/mda/mitschnitt.mp3

einer Legende wurde der Lösungsweg weiterhin erschwert. Die DSL wurde ohne Syntaxhighlighting auf dem Bildschirm repräsentiert. Die Benutzung der DSL musste sich die Expertin als ein Gedankenexperiment vorstellen. Eine Aufzeichnung[2] von dem Interview schildert die Aufgabenlösung.

[2] 42 min - http://srvme.de/mda/aufgabenloesung.mp3

5 Auswertung

5.1 Beurteilung durch Domänenexperte

Die Idee eine DSL einzuführen wurde überwiegend positiv aufgenommen, weil denkbar jede elektronische Unterstützung für die Domäne zuallererst als hilfreich bewertet wird. In dem vorhandenen PMS hat die Domänenexpertin keinen direkten Kontakt mit der Preislogik, sondern nur indirekt. Unsicherheit entstand, als die Domänenexpertin die DSL zur Preisberechnung zum ersten Mal gesehen hat, weil sie an eine rein textuelle Repräsentation nicht gewohnt ist. Die Wahrnehmung und die Erkennung der Bestandteile wurde erst bei näherem Betrachten klarer. Die Expertin war der Meinung, dass eine bessere visuelle Repräsentation der Unterscheidung helfen würde. Abschließend wurde die DSL als ein geeignetes Kommunikationsmedium gutgeheißen, da die Kommunikation mit dem Programmierer nicht immer eindeutig ist. Gegenüber einer grafischen Benutzeroberfläche wurde die DSL und deren Notation als "verwirrend" eingestuft. Jedoch wurde die DSL letztendlich als flexibler angesehen, da eigene Variablendefinitionen in vordefinierten, grafischen User Interfaces eher unüblich sind. Positiv sind die überwiegend richtigen Antworten der Fragen zu bewerten. Insgesamt wurden mit Hilfestellung 80% der Fragen beantwortet. Die Fragen waren aufgrund ihrer Nachfragen nur zu 80% eineindeutig.

5.2 Beurteilung durch Programmierer

Die Einsparung des GUI-Interfaces ist gegenüber der Entwicklungsgeschwindigkeit als positiv zu bewerten, wenn die DSL als Kommunikationsmedium mit dem Experten benutzt werden soll. Da die DSL über keinen speziellen Editor verfügt, der direkte Hilfe für die Erstellung der DSL bietet (Syntax Fehler, Autovervollständigung, Debugger, Versionskontrolle), ist ein fehlerfreies und effektives Editieren fast unmöglich und würde beim Domänenexperten schnell zu Frustration führen.

Die Möglichkeiten zur Erstellung einer DSL in Groovy sind ausreichend, aber oft in der Notation beschränkt. Die Implementierung einer eigenen DSL war für den Programmierer ein

neues Gebiet. Der Aufwand zum Erlernen der DSL-Leistungsträger in Groovy war gegenüber einer herkömmlichen statischen Implementierung hoch. Die Benutzung des MOP ist ein erheblicher Gewinn gegenüber den herkömmlichen Programmiermethoden, da nach einer Implementierung ausdrucksstärkerer Quelltext vorliegt, der die Lesbarkeit fördert. Eine zukünftige Verwendung von DSLs und dem MOP ist nach dieser Arbeit für den Autor fast nicht mehr wegzudenken, da der Nutzen den Aufwand beim Erstellen deutlich überwiegt. Lediglich beim Lernen der Konzepte gab es einen deutlichen Mehraufwand.

5.3 Zusammenfassung und Ausblick

Sprachorientierte Programmierung wurde in dieser Arbeit betrachtet. Als Mittel für die Erstellung einer DSL wurde die Groovy-MOP verwendet, weil die Wirtssprache des PMS auch Groovy ist. Die erstellte DSL diente als Bewertungsgrundlage dieses Paradigmas aus der Sicht des Programmierers und aus der des Domänenexperten. In zukünftigen Arbeiten ist die Verwendung von geeigneten Editoren für DSLs unvermeidbar, um das Paradigma Sprachorientierte Programmierung noch besser anwendbar zu machen. “Language Workbenches” bieten eine vielversprechende Lösung ([Vol11], [Fow05b] [Dmi04], [FP11]).

Literaturverzeichnis

[Bec02] Beck, Kent: *Test Driven Development. By Example (Addison-Wesley Signature)*. Addison-Wesley Longman, Amsterdam, 2002. – ISBN 0321146530

[Bie08] Bieker, F.: Metaprogrammierung und linguistische Abstraktion. (16. Dezember 2008). – http://bit.ly/wUHlxB

[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

[CM07] Cuadrado, J.S. ; Molina, J.G.: Building domain-specific languages for model-driven development. In: *IEEE software* (2007), S. 48–55

[Dmi04] Dmitriev, Sergey: Language Oriented Programming: The Next Programming Paradigm. (2004). http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf

[Fow05a] Fowler, M.: *Fluent interface*. 2005

[Fow05b] 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

[FRF02] Fowler, Martin ; Rice, David ; Foemmel, Matthew ; Professional, Addison-Wesley (Hrsg.): *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002 http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0321127420. – ISBN 0321127420

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. Addison-Wesley

Professional, 1994 http://www.worldcat.org/isbn/0201633612. – ISBN 0201633612

[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

[HL95] HRSCH, Walter L. ; LOPES, Cristina V.: Separation of Concerns. 1995. – Forschungsbericht

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

[LB] LARS BLUMBERG, Christoph Hartmann und Arvid H.: Groovy Meta Programming. . – http://www.acidum.de/wp-content/uploads/2008/10/groovy-meta-programming-paper.pdf

[PT06] PEKKA-TOLVANEN, J.: Domnenspezifische Modellierung fr vollst andige Code-Generierung,. In: *Javaspektrum* (2006), S. 9–12

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

[Spi08] SPINELLIS, D.: Rational metaprogramming. In: *Software, IEEE* 25 (2008), Nr. 1, S. 78–79

[SVEH07] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2. Heidelberg : dpunkt, 2007. – ISBN 978–3–89864–448–8

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

[Vol11] VOLTER, Markus: From Programming to Modeling - and Back Again. In: *IEEE Software* 28 (2011), S. 20–25. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MS.2011.139. – DOI http://doi.ieeecomputersociety.org/10.1109/MS.2011.139. – ISSN 0740–7459

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

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