Hochschule der Medien
Fachbereich Medieninformatik

Seminararbeit

Interne domänenspezifische Sprachen mit Groovy

erstellt von
Christian Schwörer (cs116)
und
Timo Müller (tm043)
im Seminar
generatives Computing

Februar 2009

Seminarleiter:
Timo Kehrer
FB 12, Elektrotechnik und Informatik
Fachgruppe Praktische Informatik
Universität Siegen

Inhaltsverzeichnis

1 Einleitung

„Man muss die Dinge so tief sehen, dass sie einfach werden.“

Konrad Adenauer

In der Informatik stellen die domänenspezifischen Sprachen *(Domain-Specific Languages, DSLs)* eine Möglichkeit dar, ausgewählte Problemfelder (Domänen) mit maßgeschneiderten Begrifflichkeiten zu beschreiben. Die an das Problemfeld angepasste DSL zeichnet sich durch ihre Einfachheit aus und bildet somit das Gegenteil zu den üblichen Programmiersprachen, die für verschiedenste Problemstellungen geeignet sind. Mit Hilfe einer solchen DSL können Personen, die besonders mit dem Problemfeld vertraut sind, auch ohne Programmierkenntnisse Lösungen für eine Domäne entwickeln.
Die Skriptsprache *Groovy* eignet sich aufgrund ihrer dynamischen Sprachkonstrukte hervorragend zur Definition von DSLs, die auf der Syntax von Groovy aufbauen (vgl. Barszczewski und Scharff, 2009, S. 37). Im Rahmen dieser Arbeit soll eine DSL für die Domäne eines Getränkeautomaten erstellt werden. Dabei sollen die Möglichkeiten von Groovy aufgezeigt und untersucht werden. Dafür werden zunächst die verschiedenen Typen von DSLs erläutert und die relevanten Konzepte von Groovy beschrieben. Im Anschluss erfolgt die Beschreibung der Domäne sowie die Umsetzung der DSL. Abschließend soll kurz abgeschätzt werden, wie sehr sich Groovy als Basis einer DSL eignet.

2 Domänenspezifische Sprachen

Bei einer domänenspezifischen Sprache *(Domain-Specific Language, DSL)* handelt es sich um eine formale Sprache, die für eine bestimmte Domäne, d.h. ein abgegrenztes Problemfeld, entworfen wird. Dabei wird beim Entwurf versucht, die Sprache möglichst spezifisch auf das Problemfeld auszurichten, so dass mit ihr alle relevanten Sachverhalte der Domäne ausgedrückt werden können (vgl. Wikipedia, 2009). Die Syntax einer DSL sollte dementsprechend so gewählt sein, dass ein Domänenspezialist ohne besondere Zusatzkenntnisse innerhalb seines Problemfeldes mit der DSL arbeiten kann.
Eine DSL stellt eine konkrete Notation zur Modellierung innerhalb einer bestimmten Domäne bereit. Sie stellt folglich das Gegenteil zu universell einsetzbaren Programmiersprachen *(General Purpose Languages, GPLs)*, wie z.B. Java oder C#, und universell einsetzbaren Modellierungssprachen, wie z.B. der *Unified Modeling Language (UML)*, dar.

Domänenspezifische Sprachen lassen sich auf verschiedenste Weise unterscheiden. Im Folgenden werden zwei der gebräuchlichsten Kategorisierungen vorgestellt (vgl. Stahl u. a., 2007, S. 98 ff.):

2.1 Interne vs. externe DSLs

Eine *interne DSL (internal DSL)* ist in eine andere Sprache, die so genannte *Hostsprache*, eingebettet. Man spricht daher auch von einer *eingebetteten DSL (embedded DSL)*. Die DSL stellt damit eine Untermenge der Hostsprache dar. Als Hostsprache bieten sich dynamisch typisierte Sprachen mit flexibler Syntax wie Ruby, Lisp oder Groovy an, da sie einen sehr tiefen Eingriff in ihre Sprachinterna erlauben. Allerdings sind interne DSLs, die mit den Sprachkonstrukten einer GPL definiert werden, in ihrer Syntax durch den von der Hostsprache vorgegebenen Rahmen beschränkt.
Im Gegensatz dazu wird eine *externe DSL (external DSL)* nicht in eine Hostsprache eingebettet, sondern von Grund auf neu definiert. Entsprechend können Syntax und Semantik frei festgelegt werden. Da es keine syntaktischen Beschränkungen durch eine Hostsprache gibt, ist eine externe DSL ausdrucksstärker. Allerdings ist die Erstellung aufwändig, da beispielsweise auch ein eigener Parser erstellt werden muss.

2.2 Grafische vs. textuelle DSLs

Eine *grafische DSL* besteht in der Regel aus einer Menge von Graphen mit Knoten und Kanten. Hier bietet sich beispielsweise die UML an, auf deren Basis eine eingebettete, grafische DSL entworfen werden kann. Eine grafische Notation ist besonders hilfreich, wenn Strukturen und Zusammenhänge nachvollziehbar dargestellt werden sollen. Ebenso lassen sich relativ einfach verschiedene Sichten und Abstraktionsgrade realisieren.
Dagegen besitzen *textuelle DSLs* eine textuell konkrete Syntax. Auf einer textuellen DSL basierende Modelle sind meist weniger übersichtlich als Modelle mit einer grafischen DSL. Die Übersichtlichkeit kann aber durch entsprechende Werkzeuge verbessert werden (z.B. durch Texteditoren mit Syntaxhervorhebung, Outline Views, usw.). Das Konfigurationsmanagement und somit auch das kollaborative Arbeiten mit Modellen wird bei textuellen DSLs besser unterstützt, da eine ausreichende Werkzeugunterstützung gegeben ist (SVN, CVS, Diff, Merge, usw.).

Bei der in Abschnitt 5 erstellten domänenspezifischen Sprache handelt es sich um eine *interne, textuelle* DSL. Die Hoststprache, in die die DSL eingebettet wird, ist die Skriptsprache Groovy, die im nächsten Kapitel vorgestellt wird.

3 Relevante Konzepte von Groovy

Groovy ist eine dynamisch typisierte Skriptsprache, die auf Java aufsetzt. Zum Einen bietet Groovy eine vereinfachte Syntax und zum Anderen werden neue Sprachkonzepte eingeführt. In Groovy geschriebene Programme werden in Java Bytecode übersetzt und direkt auf der Java Virtual Machine (JVM) ausgeführt. Dadurch ist es nicht nur möglich in Groovy Java-Klassen zu verwenden, sondern auch in Java auf Groovy-Klassen zuzugreifen.
Da die Syntax von Groovy sich stark an Java orientiert, soll im Rahmen dieser Arbeit keine vollständige Einführung gegeben werden. Baumann (2008) beschäftigt sich umfassend mit der Syntax und den Eigenschaften von Groovy. Stattdessen sollen die wichtigsten Eigenschaften gezeigt und die für die Erstellung der DSL relevanten Konstrukte und Konzepte vorgestellt werden.

3.1 Vollständige Objektorientierung

Wichtigster Unterschied zu Java ist die vollständige Objektorientierung. In Groovy werden ausschließlich Objekte verwendet. Dadurch entfällt die aus Java bekannte Unterscheidung zwischen dem elementaren Datentyp und einem Objekt der Wrapper-Klasse (`Integer`, `Float`, usw.). Ein beliebtes Beispiel ist die `times`-Methode, die direkt auf einer Zahl aufgerufen werden kann. Im Listing 1 wird der Anweisungsblock 28 mal aufgerufen und somit 28 mal „Hallo Welt“ auf die Konsole geschrieben.

```
28.times { println "Hallo Welt" }
```

Listing 1: Die Methode `times`

3.2 Native Verwendung von Maps

Das bekannte Collection-Framework (siehe Sun Microsystems, Inc., 2006) aus Java wird von Groovy vollständig unterstützt. Groovy vereinfacht aber mit einer neuen Notation den Umgang mit Collections erheblich. Die Inhalte einer Map müssen beispielsweise nicht mehr über die Methoden `get` bzw. `set` abgefragt und geändert werden. Dafür wird eine neue Schreibweise eingeführt, die vergleichbar mit dem Zugriff auf die Elemente eines Arrays ist. Listing 2 zeigt die Definition einer Map und die neue Art des Zugriffs.

```
1 def sampleMap = ['a': 4.5, 'b': 23.57]
2 println sampleMap['a']
3
4 sampleMap['b'] = 12.56
5 println sampleMap['b']
```

Listing 2: Maps in Groovy

3.3 Closures

Ein wichtiges Konzept in Groovy sind die Objekte erster (oder höherer) Ordnung. In Groovy wird ein solches Objekt Closure genannt. Eine Closure steht für eine Methode, die wie jedes andere Objekt als Parameter an andere Methoden übergeben oder in einer Variable gespeichert werden kann. Ein Beispiel für eine Closure wurde bereits in Listing 1 gezeigt. Der Anweisungsblock zwischen den geschweiften Klammern ist eine Closure, die als Parameter an die Methode `times` übergeben wird.

```
1 def addOperation = { a, b -> a + b }
2 println addOperation(5,6)
3 println addOperation.doCall(5,6)
```

Listing 3: Definition und Aufruf einer Closure

Listing 3 zeigt die Definition einer Closure, die die Eingabeparameter `a` und `b` erwartet und die Summe beider Werte zurück gibt. Die Eingabeparameter einer Closure werden als kommaseparierte Liste vor dem Pfeil-Symbol (->) festgelegt. Die Logik wird dann im Anschluss definiert und kann sich auch über mehrere Zeilen erstrecken. Zurückgegeben wird entweder ein Wert, der explizit mit dem Schlüsselwort `return` gekennzeichnet wird, oder der Wert der letzten Zeile.
In Zeile 1 wird die Closure in einer Variable gespeichert. Danach gibt es zwei Möglichkeiten die Methode auszuführen. In Zeile 2 wird `addOperation` direkt mit zwei Parameterwerten ausgeführt, während in Zeile 3 die Methode `doCall` mit den entsprechenden Parameterwerten aufgerufen wird. Jede Closure besitzt diese `doCall`-Methode. Intern wird der Aufruf aus Zeile 2 auch durch die `doCall`-Methode abgearbeitet.

3.4 Categories

Zur Laufzeit Methoden einer Klasse zu verändern oder neue Methoden hinzuzufügen ist in Java undenkbar. Groovy bietet diese Funktionalität in Form der so genannten Categories. Eine Category kapselt das Verhalten, das hinzugefügt werden soll, in Form von statischen Methoden (vgl. Baumann, 2008, S. 38). Angewandt wird die Category mit Hilfe des Schlüsselwortes `use`.

```
1 static class SampleCategory {
2     public static printHello(String value) {
3         return "Hello $value"
4     }
5 }
6
7 use(SampleCategory) {
8     println "Timo".printHello()
9 }
```

Listing 4: Definition und Nutzung von Categories

In Listing 4 wird zunächst eine neue Category `SampleCategory` in Form einer statischen Klasse erstellt. Diese Category erhält eine Klassenmethode `printHello`, die den übergebenen Parameter in einen neuen String einfügt und zurückgibt. In Zeile 7 wird die Category jetzt verwendet. Innerhalb des folgenden Anweisungsblocks wird zu jedem Objekt die Methode `printHello` hinzugefügt. Jetzt kann beispielsweise in Zeile 8 die Methode auf einen String aufgerufen werden und das Ergebnis („Hallo Timo“) ausgegeben werden.
Neben diesem einfachen Beispiel ist es mit den Categories auch möglich Methoden zu Überschreiben oder die Anwendung auf bestimmte Klassen zu beschränken.

3.5 Das Meta Object Protocol (MOP)

Mit Hilfe des Meta Object Protocols (MOPs) erlaubt es Groovy allen Objekten zur Laufzeit beim Aufruf einer Methode zu entscheiden, was genau passieren soll. Dieses Konzept hat seine Wurzeln in verschiedenen Lisp-Dialekten und wird beispielsweise in SmallTalk intensiv genutzt (vgl. Baumann, 2008, S. 20). Mit dem MOP können zum Beispiel Aufrufe abgefangen und verändert werden oder nicht implementierte Methoden anhand des Methodennamens an andere Objekte weitergeleitet werden.
Groovy realisiert das MOP dadurch, dass jedes Objekt einer Metaklasse zugeordnet ist. Jeder

Aufruf des Objekts wird zuerst transparent von der zugeordneten Metaklasse verarbeitet. Diese entscheidet dann, ob der Aufruf an das Objekt weitergeleitet wird, oder ob anstelle dessen einer andere Anweisung ausgeführt werden soll (vgl. Baumann, 2008, S. 284).
Der Programmierer kann das MOP erweitern und eigene Metaklassen für seine Klassen implementieren. Seit der Version 1.5 von Groovy gibt es mit dem `metaClass`-Attribut die Möglichkeit auf die Klasse `ExpandoMetaClass` zuzugreifen. Diese Klasse ermöglicht es, Methoden und Attribute einer Klasse durch einfache Zuweisungen hinzuzufügen oder zu überschreiben.

```
1 Integer.metaClass.addTwice = {
2     addedValue ->
3     return delegate + (addedValue * 2)
4 }
5
6 println 5.addTwice(6)
```

Listing 5: Definition einer neuen Methode mit Hilfe der MetaKlasse

In Listing 5 wird der Metaklasse von Integer mit Hilfe der `ExpandoMetaClass` eine neue Methode `addTwice` hinzugefügt. Diese Methode kann nun in Zeile 6 direkt auf einem Integerwert aufgerufen werden. Wenn die Methode `addTwice` bereits existiert, wird sie entsprechend von der neuen Logik überschrieben.
Neben der Definition neuer und dem Überschreiben alter Methoden und Attribute lassen sich die Aufrufe von Methoden und Attributen abfangen, die noch nicht existieren.

```
1 Integer.metaClass.propertyMissing = {
2     String propertyName ->
3     if (propertyName == 'testProperty') {
4         return 'Hallo Groovy'
5     }
6 }
7
8 println 5.testProperty
```

Listing 6: Überschreiben der `propertyMissing`-Methode

Erhält ein Objekt einen Aufruf für eine Methode bzw. ein Attribut, die jeweils noch nicht existieren, dann wird die Methode `methodMissing` bzw. `propertyMissing` der Metaklasse aufgerufen. Im Normalfall wird hier ein entsprechender Fehler erzeugt. Allerdings können

beide Methoden überschrieben und damit ein anderes Verhalten erreicht werden.
Listing 6 zeigt, wie ein Aufruf für ein Attribut abgefangen wird. In der `propertyMissing`-Methode wird zunächst der Name des Attributs geprüft, das abgefragt wurde. Wurde das Attribut `testProperty` abgefragt wird, der String „Hallo Groovy“ zurückgegeben. Auf diese Art wird erst zur Laufzeit ermittelt, was tatsächlich bei der Abfrage des Attributs passiert.

4 Die Domäne des Getränkeautomaten

Die interne DSL, die in Kapitel 5 erstellt wird, beschreibt die Arbeitsweise eines Getränkeautomaten. Diese Arbeitsweise stellt folglich die Domäne dar, die mit der DSL modelliert wird.
Aufgabe des Getränkeautomaten ist die Zubereitung von Getränken und deren Ausgabe an den Kunden. Dem Kunden stehen zwei Getränke - Tee und Kaffee - zu Auswahl. Beide Getränke werden zusätzlich mit Milch und/oder Zucker angeboten. Dazu verfügt der Automat über einen Vorrat an Zutaten, der aufgefüllt oder entleert werden kann. Welche Menge von welcher Zutat für ein Getränk benötigt wird, kann im Vorhinein definiert werden. So kann beispielsweise festgelegt werden, dass für einen Tee 200 ml Wasser und ein Teebeutel nötig sind. Ebenso können die Preise für die einzelnen Getränke individuell bestimmt werden.
Möchte ein Kunde Getränke am Automaten erstehen, wählt er zunächst ein Getränk und die gewünschte Menge. Eine solche Bestellung kann vom Kunden beliebig oft abgegeben werden. Der Automat verwaltet die Bestellliste und ermittelt den Gesamtpreis der Bestellung. Der Kunde kann sich neben diesem Gesamtpreis auch den Einzelpreis eines Getränks anzeigen lassen. Ist die Bestellung abgeschlossen, bezahlt der Kunde und die Getränke werden ausgegeben. Dabei werden die benötigten Zutaten aus dem Vorrat entfernt. Bezahlt werden kann mit drei Zahlungsmitteln: Kreditkarte, Geldkarte und Bargeld.

5 Erstellung einer internen DSL mit Groovy

In diesem Kapitel soll die interne DSL für den beschriebenen Getränkeautomaten erstellt werden. Dabei kommen die in Kapitel 3 beschriebenen Techniken zum Einsatz.
Bevor mit der tatsächlichen Umsetzung begonnen wird, soll die DSL zunächst anhand eines beispielhaften Ablaufs vorgestellt werden (siehe Listing 7). Alle Quelltexte, die in den nächsten Abschnitten erstellt werden, sind in ihrer endgültigen Form in Anhang A zu finden.

```
1  // Befüllen und Entleeren des Vorrats
2  fill 1.Kg, coffeebeans
3  fill 20.L, water
4  fill 2000.ml, milk
5  fill 50.g, sugar
6  remove 10.g, coffeebeans
7
8  // Definition der Preise und der Zutaten
9  define price.of(coffee, 1.56)
10 define ingredients.of(coffee, [(water): 200.ml, (coffeebeans): 20.g])
11
12 define price.of(coffee.with(sugar, milk), 1.70)
13 define ingredients.of(coffee.with(sugar, milk),
14             [(water): 200.ml, (coffeebeans): 20.g,
15             (milk): 5.ml, (sugar): 2.g])
16 // Abfrage der Preise
17 show price.of(coffee.with(sugar, milk))
18 show price.of(coffee)
19
20 // Ein beispielhafter Bestellvorgang
21 choose 1, coffee
22 choose 1, coffee
23 choose 3, coffee.with(sugar, milk)
24
25 // Anzeige des Gesamtbetrages der Bestellung
26 show total
27
28 // Bezahlvorgang
29 person = new Person(Name: 'Timo Müller', (creditcard): 30.00)
30 pay 'Timo Müller'.creditcard
```

Listing 7: Beispielhafter Ablauf unter Verwendung der DSL

Der gezeigte Ablauf nutzt bereits alle Konstrukte der DSL, die im Folgenden mit Groovy erstellt werden soll. Zu Beginn werden einige Zutaten in den Vorrat des Automaten gefüllt und im Anschluss 10 g Kaffeebohnen wieder entfernt (siehe Zeile 2-6). Die angegebene Menge der Zutaten enthält immer auch eine Maßeinheit. Für Flüssigkeiten (Wasser, Milch) sind hier die Maßeinheiten Liter (L) und Milliliter (ml), für Kaffeebohnen und Zucker Kilogramm (Kg) und Gramm (g) zulässig.
In Zeile 9 wird dann der Preis für einen Kaffee auf 1,56 festgelegt, um danach die Zutaten zu definieren, die für die Zubereitung verwendet werden. In diesem Fall benötigt man für einen Kaffee 200 ml Wasser und 20 g Kaffeebohnen. In Zeile 12-15 erfolgt das Gleiche für einen Kaffee mit Milch und Zucker.
Bevor der eigentliche Kaufvorgang beginnt, werden dem Kunden die Preise für die beiden Getränke angezeigt (siehe Zeile 17-18). Daraufhin wählt der Kunde zweimal einen Kaffee und einmal drei Kaffee mit Milch und Zucker. Der Gesamtbetrag der Bestellung wird danach mit `show total` angezeigt.
Die Person, die in Zeile 29 angelegt wird, bezahlt die Bestellung in Zeile 30 mit ihrer Kreditkarte. Die Getränke werden daraufhin ausgegeben und das Geld vom Guthaben der Kreditkarte abgebucht.

5.1 Die Klasse DrinkDispensingMachine

Der Getränkeautomat wird durch die Groovy-Klasse `DrinkDispensingMachine` realisiert. Hier wird der Status der Maschine verwaltet und die verschiedenen Operationen durchgeführt. Listing 8 zeigt lediglich die Klasse und ihre Attribute. Die Klassenmethoden werden in den folgenden Abschnitten schrittweise hinzugefügt.
Das Attribut `total` enthält während eines Bestellvorgangs den Gesamtbetrag der Bestellung. `choice` und `ingrediensNeeded` führen Buch darüber, welche Getränke in welcher Menge bestellt wurden und welche Zutaten insgesamt für die Bestellung aufgebracht werden müssen. Wie der Inhalt dieser Attribute verwaltet wird, wird in Abschnitt 5.6 aufgezeigt. Die nächsten beiden Attribute `ingredients` und `prices` definieren weitere Eigenschaften des Automaten. Während `ingredients` die Rezepte für die Zubereitung der einzelnen Getränke beinhaltet, enthält `prices` die Preisliste. Zu guter Letzt verwaltet der Automat die bevorrateten Zutaten im Attribut `stock`.
Alle Attribute mit Ausnahme von `total` sind durch den Mengentyp `Map` definiert, der die Zugriffe auf die Werte anhand des Getränketypen als Schlüssel erlaubt.

```
1  public class DrinkDispensingMachine{
2      // Gesamtbetrag der Bestellung
3      def total = 0
4      // Die gewählten Getränke und die Menge
5      def choice = [:]
6      // Die Zutaten, die für die Bestellung benötigt werden
7      def ingredientsNeeded = [:]
8      // Die Liste der Zutaten, die für ein Getränk benötigt werden
9      def ingredients = ['Coffee': [],
10             'CoffeeWithSugar': [],
11             'CoffeeWithMilk': [],
12             'CoffeeWithMilkAndSugar': [],
13             'Tea': [],
14             'TeaWithSugar': [],
15             'TeaWithMilk': [],
16             'TeaWithMilkAndSugar': []]
17     // Die Preise
18     def prices = ['Coffee': 0.00,
19             'CoffeeWithSugar': 0.00,
20             'CoffeeWithMilk': 0.00,
21             'CoffeeWithMilkAndSugar': 0.00,
22             'Tea': 0.00,
23             'TeaWithSugar': 0.00,
24             'TeaWithMilk': 0.00,
25             'TeaWithMilkAndSugar': 0.00]
26     // Die bevorratete Menge der Zutaten
27     def stock = ['Coffeebeans': 0.00,
28             'Teabags': 0.00,
29             'Milk': 0.00,
30             'Sugar': 0.00,
31             'Water': 0.00]
32 }
```

Listing 8: Die Klasse `DrinkDispensingMachine`

5.2 Das Ausführen der internen DSL

Um die interne DSL auszuführen, wird ein Groovy-Skript unter dem Dateinamen `Initialize.groovy` angelegt (siehe Listing 9). In diesem Skript wird ein Objekt der Klasse `DrinkDispensingMachine` erzeugt (Zeile 4). Anschließend werden in Zeile 7-11

für die einzelnen Zutaten Namenszuweisungen definiert. Das hat den Hintergrund, dass dadurch in der DSL mit diesen Variablen und nicht mit Strings gearbeitet werden kann. Entsprechend werden auch den Zahlungsarten (Zeile 14-16), den Getränken (Zeile 19-20) und dem Gesamtbetrag (Zeile 23) Variablen zugewiesen, um diese Namenszuweisungen anschließend in der DSL verwenden zu können.

```
1  import groovy.lang.GroovyShell
2
3  // Initialisieren der benötigten Klassen
4  machine = new DrinkDispensingMachine()
5
6  // Namenszuweisung der Zutaten
7  coffeebeans = 'Coffeebeans'
8  teabags = 'Teabags'
9  milk = 'Milk'
10 sugar = 'Sugar'
11 water = 'Water'
12
13 // Namenszuweisung der Zahlungsarten
14 creditcard = 'Creditcard'
15 cashcard = 'Cashcard'
16 purse = 'Purse'
17
18 // Namenszuweisung der Getränke
19 coffee = 'Coffee'
20 tea = 'Tea'
21
22 // Namenszuweisung der für den Gesamtbetrag
23 total = 'Total'
24
25 // ====================================================
26 // DSL laden
27 GroovyShell sh = new GroovyShell(binding)
28 sh.evaluate(new File("DSLExample.groovy"))
```

Listing 9: Das Groovy-Skript zur Ausführung der DSL *(Datei: Initialize.groovy)*

Die Datei, die eine konkrete Ausprägung der DSL beinhaltet und entsprechend geladen werden muss, hat in unserem Beispiel den Dateinamen `DSLExample.groovy`. Sie könnte den in Listing 7 gezeigten exemplarischen Aufbau besitzen. Die Einbindung in die

`Initialize.groovy` erfolgt in den Zeilen 27-28 über eine Instanz der Klasse `GroovyShell`. `GroovyShell` erlaubt es, Groovy-Skripte in ein Programm einzubetten und ausführen zu lassen. In diesem Fall wird dazu dem Konstruktor das `binding` übergeben, ein Objekt, das es erlaubt Informationen zwischen einem eingebetteten Skript und dem aufrufenden Programm auszutauschen. Anschließend wird mit der Methode `evaluate` das Skript in der Datei `DSLExample.groovy` ausgewertet. Damit die `GroovyShell` wie beschrieben genutzt werden kann, wird die entsprechende Klasse über die Import-Anweisung in Zeile 1 eingebunden.
Um die DSL auszuführen, wird die `Initialize.groovy` über die Kommandozeile oder eine Entwicklungsumgebung wie Eclipse gestartet (vgl. dazu ausführlich Rozanski, 2008, S. 38 ff. / S. 49 ff.). Allerdings müssen für eine korrekte Ausführung zunächst die weiteren benötigten Groovy-Artefakte aus den folgenden Abschnitten hinzugefügt werden.

5.3 Das Befüllen und Entleeren des Vorrats

Die Klasse `DrinkDispensingMachine` bietet bisher keine Methoden an, die das in Abschnitt 4 vorgestellte Verhalten realisieren. Als Erstes soll es möglich sein den Automaten mit den Zutaten, die in Listing 9 in den Zeilen 7-11 definiert sind, zu befüllen. In der Klasse `DrinkDispensingMachine` soll daher eine neue Methode angelegt werden, die als Eingabeparameter die Menge und den Zutatentyp erwartet. In der Methode wird die übergebene Menge dann zu der im Automaten vorrätigen Menge addiert. In welcher Menge eine Zutat vorhanden ist, wird, wie bereits dargestellt, in der Eigenschaft `stock` des Automaten gespeichert.

```
1 void fill(int amount, type) {
2     stock[type] += amount
3 }
```

Listing 10: Die `fill`-Methode der Klasse `DrinkDispensingMachine`

Der Vorrat des Getränkeautomaten kann jetzt durch einen Aufruf der Methode `fill` aufgefüllt werden. In der DSL soll es allerdings möglich sein die Methode ohne die Angabe der Instanzvariable auszuführen und `fill` als Schlüsselwort zu verwenden. Realisiert werden kann dies durch Methodenzeiger, mit denen in Groovy eine Methode in einer Variable abgespeichert werden kann.

```
1 // Methodenzuweisungen
2 fill = machine.&fill
```

Listing 11: Methodenzeiger für die Methode `fill` *(Datei: Initialize.groovy)*

In der Datei `initialize.groovy` wird bereits eine Instanz von `DrinkDispensingMachine` mit dem Namen `machine` erzeugt. Listing 11 zeigt, wie ein Zeiger auf die `fill`-Methode dieser Instanz in einer Variable gespeichert werden kann. Mit diesem Zeiger kann der Automat vereinfacht mit Zutaten befüllt werden.

```
fill 5, coffeebeans
```

Listing 12: Vereinfachter Zugriff auf die Methode `fill` *(Datei: DSLExample.groovy)*

Der beispielhafte Aufruf aus Listing 12 fügt fünf Kaffeebohnen zum Vorrat des Automaten hinzu. Die Verwendung der Variable `coffeebeans` verhindert zusätzlich, dass in der DSL mit Strings gearbeitet werden muss.
Genau gleich kann verfahren werden, um das Entleeren des Vorrats zu ermöglichen. Auch hier erhält die `DrinkDispensingMachine` erst eine neue Methode namens `remove`.

```
1 void remove(int amount, type) {
2     int left = stock[type]
3     if (left >= amount) {
4         stock[type] -= amount
5     } else {
6         println "Not enough $type left"
7     }
8 }
```

Listing 13: Die `remove`-Methode der Klasse `DrinkDispensingMachine`

Im Unterschied zur `fill`-Methode muss zusätzlich überprüft werden, ob die Zutat, die entnommen werden soll, überhaupt in ausreichender Menge vorrätig ist. Im Fall, dass genug vorhanden ist, wird die gewünschte Menge vom Vorrat entfernt. Andernfalls wird in einer Konsolenausgabe darauf hingewiesen, dass die vorrätige Menge nicht ausreicht, um die gewünschte Menge der Zutat zu entfernen.
Auch für diese Methode wird ein Methodenzeiger mit dem Namen `remove` registriert.

```
remove = machine.&remove
```

Listing 14: Methodenzeiger für die Methode `remove` *(Datei: Initialize.groovy)*

Anschließend kann die Methode in der DSL äquivalent zu `fill` aufgerufen werden.

```
remove 5, coffeebeans
```

Listing 15: Vereinfachter Zugriff auf die Methode `remove` *(Datei: DSLExample.groovy)*

Bisher wird beim Befüllen und Entleeren des Automaten noch keine Maßeinheit angegeben. Listing 7 auf Seite 10 zeigt wie der Aufruf letzten Endes erfolgen soll (siehe Zeile 6-11). Die Angabe der Maßeinheit erfolgt über einen Attributzugriff auf der `Integer`-Instanz. Beispielsweise wird mit `5.Kg` das Attribut `Kg` der `Integer`-Klasse abgefragt. Da dieses Attribut aber noch nicht existiert, wird zur Laufzeit eine `PropertyMissingException` geworfen. Um die `Integer`-Klasse um die geforderten Attribute `g`, `Kg`, `ml` und `L` zu erweitern (siehe Abschnitt 5), kann man die `ExpandoMetaClass` des MOP verwenden.

```
1  // Umrechnung der Einheiten für die Zutaten
2  Integer.metaClass.propertyMissing = {
3      String propertyName ->
4          def units = ['g':1, 'Kg':1000, 'ml':1, 'L':1000]
5          def unit = units[propertyName]
6
7          if (unit) {
8              return delegate * unit
9          } else {
10             println "Conversion from $propertyName to gramm not available"
11         }
12 }
```

Listing 16: Das Erweitern der Klasse `Integer` *(Datei: Initialize.groovy)*

In Listing 16 wird über das Attribut `metaClass` auf die `ExpandoMetaClass` der Klasse `Integer` zugegriffen. Im Anschluss kann das Attribut `propertyMissing` mit einer Closure hinterlegt werden. Diese Closure wird immer dann gerufen, wenn ein Attribut abgefragt wird, das Integer noch nicht bekannt ist. Die Closure besitzt einen Eingabeparameter `propertyName`, der den Namen des Attributs enthält, auf das ein Zugriff erfolgte. Anhand dieses Attributes kann entschieden werden, wie weiter verfahren werden soll.

Innerhalb der Closure wird eine Map definiert, die die Umrechnungsfaktoren der einzelnen Maßeinheiten enthält. Der Einfachheit halber werden alle Mengen immer in Gramm umgerechnet. Auch Flüssigkeiten werden in Gramm in der `DrinkDispensingMachine` gespeichert (1 g = 1 ml). Der Eingabeparameter wird im Anschluss als Schlüssel verwendet, um den entsprechenden Umrechnungsfaktor aus der Map abzufragen. Existiert kein Umrechnungsfaktor für die Einheit, enthält die Variable `unit` den Wert `null` und der `else`-Zweig in Zeile 10 wird ausgeführt. Andernfalls wird der Wert der `Integer`-Zahl, deren Attribut abgefragt wurde und der durch die Variable `delegate` innerhalb der Closure verfügbar ist, mit dem Umrechnungsfaktor multipliziert und zurückgegeben.
In der DSL kann nun der Automat unter Verwendung der definierten Maßeinheiten befüllt und entleert werden (siehe Listing 17).

```
1 // Befüllen und Entleeren des Vorrats
2 fill 1.Kg, coffeebeans
3 fill 20.L, water
4 fill 2000.ml, milk
5 fill 50.g, sugar
6 remove 10.g, coffeebeans
```

Listing 17: Nutzung der definierten Maßeinheiten *(Datei: DSLExample.groovy)*

5.4 Die Definition der Preise und der Zutaten

Damit Zutatenlisten und Preise für die einzelnen Getränke definiert werden können, wie dies im beispielhaften Ablauf der DSL (Listing 7) in den Zeilen 9-15 gezeigt wird, muss zunächst eine neue Klasse `ReturnerClass` erstellt werden. Wie in Listing 18 zu sehen ist, beinhaltet sie eine Methode `of` in zweifacher Ausprägung. Die erste Form erwartet einen String-Parameter `returnValue` und wird später bei der Anzeige des Getränkepreises Verwendung finden. Für die Definition von Getränkepreisen und -zutaten ist zunächst die zweite Form relevant. Sie erwartet einen Parameter `key`, der den Getränkenamen beinhaltet, und einen zweiten Parameter `data`, der den Preis oder die Zustatenliste enthält. Der Rückgabewert der Methode ist eine aus `key` und `data` zusammengesetzte ArrayList.

```groovy
public class ReturnerClass{
    // of-Methode zur Anzeige des Getränkepreises
    def of(String returnValue) {
        return returnValue
    }

    // of-Methode zur Definition von Preis und Zutaten
    def of(String key, def data) {
        return [key, data]
    }
}
```

Listing 18: Die Klasse `ReturnerClass` *(Datei: ReturnerClass.groovy)*

Das Skript in der Datei `Initialize.groovy` muss nun, wie in Listing 19 zu sehen ist, erweitert werden. Das heißt, zusätzlich zu der Instanziierung der `DrinkDispensingMachine`-Klasse werden zwei Objekte `price` und `ingredients` der Klasse `ReturnerClass` erzeugt.

```groovy
price = new ReturnerClass()
ingredients = price
```

Listing 19: Instanziierung der `ReturnerClass`-Klasse *(Datei: Initialize.groovy)*

Als Nächstes wird der Klasse `DrinkDispensingMachine` eine neue Methode `define` hinzugefügt. Ihren Aufbau zeigt Listing 20.

```groovy
void define(ArrayList values) {
    switch(values[1]) {
        // Wert ist vom Typ 'BigDecimal' -> Preis
        case { it instanceof BigDecimal }:
            prices?.putAt(values[0], values[1])
            break
        // Wert ist vom Typ 'Map' -> Zutaten
        case { it instanceof Map }:
            ingredients?.putAt(values[0], values[1])
    }
}
```

Listing 20: Die Methode `define` *(Datei: DrinkDispensingMachine.groovy)*

Die Methode erhält als Übergabeparameter die ArrayList `values`. Dies ist die ArrayList, die von der Methode `of` aus Listing 18 zurückgegeben wird. Position `[0]` von `values` enthält den Getränkenamen (z.B. *Coffee*), Position `[1]`beinhaltet entweder den zugeordneten Preis oder die die Zutatenliste.
Anhand einer Typüberprüfung, die in Form einer switch-Anweisung realisiert ist (Zeile 2-10), kann unterschieden werden, ob es sich bei der Zuordnung um den Preis oder die Zutatenliste handelt. Ist an der Stelle `[1]` der ArrayList `values` eine Instanz der Groovy-Klasse `BigDecimal`, soll der Preis festgelegt werden. Handelt es sich dagegen um eine Instanz der Groovy-Klasse `Map`, soll die Liste der Zutaten definiert werden. Bei der Preisdefinition wird bei der Map `prices` (die in der Klasse `DrinkDispensingMachine` in Listing 8 deklariert wurde) an der Stelle, deren Schlüssel dem Getränkenamen enstpricht, ein neuer Wert für den Getränkepreis gesetzt. Bei der Festlegung der Zutaten wird entsprechend in der Map `ingredients` die Zutatenliste gesetzt. Mit der Verwendung des Fragezeichens beim Zugriff (z.B. `prices?.putAt(...)`) wird sichergestellt, dass kein Laufzeitfehler auftritt, wenn ein Element mit entsprechendem Schlüssel in der Map noch nicht existiert. In diesem Fall wird der Map ein neues Element hinzugefügt.
Abschließend wird noch, wie zuvor bereits bei den Methoden `fill` und `remove`, in der `Initialize.groovy` ein Methodenzeiger mit dem Namen `define` registriert (siehe Listing 21).

```
1 define = machine.&define
```

Listing 21: Methodenzeiger für die Methode `define` *(Datei: Initialize.groovy)*

Somit kann nun ein exemplarischer Aufruf zur Definition des Preises eines Kaffees sowie der zugehörigen Zutatenliste den in Listing 22 gezeigten Aufbau haben. In diesem Beispiel wird festgelegt, dass ein Kaffee 1,56 kostet und sich aus den Zutaten 200 ml Wasser und 20 g Kaffeebohnen zusammensetzt.

```
1 define price.of(coffee, 1.56)
2 define ingredients.of(coffee, [(water): 200.ml, (coffeebeans): 20.g])
```

Listing 22: Vereinfachter Zugriff auf die Methode `define` *(Datei: DSLExample.groovy)*

Ergo ist es nunmehr möglich, für die Getränke „Kaffee“ und „Tee“ die Preise und Zutaten festzulegen.

Allerdings gibt es, wie in der Domänenbeschreibung in Kapitel 4 vorgegeben, die beiden Getränke auch mit Milch und/oder Zucker. Die zugehörige exemplarische Definition von Preis und Zutaten mit der internen DSL zeigt Listing 23. Dort wird bestimmt, dass das Getränk „Kaffee mit Zucker und Milch“ 1,70 kostet und sich aus den Zutaten Wasser (200 Milliliter), Kaffeebohnen (20 Gramm), Milch (5 Milliliter) und Zucker (2 Gramm) zusammensetzt.

```
1 define price.of(coffee.with(sugar, milk), 1.70)
2 define ingredients.of(coffee.with(sugar, milk),
3                       [(water): 200.ml, (coffeebeans): 20.g,
4                       (milk): 5.ml, (sugar): 2.g])
```

Listing 23: Definition von Preis und Zutaten für das Getränk „Kaffee mit Milch und Zucker“

Um die Methode `with` auf ein Getränk anwenden zu können, muss zunächst mit Groovy eine *Category* erstellt werden (siehe Abschnitt 3.4). Die Category besitzt den in Listing 24 gezeigten Aufbau und befindet sich in der Datei `UniqueNameUtil.groovy`.

```
1  public class UniqueNameUtil{
2      // Grundgetränk (Tee oder Kaffee)
3      // mit einer weiteren Zutat (Milch oder Zucker)
4      def static with(Object self, String firstIngredient) {
5          return "${self}With${firstIngredient}"
6      }
7
8      // Grundgetränk (Tee oder Kaffee)
9      // mit zwei weiteren Zutaten (Milch und Zucker)
10     def static with(Object self, String firstIngredient,
11                     String secondIngredient) {
12         if (firstIngredient == 'Sugar') {
13             return "${self}With${secondIngredient}And${firstIngredient}"
14         } else {
15             return "${self}With${firstIngredient}And${secondIngredient}"
16         }
17     }
18 }
```

Listing 24: Category `UniqueNameUtil` *(Datei: UniqueNameUtil.groovy)*

In der Category `UniqueNameUtil` befinden sich zwei statische Methoden `with`. Die erste erwartet als Parameter neben dem Objekt `self`, das die Instanz beinhaltet, auf die die `with`-Methode angewendet wird, auch eine Zeichenkette, die die zusätzliche Zutat enthält (also „Sugar“ oder „Milk“). Rückgabewert der Methode ist eine zusammengesetzte Zeichenkette aus dem Grundgetränk, dessen Name in `self` gespeichert ist, dem Verküpfungswort `With` und der weiteren Zutat. Beim Aufruf `coffee.with(milk)` würde folglich die Zeichenkette `CoffeeWithMilk` zurückgegeben werden.
Die zweite `with`-Methode erwartet als zusätzlichen Parameter eine weitere Zeichenkette, welche die zweite Zutat enthält. Auch hier wird ein zusammengesetzter String zurückgegeben, so dass zum Beispiel der Aufruf `coffee.with(milk, sugar)` die Zeichenkette `CoffeeWithMilkAndSugar` retournieren würde. Dieser Rückgabewert wird dann der Methode `of` der `ReturnerClass`-Instanz übergeben (siehe Listing 18), in der dann wie gewohnt die weitere Abarbeitung fortgesetzt wird.
Um eine Category in einem Groovy-Skript verwenden zu können, muss sie über die vordefinierte Methode `use` aktiviert werden. Damit die zuvor erstellte Category `UniqueNameUtil` in der internen DSL angewendet werden kann, bietet es sich an, die Aktivierung in der `Initialize.groovy` vorzunehmen. Listing 25 zeigt die entsprechende Erweiterung, mit der die Category im kompletten über die `GroovyShell`-Klasse eingebetteten Skript zur Verfügung steht

```
1 // UniqueNameUtil-Category aktivieren und DSL laden
2 use(UniqueNameUtil) {
3     GroovyShell sh = new GroovyShell(binding)
4     sh.evaluate(new File("'DSLExample.groovy"'))
5 }
```

Listing 25: Aktivierung der Category `UniqueNameUtil` *(Datei: Initialize.groovy)*

5.5 Die Abfrage der Getränkepreise und des Gesamtbetrages

Um eine Abfrage der Einzelpreise von Getränken sowie dem Gesamtbetrag einer Bestellung zu ermöglichen, wird als Erstes der Klasse `DrinkDispensingMachine` eine neue Methode `show` hinzugefügt, die in Listing 26 abgebildet ist.

```
1  void show(String type) {
2          String amount = 'No price available for beverage!'
3
4          switch(type) {
5              case 'Total':
6                  // Abfrage des Gesamtbetrags
7                  amount = "Total: $total"
8                  break
9
10
11
12             default:
13                 // Abfrage eines Getränkepreises
14                 if (prices.get(type) != null) {
15                     amount = "Price of $type: ${prices.get(type)}"
16                 }
17         }
18         println amount
19     }
```

Listing 26: Die Methode `show` *(Datei: DrinkDispensingMachine.groovy)*

Die Methode erwartet als Parameter die Zeichenkette `type`. Sie beinhaltet entweder den String „Total“, wenn der Gesamtbetrag angezeigt werden soll, oder den Namen des Getränks, dessen Einzelpreis abgefragt wird. Die Unterscheidung der beiden Fälle erfolgt mit einer switch-Anweisung (Zeile 4-16). Im Falle des Gesamtbetrags wird der Wert der Klassenvariablen `total` ermittelt. Zur Ermittlung eines einzelnen Getränkepreises wird mit dem Getränkenamen als Schlüssel die Map `prices` ausgelesen. Rückgabewert der Methode ist der so ermittelte Preis bzw., wenn der Name eines nicht verfügbaren Getränks übergeben wurde, die Fehlermeldung `No price available for beverage!`.
Abschließend wird auch für die Methode `show` in der Datei `Initialize.groovy` ein Methodenzeiger mit dem Namen `show` registriert (siehe Listing 27).

```
1  show = machine.&show
```

Listing 27: Methodenzeiger für die Methode `show` *(Datei: Initialize.groovy)*

In Listing 28 ist beispielhaft zu sehen, wie in der DSL die Einzelpreise für die Getränke „Kaffee“ und „Kaffee mit Zucker und Milch“ (Zeile 2-3) sowie der Gesamtbetrag der

Bestellung (Zeile 5) abgefragt werden können. Die Verwendung der Methoden `of` und `with` zur Angabe eines Getränks erfolgt analog zum Vorgehen bei der Definition von Preisen und Zutaten (siehe Abschnitt 5.4).

```
1 // Abfrage der Einzelpreise
2 show price.of(coffee.with(sugar, milk))
3 show price.of(coffee)
4 // Abfrage des Gesamtbetrages der Bestellung
5 show total
```

Listing 28: Vereinfachter Zugriff auf die Methode `show` *(Datei: DSLExample.groovy)*

5.6 Der Bestellvorgang

Damit der Kunde eine Bestellung durchführen kann, muss es ihm möglich sein, Getränke zu wählen. Dazu wird zunächst die Klasse `DrinkDispensingMachine` um die Methode `choose` ergänzt (siehe Listing 29).

```
1  void choose(amount, type) {
2          def ingr = ingredients[type]
3
4          try {
5              ingr.each {
6                  key, value ->
7                  def amount_choice = ingredientsNeeded.get(key, 0)
8                  if (stock[key] < ((value * amount) + amount_choice)) {
9                      throw new Exception("Not enough $key")
10                 }
11             }
12
13             ingr.each {
14                 key, value ->
15                 if (ingredientsNeeded.get(key)) {
16                     ingredientsNeeded[key] += (value * amount)
17                 } else {
18                     ingredientsNeeded[key] = (value * amount)
19                 }
20             }
21
```

```
22              if (choice.get(type)) {
23                  choice[type] += amount
24              } else {
25                  choice[type] = amount
26              }
27
28              total += prices[type] * amount
29          } catch (Exception e) {
30              println e.message
31          }
32      }
```

Listing 29: Die Methode `choose` *(Datei: DrinkDispensingMachine.groovy)*

Die Methode `choose` erhält als Parameter die gewünschte Menge (`amount`) und den Namen des gewünschten Getränks (`type`). Abhängig vom gewünschten Getränk werden in Zeile 2 die benötigten Zutaten ermittelt und in `ingr` zwischengespeichert. Diese Zutatenliste wird durchlaufen und so festgestellt, ob für die Bestellung noch ausreichend Zutaten bevorratet sind (Zeile 5-10). Im Anschluss werden die benötigten Zutaten zum Gesamtbedarf der Bestellung addiert (Zeile 13-20). In Zeile 22-26 wird das gewünschte Getränk der Auswahl `choice` hinzugefügt. Abschließend wird der Gesamtbetrag der Bestellung `total` entsprechend erhöht.
Wie bereits zuvor für die anderen verwendeten Methoden wird auch für `choose` ein Methodenzeiger mit dem Namen `choose` registriert, so dass in der DSL ein vereinfachter Aufruf möglich wird (siehe Listing 30).

```
1 show = machine.&show
```

Listing 30: Methodenzeiger für die Methode `choose` *(Datei: Initialize.groovy)*

Listing 31 zeigt, wie in der DSL eine Bestellung realisiert werden kann. In dem Beispiel werden zweimal ein Kaffee sowie einmal vier Kaffee mit Zucker und Milch bestellt.

```
1 choose 1, coffee
2 choose 1, coffee
3 choose 4, coffee.with(sugar, milk)
```

Listing 31: Vereinfachter Zugriff auf die Methode `choose` *(Datei: DSLExample.groovy)*

5.7 Der Bezahlvorgang

Die Bestellung, die im letzten Abschnitt aufgegeben wurde, muss selbstverständlich auch bezahlt werden. Bevor dies aber technisch realisiert werden kann, müssen zunächst zwei Fragen geklärt werden. Wer bezahlt die Bestellung überhaupt und womit bezahlt er?
In der DSL des Getränkeautomaten sollen Personen die Bestellung wahlweise mit ihrer Kreditkarte, Geldkarte oder auch bar bezahlen können. Für die Zahlungsmittel wird eine eigene Klasse `MoneyContainer` erstellt, die die Funktionalitäten der Zahlungsart kapseln könnte. In diesem einfachen Beispiel besitzt jedes Zahlungsmittel jedoch nur ein Guthaben (`value`), von dem letzten Endes der Gesamtpreis der Bestellung abgezogen wird (siehe Listing 32).

```
1 public class MoneyContainer{
2     def value
3 }
```

Listing 32: Die Klasse `MoneyContainer`

Eine Person, die später eine Bestellung bezahlt, besitzt einen Namen und drei Zahlungsmittel, jeweils in Form eines MoneyContainers (Listing 33). Der Konstruktor erwartet zum Einen den Namen der Person, zum Anderen das Guthaben der einzelnen Zahlungsmittel. Ist kein Guthaben angegeben, so wird es entsprechend auf Null gesetzt.

```
1  public class Person{
2      String name
3      MoneyContainer creditcard = new MoneyContainer()
4      MoneyContainer cashcard = new MoneyContainer()
5      MoneyContainer purse = new MoneyContainer()
6
7      Person(Map vars) {
8          name = vars.get('Name', 'Noname')
9          creditcard.value = vars.get('Creditcard', 0)
10         cashcard.value = vars.get('Cashcard', 0)
11         purse.value = vars.get('Purse', 0)
12     }
13 }
```

Listing 33: Die Klasse `Person`

Neue Personen können in der DSL damit zum Beispiel wie in Listing 34 gezeigt angelegt werden. Leider müssen die Schlüssel für die Zahlungsmittel in Klammern gesetzt werden, um die definierten Variablen zu verwenden. Andernfalls würde der Wert vor dem Doppelpunkt direkt als String interpretiert und somit die Werte im Konstruktor nicht korrekt aus der Map ausgelesen.

```
1 person = new Person(Name: 'Timo Müller', (creditcard): 30.00)
2 person = new Person(Name: 'Christian Schwörer', (creditcard): 30.00,
3                             (cashcard): 45.00, (purse): 120.00)
```

Listing 34: Anlegen einer neuen Instanz von `Person` *(Datei: DSLExample.groovy)*

Damit ist klar, wer bezahlt und womit bezahlt wird. Beim Bezahlen soll der Methode `pay` des Getränkeautomaten ein Objekt vom Typ `MoneyContainer` übergeben und damit die Rechnung für die Bestellung beglichen werden. Das übergebene Objekt entspricht also zum Beispiel der Kreditkarte einer Person. In der `pay`-Methode wird überprüft, ob das Zahlungsmittel zugelassen ist und ob genügend Guthaben vorhanden ist. Ist dies der Fall, wird der Gesamtbetrag vom Guthaben abgezogen, die benötigten Zutaten aus dem Vorrat entfernt und der Automat mittels der Methode `reset` wieder in seinen Ursprungszustand versetzt.

```
1  void pay(MoneyContainer money) {
2      if (money.value < 0) {
3          println "Paytype not supported"
4      } else if (money.value < total) {
5          println "Not enough money left"
6      } else {
7          money.value -= total
8
9          ingredientsNeeded.each {
10             key, value ->
11                 stock[key] -= value
12         }
13
14         reset()
15     }
16 }
```

Listing 35: Die `pay`-Methode der Klasse `DrinkDispensingMachine`

Die `reset`-Methode initialisiert dabei lediglich die Maps `choice` und `ingredientsNeeded` neu und setzt den Gesamtbetrag wieder zurück auf Null (siehe Listing 36).

```
1 void reset() {
2     total = 0
3     choice = [:]
4     ingredientsNeeded = [:]
5 }
```

Listing 36: Die `reset`-Methode der Klasse `DrinkDispensingMachine`

Nachdem der Zeiger auf die pay-Methode in der `Initialize.groovy` wieder der Variable `pay` zugewiesen wurde (`pay = machine.&pay`), kann in der DSL jetzt eine Bestellung mit folgender Syntax bezahlt werden:

```
1 person = new Person(Name: 'Timo Müller', (creditcard): 30.00)
2 pay person.creditcard
```

Listing 37: Bezahlen einer Bestellung *(Datei: DSLExample.groovy)*

Allerdings soll es in der DSL nicht nötig sein die Variable zu kennen in der eine Person abgelegt wurde. Stattdessen soll die Person direkt anhand ihres Namens gefunden werden. Einen beispielhaften Aufruf zeigt Listing 38.

```
1 person = new Person(Name: 'Timo Müller', (creditcard): 30.00)
2 pay "Timo Müller".creditcard
```

Listing 38: Vereinfachtes Bezahlen einer Bestellung*(Datei: DSLExample.groovy)*

Um dieses Verhalten zu realisieren verwendet man, wie in Abschnitt 5.3, die `ExpandoMetaClass` und implementiert die Methode `propertyMissing`. Da es sich beim Namen der Person allerdings um einen String handelt, muss hier Klasse `String` und nicht die `Integer`-Klasse erweitert werden.

```groovy
// Das Auffinden einer Person (Instanz) anhand ihres Namens
String.metaClass.propertyMissing = {
    String propertyName ->
        if (propertyName in ['creditcard', 'cashcard', 'purse']) {
            def test =  binding.variables.find {
                key, value ->
                if (value instanceof Person) {
                    if (value.name == owner.delegate) {
                        return true
                    }
                    return false
                }
                return false
            }

            return test.value."$propertyName"
        } else {
            return new MoneyContainer(value:-1)
        }
}
```

Listing 39: Das Erweitern der Klasse `String` *(Datei: Initialize.groovy)*

Als Erstes wird wieder geprüft, ob ein erwünschtes Attribut abgefragt wurde. Andernfalls wird ein neuer `MoneyContainer` erzeugt, der ein negatives Guthaben besitzt, das wiederum zu einem Fehler in der `pay`-Methode des Automaten führt. Wurde hingegen auf ein erlaubtes Zahlungsmittel zugegriffen, durchsucht man in der Closure das Binding nach einer Klasse vom Typ `Person`, deren Namen dem Gewünschten entspricht.
Das Binding ist eine Map, die alle Objekte enthält, die innerhalb eines Skriptes angelegt werden. Das Durchsuchen des Bindings erfolgt mittels der Methode `find`, der eine weitere Closure übergeben wird. `find` ruft die Closure für jedes Objekt des Bindings auf und gibt den aktuellen Map-Eintrag zurück, wenn die Closure den Wert `true` zurückgibt. Dies geschieht dann, wenn der Wert des aktuellen Map Eintrags vom Typ `Person` ist und der Name dieser Person dem String entspricht, auf den das Attribut aufgerufen wurde (`owner.delegate`).
Damit sind alle Funktionen des Automaten in der DSL abgebildet und es können alle Abläufe vom Verwalten des Vorrats bis zum Einkauf sehr einfach in der eigenen Sprache ausgedrückt werden.

6 Fazit

Im Rahmen dieser Arbeit wurde die Erstellung einer DSL auf Basis der Skriptsprache Groovy vorgestellt. Dazu wurde für die Domäne eines Getränkeautomaten eine einfache DSL entworfen, die in Form einer internen DSL in Groovy umgesetzt wurde.
Prinzipiell lässt sich festhalten, dass sich mit Groovy schnell und einfach eine interne DSL für einen überschaubaren Problembereich erstellen lässt. Die dynamischen Sprachkonzepte wie das MOP bieten eine gute Unterstützung bei der Erstellung.
Jedoch steigt die Komplexität mit der Größe des Problembereichs, so dass in Groovy erstellte DSLs schnell an ihre Grenzen stoßen und damit die Übersichtlichkeit und die Wartbarkeit leiden. Zudem greifen die generellen Nachteile einer internen DSL, bei der nach wie vor auf Sprachkonstrukte der Programmiersprache zurückgegriffen werden muss. Weiterhin ist anzumerken, dass eine grafische Notation zur Verständlichkeit beitragen könnte. Ein weiterer Nachteil ist das Fehlen eines Validators für die erstellte interne DSL, so dass Fehler erst zur Laufzeit erkannt werden können.
Zusammenfassend lässt sich sagen, dass sich mit Groovy erstellte DSLs vor allem für kleine, klar abgegrenzte Problembereiche eignen. In diesen Bereichen kann dadurch der Entwicklungsprozess beschleunigt und die Kommunikation verbessert werden.

A Anhang: Quelltexte

```
1  import java.util.HashMap
2  import groovy.lang.GroovyShell
3  import java.lang.String
4
5  // Initialisieren der benötigten Klassen
6  machine = new DrinkDispensingMachine()
7  price = new ReturnerClass()
8  ingredients = price
9
10 // Namenszuweiseung der Zutaten
11 coffeebeans = 'Coffeebeans'
12 teabags = 'Teabags'
13 milk = 'Milk'
14 sugar = 'Sugar'
15 water = 'Water'
16
17 // Namenszuweiseung der Zahlungsarten
18 creditcard = 'Creditcard'
19 cashcard = 'Cashcard'
20 purse = 'Purse'
21
22 // Namenszuweiseung der Getränke
23 coffee = 'Coffee'
24 tea = 'Tea'
25
26 // Namenszuweiseung der für den Gesamtbetrag
27 total = 'Total'
28
29 // Methodenzuweisungen
30 fill = machine.&fill
31 remove = machine.&remove
32 pay = machine.&pay
33 show = machine.&show
34 define = machine.&define
35 choose = machine.&choose
36
37
38
39
```

```
40 // Umrechnung der Einheiten für die Zutaten
41 Integer.metaClass.propertyMissing = {
42   String propertyName ->
43     def units = ['g':1, 'Kg':1000, 'ml':1, 'L':1000]
44     def unit = units[propertyName]
45
46     if (unit) {
47       return delegate * unit
48     } else {
49       println "Conversion from $propertyName to gramm not available"
50     }
51 }
52
53 // Das Auffinden einer Person (Instanz) anhand ihres Namens
54 String.metaClass.propertyMissing = {
55   String propertyName ->
56     if (propertyName in ['creditcard', 'cashcard', 'purse']) {
57       def test =  binding.variables.find {
58         key, value ->
59         if (value instanceof Person) {
60           if (value.name == owner.delegate) {
61             return true
62           }
63           return false
64         }
65         return false
66       }
67
68       return test.value."$propertyName"
69     } else {
70       return new MoneyContainer(value:-1)
71     }
72 }
73
74 // DSL laden
75 use(UniqueNameUtil) {
76   GroovyShell sh = new GroovyShell(binding)
77   sh.evaluate(new File("DSLExample.groovy"))
78 }
```

Listing 40: Initialize.groovy

```
1  import java.math.BigDecimal
2
3  public class DrinkDispensingMachine{
4    // Gesamtbetrag der Bestellung
5    def total = 0
6
7    // Die gewählten Getränke und die Menge
8    def choice = [:]
9
10   // Die Zutaten, die für die Bestellung benötigt werden
11   def ingredientsNeeded = [:]
12
13   // Die Liste der Zutaten, die für ein Getränk benötigt werden
14   def ingredients = ['Coffee': [],
15                      'CoffeeWithSugar': [],
16                      'CoffeeWithMilk': [],
17                      'CoffeeWithMilkAndSugar': [],
18                      'Tea': [],
19                      'TeaWithSugar': [],
20                      'TeaWithMilk': [],
21                      'TeaWithMilkAndSugar': []]
22
23   // Die Preise
24   def prices = ['Coffee': 0.00,
25                  'CoffeeWithSugar': 0.00,
26                  'CoffeeWithMilk': 0.00,
27                  'CoffeeWithMilkAndSugar': 0.00,
28                  'Tea': 0.00,
29                  'TeaWithSugar': 0.00,
30                  'TeaWithMilk': 0.00,
31                  'TeaWithMilkAndSugar': 0.00]
32
33   // Die bevorratete Menge der Zutaten
34   def stock = ['Coffeebeans': 0.00,
35                'Teabags': 0.00,
36                'Milk': 0.00,
37                'Sugar': 0.00,
38                'Water': 0.00]
39
40
41
```

```
42  // Fügt die gegebene Menge vom angegebenen Typ zum Vorrat hinzu
43  void fill(int amount, type) {
44    stock[type] += amount
45  }
46
47  // Entfernt die gegebene Menge vom angegebenen Typ vom Vorrat
48  void remove(int amount, type) {
49    int left = stock[type]
50    if (left >= amount) {
51      stock[type] -= amount
52    } else {
53      println "Not enough $type left"
54    }
55  }
56
57  // Versetzt den Automat in seinen Ausgangszustand
58  void reset() {
59    total = 0
60    choice = [:]
61    ingredientsNeeded = [:]
62  }
63
64  // Wenn genug Geld in money vorhanden ist
65  // wird der Gesamtbetrag abgezogen
66  void pay(MoneyContainer money) {
67    if (money.value < 0) {
68      println "Paytype not supported"
69    } else if (money.value < total) {
70      println "Not enough money left"
71    } else {
72      money.value -= total
73
74      ingredientsNeeded.each {
75        key, value ->
76          stock[key] -= value
77      }
78
79      reset()
80    }
81  }
82
```

```
83  // Zeigt Einzelpreis oder Gesamtbetrag der Bestellung
84  void show(String type) {
85    String amount = 'No price available for beverage!'
86
87    switch(type) {
88      case 'Total':
89        amount = "Total: $total"
90        break
91
92      default:
93        if (prices.get(type) != null) {
94          amount = "Price of $type: ${prices.get(type)}"
95        }
96    }
97    println amount
98  }
99
100 // Erlaubt die Definition von Preisen und Zutatenlisten
101 void define(ArrayList values) {
102   switch(values[1]) {
103     case { it instanceof BigDecimal }:
104       prices?.putAt(values[0], values[1])
105       break
106
107     case { it instanceof Map }:
108       ingredients?.putAt(values[0], values[1])
109   }
110 }
111
112 // Die Auswahl eines Getränks in der angegebenen Menge
113 void choose(amount, type) {
114   def ingr = ingredients[type]
115
116   try {
117     ingr.each {
118       key, value ->
119       def amount_choice = ingredientsNeeded.get(key, 0)
120       if (stock[key] < ((value * amount) + amount_choice)) {
121         throw new Exception("Not enough $key")
122       }
123     }
```

```
124       ingr.each {
125         key, value ->
126         if (ingredientsNeeded.get(key)) {
127           ingredientsNeeded[key] += (value * amount)
128         } else {
129           ingredientsNeeded[key] = (value * amount)
130         }
131       }
132
133       if (choice.get(type)) {
134         choice[type] += amount
135       } else {
136         choice[type] = amount
137       }
138
139       total += prices[type] * amount
140     } catch (Exception e) {
141       println e.message
142     }
143   }
144 }
```

Listing 41: DrinkDispensingMachine.groovy

```
1  public class ReturnerClass{
2    String of(String returnValue) {
3      return returnValue
4    }
5
6    def of(String key, def data) {
7      return [key, data]
8    }
9  }
```

Listing 42: ReturnerClass.groovy

```groovy
public class UniqueNameUtil{
  def static with(Object self, String firstIngredient) {
    return "${self}With${firstIngredient}"
  }

  def static with(Object self, String firstIngredient,
        String secondIngredient) {
    if (firstIngredient == 'Sugar') {
      return "${self}With${secondIngredient}And${firstIngredient}"
    } else {
      return "${self}With${firstIngredient}And${secondIngredient}"
    }
  }
}
```

Listing 43: UniqueNameUtil.groovy

```groovy
public class MoneyContainer{
  def value
}
```

Listing 44: MoneyContainer.groovy

```groovy
public class Person{
  MoneyContainer creditcard = new MoneyContainer()
  MoneyContainer cashcard = new MoneyContainer()
  MoneyContainer purse = new MoneyContainer()
  String name

  Person(Map vars) {
    name = vars.get('Name', 'Noname')
    creditcard.value = vars.get('Creditcard', 0)
    cashcard.value = vars.get('Cashcard', 0)
    purse.value = vars.get('Purse', 0)
  }
}
```

Listing 45: Person.groovy

```
1  // Befüllen und Entleeren des Vorrats
2  fill 500.g, coffeebeans
3  fill 20.L, water
4  fill 20.L, milk
5  fill 20.g, sugar
6  remove 10.g, coffeebeans
7
8  // Definition der Preise und der Zutaten
9  define price.of(coffee, 1.56)
10 define ingredients.of(coffee, [(water): 200.ml, (coffeebeans): 20.g])
11
12 define price.of(coffee.with(sugar, milk), 1.70)
13 define ingredients.of(coffee.with(sugar, milk), [(water): 200.ml,
14     (coffeebeans): 20.g, (milk): 5.ml, (sugar): 2.g])
15
16 // Abfrage der Preise
17 show price.of(coffee.with(sugar, milk))
18 show price.of(coffee)
19
20 // Ein beispielhafter Kaufvorgang
21 choose 1, coffee
22 choose 1, coffee
23 choose 4, coffee.with(sugar, milk)
24
25 // Anzeige des Gesamtbetrages der Bestellung
26 show total
27
28 // Bezahlvorgang
29 person = new Person(Name: 'Timo Müller', (creditcard): 30.00)
30 pay 'Timo Müller'.creditcard
```

Listing 46: DSLExample.groovy

Literatur

[Barszczewski und Scharff 2009] BARSZCZEWSKI, Sebastian ; SCHARFF, Stephan: DSLs mit Groovy. In: *Javamagazin 1.09* (2009), Januar

[Baumann 2008] BAUMANN, Joachim: *Groovy - Grundlagen und fortgeschrittene Techniken*. Heidelberg : dpunkt.verlag GmbH, 2008

[Rozanski 2008] ROZANSKI, Uwe: *Groovy 1.5*. Heidelberg : mitp-Verlag, 2008

[Stahl u. a. 2007] STAHL, Thomas u. a.: *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. 2., aktualisierte und erweiterte Auflage. Heidelberg : dpunkt.verlag GmbH, 2007

[Sun Microsystems, Inc. 2006] SUN MICROSYSTEMS, INC.: The Collections Framework. In: *http://java.sun.com/javase/6/docs/technotes/guides/collections/index.html (Zugriff: Februar 2009)* (2006)

[Wikipedia 2009] WIKIPEDIA: Domänenspezifische Sprache - Wikipedia, Die freie Enzyklopädie. In: *http://de.wikipedia.org/w/index.php?title=Dom%C3%A4nenspezifische_-Sprache&oldid=56058542 (Zugriff: Februar 2009)* (2009)