Meta-Programmierung in Groovy

Lars Blumberg, Christoph Hartmann und Arvid Heise

Hasso-Plattner-Institut für Softwaresystemtechnik GmbH

Zusammenfassung. Der Name der Programmiersprache ist Programm. Dass Groovy als noch junge Sprache viele neue und interessante Sprachkonzepte mitbringt, möchten wir im ersten Teil dieses Papers aufzeigen. Ein weiteres Highlight des ersten Kapitels ist vielleicht die Erkenntnis über die gelungene, nahtlos Integration mit der Java-Plattform. Im folgenden Hauptteil wird insbesondere die Möglichkeit der Meta-Programmierung mit Groovy beschrieben. Es wird auf die Umsetzung des Meta-Object-Protocols eingegangen und erläutert, wie die zugehörigen Konzepte mit Hilfe der Java Virtual Machine umgesetzt werden konnten. Im dritten und letzten Teil wird dem Leser eine implementierte Beispiel-Anwendung vorgestellt. Sie ist wie dieses Paper im Rahmen der Veranstaltung "Metaprogrammierung und Reflection" entstanden und demonstriert eindrucksvoll die Meta-Programmierung in Groovy.

1 Einführung

1.1 Das Zusammenspiel von Groovy und Java

Aufgrund der noch jungen Entstehungsgeschichte von Groovy steht die Programmiersprache in einem engen Zusammenhang zur Sprache Java und deren Ausführungsplattform. Diese grundlegende Abhängigkeit von Groovy aber auch die daraus resultierenden Möglichkeiten für Java-Programmierer beleuchten die beiden folgenden Abschnitte näher.

1.1.1 Meine Klasse ist auch deine Klasse Es gibt diese weltweit verbreitete Redewendung - *My castle is your castle.* In leicht abgewandelter Form trifft diese Mundart auch auf das Zusammenspiel von Groovy und Java zu: *My class is your class* - Meine Klasse ist auch deine Klasse. Jede Klasse, die auf einer Java-Plattform lauffähig ist, kann in einem Groovy-Programm wiederverwendet werden. Einem Groovy-Entwickler steht somit nicht nur das gesamte Java Development Kit (JDK) – Java's außerordentlich umfangreiche und mächtige Klassenbibliothek – zur Verfügung. Auch sämtliche Java-Frameworks wie Hibernate, Struts und Swing als auch selbstgeschriebene Bibliotheken können ohne Umwegen und Einschränkungen in Groovy wiederverwendet werden. Ein in Groovy programmierender Java-Entwickler muss sich also nicht erst in neue Umgebungen einarbeiten – er kann auf die ihm vertrauten Werkzeuge zurückgreifen. Aber es können nicht nur Java-Klassen in Groovy wiederverwendet werden.

Andersherum verhält es sich nämlich genauso, wie Abbildung 1 veranschaulicht. Die gemeinsame Ausführungsumgebung, das Java Runtime Environment (JRE), in der Programme beider Sprachen abgewickelt werden, ermöglicht ebenso die Wiederverwendung von Groovy-Code in Java-Anwendungen. Jede Klasse, die mit Groovy erstellt wird, kann somit auch in Java benutzt werden. Eine technisch tiefergehende Erklärung liefert der nachfolgende Abschnitt.

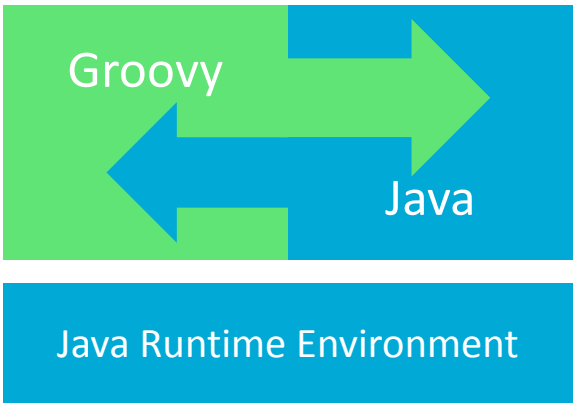

Abb. 1. Groovy und Java bilden eine enge Gemeinschaft (adoptiert von [1])

1.1.2 Zwei Möglichkeiten der Ausführung von Groovy-Code Dem Java-Entwickler wird es nicht fremd vorkommen, dass auch Groovy-Code auf alt hergebrachte Weise von einem Compiler vor der Programmabwicklung übersetzt werden kann. Hierbei besitzt der Groovy-Compiler *groovyc* eine bedeutungsvolle Gemeinsamkeit mit dem Java-Compiler *javac*: Beide erzeugen nämlich Java-Bytecode. Das ist auch schon das große Geheimnis weshalb sich Groovy und Java so gut verstehen. Aufgrund desselben Bytecodes wird jedes Groovy-kompilierte Programm direkt von der Java-Plattform ausgeführt. Der Java-Classloader, welcher die vorkompilierten Java-Klassen zur Laufzeit in die Ausführungsumgebung lädt, weiß nicht einmal, dass es sich ursprünglich um eine Groovy-Klasse handelte. Kompilierte Groovy-Klassen werden von der Ausführungsumgebung JRE somit wie kompilierte Java-Klassen behandelt.
Neu für Java-Entwickler ist allerdings die zweite Möglichkeit, nämlich Groovy als Skriptsprache zu verwenden. In diesem Fall werden Groovy-Klassen erst zum benötigten Zeitpunkt während der Laufzeit kompiliert. Wird in einer Groovy-Skriptumgebung eine noch nicht geladene Klasse verwendet, kommt der Groovy-Classloader zum Einsatz. Die Laufzeitumgebung macht den zugehörigen Groovy-Code anhand des Paket- und Klassennamens ausfindig und stößt die Kompilierung der Klasse mit Hilfe des Groovy-Compilers an. Nachdem aus dem Groovy-Quellcode Java-Bytecode generiert wurde, kann der ClassLoader die geladene

Klasse der Laufzeitumgebung zur Verfügung stellen. In beiden Fällen – sowohl bei der Verwendung von vorkompiliertem Groovy-Code als auch bei der Verwendung von Groovy als Skriptsprache – steht am Ende die geladene Klasse zur Verfügung. Die JRE macht dabei keine Unterscheidung zwischen Groovy- und Java-Klassen weil beide in Java-Bytecode formuliert sind.

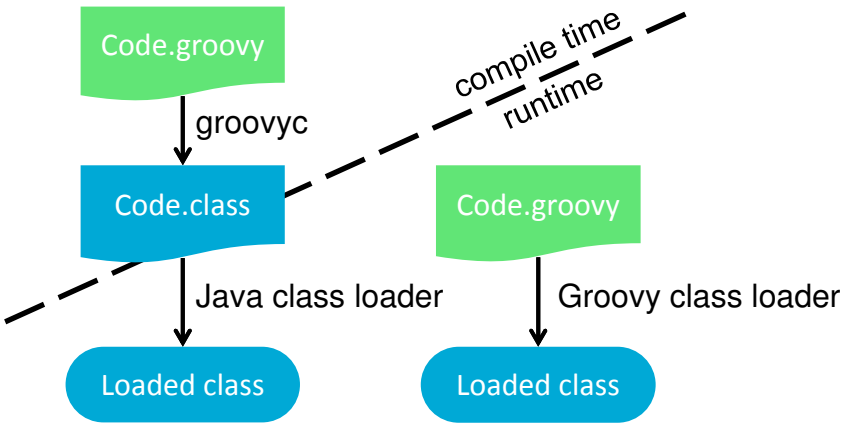

Abb. 2. Vorkompilierter Code vs. Groovy als Skriptsprache (adoptiert [1])

1.2 Eine Einführung in Groovy's Sprachkonzepte

In dieser Einführung möchten wir die interessantesten Sprachkonzepte von Groovy vorstellen. Es handelt sich also nicht um vollständige Auflistung aller Möglichkeiten, stattdessen möchten wir dem Leser einen kleinen Überblick geben und Gefühl für die Sprache vermitteln.

1.2.1 Fast jeder Java-Code kann in Groovy übernommen werden

Java-Entwickler haben es sehr leicht, Programm-Code in Groovy zu formulieren. Da Groovy die Syntax von Java bis auf sehr kleine Ausnahmen vollständig unterstützt, kann beliebiger Java-Code in ein Groovy-Programm übernommen und ausgeführt werden. Allerdings wäre Groovy nicht so "groovy", wenn sich deren Entwickler nicht so einige Vereinfachungen hätten einfallen lassen. So wollen wir mit einem Code-Beispiel beginnen, das sowohl Java- als auch Groovy-kompatibel ist – es lässt es sich also von den Compilern beider Sprachen übersetzen. Da wir uns aber auf Groovy konzentrieren wollen, laden wir den folgenden Code in die Groovy-Skriptumgebung und führen ihn aus.

```
import java.util.Date;

public class Foo {
```

```
    public static void main(String[] args) {
        int start = 1;
        int summe = 0;

        for (int i = start; i <= 5; i++) {
            summe += i;
        }

        Date now = new Date();
        System.out.println(now.toString() + " Die Summe ist: "
            + summe);
    }
}
```

Listing 1.1. Ein Groovy-Programm wie es auch mit Java ausführbar wäre

Führen wir den Quelltext nun als Groovy-Programm aus, erhalten wir dieselbe Ausgabe als hätten wir es als Java-Programm ausgeführt:

```
Mon Feb 04 15:52:15 CET 2008 Die Summe ist: 15
```

Groovy gestattet es uns für dieses Beispiel, zahlreiche Vereinfachungen anzuwenden. Schöpfen wir nun alle Möglichkeiten aus, können wir das erste Beispiel auch in folgender, verkürzter Weise notieren:

```
        start = 1
        summe = 0

        for (i in start..5) {
            summe += i
        }

        now = new Date()
        println "$now Die Summe ist: $summe"
```

Listing 1.2. Die verkürzte Fassung des vorherigen Beispiels

1.2.2 Über Closures und die Unterstützung von Arrays, Listen und Maps auf Sprachebene

In Groovy können namenlose Methoden - sogenannte *Closures* - definiert werden. Es handelt sich um ein Stück Code, das Parameter entgegennehmen und an beliebiger Stelle ausgeführt werden kann. Da Closures selbst Objekte sind, können sie als Parameter an Methodenaufrufe übergeben werden. Das Groovy-Framework (Groovy Development Kit) macht intensiven Gebrauch davon. So bieten alle Container-Typen beispielsweise die Methode *each()* an. Diese Methode erwartet als einzigen Parameter ein Closure, dessen Code für jedes Element in dem Container ausgeführt wird. Weiterhin demonstriert das folgende Code-Beispiel die native Sprachunterstützung für Aufzählungstypen, wie Maps. Solche Aufzählungen können mit einer ausdrucksstarken Syntax vorbelegt und manipuliert werden.

```
map = ["a":1, "b":2]

map.each { key, value ->
    map[key] = value*2
}
println map
```

Listing 1.3. Closures in Verbindung mit Maps

Erwartungsgemäß liefert das Skript folgende Ausgabe:

```
["a":2, "b":4]
```

1.2.3 Klassen und Beans Auch Groovy ist eine objektorientierte Sprache und Klassen werden in derselben Weise wie in Java formuliert. Interessant ist hierbei das *GroovyBean*-Konzept, das jedem öffentlichen Klassenattribut automatisch *get*- und *set*-Methoden bereitstellt. Wird in Groovy das Sichtbarkeitsattribut weggelassen, wird automatisch *private* angenommen. Gleichzeitig werden aber öffentliche get- und set-Methoden bereitgestellt, um darauf zugreifen zu können. Das nachfolgende Programmbeispiel veranschaulicht es: Obwohl wir keine get- und set-Methoden für das Attribut *title* definiert haben, können wir sie dennoch aufrufen. Außerdem stellt Groovy zur Laufzeit get- und set-Methoden auch für Java-Klassen bereit, wenn es ein öffentliches und namentlich passendes Objektattribut gibt.
Ein weiteres interessantes Feature, das wir hier benutzt haben, ist die Operatorenüberladung. Wie man dem Beispiel entnehmen kann, definieren wir einen *plus*-Operator für *Book*-Instanzen indem wir eine gleichnamige Methode definieren.

```
class Book {
    String title

    Book (String theTitle) {
        title = theTitle
    }
    Book plus(Book anotherBook) {
        return new Book(title + ", " + anotherBook.title)
    }
}
Book gina = new Book("Groovy in Action")
println gina.getTitle()

println ((gina + new Book("Groovy for Experts")).title)
```

Listing 1.4. Ein GroovyBean mit impliziten get- und set-Methoden

Führt man dieses Skript aus, zeigt die Konsole folgende Ausgabe an:

```
Groovy in Action
Groovy in Action, Groovy for Experts
```

2 Meta-Programmierung mit Groovy

Groovy ist eine mächtige, dynamische Programmiersprache, was sich vor allem in dem wohldurchdachten und sehr flexiblen Metaklassenmodell widerspiegelt. Im Sinne einer Open Implementation hat der Programmierer nahezu alle Möglichkeiten sein Programm mittels Reflection zur Laufzeit zu verändern und damit an die individuellen Bedürfnisse anzupassen. In den folgenden Abschnitten wird die Fähigkeit von Groovy zur *Reflection* beleuchtet. Dafür werden die vorgestellten Aspekte des Systems eingeteilt in Konstrukte zur *Introspection* und *Intercession*.

Introspection bezeichnet die Fähigkeit, auf Informationen über die Zustände von Objekte, deren Klassen und Verhalten zur Laufzeit zugreifen zu können.

Intercession erlaubt Zustände und Verhalten von Objekten, aber auch deren Klassen zur Laufzeit zu verändern. *Intercession* setzt meist *Introspection* voraus.

2.1 Groovy im Vergleich

Da Groovy auf der Java VM ausgeführt wird und somit auch den Grundregeln des Javaklassenmodells folgen muss, unterliegt es auch dessen Einschränkungen. In Java ist eine Veränderungen der Klassen zur Laufzeit nicht vorgesehen. Die Java Runtime Environement stellt mit dem java.reflect Package primär eine Möglichkeit zur *Introspection* bereit. Zur Laufzeit verändert werden kann lediglich der Zustand und nicht das Verhalten. Mittels Hotspot gibt es zwar eine VM Implementierung, die es erlaubt bestehende Methode zu ersetzen, allerdings dient sie hauptsächlich zum Debugging und Optimieren des Bytecodes. Erst Javassist [3] und Reflex [2] erlauben echtes dynamisches Verhalten, in dem auf Bytecode Ebene die Klasse verändert wird. Dazu muss allerdings die Klasse teilweise umständlich entladen und neugeladen werden.

2.2 Meta Object Protocol in Groovy

Ein *Meta Object Protocol* auf der Java VM kann somit nur mittels einer zusätzlichen Indirektionsschicht realisiert werden. Dazu werden bestehende Konzepte von Java benutzt und um ein Metaklassenmodell erweitert. Aus dem Java Unterbau ergeben sich folgende Grundsätze:

Wie in Java ist in Groovy jede Klasse abgeleitet von *Object*. Groovy ist in diesem Hinblick aber wesentlich konsequenter, da auf primitive Datentypen bewusst verzichtet wurde, um die Inkonsistenzen bei der Behandlung von primitiven Datentypen und Klassen zu vermeiden. Weiterhin ist jede Klasse in Groovy eine Instanz von *Class*, womit *Class* also auch in Groovy die Klasse der Klassen bleibt. Es stehen somit alle von Java gewohnten Möglichkeiten zur *Reflection* zur Verfügung, diese werden ausführlich in [4] diskutiert und werden hier bewusst ausgelassen. Allerdings sind die aus Groovy Code generierten Java Klassen mit zusätzlichem Code angereichert, der eine *Reflection* auf Java Ebene verkomplizieren könnte.

2.2.1 Erweiterungen Die zusätzliche Indirektionsschicht in Groovy spiegelt sich in zwei Interfaces wider: *GroovyObject* und *MetaClass*. Jede in Groovy geschriebene Klasse implementiert das *GroovyObject* Interface implizit, wobei die fünf deklarierten Methoden vom Groovy Compiler *groovyc* automatisch generiert werden, sofern sie nicht explizit definiert werden.

Methoden	Beschreibung
getMetaClass setMetaClass	Nicht nur eine Klasse hat eine Metaklasse, sondern auch jedes einzelne Groovy Objekt kann eine von der Klasse unabhängige Metaklasse haben (siehe 2.4). Diese instanzspezifische Metaklasse ist über die beiden Methoden zugänglich.
getProperty setProperty	Properties definieren den Zustand eines Objektes und werden in Groovy primär auf Instanz und Klassenebene abgebildet. Jede Groovy Klasse kann durch Überschreiben dieser zwei Methoden dynamische *Properties* auf Objektebene oder Klassenebene erzeugen (siehe auch 2.7).
invokeMethod	Das Verhalten eines Objektes ist wiederum eine Ebene höher angesiedelt, spielt sich also auf Klassen- oder Metaklassenebene ab. Normalerweise wird dynamisches Verhalten damit auf Metaklassenebene realisiert, sodass diese Methode der *GroovyObject*s gar nicht erst aufgerufen wird. Nur im Fehlerfall oder mit Hilfe des Tag-Interfaces *GroovyInterceptable* wird *invokeMethod* aufgerufen (in 2.5 genau beschrieben).

Tabelle 1. Methoden eines GroovyObjects

Um auch in Java geschriebene Klassen die Möglichkeit zur instanzspezifischen *Intercession* zu geben, muss das *GroovyObject* Interface implementiert werden oder man erbt gleich von *GroovyObjectSupport*. Letzteres ist eine Standardimplementation von *GroovyObject* und wird auch exzessiv von der Groovy Bibliothek benutzt, um die grundlegenden Typen wie Closures performant in Java zu implementieren. Während *GroovyObject* dynamisches Verhalten auf Objekt- und Klassenebene erlaubt, ist das zweite elementare Interface *MetaClass* die Grundlage für das sehr ausgewogene Metaklassenmodell in Groovy.

2.2.2 Metaklassen, Klassen und Instanzen Läuft ein Programm ohne *Intercession*, so gelten für die Metaklassen der Klassen und Instanzen folgende Grundaussagen: Jede Klasse hat eine Metaklasse, die eine Instanz von *MetaClassImpl* ist und damit *MetaClass* implementiert. Diese Instanzen werden dynamisch erzeugt und sind bis auf wenige Ausnahmen direkt *MetaClassImpl* Instanzen. Java Klassen erhalten eine Instanz von *ExpandoMetaClass* als Metaklasse, damit auch diesen Klassen Methoden hinzugefügt werden können. Neue Instanzen von Klassen werden über die Metaklasse der Klasse erstellt und haben diese Metaklasse als Metaklasse.

Methoden	Beschreibung
getProperties getMethods getMetaMethods	Die Methoden der *Introspection* in Groovy. Der Unterschied zwischen getMethods und getMetaMethods wird in 2.5 näher erläutert.
getClassNode	Liefert den AST der Metaklasse sofern verfügbar. Erlaubt auch die Modifikation dieses ASTs und ist damit sowohl für *Introspection* als auch *Intercession* geeignet. Aufgrund der Komplexität des ASTs wird allerdings dynamisches Verhalten häufiger über die *ExpandoMetaClass* realisiert und der AST verwendet, um Quelltextfragmente neu zu interpretieren. In den Standardbibliotheken wird so zum Beispiel eine Closure automatisch in ein SQL Statement umgewandelt, um das SELECT Statement performant zu benutzen.
invokeMethod	Jeder Methodenaufruf wird primär von der Metaklasse behandelt und entweder an die *invokeMethod* Funktion des *GroovyObject*s oder aber an die entsprechende Methode der Klasse delegiert. Das Erzeugen einer eigenen Metaklasse und Überschreiben dieser Methode ist die Hauptmöglichkeit für dynamisches Verhalten in Groovy außer der *ExpandoMetaClass*.
getProperty setProperty	Die Methoden zur Property-Unterstützung auf Metaklassenebene werden von der Standardimplementierung von den entsprechenden Methoden von *GroovyObject* aufgerufen. Die Aufrufreihenfolge ist im Vergleich zu *invokeMethod* also genau andersherum. Auch diese Methoden sind für *Intercession* geeignet.
invokeMissingMethod invokeMissingProperty	Diese Backupmethoden werden jeweils aufgerufen, wenn die normalen Aufrufmechanismen fehlgeschlagen sind. *Intercession* mit diesen Methoden erlaubt zum Beispiel die Erweiterung von Klassen und Objekten um zusätzliche Properties zur Laufzeit.

Tabelle 2. Methoden eines GroovyObjects

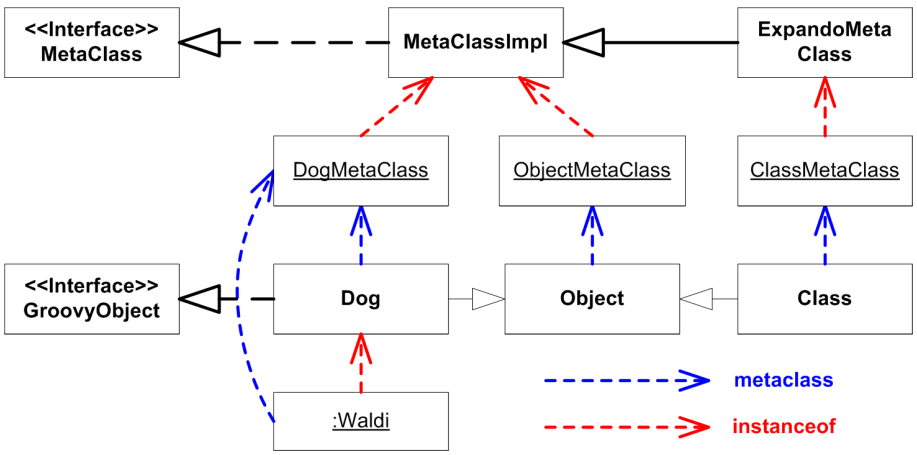

Abb. 3. Beziehung von Metaklassen, Klassen und Instanzen

Die automatisch erzeugten Metaklassen für Klassen und Objekte können jederzeit ersetzt werden, wie Listing 1.5 zeigt. Dabei wird wohl in den meisten Fälle auf *MetaClassImpl* als Basis für die eigene Implementation zurückgreifen.

```
import org.codehaus.groovy.runtime.metaclass.*

class WaldiMeta extends MetaClassImpl {
    WaldiMeta() {
        super(GroovySystem.getMetaClassRegistry(), Dog)
        initialize()
    }
}

// Change the metaClass for this instance only
waldi = new Dog(name: 'Waldi')
waldi.metaClass = new WaldiMeta()
assert waldi.metaClass != Dog.metaClass

// Change for class and every _new_ instance
registry = GroovySystem.getMetaClassRegistry()
oldMetaClass = registry.getMetaClass(Dog)
registry.setMetaClass(Dog, new WaldiMeta())
bruno = new Dog(name: 'Bruno')

assert registry.getMetaClass(Dog) != oldMetaClass
assert bruno.metaClass == registry.getMetaClass(Dog)
assert bruno.metaClass != waldi.metaClass
```

Listing 1.5. Ändern der Metaklasse für Klassen und Instanzen

2.3 Running Example

Um die Funktionsweise von Groovy und insbesondere die Indirektionsschicht zu untersuchen, wird die folgende Klasse *Dog* verwendet. Sie hat ein *Property name* und die Methode *bark*. Außerdem wurde die *toString* Methode von java.lang.Object überschrieben, um eine einfachere Ausgabe zu ermöglichen.

```
class Dog {
    String name = 'dog'

    def bark() {
        System.out.println "$name: wau"
    }

    String toString() {
        name
    }
}
```

Listing 1.6. Running Example für die Metaklassen

2.4 Objektinstanziierung

Offensichtlich kann die Objekterzeugung in Groovy nicht mit der aus Java 1:1 übereinstimmen, da jede Erzeugung von Instanzen durch die Metaklasse gesteuert wird. Außerdem muss an einem Punkt die neue Instanz mit ihrer Metaklasse verknüpft werden.

```
static void main(args) {
      new Dog()
}
```

Wird die Erzeugung eines Hundes betrachtet, so erzeugt groovyc folgenden Aufruf in Java Bytecode.

```
    ScriptBytecodeAdapter.invokeNewN(DogExample.class, Dog.
        class, new Object[0])
```

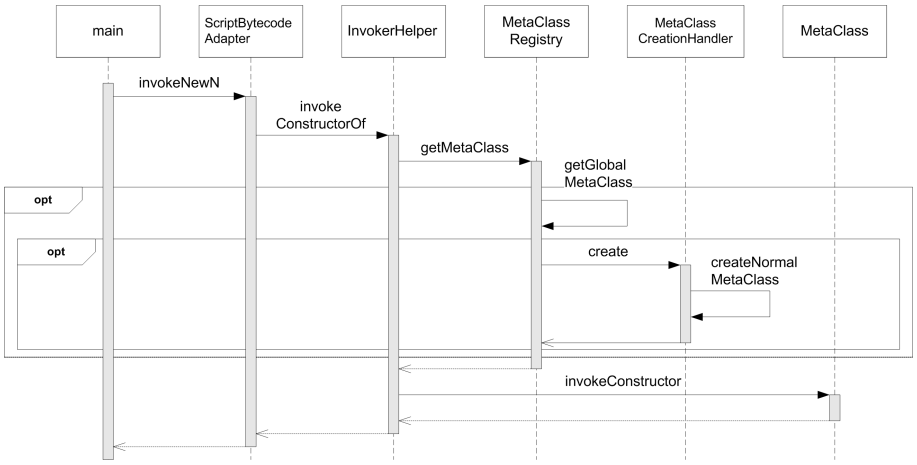

Abb. 4. Sequenzdiagramm für das Erstellen eines neuen Objektes

Der *ScriptBytecodeAdapter* ist eine Hilfsklasse, der den Zugriff auf die inneren Groovy Mechanismen zum Aufruf von Methoden, Zugriff auf Properties und die erweiterte Operatorenunterstützung kapselt. Der primäre Zweck dieser Kapselung ist die vereinfachte Generierung von Bytecode. Aufrufe auf den *ScriptBytecodeAdapter* werden in den meisten Fällen nach einer kurzen Verarbeitung direkt an die zentrale Klasse *InvokerHelper* delegiert.

InvokerHelper ist das Herz der dynamischen Programmierung in Groovy. Hier werden Methodenaufrufe an die Metaklassen, Klassen oder Instanzen weitergeleitet sowie Zugriffe auf Properties entsprechend an die GroovyBean Methoden umgewandelt. Alle in Java geschriebenen Bestandteile des Groovy Frameworks benutzen den *InvokerHelper*, um dynamisches Verhalten zu ermöglichen.

Im Falle der Objekterzeugung wird die Methode *InvokerHelper.invokeConstructorOf* aufgerufen, um dann den Aufruf an die Metaklasse zu delegieren. Da zum Zeitpunkt der Instanz noch keine instanzspezifische Metaklasse existiert, wird für die Objekterzeugung stets die Metaklasse der Klasse benutzt. Dafür holt sich der *InvokerHelper* die entsprechende Metaklasse über die *MetaClassRegistry* und leitet den Aufruf an diese weiter. Die Metaklasse benutzt nun in *invokeConstructor* die java.reflect API, um entsprechend den Parametern ein neues Objekt zu erzeugen und ggf. die *Properties*zu setzen.

Die *MetaClassRegistry* verwaltet die Metaklassen für die Klassen und sorgt vor allem für die Erzeugung von neuen Metaklassen, falls einer Klasse noch keine Metaklasse zugewiesen war. Wie bereits in Listing 1.5 gezeigt, kann die Metaklasse mit *MetaClassRegistry.*setMetaClass geändert werden. In dem Fall wird sie in dem globalen Cache eingetragen. Danach liefert *MetaClassRegistry.getGlobalMetaClass* und damit *getMetaClass* nur noch die neue Metaklasse.

2.5 Methodenaufrufe

Analog zu der Erzeugung eines Objektes wird auch jeder einfache Methodenaufruf auf ein *GroovyObject* auf den *ScriptBytecodeAdapter* und damit den *InvokerHelper* umgeleitet.

```
static void main(args) {
      dog = new Dog()
      dog.bark()
}
```

```
    ScriptBytecodeAdapter.invokeMethodN(DogExample.class, dog,
        "bark", new Object[0])
```

Der Ablauf eines Methodenaufrufs im *InvokerHelper* lässt sich grob beschreiben mit:

1. Wenn der Empfänger eine Klasse ist, rufe *MetaClass.invokeStaticMethod* auf.
2. Sonst delegiere Aufruf an *InvokerHelper.invokePogoMethod* oder *InvokerHelper.invokePojoMethod* wenn es um ein *GroovyObject* oder um ein einfaches Java *Object* handelt (Plain Old Groovy/Java Object).
3. In *invokePojoMethod*: Hole die *MetaClass* der *Class* mit *MetaClassRegistry.*getMetaClass. Leite Aufruf an *MetaClass.invokeMethod* weiter.
4. In *invokePogoMethod*:
   (a) Wenn *GroovyObject* das *GroovyInterceptable* Interface implementiert rufe *GroovyObject.invokeMethod* auf.
   (b) Sonst: hole die *MetaClass* des *GroovyObject*. Leite Aufruf an *MetaClass.invokeMethod* weiter.

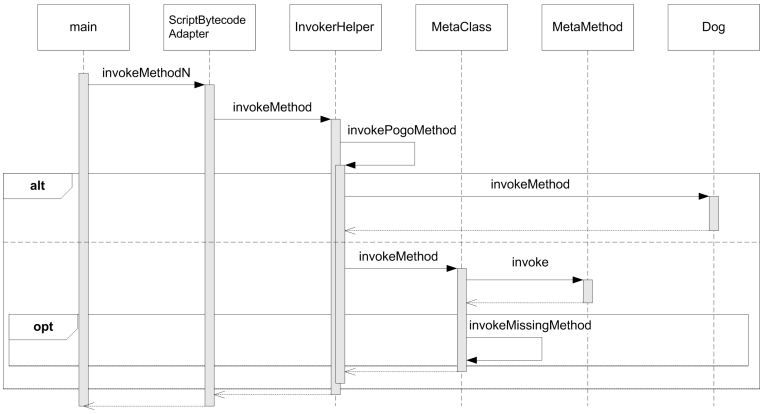

Abb. 5. Sequenzdiagramm für den Aufruf einer Instanzmethode für das *GroovyObject* dog

(c) Falls *MetaClass.invokeMethod* eine groovy.lang.MissingMethodException wirft, wird *GroovyObject.invokeMethod* als Backup aufgerufen. Diese ruft wiederum *MetaClass.invokeMethod* auf, wobei diesmal die Exception aber nicht mehr abgefangen wird.

5. Die Metaklasse behandelt entsprechend den Methodenaufruf in *invokeMethod* oder wirft in *invokeMissingMethod* eine groovy.lang.MissingMethodException.

Die *MetaClassImpl*verwaltet alle Methoden in einem *MetaMethodIndex*. Methoden sind Instanzen von *MetaMethod* und lassen sich grob in drei Kategorien unterteilen:

CachedMetaMethods kapseln Java Methoden (java.reflect.Method).

ClosureMetaMethod repräsentieren Closures, die man als anonyme, ungebundene Funktionen interpretieren kann. Mit der ClosureMetaMethod wird eine Closure konkret an eine Metaklasse gebunden und steht somit als neue Methode zur Verfügung. In Verbindung mit der *ExpandoMetaClass* wird so häufig verhaltensspezifische *Intercession* verwirklicht.

TransformMetaMethod wird benutzt, um die Parameter eines Aufrufes von einem Typ zu einem anderen zu übersetzen und diese Umwandlung performant zu cachen. Sie steht hier stellvertretend für dynamisch erzeugte Meta-Methoden.

In Tabelle 2 wurden unter anderem zwei wichtige Methoden der *MetaClass* zur *Introspection* vorgestellt:

getMethods() gibt alle öffentliche, in Java Bytecode übersetzte Methoden der Klasse und deren Oberklassen zurück, also alle *CachedMethod*s. Im Falle unseres Hundes sind das alle von Java *Object* geerbten und durch *groovyc* automatisch implementierten Methoden von *GroovyObject* sowie die *bark* und die überschriebene *toString* Methoden.

getMetaMethods() wiederum gibt nur die zusätzlich hinzugefügten Methoden zurück. Selbst bei unserem Hund sind da etliche automatisch hinzugefügt worden, die in *DefaultGroovyMethods* stehen. Neben den einleuchtenden Methoden wie *is* zur Identitätsprüfung, *isCase* für das erweiterte Switch-Statement sowie *println*, um von jedem Objekt auf die Kommandozeile ausgeben zu können, werden aber auch Methoden hinzugefügt die auf den ersten Blick wenig Sinn ergeben. So werden sämtliche von den Groovy Collections bekannten Methoden hinzugefügt, zum Beispiel *each* oder *grep* zum Iterieren über und zum Selektieren von Elementen. In dem Fall verhält sich das Objekt wie eine einelementige Liste und erlaubt so den einheitlichen Zugriff auf diese Methoden.

Obwohl das Sequenzdiagramms 5 für den Methodenaufruf schon recht komplex ist, so wurden doch einige wesentliche Aspekte übersichtshalber weggelassen. Neben dem Caching der einzelnen Methoden, finden zusätzlich noch eine Parameterumwandlung und eine Sonderbehandlung für Closures statt.

2.6 Method Intercepting

Im Folgenden wird nun gezeigt, wie man in die Mechanismen zum Methodenaufruf eingreifen kann. Die Beispiele beschränken sich auf eine einfache Logausgabe, können aber auch gegen komplexere Transformationen der Parameter und Rückgabetypen ausgetauscht werden. Es ist auch immer möglich Methodenaufrufe ganz zu verhindern oder aber auch Methoden zu simulieren. Ausgehend vom Sequenzdiagramms 5 für den Methodenaufruf ergeben sich grundsätzlich zwei Möglichkeiten in die Mechanismen einzugreifen.

2.6.1 Intercepting auf Klassenebene Implementiert eine Klasse das *GroovyInterceptable* Interface so werden sämtliche Aufrufe an *GroovyObject.invokeMethod* weitergeleitet. Dort kann dann klassenspezifisches *Intercession* stattfinden.

```
class InterceptingDog extends Dog implements
    GroovyInterceptable {
    Object invokeMethod(String name, Object args) {
        System.out.println "${this} is about to ${name}"
        metaClass.invokeMethod(this, name, args)
    }
}

dog = new InterceptingDog(name: 'Waldi')
dog.bark()
```

Listing 1.7. Interception mit *GroovyInterceptable*

In diesem Fall würde vor der Ausgabe von *bark* **"Waldi: bark"** noch die Ausgabe **"Waldi is about to bark"** erscheinen. Vorstellbar wäre aber auch instanzbasiertes *Intercession*, indem es ein Feld mit einer Closure gibt, welche

von Instanz zu Instanz variiert und entsprechend in der *invokeMethod* aufgerufen wird. Das wird vor allem in der *Expando* Klasse umgesetzt.

2.6.2 Intercepting auf Metaklassenebene Will man ein Verhalten nicht nur bei einer Klasse ändern, sondern über Klassengrenzen hinweg, so ist der eleganteste Weg das Verhalten auf Metaklassenebene zu verändern. Wie schon in Tabelle 2 aufgezeigt gibt es dafür zwei Methoden in *MetaClass*: *invokeMethod* und *invokeMissingMethod*.

```
import org.codehaus.groovy.runtime.metaclass.*

class DogMeta extends MetaClassImpl {
    DogMeta() {
        super(GroovySystem.getMetaClassRegistry(), Dog.class)
        initialize()
    }

    public Object invokeMethod(Class sender, Object object,
        String methodName, Object[] originalArguments, boolean
         isCallToSuper, boolean fromInsideClass) {
        System.out.println "$object is about to $methodName"
        super.invokeMethod(sender, object, methodName,
            originalArguments, isCallToSuper, fromInsideClass)
    }

    public Object invokeMissingMethod(Object instance, String
        methodName, Object[] arguments) {
        System.out.println "${instance} doesn't ${methodName}"
    }
}

metaDog = new Dog(name: 'Waldi')
//metaDog.metaClass = new DogMeta()
metaDog.bark()
metaDog.speak()
```

Listing 1.8. Interception mit *MetaClass*

Auch hier erscheint vor jedem Methodenaufruf eine Log Nachricht. Außerdem wird bei *speak* statt der groovy.lang.MissingMethodException nur die Nachricht ausgegeben: **“Waldi does not speak”**.

2.7 Properties in Groovy

Wie schon beim Vorstellen von *Dog* (siehe 1.6) angemerkt, werden die *Properties* von Groovy auf die JavaBean *Properties* durch den *groovyc* abgebildet. Bei unserem *Dog* gibt es damit neben einem privaten Feld *name* auch eine *setName* und *getName* Methode. Diese Methoden werden normalerweise durch *groovyc* generiert, können aber jederzeit überschrieben werden.

```
static void main(args) {
    dog = new Dog(name: 'Waldi')
    println dog.name
}
```

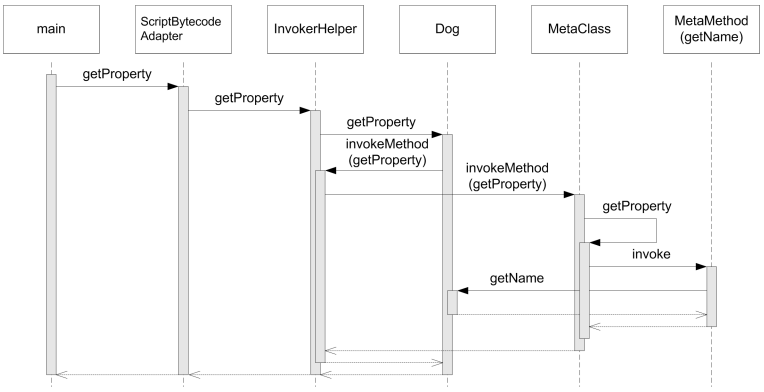

Abb. 6. Lesender Zugriff auf ein *Property*

Ein Zugriff auf ein *Property* wird zunächst vom *InvokerHelper* an das *GroovyObject* delegiert. Die vom *groovyc* generierte Methode ruft nun *MetaClass.getProperty* auf - überraschenderweise dynamisch über den *InvokerHelper*, was unperformant ist und nur bei der Metaklasse der Metaklasse ausgenutzt werden könnte. Die *MetaClassImpl* sucht nun das *MetaProperty* zu *name*, welches in diesem Fall ein *MetaBeanProperty* ist. Da für *name* der Getter *getName* beim Kompilieren erzeugt wurde, wird dieser dynamisch über den *InvokerHelper* aufgerufen. Sollte es keinen *Property* mit dem Namen geben, wird standardmäßig über *MetaClassImpl.invokeMissingProperty* eine groovy.lang.MissingPropertyException geworfen.

Alternativ zu einem *MetaBeanProperty* kann ein *Property* auch dynamisch erzeugt werden als *MetaExpandoProperty* oder ein einfaches Java Feld sein, das als *CachedField* repräsentiert wird.

Die einfachste Möglichkeiten *Properties* per *Intercession* zu simulieren, ist offensichtlich das Überschreiben von *getProperty* und *setProperty* des *GroovyObject* und als Backend eine einfache *HashMap* zu benutzen. Die Standardbibliothek wendet genau dieses Konzept auf *Expando* an, das wir in unserer Demo noch (siehe 2.11) anwenden werden.

Auf Metaklassenebene können natürlich auch die *getProperty* und *setProperty* Methoden neu implementiert werden, was vor allem die *ExpandoMetaClass* nutzt, um als *Properties* gesetzte Closures den Methoden der Klasse hinzuzufügen. Ebenso überschreibt die *ExpandoMetaClass* die *invokeMissingProperty*, um dynamisch erzeugte *Properties* der Basisklassen zu finden.

Schließlich bleibt noch die theoretische Variante auch *MetaClass.invokeMethod* zu überschreiben, um die Methodenaufrufe an sich abzufangen. Da aber schon die oben genannten Möglichkeiten genügend Spielraum für *Intercession* mit *Properties* lassen, sollte man sich auch auf diese beschränken.

Weiterhin muss natürlich hinter eine *Property* kein reales Feld existieren. Eine JavaBean *Property* kann auch berechnet sein. Dafür stellt man entsprechend nur die *get* und/oder *set* Methode bereit. Da auf berechnete *Properties* aber auch genauso zugegriffen werden kann wie auf feldbasierte, entspricht der *Property* Mechanismus in Groovy dem *Uniform access principle*.

```
class OldDog extends Dog {
    def birthDay = new Date(0)

    def getAge() {
        new Date().year - birthDay.year
    }

    static void main(args) {
        def waldi = new OldDog(name: 'Waldi')
        println waldi.name
        println waldi.age
    }
}
```

Listing 1.9. *Uniform access principle* in Groovy

2.8 Dynamische Skripte

```
static void main(args) {
      shell = new GroovyShell()
      shell.evaluate("1+1")
}
```

Bei der dynamischen Kompilierung eines Skriptes wird mittels *GroovyShell.evaluate* ein eindeutiger Skriptname generiert und anschließend der Quelltext mit dem *GroovyClassLoader* zu einem AST in der *CompilationUnit* geparsed. Der Skriptname ist ein Pseudoidentifier, der auf eine Datei verweist, die zwar nicht existiert, aber trotzdem den Java Sicherheitsrichtlinien genügt. Dann kann der AST in Java Bytecode mit *compile* umgewandelt und in den aktuellen Kontext mit *InvokerHelper.createScript* geladen werden. Das daraus erzeugte *Script* kann anschließend ausgeführt werden.

2.9 Kategorien

Mit Kategorien bietet Groovy eine Art dynamische Mix-ins. Es können innerhalb von einem Closure, beliebige Methoden zu allen Metaklassen hinzugefügt oder überschrieben werden.

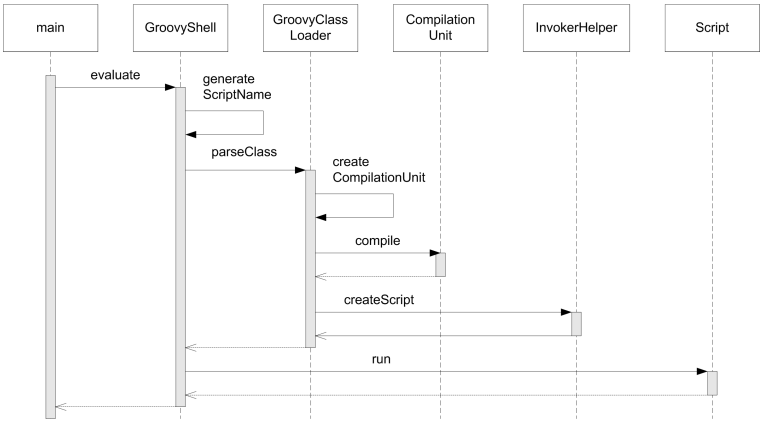

Abb. 7. Dynamisches Ausführen von Skripten

```
class DogRenamer {
    static String getName(Dog self) {
        "Bruno"
    }
}

waldi = new Dog(name: 'Waldi')
use(DogRenamer) {
    assert waldi.name == "Bruno"
}
assert waldi.name == "Waldi"
```

Listing 1.10. *Category* überschreibt *getName*

Die Methode *use* ist eine der Standardmethoden in *DefaultGroovyMethods* die jeder Metaklasse, die von *MetaClassImpl* erbt, automatisch hinzugefügt wird. Deswegen wird häufig von *use* auch von einem Sprachkonstrukt statt von einer Methode gesprochen. Mit *use* wird die in Klammern angegebene Klasse auf Klassenmethoden untersucht und in *org.codehaus.groovy.runtime.GroovyCategorySupport* verwaltet.

Es werden nur Klassenmethoden erlaubt, um Zustände in der Kategorieinstanz zu vermeiden, die nicht threadsicher wären. Der *this* Parameter wird deshalb bei Überschreiben von Instanzmethoden wie *getName* explizit als ersten Parameter angegeben. Wird nun eine Methode aufgerufen, überprüft *MetaClassImpl* in *invokeMethod* ob es eine Kategorie Methode gibt, die kompatibel ist mit dem aktuellen Objekt und den übergebenen Parametern. Gibt es solch eine wird der Aufruf delegiert, sonst wird wie in 2.5 beschrieben der normale Aufruf fortgesetzt.

Kategorien sind von Objective-C entlehnt und bieten eine einfache Möglichkeit, kurzzeitig und ohne Seiteneffekte neue Methoden hinzuzufügen oder bestehende zu überschreiben. Sie eignen sich deswegen gut für Aspekt- oder Contextorientierte Programmierung.

2.10 Performance

Als Programmierer muss man ständig ein Auge auf der Performance haben - im weitesten Sinne. Prinzipiell kann man eine Sprache daran messen, wie schnell man mit ihr bestimmte Algorithmen realisieren kann, wie schnell diese dann ausgeführt werden und wieviel Zeit die Wartung dieses Code in Anspruch nimmt.

Bei Java kann man als Entwickler mit den vorhandenen exzellenten IDEs schnell ein Programm schreiben. Es kommen zwar weit mehr Zeilen Code am Ende heraus als bei jeder Skriptsprache, dafür werden aber viele Programmfehler schon beim Schreiben erkannt. Die Ausführungsgeschwindigkeit wurde in Java in den vergangenen Jahren soweit optimiert, dass sie selbst C++ Konkurrenz macht.

Allerdings ist die Wartung des Codes nicht immer leicht, gerade durch die vielen automatisch generierten Methoden. Im Vergleich zu Java ist Groovy in der Ausführung wesentlich langsamer, erspart aber sowohl bei der Erstprogrammierung als auch bei der Wartung einiges an Zeit. Da letztere aber subjektiv sind und direkt vom Programmierer abhängen, geben wir nun einige objektiv messbare Werte zur Laufzeit.

for Schleife In einer for-Schleife wird die Zählvariable vom Typ *Long* von 0 bis 10^9 in Java und 10^7 in Groovy erhöht. In Groovy braucht pro Erhöhung etwa 3 mal so lange wie Java. Benutzt man aber in Java den wohl gebräuchlicheren primitiven Typ *long* so ergibt sich der Faktor 30.

Methodenaufruf Um einen einzelnen Aufruf einer leeren Methode messen zu können, wurde dieser Aufruf in einer for Schleife wiederholt und anschließend die Zeit für eine leere Schleife abgezogen. Groovy braucht 4000 mal so lang um eine leere Instanzmethode aufzurufen, also etwa 7 Mikrosekunden.

Objekterzeugung Gemessen wurde, wie lange es dauert 10^7 *Dog* Objekte zu erzeugen. Dabei war Groovy nur 10% langsamer, da offenbar die Objekterzeugung in Java selbst mit 1 Mikrosekunde recht lange dauert.

Rekursion Es wurde die 30. Fibonacci Zahl ausgerechnet, wobei Groovy 300 mal so lange brauchte.

Sortieren Sortieren einer Liste mit 10^7 zufälligen Zahlen dauert gleich lange, wenn man in Groovy ebenfalls auf die Java Methode zurückgreift.

Offensichtlich ist Groovy nicht zum Numbercrunchen geschaffen, was schon alleine wegen den fehlenden primitiven Typen nicht empfehlenswert ist. Erschreckend ist der Unterschied bei den Methodenaufrufen, wobei man fairerweise sagen muss, dass ein reflektiver Methodenaufruf in Java auch nur 4 mal schneller als der in Groovy ist. Außerdem wirkt das Verhältnis deshalb so krass, weil es eine leere Methode ist. Ruft man entsprechend von Groovy Java Methoden

wie *Collections.sort* auf, so gibt es keinen Unterschied mehr, wenn die Funktion längere Berechnungen durchführt. Rekursive Algorithmen sollte man jedenfalls in Groovy tunlichst meiden und lieber in eine Java Klasse auslagern. Überhaupt empfiehlt es sich bei performancekritischen Algorithmen stets auf Java auszuweichen und Groovys Flexibilität eher für die Verknüpfung der verschiedenen Java Klassen zu benutzen, um Anwendungslogik zu realisieren.

2.11 Bewertung

Die Entwickler von Groovy hatten einige Hürden zu überwinden, die ihnen die Java Runtime Environement in den Weg gestellt hat. Deshalb ist mit dem Metaklassenframework auch erheblicher Mehraufwand bei der Verarbeitung von Objektinstanziierungen, Methodenaufrufe und Zugriff auf die *Properties* zu verzeichnen. Dynamische Programmierung mit *Reflection* wäre ohne diese Metaklassenebene allerdings nicht in dieser Qualität möglich gewesen.

Die Entscheidung zu instanzbasierte Metaklassen bringt zwar auch einen gewissen Overhead mit, gibt dem Groovy Programmierer aber wiederum zusätzliche Möglichkeiten. So ist es möglich zusätzliche Funktionen in einer *ExpandoMetaClass* zu definieren und diese dann nur ausgewählten Instanzen zur Verfügung zu stellen, wobei bestehende Funktionen konzeptionell überschrieben werden. Interessant ist auch die unterschiedliche Behandlung von Methodenaufrufe und Zugriffe auf *Properties*. Hier wurde wahrscheinlich verstärkt darauf geachtet, auf welcher Ebene man das dynamische Verhalten eher braucht - bei *Property* ist es eher Instanzebene, während Methoden doch eher auf Klassenebene verändert werden.

Letztendlich stehen durch die Vielzahl der Indirektionen dem Programmierer alle Wege offen, in das dynamische Verhalten einzugreifen. Der Preis dafür ist allerdings ein teilweise erheblicher Overhead, der auch in der Performance sichtbar wird. Die Schwäche von Groovy ist aber zugleich auch dessen Stärke: kritische Algorithmen können einfach in Java Code ausgelagert werden und dennoch bequem von Groovy aufgerufen werden. Einzig auf die Groovy Magie muss man dann im Java Code verzichten.

Kleinere und größere Veränderungen in dem Metaklassenframework sind aber immer noch zu erwarten. So stießen wir bei unseren Versuchen mit Groovy 1.5.1 auf zwei Bugs, von denen einer sogar bereits in Groovy 1.5.2 gefixet wurde. Deshalb sind alle Angaben, die einen tiefen Einblick in das System erlauben, auch mit einer gewissen Vorsicht zu genießen. Es sind aber auch zusätzliche Konzepte zu erwarten, denn während *MetaMethod* und *MetaProperty* gibt, sucht man in der aktuellen Version von Groovy noch vergeblich nach *MetaConstructor*. Dieser ist laut Bug Tracker für zukünftige Versionen allerdings schon in Planung, um dann die drei Aspekte Erzeugung, Zustand und Verhalten eines Objektes per *Reflection* modifizieren zu können.

3 Eine praktische Anwendung

Viele Aspekte der Meta-Programmierung wurden ausführlich betrachtet. Um den Stand des Frameworks wirklich beurteilen zu können, sollte ein Test in der Praxis erfolgen. Groovy setzt sich zum Ziel, dass Programmierer schneller zu einer Lösung kommen als mit Java. Was liegt näher, als das auf den Datenbankzugriff auszudehnen? Dazu wird ein kleiner Objekt-relationaler Mapper in Groovy implementiert, welcher aus Zeilen einer Datenbank passende Objekte erzeugt. Allerdings sollte im Gegensatz zu gängigen O/R-Frameworks wie Hibernate und Java Persistence auf die Definition von Klassen oder XML-Strukturen verzichtet werden.

Als Lösungsidee kann das dynamische Hinzufügen von Properties oder Methoden verwendet werden. Dabei wird für jede Tabellenzeile ein Objekt erzeugt, welches die Spalten als Properties bzw. Methode während der Laufzeit hinzugefügt bekommt. Da der Nutzer Anfragen an verschiedene Tabellen stellen kann, ist die Struktur der Objekte immer nur für eine Tabellenabfrage identisch. Es kann daher nicht eine Metaklasseninstanz per Klasse verwendet werden, da sonst auch alte schon ausgelesene Objekte nachträglich neue Properties bzw. Methoden bekommen würden. Als Lösung würde sich eine Metaklasseninstanz pro Tabellenanfrage empfehlen.

Versuche auf Basis von *ExpandoMetaClass* schlugen fehl, da die API der *ExpandoMetaClass* es nicht explizit vorsieht, dass der Anwendungsprogrammierer selbst Instanzen von dieser erzeugen möchten. Im Standardfall würde die Instanziierung durch die Groovy Umgebung bei Bedarf automatisch geschehen geschehen.

Darauffolgend bestehen zwei Möglichkeiten in Groovy. Eine eigene Implementierung des Verhaltens von *ExpandoMetaClass* oder den Eingriff in den *Property* Zugriff durch Überschreiben der Methoden *getProperty* und *setProperty* auf Klassenebene (siehe 2.7). Der Entschluss fiel auf die zweite Variante. Das Implementieren einer Klasse, welche die Methoden zum setzen bzw. lesen der Properties verändert, kann genutzt werden, um die Properties selbst zu verwalten. In der Benutzung würde das wie in folgendem Beispielcode aussehen.

```
def rowObject = new DatabaseObject();
// Bestimmung des Spaltennames und Inhalt in aktueller Zeile
rowObject."$columnName" = result
```

Ein O/R-Mapper ohne Speichermöglichkeit ist in der Realität nicht sinnvoll. Dazu kann zusätzlich ein Closure verwendet werden, welches bei *setProperty* aufgerufen wird, wenn sich ein *Property* des Objekts verändert. Dieses Closure wird dem Objekt nach Instanziierung übergeben und enthält den Tabellennamen sowie den Zeilenidentifier.

Anhand des O/R Mappers konnten die Möglichkeiten durch Meta-Programmierung am praktischen Beispiel durchgeführt werden. Die Grundfunktionalität wurde implementiert. Als Erweiterungen sind viele Dinge denkbar, wie automatischer ForeignKey-Verfolgung mit z.B. row.foreignTable.foreignValue oder aber das automatische ALTER TABLE, wenn neue Properties hinzugefügt oder ge-

löscht werden. Auch die Performance wurde in der bisherigen Implementierung nicht betrachtet.

Es lässt sich zusammenfassen, dass obwohl die Meta-Struktur in Groovy noch nicht perfekt ist, selbst mit wenigen Handgriffen dieses Problem umschifft werden konnte.

Literatur

1. *Groovy In Action.* Manning Publications Co., 2007.
2. Felix Geller, Regina Hebig, and Robert Warschofsky. *Reflex.* Hasso Plattner Institute, 2008.
3. Stefan Hüttenrauch and Robert Schuppenies. *Reflection with Javassist: Architecture and Design Decisions.* Hasso Plattner Institute, 2008.
4. Victor Saar, Daniel Rinser, and Mircea Crăciuleac. *Concepts and Practice of Reflection in Java.* Hasso Plattner Institute, 2008.