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

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

Vorgelegt von: Nils Petersohn

am: 8.11.2011.

Inhaltsverzeichnis

1 Einleitung

Die Sektionen würden bei der echten Arbeit wegfallen.

1.1 Motivation (Heranführen an das Thema)

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

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

1.2 Begriffserklärung (falls notwendig)

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

1.3 Aufgabenstellung (kurz, knapp, präzise) und Erwartungen

Anhand mehrerer Problemfälle soll das Paradigma der sprachorientierten Programmierung und die damit verbundenen Konzepte der domänenorientierten Programmierung analysiert und geprüft werden. Mehrere vergleichsweise einfache DSLs sollen mit Hilfe von Groovy-

Metaprogramming als interne DSLs bzw. mit dem Meta-Programming-System (MPS) sollen überschaubare externe DSLs entworfen werden. Domänenexperten, die keine Erfahrung mit Programmierung haben, sollen an interne bzw. externe DSLs für deren bekannte Domäne herangeführt werden, um diese anschließend nach Lesbarkeit, intuitivem Verständnis und Flexibilität zu bewerten. Dabei bekommen die Probanden mehrere Aufgaben, die sie mit den gegebenen DSLs lösen sollen. Die Erwartungen sind, dass bei kleinen Systemen, der Aufwand zu hoch ist, extra eine DSL einzusetzten. Bei komplexen Systemen stellt eine geeignete DSL jedoch einen unmittelbaren Mehrwert dar.

1.4 Gliederung

Zuerst werden theoretische Grundlagen zum Thema “Sprachorientierte Programmierung” und Metamodellierung erläutert. Der praktische Teil beginnt mit der Vorstellung der Problemdomänen und deren verschiedenen Anforderungen. Dann werden die zu implementierenden DSLs konzeptionell vorgestellt. Die Umsetzung dieser Konzepte mit den Toolsets wird im nächsten Kapitel beschrieben. Weiterhin soll die Testumgebung mit den Probanden spezifiziert werden, um danach die Durchführung und die Ergebnisse auszuwerten und zu beschreiben.

1.5 Abgrenzung

Der Fokus dieser Arbeit soll auf die textuelle und nicht auf die graphische Repräsentation von DSLs abzielen. “Natural language processing” soll nur oberflächlich betrachtet werden. Domänenspezifische Modellierung mittels UML-Profilen soll nur erwähnt werden. Als Toolset sollen ausschließlich Groovy-Meta-Programming und MPS (Meta-Programming-System von Jetbrains) verwendet werden.

2 Theorie

2.1 Domain Specific Languages

Software wird mit Hilfe von universell einsetzbaren Programmiersprachen (engl. general purpose language, GPL), wie z.B. Java, C# entwickelt. Diese Sprachen sind, für nahezu jeden Anwendungszweck einsetzbar. Dadurch werden diese Sprachen komplex und deren Benutzung ist nur durch gut ausgebildete Programmierer möglich.

Sachverhalte, Objekte und Prozesse aus der realen Welt werden durch diese Pro- grammierer mit Hilfe einer Kombination der universell einsetzbaren Konstrukte aus der Programmiersprache nachgebildet.

Zur Umsetzung eines Programms werden genaue Spezifikationen geliefert.

Durch testgetrieben Entwicklung und abschliessende Akzeptanztest werden die Spezifikationen verifiziert. Diese Art der Softwareentwicklung führt in der Praxis zu verschiedenen Proble- men. Es entstehen hohe Aufwände für Spezifikation und Test. Trotzdem ist die entwickelte Software häufig fehleranfällig und entspricht oft nicht genau den Spezifikationen. Nachbesserungen sind nötig, die Geld und Zeit kosten.

Domänenspezifische Sprachen können in bestimmten Situationen helfen, um diesen Problemen zu begegnen. Die grundsätzli- che Idee ist, ausgewählte Softwareteile nicht mehr mit universell einsetzbaren Programmiersprachen zu entwickeln, son- dern stattdessen Sprachen zu benutzen, die auf die konkrete Anwendungsdomäne spezialisiert sind. Der Quelltext, der mit einer solchen Sprache entwickelt wird, kann später vollautomatisch in den Quellcode einer universellen Programmierspra- che übersetzt werden. Der Vorteil ist, dass der Sprachumfang der DSL aufgrund der Spezialisierung auf eine Domäne im Vergleich mit einer universellen Sprache viel kleiner ist. Um domänenspezifische Sachverhalte als Quellcode auszudrücken ist deutlich weniger Zeit und Quellcode nötig, zur Programmierung reicht oft das Domänenwissen des Anwendungsexperten aus. Im Extremfall könnte statt dem Programmierer der Anwendungsexperte das benötigte Programm schreiben. Die Erstellung des DSS-Quellcodes erfolgt mittels eines speziellen Editors, der die Sprachelemente als

Textbefehle oder durch grafische Elemente bereitstellt. Ein solcher Editor kann so definiert werden, dass er, entsprechend den Vorgaben der Anwendungsdomäne, nur vorher festgelegte Kombinationen von Sprachelemen- ten zulässt und damit der Sicherstellung fachlicher Rahmenbedingungen besonders Rechnung trägt.[IR07]

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

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

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

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

2.1.1 Unterscheidungen

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

“Sowohl die konkrete Syntax als auch die Semantik können frei definiert werden. SQL oder

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

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

2.2 Sprachorientierte Programmierung

"Language oriented programming is about describing a system through multiple DSLs".(vlg. [Fow05])

Sprachorientierte Programmierung ist nach Fowlers Beschreibung alsöein gesamtsystem bestehend aus subsystemen, deren Funktionen mit definierten Sprachmodellen beschrieben werden. Bestenfalls sollten diese Sprachmodelle die fachlichen Probleme auf eine natürliche Art beschreiben können ohne dabei mehr zu beschreiben als das Fachgebiet benötigt.

2.3 Groovy

Groovy ist eine dynamisch typisierte Programmiersprache und Skriptsprache für die Java Virtual Machine. Groovy besitzt einige Fähigkeiten, die in Java nicht vorhanden sind: Closures, native Syntax für Maps, Listen und Reguläre Ausdrücke, ein einfaches Templatesystem, mit dem HTML- und SQL-Code erzeugt werden kann, eine XQuery-ähnliche Syntax zum Ablaufen von Objektbäumen, Operatorüberladung und eine native Darstellung für BigDecimal und BigInteger. (vgl. [unn])

2.4 Metaprogrammierung

Metaprogrammierung ist eine Programmiertechnik, die Codegenerierung einsetzt, um bessere Abstraktion zu ermöglichen. Ein Evaluierer bestimmt den Wert eines formalen Ausdrucks. Z.B. ist der Wert des formalen Ausdrucks "5 + 3" "8". Für Metaprogrammierung ist es oft nötig formale Ausdrücke zur Lauf- zeit auswerten zu können. Programmiersprachen wie Ruby oder Lisp stellen hierfür einen Eva- luierer über eine eval-Funktion bereit. Linguistische Abstraktion bezeichnet Abstraktion auf linguistischem Sprachniveau. Dabei be- zeichnet

hier der Begriff “Sprache” primär formale Sprachen. Metalinguistische Abstraktion ist Abstraktion auf (linguistischem) Sprachniveau, die den Evalu- ierer umschreibt oder einen eigenen Evaluierer verwendet. Ein Beispiel ist ein lazy eval für eine strikt ausgewertete Sprache wie etwa Java. Der Begriff der Metalinguistischen Abstraktion ist nicht klar definiert und eine harte Abgrenzung zu anderen Konzepten (etwa Frameworks) vorzu- nehmen ist kaum möglich. Metaprogrammierung kann man als Werkzeug verstehen, das linguistische Abstraktion erzeugt.(vgl. [Bie08])

Metaprogrammierung bezeichnet eine Programmier-Technik um Code automatisch zu generieren. Es geht also um Code der Code schreibt. Der Ursprung der Metaprogrammierung geht auf das 1958 am MIT entwickelte Lisp bzw. Sche- me zurück. In Lisp gibt es (defmacro ..) und in Scheme (define-syntax ..) Makros. Ein Lisp-Makro ist einer Funktion ähnlich. Eine Funktion erhält Parameter und liefert einen oder mehere Werte zurück. Ein Lisp-Makro erhält Parameter und liefert einen oder mehere Code- Ausdrücke zurück, die wieder evaluiert werden. Vor der Entwicklung der Lisp-Makros gab es bereits selbstmodifizierenden Assembler-Code. Das Problem hiermit ist das zu niedrige Abstaktionsniveau, da man sich auf Opcode-Ebene mit der Manipulation des Codes beschäftigen muss. In Ruby gibt es die Methoden define method und define class, mit der man Methoden bzw. Klassen definieren kann. Um Metaprogrammierung betreiben zu können, ist es essentiel, dass man solche Funktionen hat, damit man dynamisch Methoden und Klassen erzeugen kann. Des Weiteren stellt Ruby eval Funktionen zur Verfügung, mit denen Code in unterschiedlichen Kontexten ausgeführt werden kann. InRubywirdkomplexereMetaprogrammierungüberInterpreter-Hooksrealisert,etwamethodmissing. method missing wird augerufen, wenn man eine nicht vorhandene Methode auf einem Objekt aufruft.DieDefault-ImplemtierungwirfteineNoMethodErrorException.Durc dieser Method kann man z.B. ein Dateisystem Objekt erzeugen, dass als Methodenaufrufe seine Unterordner kennt. Dies ermöglicht es z.B. ein Verhalten, analog zu dem Shell-Befehl cd dirname, über fsobj dirname zu realisieren. Ein anderer Interpreter-Hook ist Class.inherited, der immer ausgeführt wird, wenn man von einer Klasse erbt. Möchte man verhindern, dass von einer Klasse geerbt wird, kann man in Class.inherited eine Exception werfen.

Oft wird Metaprogrammierung als eine Form der Codekomprimierung verstanden. Es geht bei Metaprogrammierung nicht um das reine Einsparen von Zeichen bzw. Code-Zeilen, sondern um Abstraktion.

(vgl. [Bie08])

Introspection bezeichnet die Fähigkeit, auf Informationen über die Zustäe 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.[LB]

2.5 probleme und loesungsansaetze

Ich beschreibe nun einige Probleme mit Metaprogrammierung und DSLs. Es geht mir hier nicht um Vollständigkeit, sondern darum einige Probleme und potenzielle Lösungen auf zu zeigen. Eins der wichtigsten Probleme ist die hohe Komplexität. In [Diomidis Spinellis. Rational metaprogramming. IEEE Software, 25(1):78–79, January/Fe- bruary 2008.] beschreibt der Autor das Pro- blem wie folgt: “While I admire the cleverness and skill that hides behind C++ libraries (...), the fact remains that writing advanced template code is devilishly hard, (...)” Die Komplexität lässt sich durch die Verwendung bekannter Metaprog. Paradigmen und Patterns reduzieren. Lisp hat hier etliche zu bieten, etwa defmacro, Higher-Order Functions oder Curry- ing. Eine DSL bzw. ein Metaprogammierungsframework sollte im mathematischen Sinne abgeschlos- sen sein. D.h. man soll den selben Code erzeugen können, den man von Hand schreiben kann und umgekehrt. Komplexe DSLs erzeugen teilweise schwer debugbaren Code. Nach Wissensstand des Autors gibt es zur Zeit kaum Lösungsansätze für dieses Problem. Es bleibt nur anzumerken, dass auch komplexe Frameworks teilweise schwer debugbaren Code erzeugen. Eine häufige Quelle für schwer debugbaren Code ist es, den Code als String darzustellen und dannzuevaluieren.Diesf ührt oft zu sinnfreien Fehlermeldungen wie “Syntax error in line 1 at char 42”, wobei der evaluierte Code an anderer Stelle steht. Ruby und viele andere Sprachen bieten Konstrukte an, die es nicht zulassen, dass man syntaktisch falschen Code er- zeugt. In Ruby kann man hierfür Blöcke verwenden. Ein anderes Problem ist die Sprachkonsistens. Es ist schwierig gute und konsistente Sprachen zu erzeugen. Es ist aufwändig diese Sprachen zu lernen und ihr Support ist ressourcen-intensiv. Es macht daher Sinn die neue Sprache möglichst gut in die vorhandene Sprachumgebung ein zu gliedern. Eingebettete DSLs helfen hierbei. (vgl. [Bie08])

2.6 Metaprogrammierung in Groovy

Groovy besteht aus einem flexiblen Metaklassenmodell. Im Sinne einer Open Implementation hat der Programmierer nahezu alle Möglich- keiten sein Programm mittels Reflection zur Laufzeit zu verändern und damit an die individuellen Bedürfnisse anzupassen. [LB] 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 Mölichkeit zur Introspection bereit. [...] Erst Javassist, und Reflex erlauben echtes dynamisches Verhalten, in dem auf Byteco- de Ebene die Klasse verändert wird.

Dazu muss allerdings die Klasse teilweise umständlich entladen und neugeladen werden. 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 pri- mitiven Datentypen und Klassen zu vermeiden. Weiterhin ist jede Klasse in Groovy eine Instanz von Class, womit Class alsöauch in Groovy die Klasse der Klassen bleibt. [LB]

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 instanz- spezifische 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.

invokeMethod Das Verhalten eines Objektes ist wiederum eine Ebene höher angesiedelt, spielt sich alsöauf Klassen- oder Metaklassenebene ab. Normalerweise wird dynamisches Verhal- ten damit auf Metaklassenebene realisiert, sodass diese Methode der GroovyObjects gar nicht erst aufgerufen wird. Nur im Fehlerfall oder mit Hilfe des Tag-Interfaces GroovyInterceptable wird invokeMethod aufgerufen (in 2.5 genau beschrieben).

[...] Während GroovyObject dynamisches Verhalten auf Objekt- und Klassenebene erlaubt, ist das zweite elementare Interface MetaClass die Grund- lage für das sehr ausgewogene Metaklassenmodell in Groovy.

Metaklassen, Klassen und Instanzen Läuft ein Programm ohne Intercession, sögelten für die Metaklassen der Klassen und Instanzen folgen- de Grundaussagen: Jede Klasse hat eine Metaklasse, die eine Instanz von Me- taClassImpl 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 Meta- klasse, 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.

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 auto- matisch 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 GroovyObjects 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 auf- gerufen. Die Aufrufreihenfolge ist im Vergleich zu invokeMethod alsögenau anders- herum. Auch diese Methoden sind für Intercession geeignet.

invokeMissingMethod invokeMissingProperty Diese Backupmethoden werden jeweils aufgerufen, wenn die normalen Aufrufmecha- nismen fehlgeschlagen sind. Intercession mit diesen Methoden erlaubt zum Beispiel die Erweiterung von Klassen und Objekten um zusätzliche Properties zur Laufzeit. [LB]

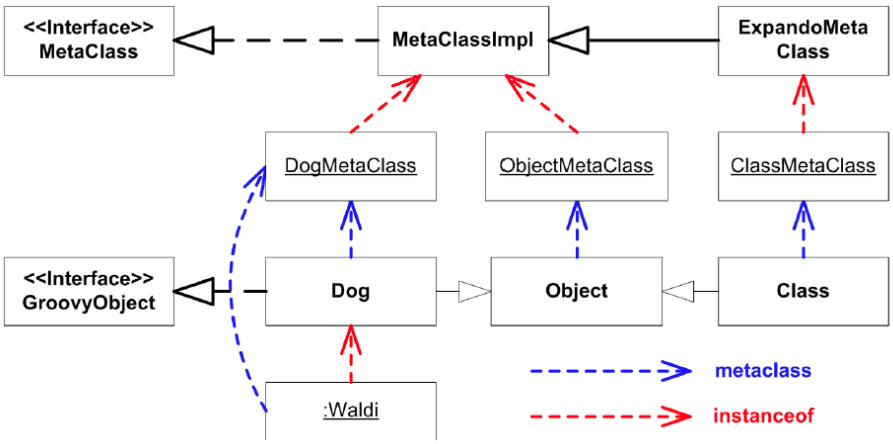

Abbildung 2.1: Beziehung von Metaklassen, Klassen und Instanzen [LB])

2.6.1 Closures

2.6.2 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. 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 Klas- senmethoden untersucht und in org.codehaus.groovy.runtime.GroovyCategory- Support verwaltet. Es werden nur Klassenmethoden erlaubt, um Zustände in der Kategoriein- stanz zu vermeiden, die nicht threadsicher wären. Der this Parameter wird des- halb bei Üerschreiben von Instanzmethoden wie getName explizit als ersten Parameter angegeben. Wird nun eine Methode aufgerufen, überprüft MetaClas- sImpl 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 Mög- lichkeit, kurzzeitig und ohne Seiteneffekte neue Methoden hinzuzufügen oder bestehende zu überschreiben. Sie eignen sich deswegen gut für Aspekt- oder Contextorientierte Programmierung. [LB]

2.6.3 Expando-MetaClass

3 MDA / MDSD und Metaprogrammierung

Model-driven Architecture (MDA) bzw. Model-Driven Software Development (MDSD) und Me- taprogrammierung bzw. DSLs haben eine vergleichbare Problemstellung. In der MDA Welt ver- wendet man auf UML etc. basierende graphische Modele. Auch das graphische Model muss eine formale Sprache sein, damit es compilerbar ist. Eine DSL kann man als textuelle Repräsentation eines Models verstehen. Die Komplexität und das Abstraktionsniveau ist abhängig von dem verwendeten Model, nicht seiner Repräsentation. Graphische Repräsentation kann man aber besser mit zusätzlichen Infor- mationen anreichern, da man diese nach Bedarf ein- und ausblenden kann. Eine DSL hat den Vorteil, das ihre Darstellung simpler ist, man kann sie beispielsweise mit einem einfachen Text Editor bearbeiten oder mit trivialen Mitteln ein diff zweier Versionen erstellen (vgl. Sachez Cuadrado and Jesu .... Building domain-specific languages for model-driven development. IEEE Softw., 24(5):48–55, 2007. & Diomidis Spinellis. Rational metaprogramming. IEEE Software, 25(1):78–79, January/February 2008.). (vgl. [Bie08])

4 Praktischer Teil

4.1 Die Fachliche Domäne

5 Auswertung

6 Zusammenfassung und Schlussbetrachtung

Literaturverzeichnis

[Bie08] Bieker, F.: Metaprogrammierung und linguistische Abstraktion. (16. Dezember 2008). – http://www.inf.fu-berlin.de/inst/ag-se/teaching/S-BSE/125$_{B}iekerF_{M}etaProg.pdf$

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

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

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

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

[IR07] IT-RADAR, LPZ E-BUSINESS |.: Domnenspezifische Sprachen. (2007). http://it-radar.org/uploads/reports/2_2007.pdf

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

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

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

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

[unn] http://de.wikipedia.org/wiki/Groovy

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

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