Seminar „Beiträge zum Software Engineering“

Winterterm 2008

Metaprogrammierung und linguistische Abstraktion

Fabian Bieker

Advisor: http://www.inf.fu-berlin.de/~oezbek/Christopher Oezbek

16. Dezember 2008 - 12:36

Abstrakt

Metaprogrammierung, linguistische Abstraktion und domänen-spezifische Sprachen sind Konzepte, die es einem Softwareentwickler ermöglichen zusätzliche Abstraktionschichten zu erzeugen, um komplexe Problemstellungen beherrschbarer machen.
Dieser Artikel betracht diese Konzepte und ihre Umsetzung in der Programmiersprache Ruby und zeigt einige Anwendungsbeispiele. Auf typische Probleme und den Einsatz im Unternehmensumfeld wird ebenfalls eingegangen. Da Java im Unternehmensumfeld eine starke Verbreitung hat, geht der Atikel kurz auf die Kombination von Ruby und Java ein. Abschließend beleuchtet er die Gemeinsamkeiten von Model-driven-Architecture und Metaprogrammierung.

Inhaltsverzeichnis

1 Einführung

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 Laufzeit auswerten zu können. Programmiersprachen wie Ruby oder Lisp stellen hierfür einen Evaluierer über eine `eval`-Funktion bereit.

Linguistische Abstraktion bezeichnet Abstraktion auf linguistischem Sprachniveau. Dabei bezeichnet hier der Begriff "Sprache" primär formale Sprachen.
Metalinguistische Abstraktion ist Abstraktion auf (linguistischem) Sprachniveau, die den Evaluierer 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) vorzunehmen ist kaum möglich.

Metaprogrammierung kann man als Werkzeug verstehen, das linguistische Abstraktion erzeugt.

Die klassische Sprache für Metaprogrammierung ist Lisp oder der Lisp-Dialekt Scheme. Dieser Artikel diskutiert Metaprogrammierung am Beispiel von Ruby, da es im vergleich zu Lisp leichter verständlich ist, wenn man schon Java oder Perl kennt. Ruby bietet einen großen Umfang an Konstrukten zur Metaprogrammierung, die sich sehr elegant in das OOP-Paradigma von Ruby einfügen.

Ich betrachten verschiedene Beispiele für Metaprogrammierung und metalinguistische Abstraktion in Ruby. Die Beispiele wurden so gewählt, dass ein möglichst grosses Spektrum Metaprogrammierungstechniken und Anwendungsgebieten abgedeckt wird. Die betrachteten Anwendungsgebiete sind u.a. funktionale Progammierung, Nebenläufigkeit und UML OCL.

1.1 Konvention

Code, Methodennamen etc. werden wie folgt dargestellt: `foo.bar()` .

1.2 Abstraktion

Abstraktion ist essentiell für gute Softwareentwicklung. Durch das Kapseln von Details und domänen-spezifische Eigenheiten in abstrakte Einheiten, werden komplexe Systeme beherrschbar. Das Buch "The Pragrammtic Programmer" [12] liefert u.a. folgende Gründe für Abstraktion:

- "Tip 53: Abstractions Live Longer than Details"
- "Tip 38: Put Abstraction in Code, Details in Metadata"

Mit der Zeit wurden in der Softwaretechnik immer neue Mittel entwickelt um Abstraktion zu erzeugen. Wichtige Entwicklungen sind u.a.:

- Abstraktion durch Assembler (statt Maschinencode / Lochkarten)
- Abstraktion durch Funktionen / Prozeduren
- Abstraktion durch Module und APIs
- Abstraktion durch Objekt-Orientierung und Komponenten-Frameworks
- Abstraktion durch Sprache (linguistische Abstraktion)

Ein Beispiel für linguistische Abstraktion ist den Satz "Das Ding mit einer CPU, vielen Schaltkreisen, einem Lüfter usw." durch "Computer" zu ersetzen.

Linguistische Abstraktion sollte nicht als Weiterentwicklung von Abstraktion durch APIs oder OOP verstanden werden. Auch ein OOP-Framework stellt ein neues linguistisches Abstraktionslevel bereit. Es ermöglicht, über Probleme abstrakter "sprechen" zu können.

2 Ruby

Ich erkläre zuerst grundlegende Sprachfeatures von Ruby, um dann Metaprogrammierung und linguistische Abstraktion in Ruby zu beschreiben.
Eine ausführlicher Einführung in Ruby findet man z.B. unter [8] oder [11].

2.1 Ruby-Syntax und -Konzepte

Die Ruby-Syntax ist stark an Java und Perl angelehnt. Ruby hat eine mächtige Grammatik, die es erlaubt Klammern, Semikola etc. weg zu lassen, solange die Semantik des Ausdrucks eindeutig bleibt.

Die Sprachkonzepte sind hauptsachlich aus Smalltalk übernommen. Lisp und Perl hatten ebenfalls einen starken Einfluss auf das Ruby-Sprachdesign. Das primäre Designziel von Ruby ist laut Erfinder Yukihiro Matsumoto "Programmer Happiness" [3].
Ruby ist ein OOP-Sprache. Alles ist ein Objekt. Z.B. ist eine "5" ein Objekt der Klasse `Fixnum`. Auch die Klasse `Fixnum` ist ein Objekt (der Klasse `Class`). Die einzige Ausnahme sind Blöcke, die in der aktuellen Sprachversion 1.9 noch keine Objekte sind. Ein Block ist vergleichbar mit einer $\lambda-$Funktion.

Ruby ist eine dynamisch getypte Sprache.
Es gibt das Konzept der "*Open Base Classes*", d.h. man kann alle Klassendefinitionen zu einen beliebigen Zeitpunkt wieder "öffnen" und neue Definitionen hinzufügen oder alte entfernen.

2.1.1 Beispiele für grundlegende Sprachfeatures in Ruby

In den folgenden Beispielen verwende ich einen sog. *Read-Eval-Print-Loop* (REPL) um die Auswertung von Ausdrücken zu veranschaulichen.
Die Semantik ist wie folgt:

- `> ..` - symbolisiert eine Eingabe in den REPL
- `=> ..` - symbolisiert die Ausgabe des Ruby-Interpreters

Kommentare, Arrays und Strings werden in Ruby wie folgt definiert:

```
# This is a one-line comment

[1,2,3,5,8] # Array

"Ruby rocks!" # String
```

Methoden-Definitionen und -Aufrufe funktionieren so:

```
def foo(a,b,c) # define method foo with params a, b, and c
    a + b + c # no return needed
end

> foo(1,2,3) # call method foo
=> 6
```

Klassen definiert man über das Schlüsselwort `class`. Mit `def` lassen sich Methoden definieren. `attr_reader` ist eine in Ruby eingebaute Funktion zur Metaprogrammierung. Sie definiert Getter für eine Liste von gegeben Attribut-Namen. Es gibt auch Methoden, die Setter definieren. Die definierten Methoden implementieren das erwarte Standardverhalten, d.h. sie setzen bzw. lesen Attribute. Die Attribute selbst muss man in Ruby nicht extra definieren, da Ruby dynamisch getypt ist und die Attribute einfach bei der ersten Verwendung generiert.

```
class A # open class A

  attr_reader :name # def method name()
  def initialize(n) # constructor hook
    @name = n       # set attribute name to n
  end

  def foo { "foo" } # def method foo()

  # def class method bar() that returns the string "bar"
  # class methods are like static methods in java
  def self.bar { "bar" }

end
```

Die oben definierten Methoden kann man wie folgt verwenden:

```
# A.new("MyName").foo() - calls foo() on new instance of class A
> A.new("MyName").foo # no '()' needed
=> "foo"

> A.bar # call class (static) method bar on class B
=> "bar"

> A.new("MyName").name # call getter name()
=> "MyName"
```

Alles ist ein Objekt:

```
> 5.class # get class of object 5
=> Fixnum

> Fixnum.superclass # get superclass of class Fixnum
=> Integer

# get class of class Fixnum
# classes are objects, too
> Fixnum.class
=> Class

> Class.superclass
=> Module

> Class.superclass.superclass
=> Object

> 5.methods # get methods of object
=> ["%", "inspect", "<<", ... ]

# get methods of class Fixnum
# returns class / static methods for object 5
> Fixnum.methods
=> ["inspect", "private_class_method",
    "const_missing", ... ]
```

Ein Beispiel für die Verwendung von Ruby-Open-Base-Classes:

```
class Object # reopen class Object
      def foo # define a foo method on _all_ Objects
         "foo"
```

```
      end
end
```

Nun kann man auf allen Objekten die Methode `foo` aufrufen:

```
> Object.new.foo
=> "foo"

> 5.foo
=> "foo"

> Fixnum.foo
=> "foo"
```

Blöcke verhalten sich ähnlich wie λ-Funktionen. Man kann einer Funktion einen Block in geschweiften Klammern übergeben. Im folgenden Beispiel führt der Block auf seinem Parameter `x` ein `+ 1` aus.

```
> [1, 2, 3].map { | x | x + 1 }
=> [2, 3, 4]

# use do ... end instead of { ...}
> [1, 2, 3].map do | x | x + 1 end
=> [2, 3, 4]
```

2.2 Ruby und Java

Ruby und Java sind gut integrierbar:

Es gibt zum Einen *JRuby* (s. `http://jruby.codehaus.org/`), eine Ruby-Implementierung für die JVM. Zur Zeit wird Ruby 1.8.5 unterstützt. Es ist mit relativ wenig Aufwand möglich aus JRuby Java-Klassen zu verwenden und umgekehrt.
Zum Anderen gibt es *Groovy* (s. `http://groovy.codehaus.org/`). Eine Spracherweiterung für Java, die etliche Ruby Konzepte und Syntaxelemente in Java umsetzt.

3 Metaprogrammierung (in Ruby)

3.1 Was ist Metaprogrammierung?

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. Scheme 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.

In Ruby wird komplexere Metaprogrammierung über Interpreter-Hooks realisert, etwa `method_missing`. `method_missing` wird augerufen, wenn man eine nicht vorhandene Methode auf einem Objekt aufruft. Die Default-Implemtierung wirft eine `NoMethodError` Exception. Durch Überschreiben 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.

Einige grundlegende Metaprogammierungs-Funktionen in Ruby sind:

- `attr_reader`, `attr_accessor`, ... - erzeugen Getter und/oder Setter.
- `instance_eval`, `class_eval`, ... - führen Code im Klassen oder Instanz Kontext au.
- `send` - sendet einen eine Nachricht (entspricht Methodenaufruf) an ein Objekt. Nützlich, wenn man erst zur Laufzeit weiß, welche Methode man aufrufen will.
- `define_method` - definiert eine Methode zur Laufzeit.

3.2 Beispiele für Metaprogrammierung

3.2.1 map_send

`map`, `filter` etc. Funktionen sind ein aus der funktionalen Programmierung bekanntes Muster zur Iteration über Listen und andere Strukturen. `map_send` soll ein Haskell ähnliches `map (+1) xs` etc. realisieren. Es werden einige Ruby-Techniken zur Metaprogrammierung verwendet, um diese zu veranschaulichen.

Es ist sinnvoll die `map`, `filter` usw. Methoden direkt im `Enumerable` Modul zu implementieren, da so alle Klassen die `Enumerable` sind über die definierten Methoden verfügen. Enumerable ist mit dem Java `Iterable`-Interface vergleichbar. In einem Ruby-Modul können aber nicht nur Methoden deklariert werden, sondern auch direkt definiert werden.

```
# reopen the Enumerable module
module Enumerable

  # "normal" way to define a public method
  # usage: [1,2,3].map_send1 :+, 1
  def map_send1(method, *args) # ruby varargs
    map { |x| x.send method, *args }
  end

  # define a public method (using define_method)
  # usage: [1,2,3].map_send2 :+, 1
  define_method('map_send2') do | method, *args |
    map { |x| x.send method, *args}
  end

  # define a bunch of methods - called map_send,
  # each_send ...
  # usage: [1,2,3].map_send :+, 1
  # usage: [1,2,3].select_send :==, 1
  # as you can see, we generate five functions with 6 lines
  # of code, which is quite efficent
  %w[select map each collect inject].each do |method|
    self.send(:define_method,
        "#{method}_send") do |m, *args|
      send(method) { |x| x.send(m, *args) }
    end
  end

end # module Enumerable
```

Verwenden kann man die definieren Methoden beispielsweise auf Arrays:

```
# increment all elements in list by one
> [1,2,3].map_send :+, 1
=> [2,3,4]

# get all elements that are equal to one
> [1,2,3].select_send :==, 1
```

```
=> [1]
```

In obigen Beispiel wird die Haskell-Syntax nicht vollständig realisiert. Das hat zwei Gründe:
Zum Einen ist es sinnvoll sich an die Ruby-Syntax anzupassen, um weiterhin in einer konsistenten Sprachumgebung arbeiten zu können.
Zum Anderen muss man den Evaluierer umschreiben, damit er die neue Syntax verarbeiten kann. Der Aufwand hierfür ist hoch und der Nutzen nicht erkennbar.

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.
`map_send` stellt eine neue Abstraktion bereit. Für Haskell Programmier, die in Ruby programmieren, ist sofort klar, welche Semantik diese Funktionen implementieren. Das die gewonnen Abstraktion im obrigen Beispiel sehr gering ist, liegt schlicht daran, dass das Beispiel leicht verständlich sein soll.

3.2.2 synchronized

Dieses Beispiel wurde aus “The Ruby Programming Language” S. 282 [11], übernommen.
Es wird Java ähnliches `synchronized` definiert, um ein neues Sprachkonzept zu erzeugen, mit dem man über Nebenläufigkeit “reden” kann.

```
require 'thread' # ruby 1.8 (Mutex defined in this lib)

# define "global" method synchronized
def synchronized(o)
  o.mutex.synchronize
end

class Object # opening class Object
  def mutex
    return @__mutex if @__mutex
    # get mutex on class obj.
    synchronized(self.class) {
      @__mutex = Mutex.new # ret-val is @__mutex
    }
  end
end

# Prevent endless recursion
# Class is an object after all
Class.instance_eval { @__mutex = Mutex.new }
```

Nun kann man `synchronized` wie folgt verwenden:

```
mutex = Object.new
synchronized (mutex) {
  print "synced on mutex: " + mutex.to_s
}
```

Dieses Beispiel verwendet keine Metaprogrammierung, erzeugt aber dennoch ein neues Sprachkonstrukt - `synchronized` - um über Nebenläufigkeit "reden" zu können. Ob `synchronized` ein gutes Konstrukt ist um über Nebenläufigkeit zu reden will ich hier nicht bewerten. Es ist auch denkbar Erlang-ähnliche *Guarding-Expressions* für Ruby zu implementieren (vgl. [14]).

4 (Meta)lingustische Abstraktion

4.1 SICP über Metalingustische Abstraktion

Das Buch "Structure and Interpretation of Computer Programs" (SICP) [9] vom MIT beschreibt Metalinguistsche Abstraktion in Kapitel 4 wie folgt:

"Metalinguistic abstraction – **establishing new languages** – plays an important role in all branches of engineering design. It is particularly important to computer programming, because in programming not only can we formulate new languages but we can also implement these languages by constructing evaluators.

An evaluator (or interpreter) for a programming language is a procedure that, when applied to an expression of the language, performs the actions required to evaluate that expression.
(...)
It is no exaggeration to regard this as the most fundamental idea in programming:

The evaluator, which determines the meaning of expressions in a programming language, is just another program.

In fact, we can regard almost any program as the evaluator for some language."

4.2 Konzept Metalinguistische-Abstraktion

Die Idee von metalinguistischer Abstraktion ist, dass man durch Umschreiben des Sprach-Evaluieres neue Sprachen bzw. Sprachkonstrukte einführt.
Anstatt eine API zu schreiben, die bei der Lösung eines Problems hilft, schreibt man eine Sprache. Ziel ist es durch einen linguistischen Ansatz ein höheres Abstraktionsniveau zu erreichen, als es durch APIs möglich ist.

Der Begriff “metalinguistische Abstraktion” ist nicht eindeutig definiert und eine Abgrenzung zu linguistischer Abstraktion ist nur unpräzise möglich. Das SICP spricht von Metalinguistische Abstraktion wenn man beispielsweise `define-syntax` Makros verwendet um in das Auswertungsverhalten des Evaluieres einzugreifen. Ein *lazy-eval* für Java ist nur durch Umschreiben des Java-Evaluieres möglich, da man sonst nicht die Auswertungsreihenfolge der Sprache beeinflussen kann.
Ein weiteres Beispiel wird im SICP [9] in Kapitel 4.4 beschrieben: Ein Evaluierer für eine prolog-ähnliche, deklarative Logik-Progmmiersprache.

In die Ruby Welt ist dieses Konzept schwer übertragbar, da es in Ruby keine lisp-artigen Makros gibt. Man verwendet wie gesagt u.a. Interpreter-Hooks. Es hat sich daher der Begriff der *domänen-spezifischen Sprachen* (DSL) eingebürgert.

Das (sehr optimistische) Ziel von metalinguistischer Abstraktion bzw. DSLs ist folgendes:
Man erhält eine Spezifikation für ein Problem und überträgt diese in eine formale, abstrakte Repräsentation. Nun wird einfach ein `(eval SPEC)` ausgeführt und das Problem ist gelöst. Der Programmierer muss “nur noch” die `eval`-Funktion implementieren, was in der Regel kein triviales Problem ist. In der Ruby Welt, nennt man diese `eval`-Funktion eine DSL.

Die Idee aus `(eval SPEC)` ein lauffähiges Programm zu erzeugen, mag zuerst unrealistisch klingen.
Aber in der Lisp Welt generiert man z.B. aus RFC-Protokollspezifikationen De- und Encoder für ein Protokoll. Der Abstraktionsgewinn ist hoch, da man nur noch in einer BNF ähnlichen Notation die Protokoll Bestandteile definiert und die En- und Decoding-Funktionen automatisch generiert werden.

4.3 Domain Specific Language (DSL)

Eine *domänen-spezifische Sprache* (DSL) ist eine Sprache um domänen-spezifische Probleme zu lösen. Das Makefile-Format, das Maven-POM-XML-Format oder das Ant-XML- Format sind domänen-spezifische Sprachen, die für die Beschreibung von Software-Build-Prozessen eingesetzt werden. Die *Object Constraint Language* (OCL) ist eine Sprache, die sich mit Objekt-Invarianten etc. auseinandersetzt.

Der Vorteil einer DSL gegenüber einer API wird an folgendem Beispiel deutlich. Das domänenspezifische Problem sei “Kaffee kaufen”.

Ein API für Kaffee kaufen kann wie folgt aussehen:
```
new CoffeeShop(''Foo'', ...).getCoffee(
new Customer(''Fabian'', ...));
```
Mit einer DSL ist Folgendes implementierbar:
```
Fabian getsCoffee @ CoffeeShop named Foo
```

Der Vorteil der DSL ist der stärkere Abstraktionsgewinn, z.B.: ist "Fabian ist ein `Customer` Objekt" nicht relevant. Diesen Vorteil erkauft man durch einen höheren Implementierungsaufwand.

Der Aufwand eine DSL zu schreiben kann drastisch reduziert werden, wenn man die DSL ein eine bestehenden Sprache - etwa Ruby - einbettet. Der Ruby-Evaluierer wird nur an einigen Stellen erweitert um die neue Sprache behandeln zu können.

Nicht eingebettete DSLs werden über externe Parser-Generatoren, wie z.B. Yacc [13] oder ANTLR [4], erzeugt. Sie sind in der Syntax flexibler als eingebettete DSLs, da der erzeugte Parser beliebige formale Sprachen auswerten kann. Allerdings steigt der Implementierungsaufwand mit der Flexibilität der Sprache drastisch an (vgl. [15]). Über die Zeit tendieren DSLs dazu, sich zu "richtigen" Programmiersprachen zu entwickeln, da man für komplexere Probleme Fallunterscheidungen (also ein `if` o.Ä.) braucht. Auch Iterationskonzepte usw. finden dann relativ schnell Einzug in die DSL.

Wird die DSL von Anfang an in ein bestehende Programmiersprache eingebettet, kann man obiges Problem umgehen. Bei Bedarf kann man einfach auf die umgebende Sprache zurückgreifen.

4.3.1 Beispiel für eine in Ruby eingebettete DSL

Das folgende Beispiel aus [10] übernommen.
Die *Object Constraint Language* (OCL) ist eine Sprache mit der Invarianten etc. auf UML-Klassen definiert werden. Im folgenden Beispiel wird erzwungen, dass alle UML Klassen die ein Attribut mit dem Namen "`id`" haben, dieses Attribut mit dem Typ `Integer` definieren.

```
context UML::Class
inv: self.attributes->forAll(attr |
    attr.name == 'id' implies
    attr.type.oclKindOf(UML::Integer))
```

Mit einer in Ruby eingebetteten DSL lässt sich das wie folgt umsetzen:

```
context UML::Class do
  inv('integer-id') do
    self.attributes.forAll { |attr|
      (attr.name == 'id').implies(
        attr.type.oclKindOf(UML::Integer))
    }
  end
end
```

Im Unterschied zu OCL ist die Ruby-DSL ausführbar. Man kann sie in einem Ruby Interpreter evaluieren wenn man die benötigten Bibliotheken geladen hat.

Eine andere populäre Ruby-DSL ist *Ruby on Rails*. Ruby on Rails ist eine DSL für Webentwicklung, die wegen ihres hohen Abstraktionsgrades geschätzt wird.

5 Probleme und Lösungsansätze

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 [15] beschreibt der Autor das Problem 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 Currying.

Eine DSL bzw. ein Metaprogammierungsframework sollte im mathematischen Sinne abgeschlossen 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 dann zu evaluieren. Dies führt 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 erzeugt. 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.

6 Metaprogrammierung und DSLs in Java

In den meisten Java IDEs gibt es sog. "*One-Time Code Generators*". Sie erzeugen über Dialoge einmalig Code. In Eclipse ist die Funktion "*Source - Generate Getters / Setters*" ein Beispiel für

One-Time Code Generators.
Diese Verfahren haben den Nachteil, dass die zur Generierung verwendeten Informationen verloren gehen. Ändert sich das Klassen-Layout muss man die Getter und Setter neu erzeugen und ggf. die Alten entfernen. Des Weiteren werden etliche Zeilen Code erzeugt, die nur die Information enthalten "Es gibt Standard Getter und Setter für diese Attribute". Vgl. [15].

Java selber stellt mit Annotations und der Reflection-API einige Voraussetzungen zur Metaprogrammierung bereit. Mit Java selbst Java-Code zu erzeugen wird aber nicht unterstützt.

Es gibt Werkzeuge für Java die bei der Code- und Sprach-Generierung helfen. Es sei hier zum einem ANTLR (ANother Tool for Language Recognition) [4] genannt, das ein mächtiger Parser-Generator ist. Zum anderen gibt es OpenArchitectureWare [7], das ein Toolkit für Model Driven Architekure bzw. Model Driven Software Development ist.

7 Metaprogrammierung und DSLs in Unternehmen

7.1 DSL für Mobile AAA

Der Autor dieser Arbeit hat über zwei Jahre für die Netzwert AG bzw. Apertio Germany AG (jetzt Nokia Siemens Networks [6]) gearbeitet.
Ein Software-Produkt dieser Firma ist der *One-AAA Server* [5]. AAA steht für "authentication, authorization and accounting". Ein AAA-Server stellt u.a. Radius- und Diameter-Schnittstellen zur Verfügung, um Benutzern IP-Adressen zu zuweisen, diese zu authentifizieren und ihnen die verwendeten Dienste in Rechnung zu stellen. Der One-AAA Server wird u.a. von T-Mobil, O2 und Vodafon produktiv in deren Mobil-IP-Netzen eingesetzt.

Um den AAA Server zu konfigurieren und die kunden-spezifische Business-Logic zu implementieren kommt eine DSL zum Einsatz. Ohne DSL wäre die Komplexität der Business-Logic nicht beherrschbar. Die DSL hat zusätzlich den Vorteil, dass sie von Domänenexperten relativ leicht verstanden werden kann. Anpassungen und Änderungen können die Domänenexperten selber vornehmen, ohne auf die ständige Mithilfe der Softwareentwickler angewiesen zu sein.

7.2 Lisp DSL für Spieleentwicklung

Die Computerspiele-Firma *Naughty Dog* verwendet für einige ihrer Spiele Lisp. Naughty Dog entwickelt u.a. "Crash Bandicoot".

Das Spieleprojekt "Jax and Daxter" ist gescheitert. Einer der Gründe dafür war die Verwendung einer Lisp DSL namens *GOAL*. GOAL steht für "Game Oriented Assembly Lisp". Ein Mitarbeiter schreibt über GOAL:

"GOAL rocks! (...)

GOAL sucks! While it's true that GOAL gave us many advantages, GOAL caused us a lot of grief. A single programmer (who could easily be one of the top ten Lisp programmers in the world) wrote GOAL. (...)

While he called his Lisp techniques and programming practices 'revolutionary', others referred to them as 'code encryption', since only he could understand them. Because of this, all of the support, bug fixes, feature enhancements, and optimizations had to come from one person, creating quite a bottleneck. Also, it took over a year to develop the compiler, during which time the other programmers had to make do with missing features, odd quirks, and numerous bugs." [2]

Weiterführende Inforamtionen zu dem Projekt findet man unter [1].

8 MDA / MDSD und Metaprogrammierung

Model-driven Architecture (MDA) bzw. *Model-Driven Software Development* (MDSD) und Metaprogrammierung bzw. DSLs haben eine vergleichbare Problemstellung. In der MDA Welt verwendet 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 Informationen 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. [10] & [15]).

9 Zusammenfassung

Metaprogrammierung und (meta)linguistische Abstraktion sind mächtige Werkzeuge. Wie mit allem mächtigen Werkzeugen, kann man auch mit ihnen mächtig viel Schaden anrichten, wenn man sie falsch eingesetzt. Nicht jedes Problem sollte nach meiner Meinung mit Metaprogrammierung oder DSLs gelöst werden, nur weil es möglich ist. Der potenzielle Nutzen sollte immer gegen die zusätzliche Komplexität abgewogen werden.

DSLs und metalinguistische Abstraktion sind keine Werkzeuge, die man braucht um Probleme zu lösen, die sich innerhalb von einigen Mann-Tagen erledigen lassen. Für Software-Projekte, die eine Dauer von mehreren Jahren haben und viel domänen-spezifisches Wissen beinhalten, kann eine DSL meiner Meinung nach sehr vorteilhaft sein.

Sprachen wie Ruby, Python, Lisp oder Smalltalk senken den Implementierungsaufwand für eingebettete DSLs erheblich. In Ruby kann man mit wenigen hundert Zeilen Code nützliche DSLs erzeugen, die dann über die Zeit reifen und wachsen können. Man vergleiche hierzu [10], Ruby on Rails oder Rake.

- Paper im Netz: `https://page.mi.fu-berlin.de/bieker/paper-metaprog.pdf`
- Lizenz: Creative Commons 3.0 - by-nc-sa (de) `http://creativecommons.org/licenses/by-nc-sa/3.0/de/`
- Der Autor dankt Uwe Kamper und Stefan Plantikow für konstruktive Kritik.

10 Quellenverzeichnis

Literatur

[1] Postmortem: Naughty dog's jax and daxter: the precursor legacy, 07 2002. URL: http://www.gamasutra.com/features/20020710/white_02.htm.

[2] Lisp in games: Naughty dog's jax and daxter, 12 2005. http://ynniv.com/blog/2005/12/lisp-in-games-naughty-dogs-jax-and.html.

[3] Ruby on rails - and why ruby?, 02 2006. URL: http://www.theregister.co.uk/2006/02/16/ruby_rails/.

[4] *ANTLR v3 documentation*, 09 2007. URL: http://www.antlr.org/wiki/display/ANTLR3/ANTLR+v3+documentation.

[5] Apertio one-aaa, 11 2008. URL: http://www.apertio.com/products/apertio-one-applications/oneaaa.

[6] Nokia siemens networks expands leadership in subscriber data management solutions, 01 2008. URL: http://www.nokiasiemensnetworks.com/global/Press/Press+releases/news-archive/Nokia_Siemens_Networks_expands_leadership_in_subscriber_data_management_solutions.htm.

[7] *openArchitectureWare User Guide, Version 4.3*, 04 2008. URL: http://www.eclipse.org/gmt/oaw/doc/4.3/html/contents/index.html.

[8] Ruby in twenty minutes, 11 2008. URL: http://www.ruby-lang.org/en/documentation/quickstart/.

[9] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, second edition, 1996.

[10] Jesús Sánchez Cuadrado and Jesús García Molina. Building domain-specific languages for model-driven development. *IEEE Softw.*, 24(5):48–55, 2007.

[11] David Flanagan and Yukihiro 'Matz' Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc., first edition, 2008.

[12] Andrew Hunt and David Thomas. *The Pragrammtic Programmer*. Addison-Wesley Professional, 1999.

[13] Stephen C. Johnson. *Yacc: Yet Another Compiler-Compiler*. AT&T Bell Laboratories, Murray Hill, New Jersey 07974. URL: http://dinosaur.compilertools.net/yacc/.

[14] Sven-Olof Nystroem. Concurrency in java and in erlang, 05 2004. URL: http://prog.vub.ac.be/~wdmeuter/PostJava04/papers/Nystrim.pdf.

[15] Diomidis Spinellis. Rational metaprogramming. *IEEE Software*, 25(1):78–79, January/February 2008.