Bachelorarbeit

Ruben Schempp

Aktuelle Entwicklungstrends
im Bereich von Programmiersprachen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Ruben Schempp

Aktuelle Entwicklungstrends im Bereich von Programmiersprachen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Angewandte Informatik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Guido Pfeiffer
Zweitgutachter: Prof. Dr. rer. nat. Michael Böhm

Abgegeben am 30. August 2006

Ruben Schempp

Thema der Bachelorarbeit

Aktuelle Entwicklungstrends im Bereich von Programmiersprachen

Stichworte

Programmiersprachen, Java, Ruby, Ruby on Rails, Aspektorientierte Programmierung (AOP), Language Oriented Programming (LOP), Language Workbench, Domänenspezifische Sprachen (DSL)

Kurzzusammenfassung

Diese Arbeit befasst sich mit aktuellen Entwicklungen im Bereich von Programmiersprachen und versucht einen Ausblick auf die nächsten Jahre zu geben. Es werden Probleme der heutigen Programmiersituation diskutiert, insbesondere von Java, und es werden verschiedene Programmiertechniken behandelt.

Die Möglichkeiten und Grenzen der Java-Plattform werden – nach heutigen Gesichtspunkten – diskutiert. Die Sprache Ruby wird als derzeit interessanteste alternative Programmiersprache (zu Java) besprochen.

Ein weiteres Thema behandelt die Chancen der aspektorientierten Programmierung in naher Zukunft.

Interessante Entwicklungen finden im Bereich des „Language Oriented Programming“ statt. Insbesondere die neuartige Einsatzweise domänenspezifischer Sprachen verspricht bessere Abstraktionsmöglichkeiten, da sie Modellierungslücke der Programmiersprachen verringern können.

Ruben Schempp

Title of the paper

Current Trends in the Field of Programming Language Development

Keywords

Programming Languages, Java, Ruby, Ruby on Rails, Aspect Oriented Programming (AOP), Language Oriented Programming (LOP), Language Workbench, Domain Specific Languages (DSL)

Abstract

This work deals with current trends in the field of programming languages and attempts to present likely developments for the coming years. Current difficulties faced by programmers, particularly in Java, are discussed, as are various programming languages.

The possibilities and limitations of the Java platform are discussed from a present-day point of view. In this context the Ruby language is considered as the most interesting alternative programming language (to Java).

Another issue of this work are the chances which Aspect-Oriented Programming may present in the near future.

Interesting developments are to be found in the field of Language Oriented Programming. In particular, the new approach to using Domain Specific Languages promises better possibilities for abstraction, because these languages reduce the modelling gap inherent in general purpose programming languages.

Danksagung

Zuallererst möchte ich mich bei meinem betreuenden Dozenten Herrn Prof. Dr. Guido Pfeiffer für die großartige Unterstützung bei meiner Bachelorarbeit bedanken. Es war mir eine Freude mit Ihnen zusammenzuarbeiten!
Ebenfalls bedanke ich mich bei meinen Zweitprüfer Herrn Prof. Dr. Michael Böhm.

Mein herzlicher Dank gilt auch meinem Gott, der mich auch in schwierigen Zeiten immer begleitet hat.

Und herzlich möchte ich auch meiner Familie und meiner Freundin Sabine danken, die mir während dieser Zeit zur Seite standen und mich unterstützt haben.

Inhaltsverzeichnis

„When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem." – Edsger W. Dijkstra [053]

1 Einleitung

1.1 Themenbereich der Arbeit

Diese Arbeit behandelt Aspekte aus dem Gebiet der Programmiersprachen. Es werden Neuerungen, richtungsweisende Entwicklungen und damit verbundene Ideen und Konzepte im Umfeld der Programmiersprachen vorgestellt, diskutiert und - nach Möglichkeit - bewertet. Die Arbeit versucht, einen Überblick auf den Stand der Dinge bei Programmiersprachen und über die daraus möglichen Entwicklungen für die nächsten Jahre zu geben.

Die Entwicklungen von Sprachen und deren zukünftige Bedeutung vorauszusagen, ist problematisch. Der Versuch einer Einschätzung von Sprachen ist nur möglich, wenn dabei auch bisherige Entwicklungen und Entwicklungslinien betrachtet werden. Dazu gehört natürlich auch ein Blick auf das technische Umfeld der Sprache.

In dieser Arbeit werden einige Kernpunkte der Überlegungen durch mal mehr, mal weniger „neutrale" Zitate unterstrichen. Obiges Zitat von E. W. Dijkstra beschreibt in einer leicht sarkastischen, aber treffenden Art das „*Programmierproblem*" der heutigen Zeit. Denn seit der Existenz erster Programmiersprachen gibt es lebhafte, häufig ideologisch geführte, Auseinandersetzungen über die „beste" Programmiersprache. Alle Versuche, eine solch universelle, geradezu ideale Programmiersprache zu erschaffen, sind bislang gescheitert. Und weder heute noch morgen ist eine solche Sprache in Sicht.

Unabhängig von dieser Diskussion gab es in der Vergangenheit immer Programmiersprachen, die den Markt mehr oder weniger dominierten. Beispiele dafür sind Fortran oder C++.

Heute besitzt Java in vielen Bereichen eine führende Stellung und kann deshalb gewissermaßen als Marktführer angesehen werden. Java ist eine Entwicklung der neunziger Jahre, die in der schnelllebigen Informatikbranche mittlerweile Probleme und Fragen aufwirft, die nicht mehr so einfach in den Griff zu bekommen sind. Das „Thema Java" und die Frage nach Alternativen werden daher genauer untersucht. Neben Java werden auch Anforderungen an eine zukunftsfähige Programmiersprache, Stärken und Schwächen potentieller Konkurrenten sowie neuartige Programmierkonzepte zur Sprache kommen.

Über das, was „nach oder über Java hinaus" geschehen wird, gibt es nur wenig Literatur. Es existieren einige Bücher, aber die zum Teil brennende Diskussion findet im Internet statt. Dies belegt ein kurzer Blick in das Literaturverzeichnis.[1]

Besonders ergiebige Quellen für die Thematik dieser Arbeit sind Entwickler oder Berater wie Martin Fowler, Bruce Tate, Paul Graham oder Markus Völter, deren Publikationen diese Arbeit maßgeblich beeinflusst haben.

Die Arbeit ist in drei Themenbereiche gegliedert.

[1] Abweichend von der üblichen Notationsweise für Referenzen werden Quellen aus dem Internet nur mit einer fortlaufenden Nummerierung gekennzeichnet.

Im zweiten Kapitel dieser Arbeit findet sich eine Analyse der heutigen Situation rund um Programmiersprachen. Es werden aktuelle Probleme von Programmiersprachen angesprochen. Dabei wird diskutiert, welche Anforderungen eine „ideale“ Sprache aus heutiger Sicht erfüllen sollte. Schwerpunkte in der Diskussion stellen die Sprachen Java und Ruby dar.

Das dritte Kapitel beschäftigt sich mit dem Paradigma der *aspektorientierten Programmierung*. Hier liegt der Schwerpunkt im Wesentlichen auf neueren Entwicklungen der Aspektorientierung. Und natürlich wird auch *AspectJ* behandelt.

Thema des vierten Kapitels ist das so genannte *Language Oriented Programming*, welches *Domänenspezifische Sprachen* und *Language Workbenches* umfasst.

Kapitel fünf stellt die in den drei vorherigen Kapiteln behandelten Entwicklungslinien im Zusammenhang dar und behandelt die Anwendung *Ruby on Rails*.

“The best way to predict the future is to invent it.” – Alan Kay [052]

1.2 Motivation und Ziel

Die Arbeit stellt nicht den Anspruch, die Zukunft vorauszusagen. Vielmehr geht es um die Vorstellung und Diskussion viel versprechender und interessanter Entwicklungslinien, deren zukünftige Beobachtung lohnenswert erscheint. Dabei wird kein Anspruch auf Vollständigkeit bezüglich aller Entwicklungen erhoben; Abgrenzungen bleiben dabei unerlässlich.

In erster Linie geht es um einen Blick auf den Markt und dessen technische Konzepte zur effektiven Lösung von Programmieraufgaben. Es ist an der Tagesordnung, neue Aufgaben bzw. Aufgabenbereiche mit überholten Mitteln und Vorgehensweisen zu lösen. Diese finden sich leider zu oft in den Unzulänglichkeiten von (z.T. falsch ausgewählten) Programmiersprachen wieder. Es entstehen Probleme und Schwierigkeiten, die leicht zu Fehlern führen können! Edsger W. Dijkstra [056] brachte es auf den Punkt:

“If debugging is the process of removing bugs, then programming must be the process of putting them in.”

Die heutige Programmiersituation ist nicht zufrieden stellend. Nach wie vor ist die Suche nach besseren Lösungswegen aktuell.

In dieser Arbeit werden folgenden Fragen behandelt:

- Was ist „falsch“ an der heutigen Art zu programmieren?
- Ist die Objektorientierung am Ende?
- Wie steht es um Javas marktbeherrschende Stellung?
- Was tut sich bei der Konkurrenz?
- Wieso ist Ruby in aller Munde?
- Wie entwickelt sich die aspektorientierte Programmierung weiter?
- Was steckt hinter Language Oriented Programming?
- Welches sind die ernstzunehmenden Alternativen?
- Welche Entwicklungslinien erscheinen richtungsweisend?

“Java was, as Gosling says in the first Java white paper, designed for average programmers. It's a perfectly legitimate goal to design a language for average programmers. (Or for that matter for small children, like Logo.) But it is also a legitimate, and very different, goal to design a language for good programmers.” – Paul Graham [054]

2 Gegenwärtige Situation

Dieses Kapitel ist in Anlehnung an das Buch *Beyond Java* [Tate-05] entstanden. Es behandelt die Defizite und Möglichkeiten der gegenwärtigen Programmiersituation. Dabei geht es nicht vorrangig um das Aufzeigen von Problemen der den Markt dominierenden Sprache Java, sondern generell darum, auf Unzulänglichkeiten und daraus entstehende Modellierungsprobleme auf dem Markt befindlicher Programmiersprachen hinzuweisen. Behandelt wird natürlich auch die Sprache Java in ihrer Position als Marktführer, aber auch deren Konkurrenten.

"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization." - Weinberg's Second Law [056]

2.1 Probleme und Defizite

In der Modellierung und Programmierung von Software treten immer wieder Probleme durch die verwendeten Programmiersprachen auf. Diese sind für bestimmte Anwendungsklassen unterschiedlich geeignet. Mit einer gut geeigneten Sprache sollten einfache Dinge leicht und auf kurzem Wege erreichbar sein. Doch dies ist leider oft nicht gegeben.

In der Evolution der Programmiersprachen wurden die einzelnen Sprachen tendenziell immer mächtiger. Doch spätestens mit Java wird deutlich, dass mit dieser „Tradition“ gebrochen wird. Java wurde etwa 30 Jahre nach Lisp entwickelt und sollte demnach Java deutlich weiter entwickelt sein. Doch es ist unumstritten, dass Lisp Javas Mächtigkeit übertrifft. Dies ist begründet in Lisps größerer Ausdrucksstärke, die viele Problemklassen durch mächtige Paradigmen viel einfacher lösbar macht. Trotz dieser Unterlegenheit bietet Java in anderen Bereichen wesentliche Vorteile gegenüber Lisp. Dennoch werden für die Java-Programmierung weit mehr Hilfsmittel und Konstrukte benötigt, die einen höheren Aufwand mit sich bringen.

In Anlehnung an die Überlegungen von Hartmut Krasemann [046] sollten bestimmte Programmierparadigmen nicht fest mit der Sprache verbunden sein, sondern optional zu benutzen sein. Es ist z.B. offensichtlich, dass das Paradigma der statischen Typisierung in Java die flexible Verwendung von Containern verhindert und die Wiederverwendung von Code einschränkt. Ebenso lassen sich auch Einschränkungen durch die Objektorientierung finden, beispielsweise bei der Kopplung von Modulen.

Um diese, durch nicht universell anwendbare Paradigmen gegebenen, Grenzen zu überwinden, bedarf es zum Teil großen Aufwands. Dieser Aufwand drückt sich durch zusätzlich benötigten Code aus, der wiederum das Arbeiten mit der Programmiersprache erschwert.

Kurz gesagt: Die eingesetzten Programmiersprachen befinden sich oft auf einer Ebene, die den an sie gestellten Anforderungen, bei der Lösung von Aufgaben bzw. Problemstellungen, nicht gerecht wird.

In Kapitel 2.3 werden die allgemein gültigen Anforderungen an eine Programmiersprache aufgeführt. Diesen Kriterien werden nur wenige Sprachen annähernd gerecht. So müssen beim Programmieren oft Einschränkungen in Kauf genommen werden, die die Produktivität der Entwickler und die Softwarequalität negativ beeinflussen.

In der Softwareentwicklung wird heute im großen Umfang die Sprache Java eingesetzt, weshalb im Folgenden immer wieder Bezug auf sie genommen wird. Java entspricht in vielen Punkten nicht dem, was ein gutes Programmierwerkzeug ausmacht. Es gibt einige Alternativen, die größere Beachtung verdient haben und sich – da sie nicht erst seit gestern verfügbar sind – bereits in ihrer Eignung bewiesen haben. Im folgenden Unterkapitel werden diese Programmiersprachen hinsichtlich ihrer Eignung untersucht.

2.2 Beurteilung ausgewählter Programmiersprachen

In diesem Abschnitt werden Konzepte und Stärken bzw. Schwächen einiger derzeit im Mittelpunkt des Interesses stehender Sprachen kurz diskutiert, um ihre Bedeutung jetzt oder in Zukunft besser abschätzen zu können. Der Fokus dieses Unterkapitels liegt bei Java und dem zukünftig wohl stärksten Konkurrenten Ruby.

Skriptsprachen

Viele der in diesem Kapitel behandelten Sprachen sind so genannte *Skriptsprachen*. Die Bezeichnung *Skriptsprache* ist nicht klar eingegrenzt und wurde ursprünglich für *Kommandosprachen* verwendet. Ihr Aufgabengebiet beschränkte sich meist auf einen kleineren Bereich, wodurch Skriptsprachen in der Regel einen geringeren Umfang an Anweisungen umfassten als normale Programmiersprachen.

Im Laufe der Jahre tendierte die Entwicklung der Skriptsprachen hin zu ausgewachsenen Programmiersprachen. Mittlerweile ist eine klare Abgrenzung einiger Skriptsprachen zu objektorientierten Programmiersprachen nicht mehr möglich (siehe etwa Python und Ruby).

Aufgrund ihrer Ausrichtung auf ein schnelles Feedback werden Skriptsprachen typischerweise interpretiert statt übersetzt und sind größtenteils dynamisch typisiert.

Das Einsatzgebiet von Skriptsprachen liegt vorwiegend bei Web-Anwendungen.

"If it was hard to write it should be hard to read" – Unbekannte Quelle [056]

2.2.1 Perl

Perl[2] ist eine *Open-Source*-Skriptsprache aus dem Jahr 1987 von Larry Wall. Ihre Stärken liegen in den Bereichen *Shell Script*[3] und in der Stringverarbeitung, da sie textorientiert und stark textmanipulativ ist. Perl besitzt ein schnelles Feedback, ist sehr effizient, dynamisch typisiert und mäßig gut objektorientiert.

Leider ist Perls Syntax recht kryptisch, weswegen Perl nachgesagt wird, eine „write-only“ Sprache zu sein. Dies ist zugleich auch der größte Kritikpunkt an Perl: Effizienter Code ist nur schwer lesbar. Eine vernünftige Wartung von Perl-Code ist daher kaum möglich. Brian Hook [056] hat also nicht ganz Unrecht, wenn er sagt:

"I have a pretty major problem with a language where one of the most common variables has the name $_"

Perl erfreut sich in dem beschränkten Umfeld der Webprogrammierung und Systemadministration großer Beliebtheit. Dies ist darin begründet, dass die Sprache Perl das Aufgabengebiet sehr gut unterstützt, für das sie gemacht bzw. entwickelt wurde.

[2] http://www.perl.org/
[3] Shell Script bezeichnet den Kommandozeileninterpreter unter UNIX-Betriebssystemen.

"Python is more concerned with making it easy to write good programs than difficult to write bad ones." - Steve Holden [057]

2.2.2 Python

Python[4] ist ein starker Konkurrent zu Perl, Ruby und Java. Entstanden ist die objektorientierte und dynamische Open-Source-Skriptsprache im Jahr 1990. Ihr Einsatzgebiet entspricht, neben Web-Anwendungen, im Wesentlichen dem Einsatzgebiet einer gewöhnlichen Programmiersprache. Python besitzt eine Version namens *Jython*, welche auf der Java Virtual Machine (JVM) lauffähig ist und die von der vorhandenen Java-Infrastruktur profitiert.

Python besitzt viele gute Eigenschaften, die eine Anwendungsprogrammiersprache aufweisen sollte. Einige Parallelen zwischen Python und Ruby sind vorhanden, doch ein entscheidender, nicht sprachspezifischer, Punkt macht den Unterschied: Python fehlt eine *Killer-Applikation*[5] wie das Rails-Framework für Ruby.

Auf Python wird nicht näher eingegangen, da Ruby weitaus mehr von sich reden macht und inzwischen die größeren Chancen auf eine erfolgreichere Marktposition besitzt.

"And if you had shown people Ruby in 1975 and described it as a dialect of Lisp with syntax, no one would have argued with you." – Paul Graham [Graham-04]

2.2.3 Ruby

Abbildung 1: Der Rubin als Symbol für Ruby

"The metaprogramming capabilities of Ruby lie so close to the surface and are quite accessible to the average Ruby programmer." – Jim Weirich [Tate-05]

Ruby[6] ist eine frei verfügbare, von Grund auf objektorientierte und hocheffiziente Skriptsprache. Ruby ist stark verwandt mit den Skriptsprachen Perl und Python (siehe Abbildung 1). Unverkennbar ist auch die große Ähnlichkeit zu Smalltalk; beide Sprachen haben Wurzeln in Lisp, die bei Ruby eine durchaus stärkere Ausprägung haben.

Ruby ist im Moment „der Superstar unter den Programmiersprachen" – sie ist zurzeit die Sprache mit dem größten Zukunftspotential. Sie befindet sich in der Diskussion und hat eine stark wachsende Benutzergruppe. Ruby erfreut sich nicht nur steigender Beliebtheit, sondern stellt im Moment die Basis für Innovationen dar: *Metaprogrammierung* gelingt in Ruby nahezu natürlich und neue Frameworks wie *Active Record* machen von sich reden. Rubys innovatives Aushängeschild ist das Framework Ruby on Rails, welches den Stand der Technik in der Web-Programmierung darstellt.

[4] http://www.python.org/
[5] Siehe letzter Punkt Kapitel 2.3.2
[6] http://ruby-lang.org

Entwickelt wurde Ruby von Yukihiro Matsumoto im Jahr 1993 in Japan. Dort wurde die Sprache 1995 veröffentlicht und wird seitdem immer bekannter. Den Sprung nach Amerika und Europa schaffte Ruby erst nach der Jahrtausendwende, als englischsprachige Dokumentationen verfügbar waren.

Bemerkenswert ist, dass mit Ruby eine stabile, ausgewachsene und intuitive Sprache zum Vorschein kam, die bis dato nur in Japan bekannt war. Mit Ruby on Rails (siehe Kapitel 5.3.1) ist eine Anwendung verfügbar, die das Interesse vieler Programmierer erweckt und Ruby zu schneller Verbreitung verhilft. Diese beiden Gegebenheiten sind Grund genug die neue Sprache Ruby genauer anzuschauen:

Einige weitere Details

Ruby bietet neben einer hohen Portabilität auf alle wesentlichen Betriebssysteme auch Multithreading-Unterstützung an, und dies unabhängig davon, ob diese Plattform Multithreading unterstützt oder nicht.

Mit Ruby lassen sich exzellent *Domain Specific Languages* (DSL) erstellen. DSLs sind domänenspezifische Sprachen, die nur für einen speziellen Einsatzbereich optimiert sind, diesen aber sehr gut unterstützen. Beispielhaft sei hier Rake erwähnt, das Rubys Pendant zu *Make* und *Ant*[7] darstellt. (Mehr dazu in Kapitel 4.)

Ruby unterstützt *Unit Testing*, *Exception Handling*, *Reflection*, *reguläre Ausdrücke*, Metaprogrammierung, *Closures* und *„duck typing“*[8]. Wie auch in Smalltalk gibt es nur einfache Vererbung, dafür aber *Mixins*[9]. Jedes Datum ist in Ruby ein Objekt. Die Zahl 42 gehört beispielsweise der Klasse *Fixnum* an.

In Ruby lassen sich Klassen dynamisch, also zur Laufzeit, erweitern. Ein Beispiel [015] zur Erweiterung von Rubys Basis-Klasse *Fixnum* mit römischen Zahlen soll dies deutlich machen (siehe Codebeispiel 1):

```
class Fixnum
 def to_roman
  value = self
  str = ""
  (str << "C"; value = value - 100) while (value >= 100)
  (str << "XC"; value = value - 90) while (value >= 90)
  (str << "L"; value = value - 50) while (value >= 50)
  (str << "XL"; value = value - 40) while (value >= 40)
  (str << "X"; value = value - 10) while (value >= 10)
  (str << "IX"; value = value - 9) while (value >= 9)
  (str << "V"; value = value - 5) while (value >= 5)
  (str << "IV"; value = value - 4) while (value >= 4)
  (str << "I"; value = value - 1) while (value >= 1)
  str
 end
end
```

Codebeispiel 1: Erweiterung der Ruby Klasse Fixnum mit römischen Zahlen.

[7] Make ist ein UNIX-Werkzeug, Ant ein Java-Werkzeug. Beide dienen zur automatisierten Erstellung von Anwendungen auf der Basis von Quelltexten.

[8] „Duck typing“ ist eine Bezeichnung der Ruby-„Szene“ für dynamische Typisierung, die Typen nicht an Variable bindet. Es heißt über duck typing: *„if it accepts a ‚quack' message, it must be a duck.“*

[9] Mixin: An Stelle von Mehrfachvererbung können Klassen in Ruby einen Modul importieren, um auch dessen Methoden zu implementieren. Module sind Sammlungen von Methoden und Klassen. Mixins sind also die Implementierungen von Interfaces.

Ruby besitzt einen Interpreter namens *irb*. Ein einfach gehaltenes Beispiel [015] in irb soll zur Verdeutlichung von Rubys einfacher und effizienter Syntax dienen. Es wird hier eine Klasse *Person* mit den Attributen *Name* und *eMail* definiert, die daraufhin instanziiert wird und den Namen *Elvis* zugewiesen bekommt (siehe Codebeispiel 2):

```
irb(main):001:0> class Person
irb(main):002:1> attr_accessor :name, :email
irb(main):003:1> end
=> nil
irb(main):004:0> person = Person.new
=> #<Person:0x2b61a80>
irb(main):005:0> person.name = "Elvis"
=> "Elvis"
irb(main):006:0>
```

Codebeispiel 2: Definition und Erzeugung einer Klasse in Rubys Interpreter irb. Die Attribut-Zugriffsmethoden für Name und eMail werden generiert.

Wichtig für einen möglichst praktischen Umgang mit einer Programmiersprache ist eine ausgereifte *Entwicklungsumgebung*. Für Ruby steht mittlerweile ein Plugin [058] für die Entwicklungsumgebung *Eclipse*[10] in der Version 0.80 bereit, das auf dem besten Weg zu einer guten Integration von Ruby ist.

Rubys Entwickler sind sehr daran interessiert, weiterhin eine klare, einfache und gut lesbare Syntax zu bewahren. Dies schlägt sich auch in den Bibliotheken nieder, welche bisher jedoch, im Vergleich zu Javas Bibliotheken, noch kein so großes Gebiet abdecken.

Außerhalb Japans ist Ruby bisher kommerziell nur wenig verbreitet. Doch die Verbreitung wird zurzeit durch Ruby on Rails vorangetrieben. Sie würde sicher noch weiter beschleunigt, wenn die JVM-Adaption namens *JRuby*[11], nach einigen Startschwierigkeiten, einmal fertig gestellt wäre. Zudem hat Microsoft verlauten lassen, dass ihre *Common Language Runtime* eine Ruby-Unterstützung bieten wird – dies ermöglicht interessante Perspektiven und könnte Ruby einen weiteren Schub geben.

[10] http://www.eclipse.org/
[11] http://jruby.codehaus.org/

"In essence, Smalltalk is a programming language focused on human beings rather than the computer" - Alan Knight [070]

2.2.4 Smalltalk

Smalltalk[12], eine dynamisch typisierte Entwicklung der frühen siebziger Jahre, war seinerzeit die erste rein objektorientierte Programmiersprache. Sie wurde unter der Leitung von Alan Kay am *Xerox PARC*[13] entwickelt. Neben Kays Vision einer grafischen Umsetzung übernahm Smalltalk die Objektorientierung von *Simula*[14] und die Closures von Lisp.

Smalltalk kam erst Mitte der neunziger Jahre in einigen Branchen, wie etwa bei Versicherungen oder Banken, zu größerer Bedeutung. Größere allgemeine Bedeutung konnte Smalltalk nicht erlangen, da seine Verbreitung zu Gunsten von Java zurückging. Zwischenzeitlich kehren einige Firmen wieder zu Smalltalk zurück, da Java sie vor massive Probleme stellt.

Smalltalks Stärken liegen in einer hohen Ausdrucksfähigkeit, einem sauber implementierten Objektmodell, intelligentem Design und der anhänglichen Entwicklergemeinde. Smalltalk macht zudem massiven Gebrauch von Closures. Aufgrund dieser Spracheigenschaften ist Metaprogrammierung in Smalltalk einfach umsetzbar.

Gewissermaßen als Nachfolger der, aus dem Xerox PARC stammenden, *Smalltalk-80* Version, sind heute mehrere Derivate von Smalltalk verfügbar. Darunter sind die Open-Source-Version *Squeak*[15] und das kommerzielle *Visualworks*[16] weit verbreitet. Beide bringen eine eigene *virtuelle Maschine* mit sich und weisen dadurch eine hohe Portabilität auf.

Für weniger Akzeptanz sorgten die Smalltalk-eigenen Entwicklungsumgebungen, die teils gewöhnungsbedürftig sind und dadurch viele Nutzer abschreckten. Laut Aufwandsschätzungen sind Smalltalks Projektkosten verglichen mit Java um das drei- bis fünffache günstiger.

Glenn Vanderburg [Tate-05] vermutet, dass die neueren Entwicklungen der Programmiersprachen schlussendlich wieder zu Smalltalk zurückkehren, allerdings unter einem anderen Namen. Vanderburgs Vermutung wird durch die Entwicklung von Ruby bestätigt. Ruby erbt stark von Smalltalk, weist aber eine stärkere Verwandtschaft zu Lisp und der C-Syntax auf.
Aufgrund Rubys steigender Präsenz ist es unwahrscheinlich, dass Smalltalks Bedeutung noch einmal zunehmen wird.

2.2.4.1 Continuation Servers

Dieses Unterkapitel behandelt keine Programmiersprache, sondern eine neue Programmiertechnologie für eine Klasse an Web-Servern namens *Continuation Servers*. Das populärste Framework für Continuation Servers ist derzeit *Seaside*[17], das gleichzeitig einen Webserver mitbringt. Avi Bryant entwickelte Seaside ursprünglich in Ruby, doch nach anfänglichen (und inzwischen behobenen) Problemen wechselte er zu Smalltalk. Deshalb findet sich dieses Unterkapitel im Smalltalk-Bereich wieder.

Die Idee hinter Continuation Servers ist recht einfach: Web-Programmierung ist zustandslos, was prinzipiell in Ordnung ist. Doch entsteht durch das Fehlen von Zuständen für den Web-Entwickler ein großer Aufwand. Die Behandlung der Navigation einer Client-Sitzung kann leicht kompliziert werden, wenn etwa der Zurück-Knopf des Browsers gedrückt wird. Dann mag es vorkommen, dass Annahmen nicht mehr gelten, die zuvor in der Sitzung getroffen wurden. Ei-

[12] http://www.whysmalltalk.com/
[13] Palo Alto Research Center: Ein Entwicklungslabor der Firma Xerox.
[14] Simula war der erste objektorientierte Ansatz einer Programmiersprache der sechziger Jahre. Sie fand keine große Verbreitung, stellte aber quasi den Startschuss zur objektorientierten Programmierung dar.
[15] http://www.squeak.org/
[16] http://smalltalk.cincom.com/
[17] http://www.seaside.st/

nen Ausweg bietet die Einführung von Zuständen durch so genannte Snapshots[18]. Hier wird also der Zustand in Form des Stacks[19] gespeichert und kann bei Bedarf wiederhergestellt werden. Dies heißt hier *Continuations*.

Rubys Unterstützung von Continuations ist in die Sprache integriert. Ein leicht lesbares und einfaches Beispiel für Continuations [Tate-05] sieht folgendermaßen aus:

```
callcc do |continuation|
    for i in 1..10 do
    continuation.call if (I == 7)
    puts i
  end
  puts 'This never happens.'
end
puts 'Good bye.'
```

Codebeispiel 3: Continuations in Ruby

```
1
2
3
4
5
6
Good bye.
```

Codebeispiel 4: Ausgabe des vorigen Codebeispiels (3).

Zur Erklärung: *callcc* führt einen Snapshot aus und speichert den Anwendungsstack (siehe Codebeispiel 3). Daraufhin läuft eine Schleife, in deren siebten Durchlauf der *contiuation.call* ausgeführt wird. D.h. der im Snapshot gespeicherte Zustand (Anwendungsstack) wird wiederhergestellt und die nächste Zeile nach dem Continuation Codeblock (hier: *puts 'Good bye.'*) wird ausgeführt. (Ausgabe siehe Codebeispiel 4.)

Durch die Continuations wird es möglich, den Ablauf einer Web-Anwendung umzudrehen: Nicht mehr der Client stellt Anfragen an den Server, sondern der Server wartet auf Antworten des Clients (siehe Abbildung 2).

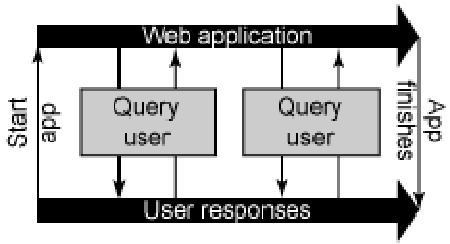

Abbildung 2: Continuations erlauben einen natürlicheren Anwendungsablauf. Die Kontrolle geht vom Client (User responses) zum Server (Web Application) über.

Der Server erhält also die Möglichkeit, seine Ressourcen für andere Aufgaben zu verwenden, während er die aktuelle Sitzung zwischenspeichert (und auf eine Antwort des Clients wartet). Dadurch wird die Handhabung der verschiedenen Stati bei der Verwendung des Zurück-Knopfs oder mehreren Fenstern im Browser wesentlich vereinfacht.

[18] Ein Snapshot ist die Aufnahme eines Zustands zu einem bestimmten Zeitpunkt.
[19] Mit Stack ist hier der Anwendungskontext gemeint, der im Programmstack enthalten ist.

Laut Glenn Vanderburg [Tate-05] ist es mit Seaside nun möglich, für Webanwendungen ähnlichen Code wie für Desktop-Anwendungen zu schreiben: Eine An- oder Rückfrage des Servers an den Client bzw. Benutzer wird wie eine gewöhnliche Anfrage einer Desktop-Anwendung behandelt. Die Continuations und der Dialog mit dem Benutzer werden automatisiert. Vereinfacht könnte man sagen, dass der Thread einer (Web-)Anwendung auf eine Rückmeldung vom Benutzer wartet.

Die hauptsächlichen Vorteile von Continuation Servers sind, neben einer deutlichen Produktivitätssteigerung, die einfachere, abstraktere und dadurch natürlichere Web-Entwicklung. Die dazu nötigen Techniken wie Closures unterstützen weder Java noch C#, jedoch Smalltalk, Lisp, Python und Ruby. Closures sind also, neben einer dynamischen Sprache, *das* Mittel, um Continuations zu ermöglichen.

Paul Graham hat die Technologie der Continuation Servers bereits erfolgreich in einem großen Lisp-Projekt eingesetzt. Er und andere namhafte Entwickler gehen davon aus, dass die interessante Technik der Continuation Servers größere Bedeutung erlangen wird. Wohl nicht in Smalltalk oder Lisp, sondern eher mit Python oder Ruby.

"C# is effectively a Java clone. ..." – Bruce A. Tate [Tate-05]

2.2.5 C# & .NET

C#[20] ist eine der neueren Programmiersprachen von Microsoft, die in die hauseigene .NET-Entwicklungsumgebung integriert ist.

Obwohl Microsoft es vehement bestreitet, ist C# eher eine Kopie von Java mit sehr ähnlichen Fähigkeiten und Problemen, aber einigen moderneren Eigenschaften. Problematisch ist die Abhängigkeit der Sprache C# von Microsoft und deren .NET-Plattform. C# ist eine proprietäre Sprache, an der Teile durch Microsoft patentiert sind.

Der Erfolg von C# hängt sehr davon ab, wie Microsoft seine .NET-Plattform auf dem Markt etablieren kann und beispielsweise in Sachen Portabilität auf Nicht-Microsoft-Plattformen weiterentwickeln wird.

.NET besitzt umfangreiche Bibliotheken und eine virtuelle Maschine namens Common Language Runtime (CLR). Sollte sich .NET mit Microsofts, auf unterschiedliche Zielgruppen ausgerichteten, Programmiersprachen gut entwickeln, steht damit wieder ein starker Konkurrent bereit. Vielleicht nicht unbedingt durch Sprachen wie C#, wohl aber mit der CLR-Unterstützung für andere Sprachen wie Ruby oder Microsofts in der Entwicklung befindliche Sprache „C Omega". Aufgrund Microsofts Marktstellung wird deutlich:

„In fact, Microsoft's .NET environment threatens Java now." – Bruce Tate [Tate-05]

C# wird hier nicht weiter diskutiert, da es zu große Gemeinsamkeiten mit Java aufweist und die meisten Kritikpunkte an Java dementsprechend auch für C# gelten.

[20] http://msdn.microsoft.com/vcsharp/

"Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot." – Eric Raymond [Graham-04]

2.2.6 Andere Sprachen

In diesem Unterkapitel werden einige Programmiersprachen kurz erwähnt, die eher nicht mehr zu größerer Bedeutung kommen werden oder in ihrer Entwicklung zu einer ausgereiften Sprache noch nicht so weit sind.

PHP

PHP[21] ist eine Open-Source-Skriptsprache der letzten 10 Jahre. Sie wird in HTML-Code eingebettet und von Web-Servern interpretiert, um den Zugriff auf Datenbanken oder andere Backend-Systeme[22] über Webseiten zu realisieren.

Die Sprache PHP ist beliebt, weil sie effektiv ist und schnell zu einem brauchbaren Ergebnis führt. Allerdings hat PHP den Nachteil, dass es Benutzeroberflächen zu eng an die Datenbank koppelt. Änderungen an einer der beiden Komponenten können folglich unerwünschte Konsequenzen für die andere bedeuten.

Weitere Kritikpunkte an PHP sind die Schwächen in der Struktur, die nicht zufrieden stellend umgesetzten objektorientierten Fähigkeiten und die fehlende Unterstützung für die Metaprogrammierung.

Mit fortlaufend steigendem Bekanntheitsgrad von Ruby on Rails wird es PHP schwerer haben, seine Marktposition zu behaupten, da das Rails-Framework auf dasselbe Gebiet ausgerichtet ist. Erste Anzeichen zu Gunsten von Ruby on Rails sind hier zu erkennen.

Als universelle Programmiersprache wird PHP aufgrund seiner engen Ausrichtung keine Zukunft haben. Jedoch gewann die Sprache 2005 als rein web-basierte Skriptsprache an Bedeutung, nicht zuletzt durch den zwischenzeitlich von IBM angebotenen PHP-Support.

"I invented the term Object-Oriented, and I can tell you I did not have C++ in mind." – Alan Kay [052]

C++

C++ wurde in den achtziger Jahren entwickelt und war *die* Sprache der neunziger Jahre. Als direkter Nachfolger von C bietet C++ neben einer Reihe von obligatorischen Neuerungen auch Unterstützung für die objektorientierte Programmierung.

Besondere Eignung erfährt C++ als Systemprogrammiersprache, weniger als Anwendungsprogrammiersprache. Dennoch sind schon sehr große Anwendungen in C++ umgesetzt worden.

Aufgrund einiger Eigenschaften von C++ müssen Probleme gelöst werden, die für die Problemlösung nicht relevant sind. Dies erfordert tiefergehende Fähigkeiten der Programmierer, erschwert den Entwicklungsprozess und senkt die Produktivität. Die Hauptprobleme der Programmierung in C++ sind die Vermischung der Referenz- und Wertsemantik, die fehlende automatische Speicherverwaltung und die mangelnde Portabilität.

[21] PHP: Hypertext Preprocessor, http://www.php.net/

[22] Das Backend ist der Teil einer Anwendung, der eigentliche Aufgabe dieser übernimmt. Es stellt das Gegenstück zum Frontend, der Benutzerschnittstelle, dar.

Die Bedeutung von C++, als Sprache für die Erstellung von Anwendungen, wird abnehmen. Wegen seiner Präsenz und dem Einsatz in der Systemprogrammierung wird C++ jedoch sicher über einen längeren Zeitraum ein Begriff bleiben.

Alan Kay [054] über Lisp: "The greatest single programming language ever designed."

Lisp

Lisp[23] ist eine extrem ausdrucksstarke, produktive und funktionale Sprache. Sie wurde Ende der fünfziger Jahre von John McCarthy am Massachusetts Institute of Technology (MIT) erfunden. Lisp ist nach *Fortran*[24] die älteste Programmiersprache und hat seine Bedeutung – nach wie vor – im Bereich der *künstlichen Intelligenz* (KI). Sie besteht nur aus wenigen Syntaxelementen, atomaren Elementen und Listen. Lisps Spezialität ist die Anbindung an den Lambda-Kalkül.

Besonders beliebt ist Lisp wegen seiner sehr guten Metaprogrammierungsfähigkeiten, die es erlauben, so gut wie jede Technik und jedes Konzept in Lisp zu implementieren.

Die Lisp-Benutzergemeinde ist, wie die von Smalltalk, zwar relativ klein, aber umso leistungsfähiger in Bezug auf die Lösung von (schwierigen) Problemen.

Trotz diverser Dialekte[25] hat Lisp den Sprung zu kommerziellen Anwendungen, bis auf Ausnahmen, nicht geschafft. Lisp bleibt eine Sprache, die im akademischen und KI-Bereich angesiedelt ist. Zukünftig wird sich daran wohl nicht viel ändern, denn Lisp fehlt eine *Killer-Applikation*. Mittlerweile setzt Ruby viele Fähigkeiten von Lisp um und macht diese zugänglicher.

Paul Graham empfiehlt Entwicklern, sich einmal genauer mit Lisp auseinander zu setzen, da Lisps Paradigmen und deren Mächtigkeit den Horizont eines Programmierers erweitern, selbst wenn dieser daraufhin wieder zu den Sprachen zurückkehrt, die er ursprünglich verwendet hat (siehe Zitat Kapitel 2.2.6).

Lisps Ideen sind nach wie vor aktuell und beeinflussen die heutige Entwicklung. Ein in der Java-Welt weit verbreitetes Hilfsmittel ist XML, welches in der Intention eng mit Lisps Fähigkeiten der Spracherweiterung zur Laufzeit verbunden ist. Paul Graham schreibt dazu:

„If you want to trick a pointy-haired boss into letting you write software in Lisp, you could try telling him it's XML." [Graham-04]

Graham weist in seinem Buch *Hackers & Painters* [Graham-04] auch auf neun bedeutende Ideen hin, die der Lisp-Entwicklung zu Grunde lagen. Von diesen Ideen wurden in der Entwicklung von Programmiersprachen immer mehr umgesetzt, sodass sich inzwischen die meisten von ihnen in dynamischen Programmiersprachen wieder finden. Doch die letzte Idee, die sich um den Wegfall der Unterscheidung zwischen der Bearbeitungs-, Übersetzungs- und Laufzeit dreht, ist bisher, von Lisp abgesehen, nicht umgesetzt worden. Interessant ist, dass dieses Konzept gerade mit *Subtext* zu neuer Bedeutung kommt.

Subtext

Subtext ist eine aktuell in der Entwicklung befindliche Art zu Programmieren, die von Jonathan Edwards am MIT entwickelt wird. Es ist keine Sprache im eigentlichen Sinne, sondern setzt, wie der Name schon andeutet, tiefer an: Subtext ist Programmieren durch Beispiele auf einer dem Syntaxbaum ähnlichen Struktur. Ein Compiler, der die Bedeutung des Codes in einen Baum extrahiert, bevor er ihn dann in ein ausführbares Programm übersetzt, ist hier überflüssig.

[23] Lisp: List Processing, http://www.lisp.org
[24] Fortran ist die erste höhere Programmiersprache. Entwickelt wurde sie Mitte der fünfziger Jahre bei IBM.
[25] Dialekte bedeuten hier sich sprachlich unterscheidende Lisp-Versionen, etwa CommonLisp, Scheme oder Dylan.

Subtext wird in Kapitel 4.4 näher vorgestellt.

Groovy

Groovy[26] ist eine dynamisch typisierte Skriptsprache, die in der letzten Zeit von der Java-Entwicklergemeinde entworfen wird. Groovy ist auf die Entwicklung von Web-Anwendungen und die Java-Plattform ausgerichtet.

Groovy läuft auf der JVM und hat damit die Chance in der Java-Welt als dynamische Sprache auf schnelle Akzeptanz zu stoßen, da die Java-Entwicklergemeinde selbst, ja sogar *Sun Microsystems*, hinter der Sprache steht. Groovy könnte die „dynamische Hintertür" für diejenigen werden, die einen Umstieg zu Sprachen wie Ruby scheuen.

Trotz dieser guten Ausgangsposition hat es Groovy noch nicht allzu weit gebracht: Die Sprache leidet unter einer inkonsistenten Syntax und ist voll von Fehlern. Es wird versucht, die meisten Fähigkeiten dynamischer Sprachen zu implementieren, aber viele dieser Punkte sind seit den frühesten Versionen unvollendet geblieben. So gut wie jede größere Beta-Version von Groovy führt mit ihren Neuerungen dazu, dass bestehende Anwendungen nicht mehr lauffähig sind. Zudem weichen immer mehr Groovy-Entwickler von den ursprünglichen Zielvorstellungen ab. So gut die Idee hinter Groovy, eine Ergänzung zu Java zu sein, auch ist, lassen all diese Schwierigkeiten an Groovys Entwicklung Zweifel aufkommen. Bruce Tate sieht Groovy aufgrund seiner Probleme solange als Versuch an, bis es eine stabile Version gibt, die sich in einer Nische des Marktes behaupten kann.

[26] http://groovy.codehaus.org/

Guy Steele [054] über Java: "And you're right: we were not out to win over the Lisp programmers; we were after the C++ programmers. We managed to drag a lot of them about halfway to Lisp. Aren't you happy?"

2.2.7 Java

Java[27], eine Entwicklung des Hardwareherstellers *Sun Microsystems*, stellt derzeit für viele Entwicklungen *die* Programmiersprache der Wahl dar und ist weit verbreitet. Aufgrund ihrer teils schlechten Eigenschaften ist Java umstritten, wird in letzter Zeit besonders intensiv diskutiert und muss starke Kritik über sich ergehen lassen.

In diesem Unterkapitel werden die wesentlichen Stärken und Schwächen von Java behandelt. (Details folgen in Kapitel 2.3.)

Javas Anfang

Die Objektorientierung, ein lange bekanntes Paradigma, war Mitte der neunziger Jahre noch nicht in aller Munde. Zu dieser Zeit entwickelte sich das Internet stark, sodass die C++ Programmierer geradezu auf neue Entwicklungen und Möglichkeiten in diesem Bereich warteten.

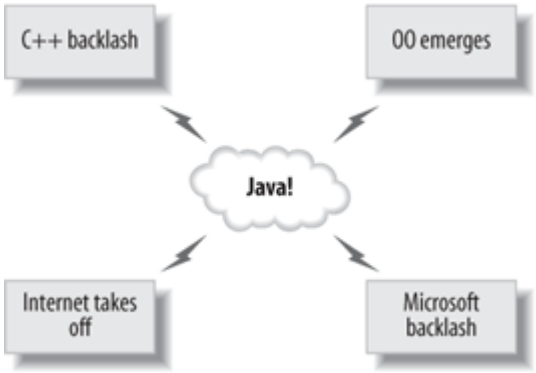

Abbildung 3: Kräfte und Entwicklungen, die um das Jahr 1995 aktuell waren, während Java entwickelt wurde.

In einer recht turbulenten Zeit (siehe Abbildung 3) ebnete Java der bestehenden C++ Entwicklergemeinde gewissermaßen den Weg ins Internet. Anfangs waren Applets[28] und später die so genannten Servlets[29] wesentliche Konzepte, die Java gegen Ende der neunziger Jahre zu großer Verbreitung verhalfen. Java gelang auch der Einzug in andere Gebiete, beispielsweise in die Handysparte mit einer mobilen Java-Version. Heute stellen datenbankbasierte Internetanwendungen ein erhebliches Einsatzgebiet Javas dar (siehe Gegenüberstellung mit Ruby in Kapitel 5.3).

Java entwickelte sich zu einer ernstzunehmenden, stabilen und universell verwendeten Programmiersprache, oder präziser: zu einer Plattform. Denn Java ist nicht allein eine Sprache sondern besitzt auch eine virtuelle Maschine (VM).

Die Java Virtual Machine (JVM)

Java etablierte das Konzept der VM auf dem Markt. Diese – für viele Java-Entwickler – vermeintliche Neuheit war schon seit langem bekannt und wurde etwa bereits mit Smalltalk erfolgreich eingesetzt.

Ein großer Vorteil der JVM liegt in der (fast vollständigen) Plattformunabhängigkeit. Diese ermöglicht es – für viele C++ Entwickler erstmalig – ein auf verschiedenen Plattformen lauffähi-

[27] http://java.sun.com/
[28] Ein Applet ist eine Art Java Programm, welches von Internetservern in den lokalen Browser geladen wird und dort ausgeführt wird.
[29] Servlets erstellen auf Web-Servern dynamische Web-Seiten, die als Antwort an Clients zurückgesendet werden.

ges Programm (im Idealfall) nur einmal übersetzen zu müssen. Ein anderer wesentlicher Vorteil ist die erhöhte Sicherheit, die grundlegend in die JVM integriert ist.

Javas Sprachfähigkeiten

"Java is C++ without the guns, knives, and clubs" - James Gosling [056]

Nach James Gosling, einem der Erfinder Javas, ist Java der Nachfolger von C++, jedoch ohne dessen spezielle Probleme. Java besitzt im Vergleich zu C++ beispielsweise eine reine Referenzsemantik, die eine automatische Speicherbereinigung ermöglicht. Dennoch gibt es wesentliche Kritikpunkte an Java:

- Java ist eine hybride Sprache. Sie besteht aus der objektorientierten Welt und einer primitiven C-Welt. Der Objektorientierung fehlen in Java die Operatoren, sodass die Operatoren der primitiven Typen genutzt werden müssen. Dies macht den Einsatz von Wrappern und Typkonversionen zwischen der Objektwelt und der primitiven C-Welt unumgänglich.
- Javas statische Typisierung steht dynamischen Konzepten und Frameworks im Weg.
- Die Metaebene ist in Java unzureichend umgesetzt. Reflection und Metaprogrammierung sind schwierig bis unmöglich umzusetzen, da Java Closures fehlen.
- Das Keep It Simple (KIS) Prinzip ist in Java nahezu nicht umgesetzt. Beispielhaft sei hier das mangelhaft umgesetzte Konzept der Generics erwähnt.
- Javas Zugänglichkeit ist mangelhaft. Laut Erik Hatcher [Tate-05] ist es kaum möglich in Java, ohne eine größere Lernleistung, die trivialsten Anwendungen schreiben zu können. Ebenso ist das Erstellen anspruchsvollerer Anwendungen ohne die Einarbeitung in gängige Werkzeuge nahezu unmöglich.

Viele Versuche, Javas Probleme zu beheben, sind in der Vergangenheit nur zögerlich oder unzureichend umgesetzt worden. Javas Syntax, die allgemein zu bürokratisch gehalten ist, wird durch manche Neuerungen immer komplexer, ohne wirklich Vorteile daraus ziehen zu können.

Der Berater Bruce Tate veranschlagt derweil für seine Kunden, die auf die Java-Plattform umsteigen wollen, ein Grundtraining von fünf Wochen. Dieser hohe Lernaufwand schreckt die meisten wechselwilligen Firmen ab, sodass diese in Java keineswegs mehr eine Alternative sehen.

Guy Steele versucht Java positiv zu sehen (siehe Zitat oben), indem er die Betrachtungsweise ändert: Java wurde nicht entwickelt, um in direkter Konkurrenz zu Lisp zu stehen – dazu ist Javas Mächtigkeit zu begrenzt – sondern um die C++ Programmierer ein Stück näher an Lisp heranzuführen. Diese befinden sich mit Java quasi auf dem Weg hin zu dynamischeren und mächtigeren Programmiersprachen. Doch Alan Kay, dessen Anspruch an Programmiersprachen Java nicht gerecht wird, sieht dies anders:

“Java and C++ make you think that the new ideas are like the old ones. Java is the most distressing thing to hit computing since MS-DOS.” [052]

Erfolgsfaktoren

Java ist als Sprache bestimmt nicht das Maß der Dinge, jedoch haben andere Faktoren zu Javas breitem Erfolg beigetragen: Java profitiert stark von seiner großen Entwicklergemeinde. Diese umfasste, als Java neu auf den Markt kam, einige kluge Köpfe, die wesentlich zu Javas Sprachentwicklung beitrugen. Inzwischen haben sich diese Entwickler jedoch anderen Projekten zugewandt. Des Weiteren erlangte Java Unterstützung durch die Entwicklungsumgebungen IntelliJ von JetBrains und das freie Eclipse-Projekt.

Der Erfolg Javas ist nicht zuletzt der Firma IBM zu verdanken, die Java durch ihre Berater immer weiter verbreitete. Viele verantwortliche Entscheidungsträger waren und sind sich sicher, mit Java zumindest nicht auf das falsche Pferd zu setzen, da sie mit ihrer Wahl nicht alleine dastehen.

“Simple things should be simple, complex things should be possible.” – Alan Kay [052]

2.3 Anforderungen an eine „gute“ Programmiersprache

Dieses Unterkapitel behandelt die Eigenschaften einer Programmiersprache, die diese mitbringen sollte, um möglichst universell[30] und produktiv eingesetzt werden zu können.

Welche Kriterien eine „gute“ Programmiersprache erfüllen muss, ist nicht so einfach zu klären. Auf diesem Gebiet gab es schon viele Irrtümer. Aus diesem Grund werden hier einige wertvolle Konzepte vorgestellt und diskutiert, die in einer „guten“ Sprache enthalten sein sollten.

Im folgenden Kapitel 2.3.1 werden zuerst die technisch relevanten Aspekte einer Sprache diskutiert. In Kapitel 2.3.2 folgen einige Punkte, die vornehmlich nicht technischer Natur sind, aber den Erfolg und die Verbreitung von Programmiersprachen beeinflussen. Abschließend wird die Produktivität der Sprachen betrachtet (siehe Kapitel 2.3.3).

Grundlegende Voraussetzungen

Das wohl wichtigste Kriterium einer Sprache, die in der Informatikbranche Anerkennung und Verwendung finden möchte, ist das *Keep It Simple* Prinzip. Es beschreibt die Einfachheit einer Programmiersprache in deren Aufbau und Verwendung:

Einfache Dinge müssen einfach zu lösen sein, komplexe Dinge müssen möglich sein. (Obiges Zitat von Alan Kay frei übersetzt.)

Laut Bruce Tate sollten zukünftig relevante Programmiersprachen eine Ausrichtung auf eine klare Definition der Sprache und eine klar strukturierte, performante Bibliothek für den Zugriff auf relationale Datenbanken haben. Ferner sollten Möglichkeiten zur Sicherung[31] von Internet- und Enterprise-Anwendungen vorhanden sein.

2.3.1 Technische Aspekte

Objektorientierung

Das Paradigma der Objektorientierung wurde in den sechziger Jahren durch Simula erfunden und spätestens durch Smalltalk in den siebziger Jahren bekannt und beliebt gemacht. Seitdem hat sich ihr Konzept als außerordentlich erfolgreich erwiesen.

Die Objektorientierung ist bis heute in populären Programmiersprachen, wie C++ oder Java, selten gut umgesetzt worden. Dennoch ist eine anhaltende Bewegung hin zur Objektorientierung unverkennbar; und unverzichtbar für eine heutige Programmiersprache.

Objektorientierung stellt hauptsächlich eine Abstraktionstechnik dar, die zur Modellierung von Softwaresystemen als Menge gekoppelter und interagierender Objekte eingesetzt wird. Die objektorientierte Programmierung (OOP) birgt, bei korrekter Anwendung, wesentliche Vorteile gegenüber der prozeduralen Programmierung. Die potentiellen Vorteile der Objektorientierung sind bessere Modellierungsmöglichkeiten durch ihr Potential zu höherer Flexibilität, Wiederverwendbarkeit und Wartbarkeit.

Doch auch die Objektorientierung hat ihre Grenzen und stellt kein Allheilmittel gegen schlechte Programmierung dar. Nicht triviale Probleme finden sich etwa bei der Konstruktion verteilter Systeme oder bei der Kopplung von Modulen wieder.

Die Objektorientierung ist zwar nicht das letzte Wort in der Softwareentwicklung, aber ein elementarer Bestandteil, der nicht wieder wegzudenken ist.

---

[30] *Verteilte Systeme* werden hier aufgrund einer deutlich anderen Ausrichtung nicht weiter berücksichtigt.
[31] Mit Sicherung ist hier Verlässlichkeit, Datenschutz und Datensicherheit gemeint.

Typsicherheit

Grundsätzlich werden Typisierungskonzepte in Programmiersprachen verwendet, um das Zusammenspiel von Operationen auf Datentypen zu regeln.

Unter Typsicherheit wird die grundlegende Eigenschaft einer Programmiersprache verstanden, die eine Verwendung der Datentypen gemäß ihrer Definition garantiert. Typsicherheit gewährleistet also in jedem Fall, dass nur legale bzw. definierte Operationen ausführbar sind. Sie wird entweder zur Übersetzungszeit (*statisch*) oder zu Laufzeit (*dynamisch*) garantiert und durch Fehlermeldungen von Typverletzungen quittiert.

Die Assemblersprache ist hier ein extremes Beispiel, da sie keine Typisierung besitzt. Sie führt weder zum Übersetzungszeitpunkt noch zur Laufzeit eine Typüberprüfung durch.

Starke und schwache Typisierung (strong versus weak typing)

Die Meinungen über das Verständnis der starken bzw. schwachen Typisierung gehen weit auseinander. Deshalb soll dieses Thema hier konkretisiert werden:

Starke oder schwache Typisierung ordnen an, welche Einschränkungen es bei der Anwendung von Operatoren auf typisierte Daten gibt. Die starke Typisierung[32] erlaubt dabei nur die Ausführung von exakt für einen Datentyp definierten Operationen. Diese Operationen sind entweder bereits im Kern der Sprache vorhanden oder werden vom Programmierer (dynamisch) hinzugefügt.

Die starke Typisierung verbietet beispielsweise die Anwendung numerischer Operationen auf booleschen Datentypen (*true* oder *false*). Explizite oder implizite Typumwandlungen (*Casts*) sind damit ausgeschlossen.

Anders sieht dies bei schwacher Typisierung aus. Dort sind explizite (d.h. manuelle) und implizite (d.h. automatische) Typumwandlungen erlaubt. In schwach typisierten Sprachen (z.B. C oder C++) können Variable implizit bzw. automatisch erweitert werden, falls diese nicht vom benötigten Typ sind (siehe Codebeispiel 5).

```
char a = 'a';
char b = 'b';
int  i = a + b;
```

Codebeispiel 5: C erweitert den Wert der Addition der Character a und b nach int.

In stark typisierten Sprachen wie Java[33], Ruby, Smalltalk oder Python, wird hingegen Kompatibilität zwischen den Operationen und Datentypen erwartet:

```
irb(main):003:0> i=1
=> 1
irb(main):004:0> puts „Wert der Variablen i: “ + i
TypeError: cannot convert Fixnum into String
    from (irb):4:in '+'
    from (irb):4
```

Codebeispiel 6: Ruby hängt eine Zahl vom Typ Fixnum nicht automatisch an einen String an.

[32] Der Begriff *starke Typisierung* ist dabei nicht automatisch gleichzusetzen mit statischer Typisierung (siehe nächste Überschrift), wenngleich es Sprachen gibt, die diese beiden Konzepte zugleich umsetzen (z.B. Java).

[33] Java und Ruby entstammen unterschiedlichen Sprachkategorien, die zwar beide stark typisiert sind, aber im Falle Javas statisch und im Falle Rubys dynamisch typisiert sind. (Siehe statische und dynamische Typisierung.)

Das Codebeispiel 6 zeigt, wie Ruby die Aktion des Konkatenierens[34] von *i* an den gegebenen String ablehnt. Ruby besitzt keine Anweisungen, um den Parameter *i* der Methode + in einen String umzuwandeln, der hier als Typ gefordert wird.

Dennoch ist es möglich, diese Aktion zum Erfolg zu führen. Ruby bietet die nötigen Sprachmittel, um seine Klassen oder Objekte zur Laufzeit in der Form zu erweitern, sodass diese Konvertierung problemlos gelingt.

Vereinfacht und kurz formuliert, entscheiden starke oder schwache Typisierung also über den Grad der Einhaltung eines Variablentyps. Starke Typisierung ist eine sinn- und wertvolle Eigenschaft einer Sprache, die zum Beispiel in Java und Ruby gleichermaßen vertreten ist.

„Compile-time checking of exception and types adds safety, but comes at a cost of additional time and syntax.“ – Bruce Tate [Tate-05]

Statische und dynamische Typisierung (static versus dynamic typing)

Eine mit der starken oder schwachen Typisierung verbundene, aber grundlegendere Art der Typisierung ist die statische oder dynamische Typisierung.

Bei statischer Typisierung wird ein bestimmter Typ fest an eine Variable gebunden. Diese Bindung muss extra angegeben werden und wird zur Übersetzungszeit überprüft. Ein Vertreter dieser Kategorie ist Java:

Die dynamische Typisierung bindet hingegen den Typ an ein Objekt und macht daher keinerlei Vorgaben über den Typ einer Variablen. Da sie keine Variablentypisierung kennt, kann sie die Typsicherheit erst zur Laufzeit garantieren. Vertreter dieser Kategorie sind etwa Smalltalk und Ruby (siehe Abbildung 4).

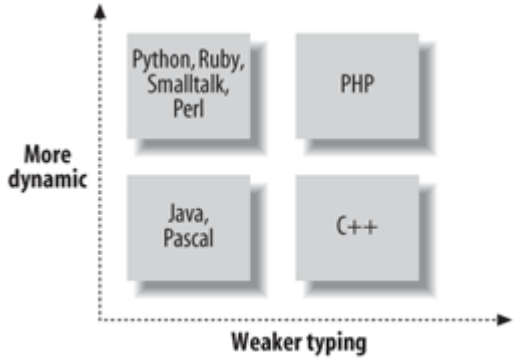

Abbildung 4: Klassifizierung der Sprachen bezüglich ihrer Typisierung

In der andauernden Diskussion um statische und dynamische Typisierung ist das häufigste Argument, das für statische Typisierung spricht, die durch die Erkennung von Typfehlern bei der Übersetzung verbundene höhere Sicherheit. Jedoch ist dieses Argument mit der Einführung von Unit-Tests zur Laufzeit nahezu hinfällig. Denn Fehler durch unpassende Typen treten relativ selten auf, sind schnell und einfach zu finden und werden durch Unit-Tests abgedeckt, die zudem noch die möglichst einwandfreie Funktion des Programms gewährleisten.

Ein Vorteil der statischen Typisierung ist die bessere Kontextunterstützung durch Entwicklungsumgebungen, die etwa aus den Rückgabetypen von Methoden Kontextinformationen sammeln können.

Statische Typisierung ist grundsätzlich ein sinnvolles Konzept. Allerdings wirft sie in der Objektorientierung Probleme auf. Objektorientierung ist hoch dynamisch und daher mit statischer Typisierung nicht immer vereinbar.

[34] Konkatenieren bedeutet hier *Anhängen*.

Um statisch typisierte und zugleich objektorientierte Programmierung umzusetzen sind Typumwandlungen bzw. Casts unumgänglich. Java und C# machen Gebrauch davon und setzen somit statische Typisierung mit dynamischer Überprüfung um. Durch die dynamische Überprüfung wird also das statische Typsystem umgangen. Dies stellt eine schlechte Lösung dar und kann leicht zu unleserlichem oder gar fehlerhaftem Code führen.

In der prozeduralen Programmierung (ohne objektorientierte Konstrukte) in Sprachen wie Pascal oder C, steht der statischen Typisierung nichts im Weg. Auch in den (objektorientierten) Bereichen der System- und Middlewareprogrammierung ist zu überlegen, ob die Vorteile der statischen Typisierung die höheren Entwicklungskosten kompensieren können.

Die Vorteile der statischen Typisierung stehen der objektorientierten Programmierung eher im Weg. Sie verursacht zusätzliche, häufig anfallende Arbeit, die Zeit kostet. Sinnvoller ist hingegen die dynamische Typisierung, die an dieser Stelle mehr Freiheiten lässt, Metaprogrammierung unterstützt und im Vergleich die Produktivität erhöht.

Vorteilhaft wäre für die Objektorientierung eine Verbindung von statischer und dynamischer Typisierung, bei der der Programmierer entscheidet, welche Typisierung er wann verwendet. Eine denkbare Ergänzung dieser Art sind die so genannten *Kontrakte*. Kontrakte stellen Überprüfungen von Variablentypen und die Einhaltung von Vor- und Nachbedingungen so wie *Invarianten*[35] sicher. Ihr Einsatz ist besonders an Modulgrenzen und Schnittstellen sinnvoll, sollte aber gut durchdacht sein.

Parametrische Polymorphie[36]

Unter parametrischer Polymorphie versteht man allgemein in der Objektorientierung, dass eine Methode, je nach Typ des Arguments bzw. Parameters, unterschiedlichen Code ausführt. Diese Methode wird in dynamisch typisierten Sprachen genau einmal geschrieben und wendet bestimmte Operationen für das übergebene Parameter-Objekt an. Je nach Klassenzugehörigkeit des Objektes wird die für den Operator auszuführende Methode bestimmt und so der tatsächlich auszuführende Code ausgewählt. Deutlich wird dies am Beispiel der Klassen *Integer* und *String*, die unterschiedliche Implementierungen des Operators + haben. Erstere addiert Zahlen, letztere hängt dagegen Zeichenketten aneinander.

Java und C++ unterstützen diese Art der parametrischen Polymorphie nicht. Sie sind durch die statische Typisierung, die eine Methode für genau einen Datentyp definiert, beschränkt. Die beiden Sprachen setzen parametrische Polymorphie *statisch* auf unterschiedliche Weise um. In C++ gibt es die so genannten *Templates*, was im Wesentlichen eine erweiterte Makrotechnik ist, und in Java werden die so genannten *Generics* dafür eingesetzt:

```
ArrayList<String> animals = new ArrayList<String>();
animals.add(“elephant”);
String elephant = animals.get(0);
```

Codebeispiel 7: Generics sollen in Java die Typsicherheit der Listen wiederherstellen.

Codebeispiel 7 [Tate-05] zeigt die Verwendung der Generics in Java. Beim Entnehmen eines Objektes aus einer Liste ist keine Typumwandlung mehr notwendig.

Doch das Konzept der Generics ist in Java schlecht umgesetzt: Die Implementierung der Generics basiert weiterhin auf der standardmäßigen *ArrayList*, die mit Objekten des Typ *Object* arbeitet. Die nötigen Typumwandlungen finden also weiterhin (im Hintergrund) statt.

Falls nun eine generische Liste auf eine nicht generische abgebildet wird[37], geht die durch die Generics neu gewonnene Typsicherheit wieder verloren, obwohl beide Referenzen der *ArrayList* auf den gleichen Speicherplatz zeigen!

[35] Eine Invariante ist eine Bedingung, die sowohl vor als auch nach einer abgeschlossenen Aktion gültig ist.
[36] Parametrische Polymorphie wird im Java-Umfeld als Generics bezeichnet.

Die Generics sind in Java nur auf Sprachebene, nicht in der VM implementiert. Sie sind deshalb eine inkonsistente unvollständige Umsetzung des Konzepts der parametrischen Polymorphie.

Feedback Loop / Code Compile Cycle

Ein immer bedeutender werdendes Problem ist das Feedback der Programmiersprachen. Die Rückmeldung über das gerade geschriebene Codestück kommt, abhängig von der verwendeten Programmiersprache, durch den teils langen Code-Compile-Zyklus erst zu spät beim Programmierer an. Vor allem bei statisch typisierten Sprachen ist dies beim Experimentieren mit Code der Fall, denn die Typisierung fällt hier negativ ins Gewicht. Erklärtes Ziel ist es, den Code-Compile-Zyklus möglichst kurz zu halten.

Durch den mühseligen Prozess des Editierens, Kompilierens, Linken/Buildens[38] und Testen sitzen zu viele Stufen zwischen dem Erstellungsprozess und dem Ausführen bzw. Testen der Software. Dynamisch typisierte Sprachen sind hier im Vorteil, da sie kein Linken/Builden erfordern.

Ein neuerer Ansatz entsteht durch das Einsparen des Compilers. Beispiele dafür finden sich im Kapitel 4 wieder, etwa Subtext oder JetBrains MPS.

Lexical Closures (auch: Closures)

Lexical Closures, in Smalltalk und Ruby auch Blöcke genannt, gehen aus Alonzo Churchs λ-Kalkül der dreißiger Jahre hervor und wurden erstmalig in Lisp als Programmiersprachen-Konstrukt umgesetzt.

Wie Abelson und Sussman in den Lambda-Papieren ([072] und [073]) zeigten, lassen sich grundsätzlich alle Kontrollstrukturen auf Lexical Closures zurückführen. Damit besitzen Programmiersprachen, welche über Closures verfügen, eine hohe Mächtigkeit. Smalltalk setzt dieses Prinzip beispielhaft um, indem es alle Kontrollstrukturen auf Closures abbildet.

In dieser Arbeit erwähnte Sprachen, die Closures unterstützen, sind: Lisp, Smalltalk, Python, Perl, Ruby und C#[39].

Eine Closure ist ein Code-Block, der bei einem Methodenaufruf als Argument übergeben wird. Dabei behält der Block die Bindungen an die Variablen aus seiner Umgebung bei. Diese Bindungen an Variable – auch freie Variable genannt – werden zum Zeitpunkt der Block-Definition erstellt. Sie fangen somit die Umgebung des Blocks ein, die sich dynamisch weiterentwickeln kann, während bei der Ausführung des Blockes die entsprechenden Objekte aus dem lexikalischen Umfeld des Blockes referenziert werden.

Ein Beispiel von Martin Fowler [011] soll dies verdeutlichen. Aus einer Liste von Arbeitnehmern (*employees*) werden die Manager herausgesucht (siehe Codebeispiele 8 und 9):

```
def managers(employees)
  return employees.select { |e| e.isManager }
end
```

Codebeispiel 8: Selektion der Manager mittels Closure in Ruby

[37] Der entsprechende Code sieht wie folgt aus: *List notGeneric = genericList;*
[38] Unter Linken/Builden versteht man das Zusammenstellen einzelner Programmmodule zu einem fertigen Produkt.
[39] Closures werden in C# erst ab der Version 2.0 durch Verwendung der so genannten Delegates unterstützt.

```
public static IList Managers(IList employees) {
  IList result = new ArrayList();
  foreach(Employee e in employees)
    if (e.IsManager) result.Add(e);
  return result;
}
```

Codebeispiel 9: Der gleiche Code in C# ohne Closure.

Der Unterschied zwischen dem Ruby- und dem C#-Code ist deutlich zu sehen. Ruby verfügt über eine eingängige Verwendung von Code-Blöcken, während in C#, wie auch in Java, eine ganze Menge Code nötig ist, um diese einfache Operation auszuformulieren.

Der Zugriff auf freie Variable ist in Java praktisch nicht möglich, da Java keine Closures unterstützt. In C# wurden die so genannten *Delegates* in der Version 2.0 hinzugefügt. Ihre Verwendung ermöglichen die Bindung der C#-Closures an freie Variable:

```
def highPaid(employees)
  threshold = 150
  return employees.select {|e| e.salary > threshold}
end
```

Codebeispiel 10: Zugriff aus einem Codeblock auf eine lokale Variable in Ruby.

```
public List<Employee> HighPaid(List<Employee> emps) {
  int threshold = 150;
  return emps.FindAll(delegate(Employee e) {
    return e.Salary > threshold;
  });
}
```

Codebeispiel 11: Zugriff aus einer C#-Closure auf eine freie Variable mittels Delegate..

Die Codebeispiele 10 und 11 [075] zeigen deutlich die natürlichere Verwendung von Closures in Ruby. Doch stehen diese einfachen Beispiele nicht stellvertretend für alle Einsatzmöglichkeiten der Closures.

Closures steigern durch ihre simplen und klaren Anwendungsmöglichkeiten die Produktivität in einer für Java-Programmierer bisher unbekannten Weise.

Metaprogrammierung / Metaprogramming

Metaprogrammierung ist *"programming your programming"*, so Glenn Vanderburg [045]. Sie ist ein Weg, die Art und Weise der Programmierung in einer Programmiersprache selbst zu verändern.

Durch Metaprogrammierung entstehen neue, nicht zu unterschätzende Möglichkeiten der Problemlösung. Sprachen lassen sich durch diese Technik entsprechend den eigenen Bedürfnissen anpassen, oder sogar zu einer eigenen Sprache weiterentwickeln (siehe Kapitel 4).

Codebeispiel 12 zeigt zur Verdeutlichung von Metaprogrammierung eine beispielhafte Implementierung von Gettern in Ruby:

```
class Module
  def attr_reader (*syms)
    syms.each do |syms|           # Arbeite Attributliste ab
      class_eval %{def #{sym}     # Definiere Getter-Methode
                     @#{sym}      # Rückgabewert: Attribut
                   end}
    end
  end
end
```

Codebeispiel 12: Metaprogrammierung in Ruby. Als Parameter werden diejenigen Attribute übergeben, die Getter erhalten sollen. Ruby fügt der Klasse für diese Attribute Getter-Methoden hinzu.

Zur Erklärung: Um die Getter *name* und *address* zu implementieren reicht der Befehl: *attr_reader :name :address*. Die Parameter *name* und *address* dieses Aufrufs werden im Codebeispiel 12 in einer Schleife abgearbeitet, die in jedem Durchlauf einen Getter definiert. Der fettgedruckte Code ist die eigentliche Definition, die durch Metaprogrammierung erzeugt wird. *sym* steht dabei stellvertretend für *name* bzw. *address*.

Metaprogrammierung basiert auf Closures und hat ihre frühen Ursprünge in Lisp. Ebenfalls starke Vertreter der Metaprogrammierung sind Smalltalk, Python und Ruby; in letzterem erlebt die Technik gerade eine Renaissance. Paul Graham [045] hat treffend beschrieben, wie Metaprogrammierung – hier stellvertretend in Lisp – funktioniert:

„In Lisp, you don't just write your program down toward the language, you also build the language up toward your program."

Reflection

Reflection ist eine Form der Metaprogrammierung. Jedoch geht es bei Reflection darum, dass ein Programm Informationen über seine eigene Struktur erlangen kann. In der Objektorientierung bietet Reflection zum Beispiel die Funktionalität an, Informationen über das eigene Klassenmodell oder dessen Instanzen abzufragen.

Java hat hier wieder einmal damit zu kämpfen, dass es zu viele verschiedenartige Typen besitzt, die bei Reflections eine unterschiedliche Behandelung erfordern.

Die Codebeispiele 13 und 14, entnommen aus [Tate-05], zeigen Reflections und machen die ungleiche Komplexität der hier verwendeten Programmiersprachen deutlich:

```
irb(main):001:0> i = 4
=> 4
irb(main):002:0> i.class
=> Fixnum
irb(main):003:0> i.methods.each {|m| puts m}
```

Codebeispiel 13: Der Ruby-Code zur Ausgabe der Methoden eines Objekts.

```
public static void printMethods(Object obj) throws Exception
{
  Class cls = obj.getClass();
  Method[] methods = cls.getDeclaredMethods();
  for (int i=0, I < methods.length, i++) {
    System.out.println("Method name:" + method.getName());
  }
}
```

Codebeispiel 14: Der Java-Code zur Ausgabe der Methoden eines Objekts.[40]

Orthogonalität

Der Begriff der Orthogonalität ist aus der Geometrie entnommen[41]. In der Informatik stellt Orthogonalität eine Metapher für Programmiersprachen dar.

Eine Programmiersprache erlangt durch verschiedene Konzepte einen bestimmten Funktionsumfang. Dieser Umfang lässt sich, übertragen auf die Geometrie, auch als ausdrückbarer Raum bezeichnen. Ein Raum wird in der Geometrie durch Vektoren aufgespannt, die voneinander linear unabhängig bzw. orthogonal zueinander sind. Jeder Vektor gibt dem Raum eine unterschiedliche Eigenschaft, auch Dimension genannt.

Würde man für einen Raum nicht orthogonale bzw. linear abhängige Vektoren verwenden, so würden diese die gleiche Dimension beschreiben. Sie wären also redundant.

Übertragen auf Programmiersprachen bedeutet dies, dass jede Programmiersprache, die aus verschiedenen, voneinander unabhängigen, Konzepten aufgebaut ist, orthogonal ist.

Der Verzicht auf eines dieser Konzepte (bzw. eines linear unabhängigen Vektors) entspräche einer Verkleinerung der Ausdrucksfähigkeiten der Programmiersprache. Damit geht eine Reduktion der Dimension des Raums einher, der durch die Konzepte einer Programmiersprache ausgedrückt wird.

Konkret bedeutet dies: Für die Programmiersprache Lisp wird dieser Raum durch die Lambdafunktion aufgespannt (siehe Closures). Smalltalk wird durch die Objektorientierung, Message Passing[42] und Closures definiert.

Kurz: Orthogonalität beschreibt die Kombinierbarkeit voneinander unabhängiger Konzepte. Sie stellt damit eine grundlegende Anforderung an die vorhandenen Bibliotheken und Konzepte einer Programmiersprache dar.

Java und C++ sind in großem Umfang nicht orthogonal, da sie auf vielen redundanten Konzepten beruhen. Beispielhaft sei das *java.util.Array* Interface erwähnt, welches mehr als 70 Methoden aufweist, deren tatsächlicher Funktionsumfang aber nur rund 10 Methoden umfasst.
Ein orthogonales Beispiel stellt die aspektorientierte Programmierung dar (siehe Kapitel 3).

[40] Die Java-Variante lässt die unterschiedlichen Parametersignaturen der Methoden außer Acht. Dafür sind weitere vier Zeilen Code nötig.
[41] In der Mathematik sind zwei Linien bzw. Vektoren orthogonal, wenn sie rechtwinklig zueinander stehen. Dies entspricht einem kartesischen Koordinatensystem.
[42] Unter Message-Passing versteht man in der Objektorientierung den Nachrichtenaustausch zwischen Objekten.

Unklare Semantik

Die Semantik eines Programmteils wird nicht durch den Code allein deutlich. Dies zeigt das aus [046] entnommene Codebeispiel 15:

```
public interface Stack {
  public int size();
  public boolean isEmpty();
  public void push(Object element);
  public Object pop()
    throws EmptyStackException;
}
```

```
public interface Queue {
  public int size();
  public boolean isEmpty();
  public void put(Object element);
  public Object get()
    throws EmptyQueueException;
}
```

Codebeispiel 15: Der Semantik-Unterschied zwischen Stack und Queue ist unklar.

Die Interfaces der weitläufig bekannten Behälter *Stack* und *Queue* ähneln sich sehr: Die Methoden *size* und *isEmpty* sind selbsterklärend und unterscheiden sich in ihrer Funktion nicht. Doch die Methoden für das Hinzufügen von Objekten (*push* und *put*), sowie die zu Entnahme (*pop* und *get*) unterscheiden sich. Sie besitzen eine grundlegend andere Funktionsweise, die den *Stack* von einer *Queue* unterscheidet. Doch dieser Unterschied wird durch das, in der Interfacebeschreibung verwendete, Abstraktionsniveau nicht klar ausgedrückt.

Unklare Semantik stellt einen Aspekt der Modellierungslücke heutiger Programmiersprachen dar. Abhilfe schaffen hier weniger neue Programmiersprachen, sondern vielmehr eine gute Dokumentation oder neue Programmierparadigmen wie das Language Oriented Programming (siehe Kapitel 4).

Codegenerierung (Code Generation)

Codegenerierung ist eine Abstraktionstechnik der Programmierung. Sie ermöglicht es Wiederholungen zu vermeiden und entspricht dem *Don't Repeat Yourself* Prinzip [Hunt-99]. Dabei werden Ideen oder Konzepte an einer Stelle definiert und an verschiedenen Stellen in einem Softwaresystem eingesetzt. Die Techniken der Codegenerierung sind ein wesentlicher Bestandteil domänenspezifischer Sprachen (siehe Kapitel 4).

Der Einsatz von Codegenerierung entbindet Programmierer nicht davon, den generierten Code zu verstehen.

2.3.2 Rahmenbedingungen und Unterstützung

Akzeptanz

Akzeptanz ist ein bedeutender Faktor für den Erfolg und die Unterstützung einer Programmiersprache. Hier erwies sich Javas Ansatz, die C++ Entwickler mit einzubeziehen, als wichtige und einflussreiche Entscheidung.

Um Erfolg zu haben muss eine Programmiersprache eine gewisse Attraktivität mitbringen, die Entwicklern Grund genug gibt, sie einzusetzen. Sympathie ist dafür sicher nicht das richtige Argument. Klarheit und Verständlichkeit der Sprache sind hier von großer Bedeutung, denn eine Sprache mit einer steilen Lernkurve stellt zu hohe Anforderungen an Einsteiger, bevor mit ihr produktiv gearbeitet werden kann. Sie ist damit schlecht zugänglich und entspricht nicht dem *Keep It Simple* Prinzip, welches beispielsweise Smalltalk und Ruby umsetzen.

Eine benutzerfreundliche, produktive Syntax und ein vertrautes Objektmodell der Programmiersprache zeichnen eine gute Zugänglichkeit aus.

In Anlehnung an R.P. Gabriel [071] muss eine Programmiersprache auch abstrakt genug sein, um Problemen auf einer entsprechenden Ebene gegenüber zu stehen. C ist beispielsweise für viele Problembereiche aufgrund niedriger Abstraktheit schlecht geeignet. Wenngleich C auf-

grund geringer mathematischer Anforderungen (besonders im Vergleich zu Lisp) weit verbreitet ist.

Um sich behaupten zu können, muss eine Sprache schließlich auch die Möglichkeit bieten, mit ihr Geld zu verdienen. Denn kein professioneller Ruby-Programmierer könnte überleben, wenn ausschließlich Java-Programmierer gesucht werden würden! Es müssen also Möglichkeiten und Gründe für den professionellen Einsatz einer Programmiersprache vorhanden sein, damit sich deren Einsatz, auch im Sinne des Verdiensts, lohnt.

Neben der (Open-Source-) Community, die im nächsten Punkt erwähnt wird, sollte eine Sprache „politisch korrekt“ sein. Dies beschreibt eine Sprache, die nicht von einem großen Unternehmen kommt, sondern Open Source ist und damit bessere Aussichten aufweist, akzeptiert zu werden. Mit ihr sind keine politischen Interessen verbunden, (Sprach-)Standards können sich unabhängig entwickeln und die Sprache kann frei eingesetzt werden, ohne Bindung an einen Hersteller oder bestimmte Betriebssysteme.

Open Source / Community

Open Source beschreibt an sich die Offenheit des Quellcodes. Diese enthält die Erlaubnis zum Weitergeben, Lesen und Verändern der Programme. Diese Möglichkeiten bieten großes Potential für die Nutzer einer Programmiersprache. Verschiedene Softwareprojekte wie Eclipse, OpenOffice[43] oder Firefox[44] haben bereits bewiesen, wie erfolgreich Open-Source-Software sein kann.

Nicht nur Softwareprodukte, auch Programmiersprachen, insbesondere Java, profitieren von der Open-Source-Community, etwa durch diverse populäre Frameworks. Viele freie Entwickler oder auch Firmen, die die Open-Source-Produkte selbst einsetzen, tragen mit ihren Entwicklern dazu bei, die Projekte voranzubringen und die Qualität zu sichern.

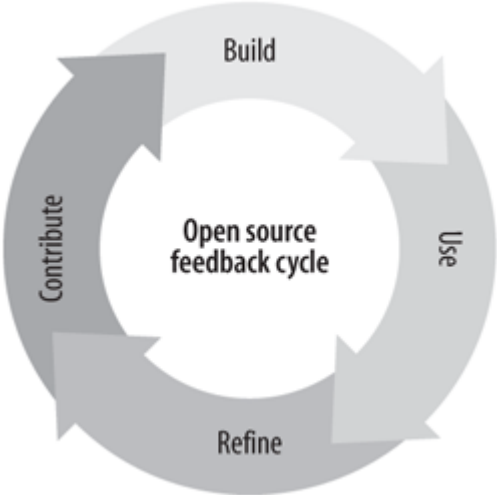

Abbildung 5: Der Open-Source-Feedback-Cycle ist ein gutes Modell, welches die Funktionsweise von Open Source beschreibt.

Es bilden sich zudem in der Regel Gruppen (engl. *Communities*) um eine Programmiersprache, in denen man sich gegenseitig bei Problemen hilft. Diese Communities werden durch Open Source gestärkt und bieten in der Regel Support für eine Open-Source-Sprache an.

Nach R.P. Gabriel sind Communities der Ort, an dem der soziale Prozess der Akzeptanz einer Sprache – unabhängig von ihrer technischen Eignung – stattfindet. Hier spielen Ähnlichkeiten zu bekannten und beliebten Sprachen eine große Rolle, z.B. eine an C angelehnte Syntax.

[43] http://www.openoffice.org/
[44] http://www.mozilla-europe.org/de/products/firefox/

Entwicklungsumgebungen / IDE (Integrated Development Environment)

Entwicklungsumgebungen sind eine wesentliche Hilfe für Entwickler, indem sie die Programmierung unterstützen. Abhängig von der eingesetzten Programmiersprache, deren Komplexität und IDE-Integration, stellen Entwicklungsumgebungen ein nahezu unverzichtbares Hilfsmittel dar. Dies trifft insbesondere auf Java zu. Ruby belegt hier beispielhaft, das die Programmierung auch ohne IDE gelingen kann.

Dokumentation

"Your check is your receipt. Your source code is your documentation" – Rob Griffin [056]

Dokumentation ist in zweierlei Hinsicht für die Zugänglichkeit einer Programmiersprache entscheidend: Zum einen sollten genügend Einführungen in Form von guten Tutorien oder Büchern vorhanden sein. Zum anderen ist eine gute Dokumentation des Quellcodes und der Bibliotheken für die (Wieder-)Verwendbarkeit und Wartung wichtig.

Eine optimale Dokumentation ist weder von fehlenden Kommentaren noch von größeren Textpassagen geprägt. Der Schlüssel steckt in der Sprache selbst: Was in ihr leicht auszudrücken ist, ist selbsterklärend und braucht nicht kommentiert zu werden.

Erklärungen und Spezifikationen sind „nur" Kommentare, sie befinden sich auf einer höheren Ebene als der Code und tragen damit zur Modellierungslücke bei. Sie sind nur dann einzusetzen, wenn die Bedeutung durch spracheigene Mittel nicht auszudrücken ist. Diese Ausdrucksschwäche ist vor allem bei schwierigeren Zusammenhängen oder schlecht strukturierten Programmen gegeben. Abhilfe schaffen hier meist ein *Refactoring* oder eine Anpassung des Sprachniveaus (siehe Kapitel 4).

Killer-Applikation

„Eine Killer-Applikation (engl. killer application) ist eine konkrete Anwendung, die einer neuen Technologie zum Durchbruch verhilft. Der Begriff leitet sich daher ab, dass konkurrierende, ähnliche und oft ältere Technologien schnell verdrängt werden." – Wikipedia [078]

Der große Nutzen einer Killer-Applikation vermag es, eine Technologie bekannt zu machen, die ohne sie nie zu einer solch großen Bedeutung gekommen wäre. In der Welt der Programmiersprachen sind Killer-Applikationen mittlerweile ein wichtiges Mittel um neue Sprachen oder Programmierparadigmen bekannt zu machen und ihnen zum Durchbruch zu verhelfen.

Durch Javas Erfolg und Marktbedeutung sind die Anforderungen an einen etwaigen Nachfolger unterdessen hoch anzusetzen. Eine Killer-Applikation als Katalysator ist quasi Bedingung, um überhaupt den nötigen Bekanntheitsgrad für ein Bestehen auf dem Markt zu erlangen.

Eine Killer-Applikation, die aktuell große Aufmerksamkeit auf sich zieht, ist *Ruby on Rails*. Ruby on Rails ist ein gut gestaltetes und derartig produktives Framework, dass es der ihm zugrunde liegenden Technologie – der Sprache Ruby – den Weg hin zu größerer Aufmerksamkeit ebnet (siehe Kapitel 5.3).

2.3.3 Produktivität

Produktivität beschreibt den umsetzbaren Funktionsumfang pro Zeiteinheit (siehe H. Krasemann [076]). Der Umfang bzw. die Produktivität hängt wesentlich von der eingesetzten Programmiersprache und deren Ausrichtung auf das Aufgabengebiet ab.

Aufgrund der Abhängigkeit der Produktivität von der gewählten Sprache spielen viele der bisher erwähnten Aspekte eine Rolle. Eine „gute" und mächtige Programmiersprache allein ist nicht

ausreichend, um produktiv zu sein. Die Fähigkeiten der Programmierer im Umgang mit der Sprache sind von großer von Bedeutung[45].

Es existieren unterschiedliche Bewertungsverfahren der Produktivität, die alle auf die eine oder andere Art korrekt sind. Es gibt beispielsweise Ansätze, die als Bewertungskriterium die Speicherauslastung, die Laufzeit eines Programms oder die Ausrichtung der Sprache (auf das jeweilige Fachgebiet) wählen.

Als recht sinnvolles und umsetzbares Kriterium haben sich die Codezeilen (*Lines of Code, LOC*) pro Aufgabeneinheit (*Function Point, FP*) erwiesen (siehe [Graham-04]). Dieses Kriterium liefert kein allgemein gültiges Ergebnis, doch ist es gleichwohl als Orientierungshilfe geeignet.

Konkrete Zahlen sind schwer zu finden, doch um überhaupt Daten nennen zu können, hier stellvertretend eine Tabelle von C. Jones [077] aus dem Jahr 1996:

Sprache	**Produktivitätslevel**	**durchschnittliche LOC je FP**
Assembler	1	320
C	2,5	128
Fortran	3	107
Lisp	5	64
C++, Java	6	53
Perl, Smalltalk	15	21
SQL	25	13
Excel	57	6

Tabelle 1: Darstellung der durchschnittlichen Produktivität verschiedener Sprachen.

Die Darstellung der Produktivität in Tabelle 1 bedarf einiger Erklärung:

Die Codezeilen je Funktionspunkt (*LOC je FP*) stellen ein Maß für die Funktionalität der Programmiersprache dar. Von den LOC abgeleitet ist der Produktivitätslevel, der eine Einteilung der Produktivität von Sprachen in verschiedene Stufen bzw. Klassen darstellt. Diese Aussagen sind recht allgemein gehalten und treten daher häufig nicht in dieser Art ein.

Zu den einzelnen Sprachen im Vergleich untereinander sind noch einige Ergänzungen und Kommentare nötig:

- C#, hier nicht erfasst, sollte in etwa ein wenig besser platziert sein als Java.
- Python liegt etwa gleich auf mit Perl.
- Smalltalk ist in der Regel produktiver als Perl, z.B. aufgrund seiner Umsetzung einiger Sprachkonzepte.
- Java ist i.d.R. um den Faktor 3 produktiver als C++. Dies ist in den schon erwähnten Spracheigenschaften begründet.
- Lisps Stärken scheinen hier nahezu unberücksichtigt zu sein. Paul Graham [Graham-04] berichtet darüber, dass Lisp sehr produktiv bzw. weitaus produktiver als Java ist.
- Excel und SQL sind hier als Vertreter domänenspezifischer Sprachen aufgeführt. Sie sind daher nicht direkt mit den anderen Programmiersprachen zu vergleichen. (Näheres folgt in Kapitel 4.)

Für den eingeschränkten, aber doch bedeutenden Bereich der Programmierung von Web-Anwendungen liefert Bruce Tate einige aussagekräftige Metriken. Diese Daten ermittelte B. Tate in einem seiner Projekte, welches zuerst in Java geschrieben wurde, danach aber nach Ruby portiert wurde. So gesehen, konnten bei der Ruby-Variante einige Überlegungen aus der vorhandenen Java-Implementierung übernommen werden, sodass die erhöhte Produktivität zu einem ge-

[45] Dies belegen Untersuchungen [067] und R.P. Gabriels Artikel [071].

ringen Teil auf diese Überlegungen zurückgeht. Dies ändert aber nichts an der grundsätzlichen Aussage der Daten (siehe Tabelle 2):

Metrik	**Java mit Spring, Hibernate**	**Ruby mit Rails**
Produkteinführungszeit	4 Monate (~ 20 Std./Woche)	4 Tage (5 Std./ Tag)
Lines of code (LOC, Codezeilen)	3.293	1.164
Lines of configuration (Konfigurationszeilen)	1.161	113
Anzahl an Klassen/Methoden	62/549	55/126

Tabelle 2: Darstellung der höheren Produktivität von Ruby durch den Vergleich eines Projektes in Java und Ruby.

Im Allgemeinen ist Ruby etwa um 3- bis 5-mal produktiver als Java. Setzt man etwa Ruby on Rails ein, kann die Produktivität gegenüber Java (mit ihren Frameworks) noch erheblicher gesteigert werden.

In der Regel ist Java deutlich produktiver als C und C++. Alle drei Sprachen liegen in ihrer Produktivität zumeist klar unter Sprachen wie Perl und Python, und diese vorwiegend unter Smalltalk, Ruby und Lisp.

"There is no programming language, no matter how structured, that will prevent programmers from making bad programs." – Larry Flon [056]

2.4 Zusammenfassung

Die wichtigsten Programmiersprachen, Anforderungen und Probleme wurden in diesem Kapitel angesprochen.

Die Sprache Java hat größere Probleme. Ihre Zugänglichkeit leidet durch unglückliche Design-Entscheidungen und die inzwischen zur Programmierung nötigen Hilfsmittel.

Javas virtuelle Maschine bildet eine erfolgreiche technische Grundlage, die mit den technischen und konstruktiven Schwächen der Sprache zu kämpfen hat. Suns Haltung steht einem wirklichen Fortschritt entgegen. Anstatt die virtuelle Maschine zu verbessern, versucht Sun Probleme lieber auf der Sprachebene zu lösen, die – wie bereits erwähnt – von geringem Erfolg gekrönt sind. Sun hält daran fest, dass Java und die JVM eng miteinander verbunden sein müssen. Doch dieser Eindruck trügt, wie die Entwicklung von Groovy oder JRuby deutlich macht.

Trotzdem hat Java (durch seine Verbreitung) hohe Hürden für potentielle Nachfolger gesetzt. Viele Unternehmen haben einiges in die Java Infrastruktur investiert. Zwischenzeitlich zeichnen sich jedoch ernstzunehmende Alternativen ab.

Abseits dieser rein technischen Kriterien ist die Akzeptanz von Programmiersprachen ein wichtiger Faktor. Richard P. Gabriel schreibt in einem Artikel [071] über Sprachen, die ein höheres mathematisches Verständnis erfordern. Er erklärt die große Verbreitung der Sprache C und begründet diese darin, dass C eine geringe Komplexität aufweist und die C-Programmierung keine mathematische Kompetenz verlangt.

Demnach ist die Verbreitung einer Programmiersprache umso größer, je ähnlicher sie der Sprache C ist. Dies wäre beispielsweise eine Erklärung den Erfolg von Java und den Misserfolg von Sprachen wie Smalltalk.

Veränderungen in der Entwicklung aller Sprachen erfordern Einfachheit, da viele Dinge schon kompliziert genug sind und nicht noch komplexer werden dürfen. Eine einfache, unkomplizierte Syntax und ein kurzer Feedback-Zyklus werden als Charakteristiken einer Sprache immer wichtiger. Beide Eigenschaften werden durch konsistente, dynamische Sprachen unterstützt, die dem *Keep It Simple* Prinzip entsprechen.

Von den angesprochenen Programmiersprachen stellt keine die ideale Antwort auf die Frage nach einer universellen Programmiersprache, die allen Anforderungen gewachsen ist, dar. Einige Kandidaten sind gute Alternativen zu Java, aber nur Ruby hat das Zeug, sich auf dem Markt zu etablieren – nicht zuletzt dank der Killer-Applikation Ruby on Rails.

Außer Ruby spielen auch die Sprachen Perl, Python und PHP eine tragende Rolle in der so genannten *Web2.0*-Bewegung, die Tim O'Reilly in einem Artikel beschreibt [074].

Java ist bestimmt noch nicht am Ende, weist aber deutliche Unzulänglichkeiten auf, die Javas zukünftige Entwicklung beeinträchtigen. Ob Ruby nun die nächste Sprache darstellt oder nicht, wird sich zeigen. Jedenfalls stehen die Chancen nicht schlecht, da Ruby sehr produktiv, nicht kommerziell und für viele Einsatzzwecke gut geeignet ist.

Wer eine noch höhere Produktivität für einen begrenzten Problembereich benötigt, kann domänenspezifische Sprachen selbst entwickeln und bauen (siehe Kapitel 4).

Doch, wie Larry Flon feststellt (siehe Zitat oben), egal wie gut eine Programmiersprache (strukturiert) ist, sie wird Programmierer nicht davon abhalten, schlechte Programme zu entwickeln.

“In Lisp, if you want to do aspect-oriented programming, you just do a bunch of macros and you're there. In Java, you have to get Gregor Kiczales to go out and start a new company, taking months and years and try to get that to work. Lisp still has the advantage there, it's just a question of people wanting that.” – Peter Norvig [054]

3 Aspektorientierte Programmierung

Neben Paradigmen, die rein auf die Programmiersprachen bezogen sind, gibt es noch andere Entwicklungslinien. Eine davon ist die Aspektorientierung, die in diesem Kapitel kurz vorgestellt werden soll. Dabei wird weniger auf die Details der Implementierung eingegangen, sondern das Paradigma an sich behandelt.

3.1 Vorstellung

Die aspektorientierte Programmierung (engl. *Aspect Oriented Programming, AOP*) ist eine im Jahr 1997 von Gregor Kiczales am Xerox PARC entwickelte Programmiertechnik. Genauer gesagt ist AOP ein Programmierparadigma, welches die Modularität von Programmen in Design und Implementierung auf eine neuartige Weise unterstützt. AOP versteht sich dabei als Erweiterung bzw. Ergänzung der objektorientierten Programmierung.

Konzept

AOP ermöglicht die Modularisierung von querschneidenden Belangen eines Systems. Querschneidende Belange (engl. *Crosscutting Concerns*) sind modulübergreifende Anwendungsbestandteile, die an viele verschiedene Stellen im Quelltext der Anwendung verteilt sind. AOP versucht nun, diese räumlich getrennten, modulübergreifenden Bestandteile an einer zentralen Stelle zusammenzufassen bzw. zu modularisieren.

Die Implementierung eines modulübergreifenden Anliegens wird *Aspekt* genannt. Sind diese Aspekte nun an einem Ort gekapselt, können sie an den verschiedenen Stellen des Systems eingesetzt werden. Diese Technik wird *Crosscutting* (zu deutsch: Querschneiden) genannt. Mittels Crosscutting werden vorwiegend technische Belange eines Systems[46] als Aspekte umgesetzt und somit modularisiert.

Vor dem Kompilieren der Anwendung werden die Aspekte mit dem Anwendungscode verbunden. Dieser Vorgang wird *Weben* (engl. *Weaving*) genannt und von einem so genannten Aspekt-Weber – gewöhnlich ein Präprozessor –ausgeführt (siehe Abbildung 6).

[46] Als Beispiel sei hier Logging genannt, das die Vorgangsprotokollierung einer Anwendung bezeichnet.

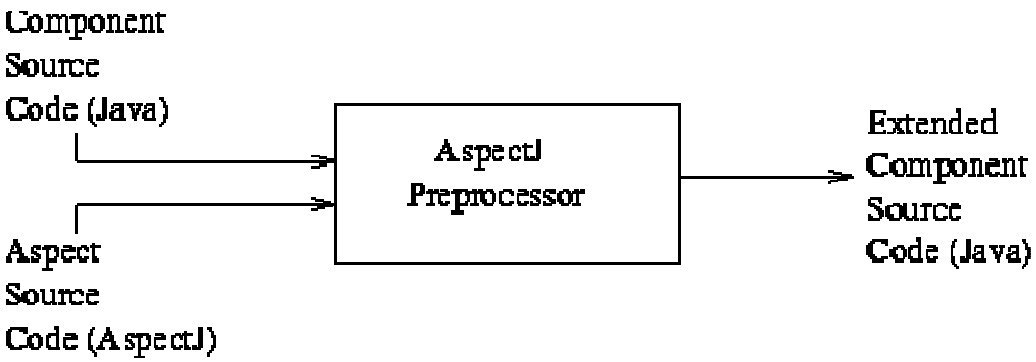

Abbildung 6: Weaving von Java- und AspectJ-Code

AOP ist ein orthogonales Konzept, welches i.d.R. nur die Teile des Verhaltens einer Anwendung beschreibt, die nicht zu deren Hauptfunktionalität gehören. Die Hauptfunktionalität wird auch als *fachliche Belange* bezeichnet. Die fachlichen Belange werden in der Regel mittels Objektorientierung modelliert und durch Klassen und Objekte (in Module) gekapselt.

Die *technischen Belange* können nun, wie eben erwähnt, durch die AOP mittels Crosscutting gekapselt werden.

Um modulübergreifende Anliegen als Aspekte (modularisiert) beschreiben zu können, werden verschiedene Sprachmittel benötigt, die in der Aspektorientierung durch die Programmiersprache selbst unterstützt werden.

Grundsätzlich ist der Funktionsumfang der AOP auch mit der OOP zu realisieren, nur ist dies viel aufwändiger. Sprachen wie Java werden an dieser Stelle durch eine Spracherweiterung ergänzt. Ruby bringt große Teile des dazu nötigen Werkzeugs von Haus aus mit.

AOP verwendet Metaprogrammierung, in dem sie Methoden und Klassen erweitert, ohne sie zu verändern. Diese Veränderung wird durch die Aspekte realisiert. Zugleich stellt die aspektorientierte Programmierung eine domänenspezifische Sprache dar, da sie auf ein spezielles Fachgebiet ausgerichtet ist (vgl. Kapitel 4).

Nutzen und Einsatzgebiete

Mit dem Einsatz der aspektorientierten Programmierung lässt sich die Komplexität von Softwaresystemen reduzieren. Die Übersichtlichkeit und Transparenz sowie die Wartbarkeit werden durch die Modularisierungsmöglichkeiten der AOP verbessert. Dabei besteht eine Trennung zwischen den Aspekten, die die querschneidenden, technischen Belange implementieren und den restlichen Komponenten der Anwendung, welche die eigentliche Aufgabenbewältigung[47] des Systems übernehmen. Änderungen an den gekapselten Aspekten können einfach und zügig an einer Stelle durchgeführt werden. Dadurch wird eine Zeit- und Kostenersparnis erzielt, die eine Steigerung der Produktivität nach sich zieht.

Prinzipiell bedeutet diese Aufteilung in fachliche und technische Belange, dass eine Anwendung sogar ohne ihre Aspekte eingeschränkt lauffähig ist.[48]

Je nach bereits vorhandenen Mitteln einer Sprache zur Modularisierung von Anliegen, können die durch die Aspektorientierung eingeführten Modularisierungsarten zur Steigerung der Softwarequalität beitragen. Dabei spielen auch optionale Architektur-Überprüfungen eine Rolle.

Querschneidende Belange, die den Einsatzbereich von aspektorientierter Programmierung ausmachen, sind typischerweise:

- *Fehlerbehandlung*: Erkennung und Behandlung von Fehlern.

[47] Auch als Geschäftslogik bezeichnet.
[48] Falls fachliche Belange durch Aspekte realisiert werden, etwa wegen einer besseren Programmstruktur, sind die Aspekte hier natürlich nicht optional. Sie gehören in diesem Fall fest zum System.

- *Sicherheit*: Authentifizierung und Autorisierung über ein gesamtes System lokal gekapselt.
- *Protokollierung*: Tracing[49], Logging und Monitoring[50] sind leicht zentral implementierbar.
- *Performance*: Eine gezielte Optimierung der Geschwindigkeit ist durch vereinfachtes Profiling[51] möglich.
- *Nebenläufigkeit*: Nebenläufige Transaktionen[52] können voneinander abgegrenzt werden, um eine gegenseitige Beeinflussung etwa durch gemeinsam genutzte Ressourcen zu vermeiden.
- *Plausibilität*: Plausibilitätsprüfung bzw. Verifikation durch Aspekte, getrennt vom fachlichen Code. Auch Vor- und Nachbedingungen sowie Invarianten lassen sich einbauen.
- *Persistenz*: Einfachere zentrale Implementierung von Persistenzmechanismen.
- *Architektur*: Durch zusätzliche Definitionen lassen sich Beschränkungen oder Rahmenbedingungen für die Softwarearchitektur festlegen. Somit können Verletzungen der Architekturprinzipien einer Anwendung bei der Erstellung vermieden werden.

3.2 AspectJ und andere Spracherweiterungen

Aspektorientierte Programmierung besitzt noch keinen allzu großen Stellenwert, wird aber im Laufe der Zeit immer bekannter und häufiger eingesetzt. Derweil gibt es für die meisten objektorientierten Sprachen Erweiterungen, die die Sprachmittel für die aspektorientierte Programmierung zur Verfügung stellen.

Einer der bekanntesten Vertreter der AOP ist *AspectJ*, eine aspektorientierte Erweiterung für Java. Neben AspectJ gibt es für Java noch *HyperJ*. Die beiden Produkte unterscheiden sich grundsätzlich in der Art des Webens. AspectJ verwendet *statisches Weben*, während HyperJ *dynamisch webt*.

Das statische Weben verbindet den Aspektcode durch einen Präprozessor fest mit dem Programmcode. Dabei ist eine Optimierung des Codes möglich. Beim dynamischen Weben findet diese unveränderliche Verbindung nicht statt, sondern geschieht zur Laufzeit. Im Vergleich zum statischen Weben hat dies den Vorteil, dass auf Veränderungen der Aspekte zügig reagiert werden kann. Dies zieht allerdings eingeschränkte Möglichkeiten zur Codeoptimierung nach sich, erspart aber dafür Verwebungsvorgänge. Je nach Einsatzkriterien sollte an dieser Stelle abgewogen werden, welche Variante zum Einsatz kommt.

Nachfolgend aufgeführt sind einige ausgewählte Produkte der aspektorientierten Programmierung:

- *AspectC++* [061] ist eine Spracherweiterung für C++, welche in einer aktuellen Version vorliegt.
- *LOOM.NET*[53] ist eine Spracherweiterung für die Microsoft .NET-Plattform, die einen statischen und dynamischen Weaver enthält.
- *AspectOrientedRuby* [064] ist die aktuelle Ruby Variante für AOP. Das Projekt AspectR [063], dessen letztes Update über vier Jahre zurückliegt, scheint gescheitert zu sein.
- *AspectWerkz*[54] stellt eine dynamische AOP-Erweiterung für Java dar, die mittlerweile in der Version 2 vorliegt.

[49] Tracing entspricht der Ablaufverfolgung.
[50] Monitoring bezeichnet Überwachungsfunktionen.
[51] Als Profiling bezeichnet man Auswertung und Statistiken bezüglich der Geschwindigkeit einer Anwendung.
[52] Eine Transaktion beschreibt eine zusammengehörige Folge von Operationen, die entweder ganz oder gar nicht ausgeführt werden.
[53] http://www.dcl.hpi.uni-potsdam.de/research/loom/

- *JFluid*[55] wird von Sun produziert. Es ist eine dynamische AOP-Variante für Java, die mittlerweile den Beta-Status überstanden hat und nun in die so genannten *Sun Tools* integriert werden soll.
- *HyperJ*[56] (auch Hyper/J) ist ein Werkzeug von IBM, welches dynamische AOP-Unterstützung für Java bietet. Es ist im Vergleich zu AspectJ wesentlich mächtiger, aber auch entsprechend komplizierter, sodass auch der Einarbeitungsaufwand erheblich größer ist. Im Gegensatz zu AspectJ arbeitet HyperJ nur auf der Ebene des Bytecodes der JVM. Aufgrund der erheblichen Komplexität des HyperJ-Ansatzes wird hier nicht weiter auf diese Erweiterung eingegangen.
- *AspectJ* [062] ist die populärste Spracherweiterung der AOP. Sie stellt eine statische AOP-Unterstützung für Java bereit und wird im Folgenden näher betrachtet.

AspectJ

Eines der bedeutendsten, frei verfügbaren Produkte der aspektorientierten Programmierung, ist AspectJ, eine in Eclipse integrierte, statische Spracherweiterung für Java. Im Jahre 1997 am Xerox PARC entwickelt, erschien vier Jahre später die AspectJ Version 1.0. Inzwischen ist AspectJ, wie Java selbst, bei der Version 5 angekommen und wird seit 2002 von Unternehmen wie Siemens und der U.S. Airforce verwendet. Auch IBM verwendet AspectJ, ist an der Entwicklung maßgeblich beteiligt und hat die technische Leitung des Projekts übernommen.

AspectJ unterstützt alle Java-konformen Programme und läuft, wie diese, auf der Standard-JVM.

In AspectJ ist die Notationsform der Aspekte an Java-Klassen angelehnt, sie verwendet beispielsweise statt des Schlüsselworts *class* das Schlüsselwort *aspect*.

```
// Einfache Java Klasse inkl. Setter:
public class Person {
  private String name, vorname;

  public void setName(String name) {
    this.name = name;
  }

  public void setVorname(String vorname) {
    this.vorname = vorname;
  }
}

// Ein einfacher, dynamischer Aspekt:
public aspect PersonChange {
  pointcut personChanged():
    execution(void Person.setName(String)) ||
    execution(void Person.setVorname(String);

  after(): personChanged() {
    System.out.println("Personendaten wurden geaendert");
  }
}
```

Codebeispiel 16: Ein Beispiel für einen dynamischen Aspekt in AspectJ.

[54] http://aspectwerkz.codehaus.org/
[55] http://profiler.netbeans.org/jfluid.html
[56] http://www.alphaworks.ibm.com/tech/hyperj

Das Codebeispiel 16 zeigt einen Aspekt, der nach jeder Ausführung der beiden Setter-Methoden der Klasse Person ausgeführt wird.

Unterschieden wird bei den Aspekten zwischen statischem und dynamischem Crosscutting. Das dynamische Crosscutting bezieht sich dabei auf Laufzeitereignisse, wie etwa die Ausführung bestimmter Methoden (siehe Codebeispiel 16). Bei der statischen Variante wird der Code einer Klasse um Ergänzungen erweitert (siehe Codebeispiel 17):

```
// Klassendefinition:
public class KlasseA {
}
public class KlasseB {
  public void methodeB() {
    throw new Exception();
  }
}
// Ein einfacher, statischer Aspekt:
public aspect BeispielAspekt {
  // Ergänze KlasseA & KlasseB um öffentliche Variable foo:
  public String ( KlasseA || KlasseB ).foo;
  // Ergänze KlasseA um eine neue Methode bar:
  public void KlasseA.bar() {
  }
  // Setze Oberklasse von KlasseA auf ArrayList:
  declare parents: KlasseA extends java.util.ArrayList;
  // Ergänze Klasse B um das Interface Serializable:
  declare parents: KlasseB implements java.io.Serializable;
  // Ergänze Behandlung der Exception in KlasseB MethodeB:
  declare soft: Exception: execution( void B.methodeB() );
}
```

Codebeispiel 17: Möglichkeiten der Erweiterung durch einen statischen Aspekt.

Diese beiden Beispiele sollen ausreichen, um aufzuzeigen, welche Möglichkeiten der Einsatz von Aspekten mit sich bringt. Auf eine weitere Vertiefung von AspectJ und dessen teils spezifischen Fachbegriffen wird an dieser Stelle verzichtet, da es nicht dem Ziel dieser Arbeit entspricht, ein spezielles AOP-Produkt im Detail vorzustellen.[57]

Aufgrund der engen Anlehnung von AspectJ an Java, sind mit dem Einsatz von AspectJ auch Javas Vor- und Nachteile verbunden. AspectJ besitzt eine nicht besonders zugängliche Syntax sondern passt sich hier die von Java vorgegebenen Konventionen an.

Der Einsatz von AspectJ oder anderen aspektorientierten Erweiterungen bringt nicht automatisch die erwähnten Vorteile mit sich. Es ist hier besonders wichtig, die AOP-Technik und die Programmiersprache gut zu kennen, um AOP angemessen einsetzen zu können. Durch übereifrige Verwendung der AOP-Techniken lässt sich genau das Gegenteil der erhofften Verbesserungen erreichen. Hier sei stellvertretend auf die Gefahr von Konflikte verschiedener Aspekte hingewiesen, deren Abhängigkeiten bei unvorsichtigem Einsatz unüberschaubar werden können. Bei einem mäßigen, gut durchdachten Gebrauch der AOP-Techniken ist jedoch ein Gewinn im Sinne der erwähnten Vorteile zu erwarten. Die Lernkurve für einen solchen, angemessenen Einsatz von AOP sollte jedoch nicht unterschätzt werden, ist aber dem Nutzen durchaus angemessen. Doch dieser variiert natürlich je nach Einsatzzweck und verwendeter Sprache.

[57] Nähere Informationen zu AspectJ finden sich in den Quellen [059], [060] und [062].

3.3 Erwartungen

Die aspektorientierte Programmierung bietet eine sinnvolle Ergänzung der objektorientierten Programmierung. Bei richtiger Verwendung stellt sie eine Steigerung der Softwarequalität und der Produktivität in Aussicht.

Wesentliche Veränderungen gab es im Bereich der AOP in der letzten Zeit nicht. Es kamen eine Reihe neuer Produkte auf den Markt und die Akzeptanz und Verbreitung der AOP wächst stetig. AspectJ bleibt aber das am weitesten verbreitete Produkt, das sich, gebunden an Java, weiterentwickelt.

Der Einsatz neuer Technik birgt oft auch Risiken. Doch inzwischen kann die AOP einige Projekterfolge vorweisen. Die wachsende Erfahrung im Umgang mit AOP trägt zu einer Minderung des Risikopotentials bei und fördert deren Verbreitung.

Die immer größer werdende Unterstützung der Aspektorientierung lässt sich ebenfalls an der internationalen *Aspect-Oriented Software Development*[58] (AOSD) Konferenz erkennen, die seit dem Jahr 2001 jährlich stattfindet.

Vor- und Nachteile

Kritisch zu betrachten ist der für einen korrekten Umgang mit der AOP-Technik nötige Lernaufwand. Im Falle Javas bedeutet dies, wieder einmal, ein neues Werkzeug in die Entwicklungsumgebung mit einzubeziehen. Andere Sprachen, wie etwa Lisp oder Ruby, stehen hier besser da (vgl. Zitat am Anfang des Kapitels). Sie bringen nötige Sprachmittel von Haus aus mit und decken damit einen großen Teil der AOP-Techniken ab.

Die Idee bzw. das Konzept der aspektorientierten Programmierung schmälert diese Argumentation keineswegs. Sie lädt aber dazu ein, die Vor- und Nachteile einer solchen Lösung genau abzuwägen, den nötigen Lernaufwand zu berücksichtigen und einmal mehr über den Einsatz einer geeigneten Programmiersprache nachzudenken.

Speziell über AspectJ lässt sich sagen, dass die Implementierung nicht sonderlich gut gelungen ist. Die Analogie eines Aspektes zu einer Klasse senkt die Übersichtlichkeit und verwässert den Unterschied zwischen Aspekten und Komponenten. Verbessernd wäre hier zum Beispiel eine grafische Darstellung geeigneter Aspekte denkbar (vgl. Domain Specific Language, Kapitel 4). Ebenso ist die Unterstützung für das Refactoring noch nicht zufrieden stellend gelöst. Sie wird, genau wie die Fehlersuche, durch das Weben des Codes erschwert.

Dennoch gibt es mit der Überwachung von Anwendungen oder Authentifizierung und Autorisierung geradezu ideale Einsatzgebiete für die AOP (in Java).

Bei der Überwachung von Softwaresystemen rückt eine weitere Schwäche von AspectJ ins Licht: Die aspektorientierte Lösung gilt nur für Klassen eines Systems, die auch mit AspectJ übersetzt worden sind. Eine Überwachung der Collections[59] ist also nur indirekt bzw. eingeschränkt möglich.

Ein auf längere Sicht möglicher Einsatzzweck der AOP ist die ebenfalls querschneidende Sparte der Middleware. Hier gibt es Chancen die modularisierbaren Aspekte gewinnbringend einzusetzen und die Entwicklung von Applikationen zu verändern.

Im Vergleich zu der bei Java verfolgten Strategie, neue Ideen und Konzepte nur sehr zögerlich in den Sprachstandard aufzunehmen, ist diese Eigenschaft bei AspectJ gegenläufig ausgeprägt. AspectJ ist, da Open Source, stetigen Entwicklungen unterworfen. Allerdings fehlt hier ein einheitlicher Standard, der AspectJ und andere AOP-Werkzeuge, wie HyperJ und AspectWerkz, mit einbezieht. Die verschiedenen AOP-Werkzeuge drohen ohne Standardisierung in ihrer Entwick-

[58] http://www.aosd.net/

[59] Collections sind die Bezeichnung für Behälter einer Programmiersprache, die Objekte in Form von Listen organisieren und beinhalten.

lung immer weiter auseinander zu driften. AspectJ hat zwar in seiner verhältnismäßig langen Entwicklungszeit einen festen „Sprachkern“ entwickelt, doch stellt dieser Kern bisher leider nur einen Quasi-Standard dar.

Eine denkbare Standardisierung wäre ein fester Kern an Elementen, der AspectJ umfasst. Dieser Weg erscheint sinnvoll, da AspectJ durch seine Integration in Java, die IDE Eclipse und deren Open-Source-Community eine starke Unterstützung erfährt.

Fazit

Die aspektorientierte Programmierung ist eine Erweiterung der objektorientierten Programmierung, die die Übersichtlichkeit steigern und Softwarequalität erhöhen kann. Zwar ist sie kein Allheilmittel gegen schlechte Programmierung oder schlechtes Design, aber bei richtiger Verwendung und Dosierung führt sie zu einem Mehrwert, der sich nicht zuletzt in erhöhter Produktivität und Senkung der Kosten widerspiegelt.

Vorsicht ist vor dem teils vorhandenen AOP-Hype geboten, denn die AOP birgt auch unüberschaubare Risiken. Deshalb ist es wichtig, den Einsatz von AOP-Techniken genau zu überlegen und ihre Stärken und Schwächen abzuwägen.

Bereiche, die gut objektorientiert gelöst werden können, sollten auch weiterhin ohne AOP umgesetzt werden. Eine alternative AOP-Einsatzmöglichkeit ist der ausschließliche Einsatz bei der Entwicklung einer Software, sodass hier eine Unterstützung erfolgen kann, während das auszuliefernde Endsystem pures Java ist.

Die Weiterentwicklung, hilfreiche Standardisierung und Verbreitung der AOP ist lohnenswert zu beobachten. Eine Killer-Applikation als Referenz wäre eine willkommene Gelegenheit, welche es der AOP ermöglichen würde sich weiter zu verbreiten, entwickeln und seine Stärken unter Beweis zu stellen.

“A programming language is for thinking of programs, not for expressing programs you've already thought of.” – Paul Graham [Graham-04]

4 Language Oriented Programming

Eine weitere Entwicklungslinie abseits der „reinen“ Programmiersprachen ist das Paradigma des Language Oriented Programming (zu deutsch: sprachorientierte Programmierung). Dieser Bereich befindet sich noch in der Entwicklung, zeigt aber neuartige, viel versprechende Wege in der Softwareentwicklung auf.

Das Thema Language Oriented Programming orientiert sich an den Beiträgen von Martin Fowler, Sergey Dmitriev, Markus Völter und Hartmut Krasemann.

4.1 Vorstellung

Language Oriented Programming (LOP) beschreibt eine, vornehmlich im letzten Jahrzehnt aufgekommene, Entwicklungsweise, die Systeme mit Hilfe verschiedener domänenspezifischen Sprachen (engl: *Domain Specific Language*, *DSL*) beschriebt. Jede domänenspezifische Sprache ist auf ein begrenztes (fachliches) Gebiet ausgerichtet.

Diese Art der Entwicklung ist an sich nicht neu, sie stellt prinzipiell eine Erneuerung altbekannter Ideen dar. Domänenspezifische Sprachen gibt es beispielsweise schon länger in Form der *Structured Query Language* (SQL) oder den so genannten *UNIX little languages*.

Im Bereich des Language Oriented Programming gibt es seit den letzten Jahren verschiedene Bemühungen, die Arbeit mit domänenspezifischen Sprachen durch entsprechende Werkzeuge bzw. Entwicklungsumgebungen zu unterstützen. Diese Produkte befinden sich noch in der Entwicklung. Zu ihren bekanntesten zählen das *Meta Programming System* von JetBrains und *Software Factories* (früher *Intentional Programming*) von Microsoft.

Das Thema Language Oriented Programming umfasst ein in der Entwicklung befindliches Gebiet, das sich grob in zwei Bereiche aufteilen lässt: Der eine Bereich ist die Entwicklung von Anwendungen mit Hilfe domänenspezifischer Sprachen, der andere Bereich stellt die Entwicklung eben dieser Sprachen dar. Letzterer wird in diesem Kapitel behandelt.

Grundlegende Gedanken

Sergey Dmitriev, der Chefentwickler von JetBrains, schreibt über seine Motivation zur Entwicklung des Meta Programming Systems:

„Today’s mainstream approach to programming has some crucial built-in assumptions which hold us back like chains around our necks, though most programmers don’t realize this. With all the progress made so far in programming, we are still in the Stone Age.” [020]

Die Feststellung, die dem Language Oriented Programming zu Grunde liegt, beschreibt die Transformationsschwierigkeiten der bisherigen Entwicklungsweise: Es besteht eine Lücke zwischen dem Verständnis zur Lösung eines Problems und der Umsetzung dessen in ein, für den Computer verständliches Programm. Anders formuliert bedeutet dies, dass Programme näher an dem Fachgebiet des Aufgabenbereichs geschrieben werden sollten, anstatt sich an den Befehlen einer computergeeigneten Programmiersprache zu orientieren.

Ein mit der Problemdomäne vertrauter Entwickler, der fachspezifische Aufgaben ausprogrammieren möchte, sollte die Programmiersprache dazu nutzen können, sich über ein Programm Gedanken zu machen und dieses, auf demselben Niveau, ausformulieren zu können. (Siehe Zitat von Paul Graham am Kapitelanfang.)

Klassische Programmiersprachen stellen ein allgemeines, vielseitig einsetzbares Programmierwerkzeug dar, das i.d.R. nicht auf spezielle Problembereiche optimiert ist. Dieser Umstand zwingt Entwickler häufig dazu, Programme, über die sie sich schon im Klaren sind, noch einmal „computergerecht" in einer Programmiersprache auszuformulieren. Die technische Ausrichtung einer Programmiersprache steht, so gesehen, dem Denken und Entwickeln einer fachbezogenen Anwendung im Weg.

Das Language Oriented Programming basiert auf der Verwendung (mehrerer) domänenspezifischer Sprachen, die, für verhältnismäßig kleine Bereiche, maßgeschneiderte Lösungsmöglichkeiten darstellen. Der Fokus des LOP liegt dabei auf der Erstellung und Entwicklung eigener Domain Specific Languages. Doch zum komfortablen Erstellen und Arbeiten mit einer DSL gehören auch entsprechende Werkzeuge, so genannte *Language Workbenches*, die zurzeit von mehreren Firmen separat entwickelt werden und in naher Zukunft auf den Markt kommen sollen. Aufgrund der domänenspezifischen Ausrichtung der Language Workbenches, unterscheidet sich deren Ansatz von dem der so genannten *CASE-Tools*.

Domain Specific Languages sind, wie bereits erwähnt, keine neue Erfindung, sondern treten, vorwiegend unbemerkt, hier und da bereits im Alltag auf. Doch ihre Unterstützung ist oft unzureichend, was nicht zuletzt durch die Wahl der darunter liegenden Programmiersprachen und die fehlende Werkzeugunterstützung begründet ist.

4.2 Begriffe und Definitionen

Im Folgenden wird der Bereich der domänenspezifischen Sprachen und Language Workbenches näher vorgestellt. Dazu werden zuerst die wichtigsten Begriffe geklärt:

Domain Specific Language

Eine domänenspezifische Sprache (DSL) ist eine ausführbare Programmier- oder eine Spezifikationssprache, die auf einen bestimmten Problem- bzw. Aufgabenbereich ausgerichtet und begrenzt ist. Sie besitzt in dieser, möglichst klein gehaltenen, Domäne eine besonders hohe, zumeist deklarative, Ausdrucksfähigkeit und schließt die (Modellierungs-)Lücke zwischen allgemeinen Programmiersprachen und dem Fachgebiet einer Anwendung. Dies geschieht, in dem die DSL die entstehende Komplexität, durch eine Erhöhung der Abstraktion auf die Fachebene, kompensiert. Im Gegensatz zu einer allgemeinen Programmiersprache muss eine domänenspezifische Sprache nicht Turing-vollständig[60] sein. Sie muss also mit einer anderen Sprache kombiniert angewendet werden, um Aufgaben bewältigen zu können. Eine DSL entspricht dem *Interpreter Pattern* der so genannten *Gang of Four* [Gamma-95].

Es existieren zwei verschiedene Konstruktionsprinzipien von DSLs, die in der Literatur unterschiedlich benannt werden. In dieser Arbeit wird die Terminologie von Martin Fowler verwendet.

Der erste Typ der Konstruktionsprinzipien beschreibt *externe DSLs*. Eine externe DSL ist eine freistehende Sprache, die unabhängig von anderen Programmiersprachen ist. Sie wird üblicherweise in die Hauptprogrammiersprache des Projekts übersetzt. Der zweite Typ nennt sich *interne DSL*. Er beschreibt eine DSL, die in eine allgemeine Programmiersprache integriert ist und als Erweiterung dieser angesehen werden kann.

[60] Vereinfacht formuliert kann man mit einer Turing-vollständigen Sprache alle Arten von Problemen lösen. Eine DSL muss dies nicht können. Ein vollständiges Programm in einer DSL zu formulieren, ist damit i.d.R. unmöglich.

Die schon länger genutzten domänenspezifischen Sprachen, etwa die *UNIX little languages*, beschränken sich meist auf die textliche Repräsentationsform. Doch DSLs können auch Tabellen, Diagramme, Bäume oder grafische Notationen aller Art umfassen. Der Vorteil der grafischen Darstellung von DSLs ist die erhöhte Ausdrucksfähigkeit bei speziellen Anwendungsbereichen. So lassen sich z.B. Beziehungen bzw. Relationen verschiedener Konzepte besser visualisieren.

Eine domänenspezifische Sprache, die – wie bislang üblich – ohne eine Language Workbench verwendet und erstellt wurde, bietet zwar einige Vorteile in Bezug auf die Ausdrucksnähe zur Fachdomäne, jedoch fehlt es an weiterer Unterstützung im Umgang mit ihr. Sie stellt sozusagen das Mindestmaß dar, welches nötig ist, um Language Oriented Programming zu betreiben. Doch die Idee des LOP geht darüber hinaus und vereinfacht den Einsatz der DSLs mittels maßgeschneiderter Werkzeuge, den Language Workbenches.

Language Workbench

Um die Entwicklung und Verwendung der DSLs zu vereinfachen und zu unterstützen, werden zur Zeit Language Workbenches entwickelt. Diese nehmen dem Entwickler der DSL Aufgaben ab, indem sie ihn durch Editoren bei der Erstellung domänenspezifischer Sprachen unterstützen. Damit auch die Benutzung einer DSL vereinfacht wird, kann mit Hilfe einer Language Workbench ein, speziell auf eine DSL zugeschnittener, Editor erstellt werden. Somit bietet eine Language Workbench auch die übliche Unterstützung einer klassischen Entwicklungsumgebung. Darunter fällen etwa Refactoring, eine Quelltext-Ergänzung und natürlich auch die Integration in die Anwendungsprogrammierung mit allgemeinen Programmiersprachen.

Eine Language Workbench ist, das nötige Wissen über eine DSL vorausgesetzt, in der Lage, diese auf verschiedene Programmiersprachen abzubilden und damit Unabhängigkeit von speziellen Sprachen zu gewähren. Eine Abbildung in ausführbaren Code ist ebenfalls möglich, wie die folgende Abbildung 7 deutlich macht.

Im Vergleich zur klassischen Programmierung ist die Notationsform der Language Workbenches nicht textbasiert. Language Workbenches speichern einzig die abstrakte Repräsentation eines Programms, die einem strukturierten Syntaxbaum ähnelt. Diese abstrakte Repräsentation kann bei Bedarf, etwa zum Bearbeiten, in verschiedene Notationsformen bzw. DSLs projiziert werden. Abbildung 7 verdeutlicht dies:

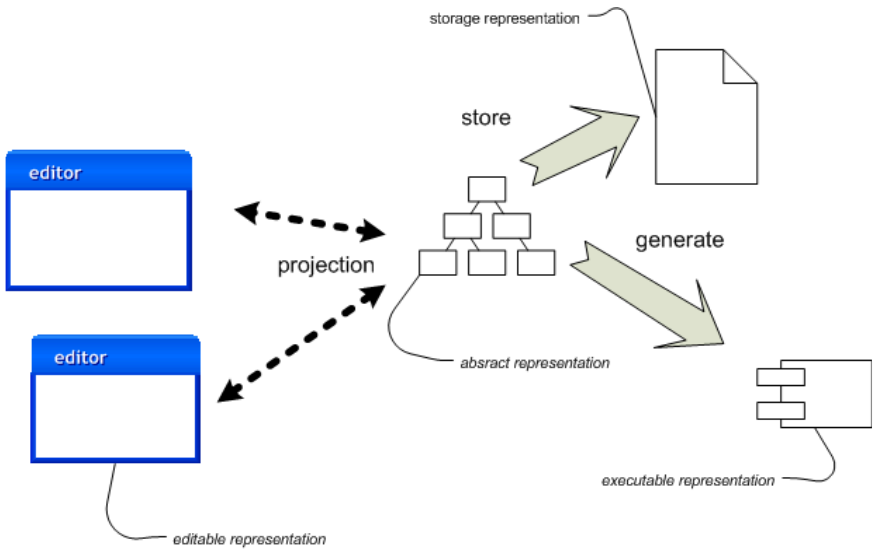

Abbildung 7: Darstellung der Projektionsformen einer Language Workbench

Language Workbenches sind nur ein möglicher Weg Language Oriented Programming umzusetzen, jedoch sind sie durch ihre DSL-Unterstützung besonders für diesen Einsatz geeignet. Viele der hier angesprochenen Eigenschaften einer Language Workbench werden in den folgenden Unterkapiteln durch Beispiele und Produkte weiter verdeutlicht.

Language Oriented Programming

LOP beschreibt das Paradigma des Programmierens mit einer oder mehreren domänenspezifischen Sprachen. Durch die Verwendung domänenspezifischer Sprachen fällt das LOP-Konzept in den Bereich der *fourth-generation programming languages*[61].

Der Einsatz und die Erstellung domänenspezifischer Sprachen verursacht Kosten. Diese stehen jedoch dem höheren Aufwand gegenüber, der ohne die Nutzung von DSLs nötig ist. Denn DSLs bieten mit Unterstützung der Language Workbenches ein hohes Potential, welches sich in der Nähe zur Fachdomäne und den damit verbundenen (Modellierungs-)Vorteilen ausdrückt.

Domänenspezifische Sprachen sind aufgrund ihrer Ausrichtung auf das Fachgebiet leicht zu lernen, da sich Entwickler ohnehin mit dem Fachgebiet auseinandersetzen müssen. Dieses Fachgebiet kann nun, ohne große Umwege durch begrenzte Ausdrucksmöglichkeiten, direkt in eine Anwendung übertragen werden. Sollte für das Fachgebiet schon ein Glossar von Fachbegriffen vorhanden sein, so kann dies, zur Anpassung bzw. Erstellung einer fachbezogenen Sprache, unterstützend hinzugezogen werden.

Eine mögliche Umsetzung des LOP-Konzeptes sind die Language Workbenches. LOP eignet sich für den Einsatz in vielen fachgebundenen Bereichen, darunter fallen insbesondere die Konfiguration, Geschäftslogik, Web-Services, Datenbanken, (Protokolle der Tele-)Kommunikation und Treiberprogrammierung.

“The Java™ programming world is full of domain-specific languages (DSLs), but options in the Java language for building DSLs are limited. Not so with Ruby. ... Ruby lets you integrate clean DSLs, giving you a new frame of reference for examining your Java options with open eyes.” – Bruce Tate [015]

4.3 Domain Specific Languages

Nach der Vorstellung des Themas, dessen Begriffen und Ideen, werden die domänenspezifischen Sprachen nun genauer vorgestellt und an Beispielen erläutert.

Interne und externe DSLs

Wie schon erwähnt, gibt es zwei unterschiedliche (textuelle) Typen von domänenspezifischen Sprachen. Interne DSL erweitern eine allgemeine Programmiersprache, während externe DSLs eine eigenständige Entwicklung darstellen.

Wird eine DSL in einer anderen Sprache als der Hauptsprache der dazugehörigen Anwendung geschrieben, so handelt es sich um eine externe DSL. Da diese DSL völlig frei steht und an keine darunter liegende Programmiersprache gebunden ist, ist die einzige Begrenzung bei der Sprachgestaltung die Übersetzbarkeit der DSL. Üblicherweise wird eine externe DSL in die Hauptprogrammiersprache der Anwendung übersetzt. Hier ist ein gewisser Aufwand nötig, doch passende Werkzeuge wie Parser- und Compiler-Generatoren sind verfügbar, wodurch sich der Aufwand für kleine DSLs in Grenzen hält.

Zwischen der Anwendungsprogrammiersprache und der DSL besteht eine symbolische Barriere: Ein in der Anwendung ausgeführtes Refactoring wird nicht automatisch auf die DSL übertragen; spätestens an diesem Punkt ist der Einsatz einer Language Workbench empfehlenswert, um Integrationsprobleme zu beheben.

[61] Zu deutsch: Vierte Generation der Programmiersprachen. Abgekürzt: 4GL. Die 4GL-Sprachen stehen eine Stufe über den 3GL-Sprachen. Mit 3GL-Sprachen werden allgemeine Programmiersprachen bezeichnet.

Statisch übersetzte Sprachen, wie Java, profitieren von der Verwendung externer DSLs. Durch die Auswertung der externen DSLs zur Laufzeit des (Java-) Programms werden Sprachänderungen der DSL zur Laufzeit möglich. Dies ist ein ausschlaggebender Grund für die große Verbreitung externer DSLs (z.B. XML) in der Java-Welt.

Das Sprachdesign einer DSL unterscheidet sich nicht wesentlich von dem einer Programmbibliothek. Bei externen DSLs ist besonders auf einen kleinen Umfang und eine leichte Erlernbarkeit der Sprache zu achten, damit ein Sprachwirrwarr ausbleibt. Werden DSLs nicht orthogonal genug umgesetzt, können Redundanzen dazu führen, dass der Sprachaufbau unklar wird. Ebenfalls empfehlenswert ist eine klare Trennung verschiedener DSLs. Jede DSL sollte nur einen (fachlich abgegrenzten) Aspekt bedienen und möglichst wenige Gemeinsamkeiten mit anderen DSLs aufweisen. Orthogonalität stellt hier das erklärte Ziel dar.

Interne DSLs

Interne DSLs besitzen die eben erwähnten Probleme nicht, da sie auf einer allgemeinen Programmiersprache basieren und auf deren Syntax aufsetzen. Sie besitzen keine integrativen Barrieren, haben sich aber dafür nach den Möglichkeiten und Grenzen der darunter liegenden Sprache zu richten. Die Sprachfähigkeiten der Basissprache sind also immer zugegen.

Dies hat zur Folge, dass sich Programmiersprachen unterschiedlich gut für die Implementierung interner DSLs eignen. Dynamische Sprachen, die Metaprogrammierung gut unterstützen, haben hier wesentliche Vorteile. Sie lassen durch ihre durch Syntax und Struktur gegebenen Grenzen genug Freiraum, um eine DSL gut zu integrieren.

Interne DSLs lassen sich mittels Präprozessoren oder Makros implementieren. Bei diesem Ansatz wird die DSL zuerst in „normalen“ Quelltext umgewandelt, bevor dieser übersetzt wird. Dementsprechend weist dieser Ansatz ein anderes Problem auf: Die Code-Überprüfung findet erst beim Übersetzen statt, sodass eventuelle Fehler des DSL-Codes erst spät bemerkt werden.

Das Feedback dieser Variante, welches auf der Ebene der Basis-Sprache stattfindet, kommt spät und macht die Programmierung somit fehleranfälliger. Geeigneter ist hier die (aufwändigere) Erweiterung der Compiler, damit die eben erwähnten Nachteile kompensiert werden können.

Interne DSLs bieten den Vorteil, dass sich Programmierer, die mit (der Syntax) der Basis-Programmiersprache vertraut sind, schnell in die DSL einarbeiten können und ihnen jederzeit die volle Funktionalität der zugrunde liegenden Sprache zur Verfügung steht. Auf der anderen Seite sollte die Syntax der Basis-Sprache die Einarbeitung in die DSL nicht erschweren.

Um die Einarbeitung in eine DSL zu erleichtern, sollte natürlich auch eine Dokumentation der DSL mit Beispielen erstellt werden. Aufgrund des begrenzten Fachgebiets einer DSL sollten aber – als Daumenregel – zwei Seiten Dokumentation nicht überschritten werden.

Der Aufwand für die Implementierung einer (kleineren) DSL liegt laut Markus Völter [012] bei einigen Stunden bis Tagen. Interne DSLs sind dabei erfahrungsgemäß schneller fertig gestellt als ihre externen Pendants, da auf die Hauptprogrammiersprache mit ihrem Compiler oder Interpreter aufgebaut werden kann.

Bekannte Domain Specific Languages

- Textuelle DSLs sind seit vielen Jahren im Einsatz. Stark vertreten sind hier die *UNIX little languages*, wie *lex, yacc, make* oder *Tex*.
- Microsofts Tabellenkalkulation *Excel*, die Datenbanksprache SQL und die für die Darstellung in Browsern entwickelte *HyperText Markup Language* (HTML) sind ebenfalls bekannt. Gerade Excel ist ein schönes Beispiel für eine sehr fachnahe DSL, mit der sogar Nicht-Programmierer umgehen können. SQL hat sich im Laufe der Jahre zu einer umfassenden Zugriffs- und Definitionssprache für relationale Datenbanken entwickelt. Sie hat den kleinen Umfang einer DSL sehr deutlich überschritten und füllt ganze Bücher mit Dokumentationen, sodass eine eingehende Beschäftigung mit der Sprache notwendig ist, bevor sie umfangreich eingesetzt werden kann. Mit HTML verhält es sich ähnlich. Die Sprache erfährt eine große Verbreitung und vielerlei Erweiterungen durch Programmiersprachen, sodass auch hier nicht mehr von einer leichten Zugänglichkeit gesprochen werden kann.
- Eine schon ausführlicher behandelte DSL ist *AspectJ* als Vertreter einer AOP-Implementierung. Mit AspectJ werden vorwiegend technische Aspekte des Systemverhaltens beschrieben, die orthogonal zur Hauptfunktionalität liegen.
- Die Sprache *PostScript*, deren Domäne das Rendern von Seiten ist, und die *Extensible Markup Language XML* erfreuen sich großer Beliebtheit. XML stellt an sich – ohne Fachausrichtung – noch keine DSL dar, sondern dient im Allgemeinen als Notationsform für strukturierte Daten. In Verbindung mit den enthaltenen Daten, kann die XML-Darstellung als DSL angesehen werden.
- Active Record ist eine textuelle DSL, die ein Zugriffsmuster für relationale Datenbanken umsetzt. Das Framework findet eine starke und intuitive Verwendung in *Ruby on Rails*
- Ein Beispiel aus einem anderen Gebiet sind die so genannten GUI-Builder[62]. Diese stellen keine textuelle, sondern eine grafische DSL dar, die Benutzeroberflächen beschreibt.

Alle bisher erwähnten, vorwiegend textuellen, DSLs sind relativ standardisiert ausgerichtet und für generelle Einsatzzwecke gedacht. Sie sind nicht individualisiert und vertreten somit eher die klassische Verwendung von DSLs.

In den letzten Jahren geht der Trend immer stärker zu maßgeschneiderten Sprachen, die die individuelle DSLs darstellen. Eine Entwicklung einer solch individuellen DSL kann sich, bei entsprechendem Einsatz, schon für ein einziges Projekt auszahlen.

Erstellung einer Domain Specific Language

Die Implementierung einer textuellen DSL sollte gut überlegt und geplant sein. Unabhängig von der Fachausrichtung der DSL sollte man sich zuerst für eine interne oder externe Variante entscheiden. Bei einer internen DSL sollte eine dynamische, veränderbare Sprache mit guten Metaprogrammierungsfähigkeiten gewählt werden. Hier fallen Java oder C# eher nicht in die engere Wahl. Smalltalk und Lisp haben auf diesem Gebiet eine gewisse Tradition, und Ruby eignet sich dafür, dank der neutralen Syntax, noch besser.

Der Großteil der domänenspezifischen Sprachen benötigt einige übliche Sprachmittel, die aus allgemeinen Programmiersprachen bekannt sind; dies sind: Typen, Literale, Deklarationen, Operatoren, Kontrollstrukturen und, je nach Einsatzzweck, andere Sprachkonstrukte.

Die Definition einer DSL besteht aus drei grundlegenden Teilen: Dem *Schema* bzw. Aufbau der abstrakten Repräsentation, einem *Editor*, der die Projektionen der abstrakten Repräsentation zur Bearbeitung darstellt und die Syntax definiert, und einem *Generator*, der die Übersetzung der abstrakten in eine ausführbare Repräsentation definiert.

[62] Als GUI-Builder (Graphical User Interface) werden Werkzeuge bezeichnet, mit denen grafische Benutzeroberflächen erzeugt werden können. Das Werkzeug generiert den nötigen Code der Ziel-Programmiersprache automatisch.

Im ersten Schritt der Modellierung einer textuellen DSL müssen die domänenspezifischen Voraussetzungen geschaffen werden. Dies umfasst ein gutes Verständnis des Fachbereichs und eine klar umrissene Abgrenzung des Aufgabengebiets der DSL. Im zweiten Schritt geht es um die Bedeutung, also die Semantik, der zentralen Abstraktionen des Fachgebiets. Im darauf folgenden Schritt wird die Syntax der Sprache festgelegt. Diese sollte möglichst intuitiv sein, um die Aufmerksamkeit nicht von der Semantik abzulenken.

Martin Fowler unterscheidet hier zwischen einer *abstrakten* und *konkreten Syntax*. Die allen Darstellungsweisen zugrunde liegende Syntax ist dabei die abstrakte Syntax, sie beschreibt den (gemeinsamen) Aufbau aller konkreten syntaktischen Ausprägungsarten (z.B. XML).

Im letzen und vierten Schritt wird die DSL integriert. D.h. es wird ein Generator erstellt, der die DSL in eine weniger abstrakte Ebene übersetzt. Im Falle einer internen DSL ist hier kaum Arbeit nötig, da sie auf der Basis einer schon bestehenden Programmiersprache erstellt wurde. Vorhandene Compiler und Interpreter können weiterverwendet werden. Bei einer externen DSL müssen Compiler oder Interpreter erst (mit Hilfsmitteln) erstellt werden. Die DSL wird dabei entweder direkt in ausführbaren Byte- oder Binärcode übersetzt oder es wird aus ihr, mittels Generator, Quelltext einer allgemeinen Programmiersprache erstellt.

Um die nötige Unterstützung für die (externe) DSL in einem Projekt zu bekommen, besteht die Möglichkeit einen Eclipse-Plugin zu schreiben. Die DSL wird dann über explizite Schnittstellen in der Hauptprogrammiersprache verwendet bzw. aufgerufen.

Beispiel einer Domain Specific Language

Anhand eines Beispiels von Martin Fowler [002] soll verdeutlicht werden, wie eine DSL aussehen kann. Beschrieben wird eine DSL, die zur Systemkonfiguration für das Einlesen von Dateien in Objekte dient:

```
#123456789012345678901234567890123456789012345678901234567890123456789
SVCLFOWLER          10101MS0120050313.......................
SVCLHOHPE           10201DX0320050315.....................
SVCLTWO            x10301MRP220050329.....................
USGE10301TWO           x50214..7050329.....................
```

Codebeispiel 18: Beispielhafte Datensätze zum Einlesen in das System. Jede Zeile stellt einen Datensatz dar.

Das Codebeispiel 18 zeigt eine Datenstruktur, die in das System eingelesen werden soll. Dazu ist ein *Reader* zum Einlesen der Daten und eine separate Datenbeschreibung notwendig. Martin Fowler hat dies in Java beispielhaft gelöst, in dem er so genannte *ReaderStrategies* einsetzt, die je ein Datenformat beschreiben.

Das Datenformat setzt sich zusammen aus einer vierstelligen Beschreibung am Zeilenanfang, die ausdrückt, ob ein *Service Call (SVCL)* oder ein (Elektrizitäts-)*Verbrauch (USGE)* folgt. Je nach angezeigtem Typ folgen nun unterschiedliche Daten, deren Zusammensetzung aus dem folgenden Codebeispiel 18 ersichtlich wird. (Irrelevante Daten sind durch Punkte gekennzeichnet.)

Die verwendeten *Reader*- und *ReaderStrategy*-Klassen werden hier nicht dargestellt, weil sie hier keine größere Rolle spielen und ihre Verwendung aus dem Codebeispiel gut deutlich wird.

Das für dieses DSL-Beispiel wichtige Codestück, welches den Reader passend zu der vorgegebenen Datenstruktur konfiguriert, sieht in C# folgendermaßen aus:

```
public void Configure(Reader target) {
  target.AddStrategy(ConfigureServiceCall());
  target.AddStrategy(ConfigureUsage());
  }
private ReaderStrategy ConfigureServiceCall() {
  ReaderStrategy result = new ReaderStrategy("SVCL", typeof
(ServiceCall));
  result.AddFieldExtractor(4, 18, "CustomerName");
  result.AddFieldExtractor(19, 23, "CustomerID");
  result.AddFieldExtractor(24, 27, "CallTypeCode");
  result.AddFieldExtractor(28, 35, "DateOfCallString");
  return result;
}
private ReaderStrategy ConfigureUsage() {
  ReaderStrategy result = new ReaderStrategy("USGE", typeof
(Usage));
  result.AddFieldExtractor(4, 8, "CustomerID");
  result.AddFieldExtractor(9, 22, "CustomerName");
  result.AddFieldExtractor(30, 30, "Cycle");
  result.AddFieldExtractor(31, 36, "ReadDate");
  return result;
}
```

Codebeispiel 19: Domänenspezifischer Konfigurationscode für den Reader in C#.

Die in Codebeispiel 19 dargestellte Konfiguration ist speziell an die vorgestellten Daten angepasst, während die Einlesewerkzeuge abstrakt gehalten und damit wieder verwendbar sind. Wie in der C#- bzw. Java-Welt üblich, lässt sich die Konfiguration auch in XML ausdrücken:

```
<ReaderConfiguration>
  <Mapping Code = "SVCL" TargetClass = "dsl.ServiceCall">
    <Field name = "CustomerName" start = "4" end = "18"/>
    <Field name = "CustomerID" start = "19" end = "23"/>
    <Field name = "CallTypeCode" start = "24" end = "27"/>
    <Field name = "DateOfCallString" start = "28" end =
"35"/>
  </Mapping>
  <Mapping Code = "USGE" TargetClass = "dsl.Usage">
    <Field name = "CustomerID" start = "4" end = "8"/>
    <Field name = "CustomerName" start = "9" end = "22"/>
    <Field name = "Cycle" start = "30" end = "30"/>
    <Field name = "ReadDate" start = "31" end = "36"/>
  </Mapping>
</ReaderConfiguration>
```

Codebeispiel 20: Der Konfigurationscode als DSL in XML.

Die Darstellung in Codebeispiel 20 bietet eine verbesserte Lesbarkeit und Plattformunabhängigkeit gegenüber der C#-Variante. Dennoch ist sie noch nicht einfach zu lesen.

Codebeispiel 21 zeigt eine maßgeschneiderte Konfiguration, die als (externe) domänenspezifische Sprache in eigener Syntax dargestellt wird:

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

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

Codebeispiel 21: Konfigurationscode in eigener Notationsweise als DSL.

Alle aufgeführten Codebeispiele stellen domänenspezifische Sprachen dar. Aufgrund ihrer gemeinsamen *abstrakten* Syntax verfügen sie über einen gleichartigen Aufbau. Dieser wird jeweils in unterschiedlichen *konkreten* Syntaxen bzw. Notationsweisen dargestellt.

```
mapping('SVCL', ServiceCall) do
     extract 4..18, 'customer_name'
     extract 19..23, 'customer_ID'
     extract 24..27, 'call_type_code'
     extract 28..35, 'date_of_call_string'
end
mapping('USGE', Usage) do
     extract 9..22, 'customer_name'
     extract 4..8, 'customer_ID'
     extract 30..30, 'cycle'
     extract 31..36, 'read_date'
end
```

Codebeispiel 22: Die Konfiguration als interne DSL in Ruby.

Codebeispiel 22 zeigt eine weitere Variante in Ruby. Diese zeigt, wie geeignet Ruby und andere dynamische Sprachen für den Einsatz interner DSLs sind. Ruby kommt in diesem Beispiel recht nah an die eigens erstellte externe DSL aus Codebeispiel 21 heran. Die C# und die XML-Variante sind im Vergleich nicht so gut lesbar.

4.4 Produkte: Language Workbenches

Language Workbenches wurden bereits vorgestellt. Es folgen einige Produkte im Überblick, von denen viele noch nicht marktreif sind. Die meisten Hersteller geben nur wenige Informationen über ihre Produkte preis. Stellvertretend soll das *Meta Programming System* von *JetBrains* etwas genauer vorgestellt werden, von dem es bereits eine viel versprechende Beta-Version zum Download gibt. Es ist zu vermuten, dass die anderen Hersteller ähnliche Produkte entwickeln.

Intentional Software

Intentional Software[63] ist eine im Jahr 2002 gegründete Firma von Charles Simonyi und Gregor Kiczales. Sie basiert auf der Arbeit Simonyis bei *Microsoft Research*, die er in den Jahren zuvor durchgeführt hat und unter dem Namen *Intentional Programming* bekannt ist. Leider hält die Firma Informationen über ihr Produkt sehr knapp, sodass nicht einmal ein Erscheinungszeitraum der Software bekannt ist. Doch was an Informationen herauszubekommen ist, deckt sich gut mit den hier beschriebenen Eigenschaften des LOP und dessen Language Workbenches.

Software Factories

Software Factories[64] ist Microsofts Initiative einer Softwareentwicklung, die dem LOP auf Basis vorwiegend grafischer DSLs entspricht. Software Factories ist keine Language Workbench, sondern eine von Microsoft propagierte Methode zur Softwareentwicklung, die in Microsofts *Visual Studio 2005* und *Visual Studio Team System* integriert wird. Eine Betaversion des *Visual Studio 2005 SDK* ist inzwischen erhältlich. Software Factories ist eine Entwicklung abseits von Standards wie UML oder MDA[65]. Es ist zu erwarten, dass die Unterstützung der DSLs auf Microsoft-Plattformen und Programmiersprachen ausgerichtet ist.

Aufgrund Microsofts bedeutender Markstellung könnte Software Factories zu größerer Bedeutung kommen. Leider sind Microsofts Informationen bezüglich Software Factories dürftig und allgemein gehalten. Verfügbare Informationen werden zum Teil kontrovers diskutiert.

Ein klares Bild über den Entwicklungstand ist auch hier nicht zu erkennen.

MetaEdit+

MetaEdit+[66] ist ein Produkt der Firma MetaCase, welches *Domain Specific Modeling* unterstützt. Der Schwerpunkt liegt hier auf der vollständigen Codegenerierung. MetaEdit+ stellt eine Entwicklungsumgebung für das Domain Specific Modeling dar. Laut MetaCase müssen hier nur noch die (domänenspezifische) Sprache und deren Generator definiert werden. Grafische Editoren, Generatoren und verschiedene Plattformunterstützungen soll das Werkzeug mitbringen.

Das Produkt MetaEdit+ ist schon seit Mitte der neunziger Jahre auf dem Markt und trägt aktuell die vierte Versionsnummer. Es ist also zu vermuten, dass MetaEdit+ nicht direkt auf das Konzept des LOP ausgerichtet ist, sondern es auf eine andere Art und Weise umsetzt als dies bei den Neuentwicklungen geschieht.

[63] http://intentsoft.com/
[64] http://www.softwarefactories.com , http://msdn.microsoft.com/vstudio/teamsystem/workshop/sf/default.aspx und http://msdn.microsoft.com/vstudio/DSLTools/
[65] MDA steht für Model Driven Architecture, ein Konstruktionsprinzip für Softwaresysteme der Object Management Group (OMG).
[66] http://www.metacase.com/

XMF-Mosaic

Eine ebenfalls kommerzielle Lösung ist mit XMF-Mosaic[67] von Xanctium auf dem Markt. Laut Xanctium, gegründet im Jahr 2003, ist dieses Produkt in der Lage, jede beliebige DSL zu unterstützen. Xanctium orientiert sich dabei an den UML-, xUML-[68] und MDA-Standards und unterstützt insbesondere die Fachgebiete der Finanzen, Telekommunikation und Verteidigung. Die Ausrichtung von XMF-Mosaic tendiert eher zu kompletten (Sprach-)Lösungen für große Firmen.

Subtext

Subtext[69] ist eine im Jahr 2004 vorgestellte Neuentwicklung von Jonathan Edwards, der zurzeit am MIT forscht. Es ist keine gewöhnliche Entwicklungsumgebung oder Sprache, sondern ein radikal neues Programmierparadigma, das mit einer Art Programmiersprache verbunden ist.

Laut Edwards ist das heutige Programmierproblem darin begründet, dass die Bedeutung der Programme hinter diversen Abstraktionsschichten verborgen liegt.[70] Programmiert wird also nicht anhand eines mentalen, an Beispielen orientierten Modells, sondern in Abstraktionsschichten, die das Arbeiten deutlich behindern. Edwards Ansatz mit Subtext wird *Programming by Example* oder auch *Example Centric Programming* genannt. Es besitzt eine Verwandtschaft zu den grundlegenden Ideen des Language Oriented Programming und zu Teilen der LOP-Umsetzung. Dabei geht es aber nur sehr nachrangig um die Entwicklungsunterstützung neuer (domänenspezifischer) Sprachen.

Subtext wendet sich ab vom klassischen, textuellen Quellcode. Stattdessen wird in einer baumartigen Struktur programmiert, welche nicht die Syntax, sondern die Semantik eines Programms darstellt. Subtext verzichtet dabei ebenfalls auf Variablen. Programmiert wird anhand eines strukturierten Graphen, der den „Code“ als lebendiges Beispiel darstellt. Änderungen im Code werden direkt umgesetzt; das Programm wird also zur Laufzeit editiert.

Eine Vorstellung von Subtext lässt sich schlecht in Worte fassen, deshalb verfügt Edwards über zwei Videos ([068] und [069]) auf seiner Subtext- Homepage, in denen er Subtext visuell erklärt.

Subtext befindet sich in einem sehr frühen experimentellen Stadium auf dem Weg zu einer alternativen Programmiersprache. Doch bietet es in diesem Stadium schon innovative Möglichkeiten. Das Testen von Code ist integriert und ohne zusätzlichen Code möglich. Die aufwändige Suche von Fehlern soll damit der Vergangenheit angehören.

Doch viele Funktionen fehlen Subtext noch. Dennoch scheint Edwards Ansatz viel versprechend zu sein: Er stellt eine radikale Änderung dar, die das Programmieren von übermorgen vereinfachen soll, bzw. das heutige Programmierproblem aus der Welt schaffen soll. Doch bis dahin ist es noch ein langer Weg, auf dem ein Umdenken, weg von einigen bisherigen Vorstellungen und Paradigmen, unerlässlich ist. Jonathan Edwards [069] formuliert es so:

“What is the future of programming? - ... We are just beginning to learn how to program. We can radically simplify programming, but: we’ll need to make a few changes.”

[67] XMF steht für eXecutable Metamodelling Facility. http://www.xactium.com/
[68] xUML steht für Executable UML. Eine Methode zur Code-Generierung aus UML-Diagrammen.
[69] http://subtextual.org/ und http://alarmingdevelopment.org/
[70] Dies entspricht der erwähnten Modellierungslücke.

Meta Programming System

Das Meta Programming System[71] (MPS) von JetBrains ist eine Language Workbench, ganz im Sinne des vorgestellten Konzepts des Language Oriented Programming. Es befindet sich noch in der Entwicklung, aber es sind schon interessante und verwertbare Ergebnisse zu erkennen.

Das folgende Beispiel ist übernommen von Martin Fowler [003]. Es beschreibt eine einfach gehaltene DSL zur Abrechung von Stromeinheiten und Serviceleistungen. Als erster Eindruck dieser DSL folgt nun eine beispielhafte Darstellung der Preise eines Stromanbieters.

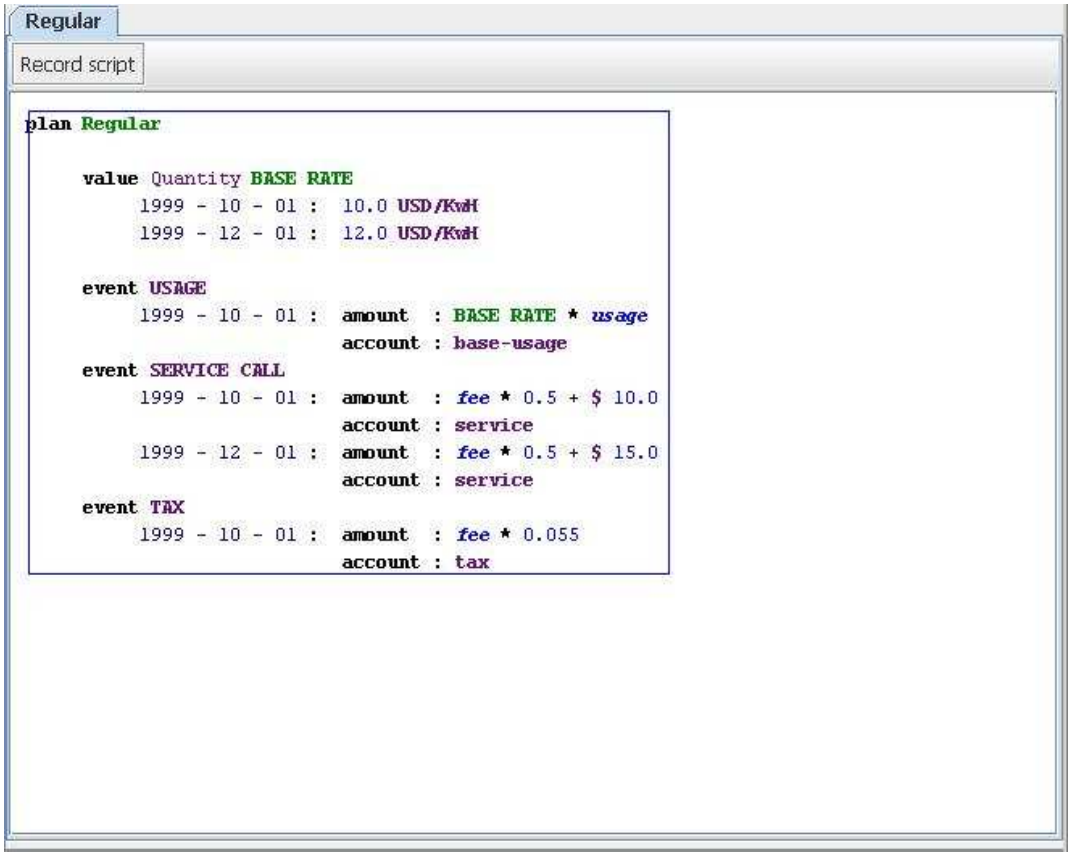

Abbildung 8: Ein regulärer Abrechnungsplan eines Stromanbieters, formuliert in einer eigens dafür erstellten DSL.

Die DSL aus Abbildung 8 beschreibt ein System, welches auf Ereignisse (engl. *events*) reagiert. Diese Ereignisse sind die Abrechnungen von Leistungen in Form von Stromverbrauch (*USAGE*), Serviceleistungen (*SERVICE CALL*) und Steuern (*TAX*). Die Abrechnung dieser Leistungen sind jeweils mit einem Anfangs-Gültigkeitsdatum versehen. Sie sind aus dem Beispiel einfach ersichtlich. Die Berechnung der Stromkosten basiert auf einer separaten Definition der Strompreise.

[71] http://www.jetbrains.com/mps/index.html

Abbildung 9 zeigt einen billigeren Stromtarif, den der Stromanbieter vertreibt:

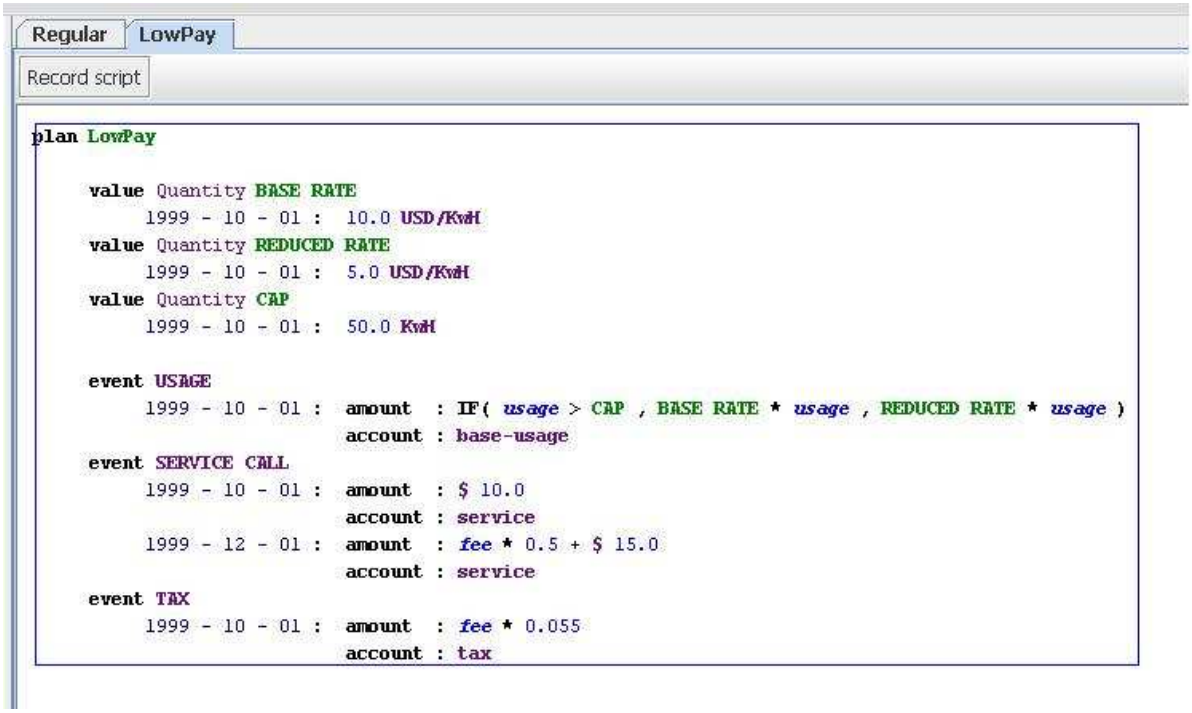

Abbildung 9: Der Plan eines Billig-Stromtarifs mit einfachen Verzweigungs-Kontrollstrukturen bei der Preisberechnung.

Die beiden, in den Abbildungen 8 und 9 dargestellten, Stromtarife sind in ihrer Notation durchaus von fachlich versierten Personen, die keine Programmierer sind, zu verstehen.

Die Darstellung dieses Abrechnungsplans stellt keinen echten Text dar, sondern ist eine Projektion der darunter liegenden abstrakten Repräsentation in Form von Text.

Das MPS unterstützt die Formulierung der Berechnungsformeln durch eine Formel-DSL. Diese bringt eine vorgegebene Notationsweise, Struktur und entsprechende Kontexthilfe mit und stellt eine gute Integration fremder DSLs dar.

Bei der Erstellung des Abrechnungsplans ist das MPS fehlertolerant. Eine nicht vollendete Definition eines Events bzw. Werts behindert die weitere Unterstützung bei der Erstellung neuer Einträge nicht.

Um die hier beispielhaft dargestellte DSL, mit Unterstützung durch das MPS, verwenden und einsetzen zu können, ist eine Definition des Aufbaus bzw. der Struktur der DSL nötig. Für diesen Zweck gibt es, wie in anderen Language Workbenches auch, eine speziell für die Definition von Sprachen geschaffene DSL. Diese trägt im MPS den Namen *Structure Language*. Mit ihr wird das Schema der Sprache festgelegt bzw. die Beziehungen oder auch Zusammensetzung der Bestandteile eines Schemas beschrieben.

Ein Beispiel für das Schema *Plan*, welches den Leistungskatalog beschreibt, ist in Abbildung 10 dargestellt:

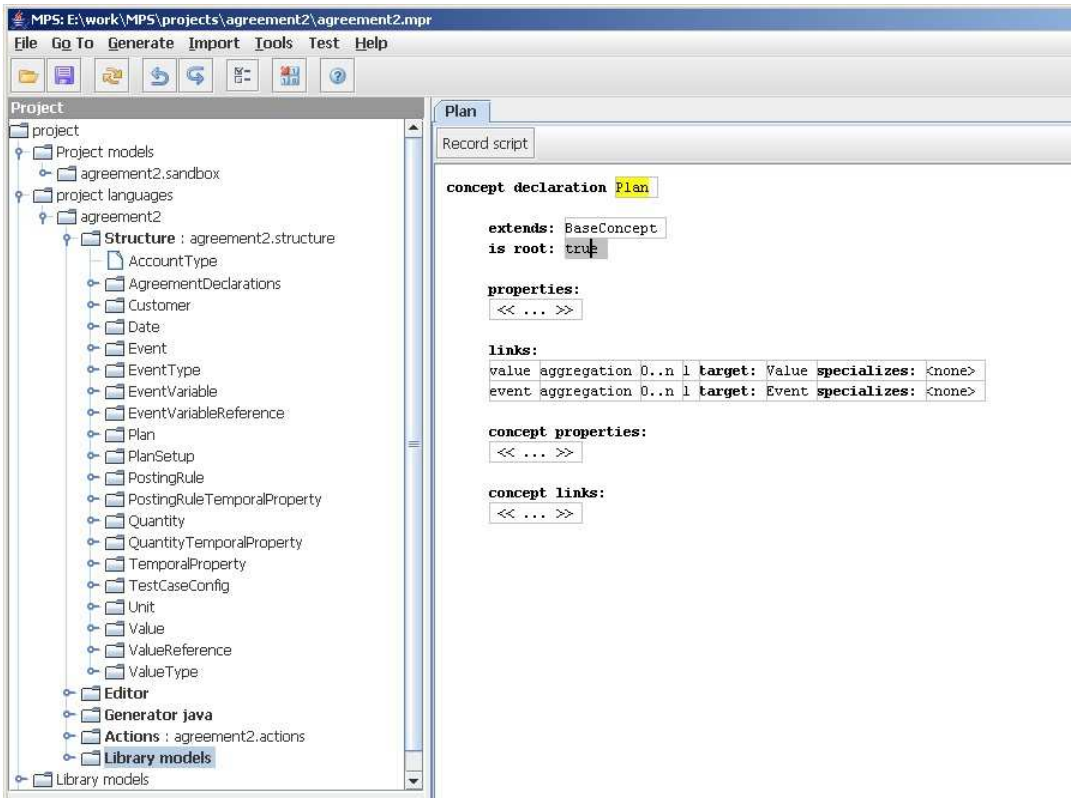

Abbildung 10: Die Definition des Schemas ‚Plan' in MPS. Sichtbar ist auch die Navigationsleiste links im Bild.

Um das Schema aus Abbildung 10 bearbeiten zu können, ist ein speziell angepasster Editor nötig, der die Syntax definiert. Um diesen Editor erzeugen zu können, wird die so genannte *Editor Language* verwendet. Diese Definition des Editors geschieht in einem Editor für Editoren, gewissermaßen einem *Meta-Editor*. Die Definition der Syntax wird auch *Konzept* genannt. Das *Konzept Plan* ist in Abbildung 11 dargestellt:

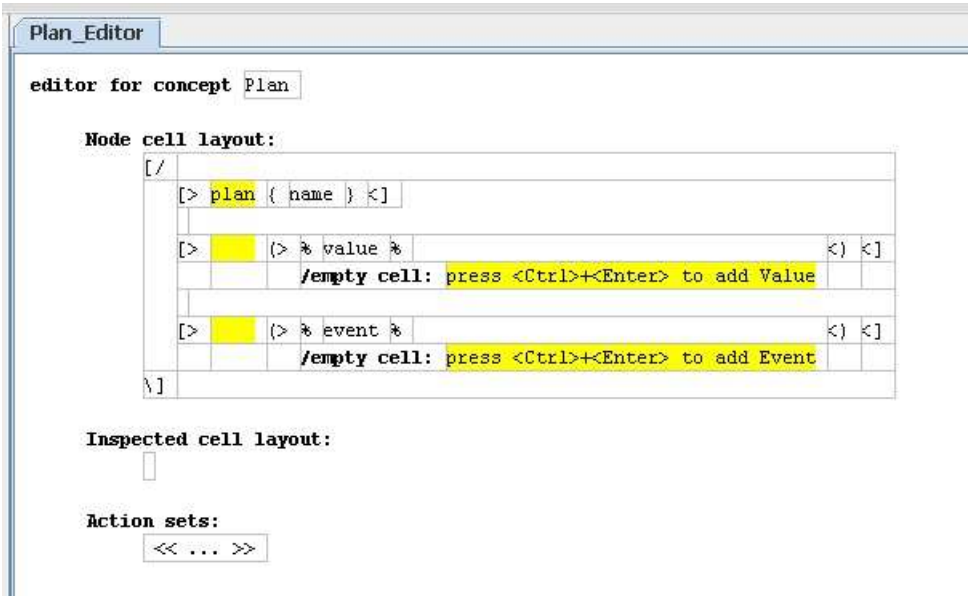

Abbildung 11: Gezeigt wird die Definition für den Editor eines Plans. Das markierte Wort ‚plan' stellt einen konstanten Bezeichner dar, der auch zur Strukturierung des Aufbaus dient.

Diese Art Editoren müssen prinzipiell für jedes Konzept erstellt werden.

Der Aufbau der Editor-Definition (siehe Abbildung 11) besteht aus vielen Zellen, in denen Konstanten oder Referenzen Platz finden. Der Aufbau erscheint aufgrund der Notationsweise recht unübersichtlich, doch ist diese leicht zu ändern und wird durch einen *Inspector*[72] unterstützt. Hier besteht noch Verbesserungsbedarf. Denkbar wären Editoren, die an spezielle Sichtweisen, Gewohnheiten oder Erfahrung der Entwickler vor-angepasst sind.

Die vertikalen Elemente des Plan-Editors sind:

- Eine Zeile, die mit *[> plan* beginnt.
- Leerzeile
- Eine Zeile, die *% value %* enthält. Eine dazugehörige folgende Zeile.
- Leerzeile
- Eine Zeile, die *% event %* enthält. Eine dazugehörige folgende Zeile.

Die drei gefüllten Zeilen entsprechen dem Aufbau des regulären Plans aus Abbildung 8. Ebenso wie für die vertikalen Elemente, gibt es Definitionen für die horizontalen Elemente, z.B. für *Event* oder *Value*. Sie sind den in Abbildung 9 und 10 dargestellten Definitionen ähnlich, werden aber hier nicht weiter erläutert.

Ist dieser Teil der Arbeit abgeschlossen, geht es an die Definition des Generators, der die DSL mittels Makrotechnik in eine allgemeine Programmiersprache übersetzt. Später soll an dieser Stelle einmal die abstrakte Repräsentation Verwendung finden.

Auf die Generierung des Codes wird hier nicht weiter eingegangen, sie kann bei M. Fowler nachgelesen werden und wird von ihm in einem gesonderten Artikel diskutiert [005].

Noch zu erwähnen ist, dass MPS einen Plugin für die Integration in IntelliJ IDEA mitbringt. Dieser befindet sich ebenfalls im Beta-Stadium. Er erlaubt die Verwendung der MPS-Konzepte in einer Java-Anwendung.

Für die Definitionen der DSLs, ihrer Werkzeuge und die Programmierung stellt JetBrains einige vordefinierte, domänenspezifische Sprachen zur Verfügung. Unter anderem sind dies die *Structure*, *Editor*, *Base*, *Collection* und *User Interface Language*.

Am Beispiel des Meta Programming System wurde gezeigt, wie eine Language Workbench aussehen kann. Das MPS befindet sich immer noch in der Entwicklungsphase, deshalb sind einige Funktionen noch nicht implementiert und an vielen Punkten, besonders in der Handhabung, besteht deutlicher Nachbesserungsbedarf. Trotzdem gibt dieses Beispiel einen Blick in Richtung der (kommenden) Language Workbenches und auf die Möglichkeiten, die das LOP mit sich bringt.

4.5 Zusammenfassung

Das Konzept des Language Oriented Programming setzt dort an, wo bisherige Programmierparadigmen, auch die objektorientierte Programmierung, ihre Probleme haben. Diese sind die Adaption mentaler Modelle einer fachbezogenen Anwendung auf die Ebene des Computers bzw. einer universell anwendbaren Programmiersprache. Die Modellierungslücke wird geschlossen, indem domänenspezifische Programmiersprachen auf der Abstraktionsebene der Fachbereiche geschaffen werden.

Das Language Oriented Programming sollte dabei, ähnlich der aspektorientierten Programmierung, nicht als Ersatz oder Nachfolger der objektorientierten Programmierung verstanden werden. Denn DSLs können mit objektorientierter Programmierung erstellt werden und von diesen verwendet werden. Eine DSL stellt häufig die Funktionalität dar, die auch durch Bibliotheken ausgedrückt werden könnte. Doch befindet sich die DSL näher an dem eigentlichen Problembereich. Das LOP stellt also eher eine Ergänzung oder Spezialisierung dar, die in Verbindung mit

[72] Ein Inspector ist ein aus IDEs bekanntes Anzeigefenster, welches die Eigenschaften eines Objekts darstellt.

der Objektorientierung die Erstellung von Anwendungen vereinfachen und die Produktivität und Softwarequalität erhöhen kann.

Durch ihre Nähe zu den Fachgebieten bieten domänenspezifische Sprachen neue Einsatzbereiche. Sie dienen Entwicklern, neben besseren Modellierungsmöglichkeiten, als Kommunikationsgrundlage mit den Fachexperten. Diese sind nun mehr denn je in der Lage, den „Anwendungscode", dargestellt in den entsprechenden Notationsformen, zu verstehen. Sie können eventuell sogar selbst in einer DSL programmieren. Doch dass Fachexperten nun eigenständig in der Lage sind Anwendungen zu erstellen, ist sehr unwahrscheinlich. Fachexperten werden aber in die Lage versetzt, die Anwendung auf der Ebene des „Quellcodes" auf Korrektheit prüfen. Sie können also so genannte *Code-Reviews* selbst durchführen und demzufolge die Softwarequalität steigern. Es ist zudem möglich, aus geeigneten DSLs eine Dokumentation zu generieren. In der Regel sollte dies jedoch nicht nötig sein, da der DSL-„Code" selbst dokumentierend sein sollte.

Unterstützt wird die Erstellung und Verwendung von DSLs durch Language Workbenches. Diese machen das LOP-Paradigma letztlich marktfähig, indem sie durch ihre Werkzeug-Unterstützung die aufwändigere, manuelle Erstellung von DSLs und die Arbeit mit ihnen erheblich erleichtern. DSLs können zwar auch ohne Language Workbenches erstellt und verwendet werden, doch fehlt ihnen dann das Potential einer auf sie zugeschnittenen Entwicklungsumgebung.

Ein wesentlicher Bestandteil der Architektur einer Language Workbench ist die Speicherung einer abstrakten Repräsentation anstatt eines Quellcodes. Diese Repräsentation kann in verschiedenen Projektionen dargestellt werden, beispielsweise grafisch oder textuell. Die Projektionen müssen nicht alle Details darstellen. Sie können, je nach Detailgrad und Projektionsform unterschiedliche Aspekte der Darstellung (der abstrakten Repräsentation) betonen.

Durch die neuartigen Darstellungsformen wird die Frage nach einer Versionskontrolle von neuem aufgeworfen. Diese ist für textuelle Programmiersprachen in den heutigen IDEs gut gelöst worden. UML-Werkzeuge, die Versionsmanagement für Diagramme anbieten, haben schon bewiesen, dass eine Versionskontrolle für grafische Notationen gut möglich ist. Doch darf man gespannt sein, was sich hier auf dem Sektor der Language Workbenches entwickeln wird.

Ein weiteres wichtiges Kriterium für den Erfolg der Language Workbenches bzw. des LOP wird die Veränderbarkeit und Wartbarkeit der DSLs in ihrer Evolution sein. Anforderungen an Anwendungen sind fast nie statischer Natur, sodass auch DSLs von Veränderungen betroffen sein werden. Noch ist es nicht an der Zeit, diesen Punkt einschätzen zu können, doch die mangelhafte Unterstützung dieses Kriteriums wäre nachteilig.

Ebenfalls zu beachten ist, dass für die Definitionen einer DSL (Schema, Editor und Generator) im Moment keine Standards in Sicht sind. Ein Wechsel der Plattform zu einer anderen Language Workbench ist also nicht möglich, ohne die Definitionen zu erneuern. Dies kann den Nachteil der Herstellerabhängigkeit bewirken, der bei einem Einsatz dieser Technologie zu bedenken ist.

Fazit

Domänenspezifische Sprachen sind keine Neuheit. Lange schon existieren mit den Unix Little Languages und anderen Sprachen, wie SQL oder auch Microsoft Excel[73], weit verbreitete Vertreter der externen DSLs. Diese sind im Vergleich zu den internen DSLs eigenständige Sprachen. Interne DSLs erweitern dagegen vorhandene Programmiersprachen. Besonders geeignet sind dazu *dynamische Programmiersprachen*[74]. Lisp hat auf diesem Gebiet eine lange Tradition.

In der letzten Zeit hat sich die Art und Weise der Auseinandersetzung mit domänenspezifischen Sprachen verändert. Sie ist ein wenig wissenschaftlicher geworden und hat sich der Probleme der heute verwendeten Programmierparadigmen angenommen. Daraus entstand ein neues Paradig-

[73] Die Produktivität von SQL und Excel stellt Tabelle 1 beispielhaft in Kapitel 2.3.3 Produktivität dar.
[74] Als *dynamisch* werden Programmiersprachen bezeichnet, deren Struktur zur Laufzeit geändert werden kann. Diese sind i.d.R. auch dynamisch typisiert. Beispiele: Lisp, Smalltalk, Ruby, Perl, Python, aber nicht C, C++, Java.

ma, das Language Oriented Programming, welches sowohl Altbekanntes, aber auch die neuen Konzepte der Language Workbenches umfasst. Language Workbenches werden zurzeit von mehreren Herstellern entwickelt, sodass in den nächsten Jahren einige interessante Produkte auf den Markt kommen werden. Verschiedene Produkte sind schon auf dem Markt, doch setzen diese das LOP nicht fokussiert um, sondern verwirklichen andere, verwandte Paradigmen.

Der Einsatz von DSLs führt zu Vorteilen wie höherer Produktivität, Verlässlichkeit, Wartbarkeit, Portabilität und Qualität. Doch stehen dem die Entwicklungs- und Schulungskosten der DSLs gegenüber. Allerdings wird dieser Aufwand durch den Einsatz von Language Workbenches auf ein vertretbares Maß reduziert, sodass das Language Oriented Programming eine ernstzunehmende Alternative zu bisherigen (Sprach-)Entwicklungsmethoden darstellt.

Die erwartete Steigerung des Potentials durch den Nutzen der Language Workbenches ist Grund genug, die Entwicklungen auf dem Gebiet des Language Oriented Programming weiter zu verfolgen. Erst wenn die Produkte auf dem Markt sind und dort eingesetzt werden, lässt sich sagen, wie hoch dieser Nutzen tatsächlich ausfällt. Bei richtigem Einsatz der Technologie, werden im Moment Produktivitätssteigerungen um den Faktor 3 bis 10 angestrebt. JetBrains Meta Programming System gibt einen guten Eindruck von dem, was von den Neuentwicklungen zu erwarten ist.

Abseits dieser Entwicklungen im „Kern“ des LOP, stellt Subtext eine sehr interessante Entwicklungsrichtung dar, der eine zukunftsweisende Denkweise zugrunde liegt.

„Java definitely was the hot technology for years. ...“ – Jason Hunter [Tate-05]

5 Einfluss und Auswirkungen

Dieses Kapitel behandelt abschließend die drei Entwicklungslinien, die in den vorigen Kapiteln vorgestellt wurden. Die Kapitel 5.1 und 5.3.1 stellen Ausnahmen dar, sie behandeln Ergänzungen zu Kapitel 4, respektive das Framework Ruby on Rails.

5.1 Standards

Dieses Unterkapitel behandelt einige Standards, die in enger Beziehung zum Language Oriented Programming stehen, aber nicht zwingender Bestandteil des LOP-Paradigmas sind.

MDA

MDA steht für Model Driven Architecture und ist ein Standard der *Object Management Group* (OMG), einem international anerkannten Konsortium mehrerer Firmen der Computerbranche. In der Diskussion der Language Workbenches geht es um ähnliche Aspekte wie bei der MDA.

Grob umrissen, beschreibt MDA ein Konstruktionsprinzip für Softwaresysteme, welches in mehrere Abstraktionsstufen aufgeteilt ist. Für jede Stufe gibt es dabei ein Modell. Die Modelle reichen von fachlich abstrakt bis hin zu plattformspezifischem Code. Alle Modellstufen, mit Ausnahme des resultierenden Anwendungscodes, verwenden den UML-Standard als Notationsform. Die OMG verfolgt mit der MDA einen ingenieurmäßigen Ansatz der Implementierung von Softwaresystemen, verbunden mit Herstellerunabhängigkeit durch Standardisierung und höherer Portabilität. Die MDA hat ihre Vorteile, ist aber nicht unumstritten. So gibt es viele Abweichler vom MDA-Standard und deutlich unterschiedliche Interpretationen dessen.

Als ein Bereich, dessen Nutzer den MDA-Standard nicht vollständig anwenden, gilt die modellgetriebene Softwareentwicklung. Im Englischen gibt es dazu gleich mehrere passende Begriffe: das *Model Driven Development* (MDD), das *Model Driven Software Development* (MDSD) und das *Model Driven Engineering* (MDE). Der Fokus dieser Begriffe liegt mehr auf der Entwicklung eines Systems unter Zuhilfenahme grafischer Notation, als auf der Implementierung bzw. Konstruktion dessen.

Im Folgenden soll ein Blick auf den Zusammenhang der Model Driven Architecture und des Language Oriented Programming mit seinen Language Workbenches geworfen werden. Einige der MDA-Werkzeuge kommen dem recht nahe, was unter eine Language Workbench verstanden wird. Doch gibt es einige grundsätzliche Unterschiede zwischen den beiden Ansätzen:

Language Workbenches bieten Unterstützung für die Entwicklung neuer DSLs und sind nicht, wie Standard-MDA-Werkzeuge, auf UML beschränkt. Die meisten MDA-Werkzeuge bieten keine Unterstützung für den Bau der, für eine DSL benötigten, Editoren und Generatoren. Ihnen fehlen, abgesehen von UML, andere Ausdrucksmöglichkeiten in Form mehrerer bzw. unterschiedlicher Notationsformen für spezielle Aufgabengebiete.

MDA und LOP besitzen Gemeinsamkeiten, doch liegen ihnen verschiedene Denkweisen zu Grunde. UML als Notationsform einzusetzen, bringt gewiss auch Vorteile für Language Workbenches mit sich. Doch ist UML nicht als domänenspezifische Modellierungssprache gedacht. Sie stellt eine universell anwendbare Notation dar und weist einen generellen Fokus auf. In der

Regel ist auch das Feedback einer Language Workbench kürzer als es beim MDA-Ansatz, durch den Einsatz mehrerer Abstraktionsstufen, der Fall ist.

Der MDA-Standard ist für das Language Oriented Programming nur von geringem Nutzen. Doch können MDA-Werkzeuge wohl das Spektrum von Language Workbenches erfüllen. Als Basis für eine Language Workbench sind die OMG Standards jedoch nicht geeignet, hauptsächlich wegen ihrer Ausrichtung auf andere Einsatzzwecke.

Die geringe Eignung der MDA kommt auch in der fehlenden Umsetzung in den Language Workbenches zum Ausdruck: Weder Microsoft, mit den Software Factories, JetBrains, mit dem Meta Programming System, noch Intentional Software halten sich an den MDA-Standard.

Andere vorgestellte Produkte, die in erster Linie keine Language Workbench darstellen, orientieren sich stärker an MDA und UML.

Charles Simonyi (Intentional Software, [016]) beschreibt MDA in Bezug auf das LOP recht zynisch, aber im Großen und Ganzen treffend:

„MDA is a kitchen-sink standard that is implementation oriented.“

UML

UML, die Unified Modeling Language, ist ebenfalls ein Standard der OMG. Sie dient zur Modellierung von Softwaresystemen und basiert auf UML-Diagrammen.

Mittlerweile ist UML zu einem komplexen Gebilde herangewachsen, das für den Bereich des LOP nur begrenzt geeignet ist. Es bringt zum einen zu viele (unnötige) Dinge mit und bietet zum anderen zu geringe Mittel, um die Entwicklung von Editoren und Generatoren einer DSL zu unterstützen.

UML besitzt eine universelle Ausrichtung hin zu einer generalisierten grafischen Programmiersprache, und ist damit nur begrenzt für den individuellen Einsatz in einer Fachdomäne geeignet. Sie umfasst mit 13 Diagrammtypen und über 1000 Elementen einen viel zu großen Umfang für ein (zu) kleines Einsatzgebiet im Rahmen des LOP.

UML findet eher in den Bereichen der MDA oder des MDD Einsatz. Hier besteht eine größere Verwandtschaft zu dem *Domain Specific Modeling* (DSM), welches im Wesentlichen mit grafischen DSLs arbeitet.

Mit *Executable UML (xUML)* steht eine neue Methode der Code-Generierung aus UML Diagrammen zur Verfügung. Dieser Ansatz ist ebenfalls umstritten, da die Interpretation von UML zu Code nicht standardisiert ist. Beim Einsatz domänenspezifischer Sprachen kann man diese Generierung hingegen selbst definieren.

Die UML erfährt, wegen ihrer Verbindung zur MDA und ihrer gleichartigen Ausrichtung, eine ähnlich geringe Eignung für den Einsatz im Language Oriented Programming. Sie stellt in der Regel einen recht umständlichen und formalen Weg dar. Dies bedeutet aber nicht, dass es im LOP keine Einsatzmöglichkeiten für die UML gibt.

5.2 Aspektorientierte Programmierung

Wie in Kapitel 4 bereits diskutiert, bringt die AOP als Erweiterung der OOP einige Vorteile mit sich. Die AOP stellt eine domänenspezifische Sprache dar, die in eine allgemeine Programmiersprache integriert wird und querschneidende Belange verbessert implementiert.

Inzwischen ist die Entwicklung der AOP an einem Punkt angekommen, an dem sie ihren Nutzen bewiesen hat und reif für den Einsatz ist. Dennoch gibt es noch ein paar offene Probleme, die eine gründliche Auseinandersetzung mit möglichen Risiken der Technik ratsam machen. Einer der Kritikpunkte ist beispielsweise die fehlende Unterstützung der AOP in der UML-Modellierung. Es ist zudem fraglich, ob das weit verbreitete AspectJ, als Vertreter der Java-Programmierung, eine gute Basis für AOP darstellt. Denn Java wird zwar vielerorts eingesetzt, weist aber, wie in Kapitel 2 diskutiert, auch deutliche Probleme auf.

Bei der Betrachtung vieler Quellen zur aspektorientierten Programmierung ist ein gewisser Hype hinter der Technik nicht zu übersehen. Einige Softwareentwickler sind überzeugt davon, dass AOP der nächste Schritt in der Entwicklung der Programmiersprachen ist. Andere Entwicklungen des Marktes, wie das aufkommende Language Oriented Programming und neuere dynamische Programmiersprachen, werden dabei meist außer Acht gelassen. Nichtsdestotrotz lässt das mit der Aspektorientierung verbundene Nutzenpotential, die AOP auf lange Sicht gesehen zu einer wahrscheinlichen Entwicklungsrichtung werden.

“If you want to work in a big company, learn how to hack Blub on Windows. If you want to work at a cool little company or research lab, you'll do better to learn Ruby on Linux. And if you want to start your own company, which I think will be more and more common, master the most powerful tools you can find, because you're going to be in a race against your competitors, and they'll be your horse.” – Paul Graham [055]

5.3 Sprachen: Java versus Ruby

Anknüpfend an Kapitel 2 behandelt dieses Unterkapitel Java und Ruby im Vergleich.

Diesem Vergleich liegt die, leicht provokative, Fragestellung zugrunde, welche der beiden Programmiersprachen zukünftig die Gunst der Entwickler gewinnen wird und somit das „Rennen“ machen wird.

Javas beste Zeit ist mit Sicherheit vorbei. In den neunziger Jahren stellte Java für die meisten Entwickler noch das Maß der Dinge dar. Heute ist Java zunehmend umstritten und stellt für viele Anwendungsgebiete nicht mehr die erste Wahl dar. Die bestehenden und aufkommenden Probleme deuten an, dass die Sprache Java, über einen längeren Zeitraum gesehen, keine Perspektive hat. Das technische Konzept der JVM steht dagegen weit besser da, so gibt es mittlerweile einige Versuche, auf der JVM aufzubauen.

Paul Graham (siehe obiges Zitat) weist darauf hin, dass eine gewöhnliche Programmiersprache, wie Java, für den Durchschnittsprogrammierer einer großen Firma durchaus geeignet ist. Doch sobald es auf Produktivität und Flexibilität ankommt, sollte man doch besser dynamische Sprachen, wie Ruby, einsetzen und die bestmöglich passenden Hilfsmittel einsetzen, die verfügbar sind.

Ein Vergleich

Java und Ruby sind beide objektorientierte Sprachen, die Einfachvererbung unterstützen. Doch trotz dieser grundlegend gleichen Ausrichtung gibt es einige wesentliche Unterschiede. Ruby besitzt einige Eigenschaften, die Java nicht aufweisen kann:

- Einfach zugängliche Metaprogrammierung
- Closures
- Reine Objektorientierung, kein hybrides Sprachkonzept
- dynamische, statt statische Typisierung
- Eine geringere Sprachkomplexität
- Erweiterung von Klassen und Bibliotheken generell, aber auch zur Laufzeit
- Unterstützung von Mixins

Besonders beim Experimentieren mit Code fällt Javas statische Typisierung negativ ins Gewicht. Die Codebeispiele 23 und 24 [Tate-05] sollen dies anhand des einfachen Rechenbeispiels der Fibonacci-Folge demonstrieren:

```
class Fib {
  public static void main (String args[ ]) {
    int x1 = 0;
    int x2 = 1;
    int total = 1;
    for (int i=0; i<10; i++) {
      System.out.println(total);
      total = x1+x2;
      x1 = x2;
      x2 = total;
    }
  }
}
```

Codebeispiel 23: Die Fibonacci-Folge in Java.

```
x1 = 0
x2 = 1
100.times do
  puts x2
  x1, x2 = x2, x1+x2
end
```

Codebeispiel 24: Die Fibonacci-Folge in Ruby.[75]

Beide Codebeispiele drücken das gleiche aus, doch die Ruby-Variante ist deutlich einfacher und schneller zu lesen als der Java-Code. In erster Linie wird hier der unterschiedlich große Aufwand beider Sprachen deutlich, der für die Implementierung der Fibonacci-Folge notwendig ist. Java benötigt dazu eine Klassendeklaration und 13 Codezeilen mit 41 Wörtern, während Ruby ohne eine Klasse auskommt und dasselbe in 6 Zeilen mit 16 Wörtern ausdrücken kann.

Sicher mag dieser Vergleich nicht für alle Gebiete stellvertretend sein, doch werden die wesentlichen Unterschiede in der Handhabung der beiden Programmiersprachen deutlich. Beim Experimentieren – und nicht nur da – verlangt Java deutlich mehr Code und Deklarationen als Ruby.

[75] Die Zuweisung *x1, x2 = x2, x1+x2* ist eine Ruby-typische parallele Zuweisung. Ausgeschrieben bedeutet sie: *temp = x1, x1 = x2, x2 = temp + x2*

Doch dieses Java-Beispiel macht noch ein anderes Problem deutlich: Würde die *for*-Schleife, wie bei der Ruby-Variante umgesetzt, einhundert Mal durchlaufen werden, so tritt ein Laufzeitfehler auf. Javas primitiver Integer-Datentyp *int* stellt für die Berechnung der Fibonacci-Folge einen zu kleinen Zahlenbereich dar. Ruby bietet an diesen Stellen eine automatische Typkonvertierung an, sodass der Programmierer dieses hausgemachte Problem nicht bedenken muss.

Die Frage, welche Programmiersprache zukünftig die Gunst der Entwickler erlangen wird, beantwortet der frühere Java-Entwickler David Heinemeier Hansson in einem Interview, in dem er seine Erfahrung mit Ruby beschreibt [029]:

"It took one day to say, 'I really like this.' It took one week to say, 'I'm never going back to PHP again.' And it took one month before my proficiency with Ruby made me run circles around my former programming capabilities in PHP. It was just such an incredibly powerful fit. Ruby fit my brain perfectly. I was having so much more fun and getting so much more done."

Diese Aussage kann sicher nicht auf alle Programmierbereiche übertragen werden, dennoch steht sie stellvertretend für eine große Entwicklergruppe – und nicht nur für die Entwickler von Web-Anwendungen.

Abschließende Überlegungen

Die Neuentwicklungen im Bereich der dynamischen Sprachen eröffnen auch für Javas Zukunft neue Möglichkeiten. Am Beispiel des in Java umgesetzten Open-Source-Frameworks *RIFE*[76] ist zu sehen, wie erfolgreiche Techniken aus der dynamischen Sprachwelt auf Java übertragen werden.

Das RIFE-Framework bildet Datenbanken auf eine bessere Art und Weise ab, als dies bisher in der Java-Welt der Fall war[77]. Es orientiert sich dabei an dem Vorbild *Active Record*, das in *Ruby on Rails* starke Verwendung findet.

Doch lässt sich die Sprache Java nicht beliebig erweitern. Javas Ausrichtung auf neue Gebiete, wie die AOP, geht nicht ohne Kosten vonstatten. Sie erfordert immer neue Werkzeuge, die die Komplexität der Java-Programmierung erhöhen. Sprachen wie Ruby haben dieses Problem beispielsweise nicht. Aspektorientierte Programmierung ist dort nahezu eine Trivialität.

Einige bekannte und führende Entwickler, die Mitte der neunziger Jahre in die aufkommende Java-Technologie involviert waren, haben sich in den letzten Jahren aus der Java-Szene zurückgezogen. Heute finden sich viele von Ihnen in der Ruby-Entwicklung wieder. Dies unterstreicht die Auffassung, dass Ruby für die meisten Aufgaben besser geeignet ist als Java.

Doch um Java zu ersetzen bzw. seine alleinige Marktführerschaft zu brechen, bedarf es mehr als einer guten Sprache. Wichtige Voraussetzungen für eine Sprache, die auf dem Markt Fuß fassen kann, sind:

- Freie Verfügbarkeit der Sprache (Open Source)
- Guter, abstrakter Sprachaufbau
- Einfache Zugänglichkeit
- Eines, besser mehrere gute Bücher über die Sprache
- Eine gute IDE-Unterstützung
- Eine Killer-Applikation

Paul Graham [Graham-04] schreibt über die Ausrichtung einer Sprache, hin zu einem mächtigen statt beschränktem Sprachwerkzeug, sehr treffend:

"I think language designers would do better to consider their target user to be a genius who will need to do things they never anticipated, rather than a bumbler who needs to be protected from

[76] http://www.rifers.org/

[77] Das Verb „abbilden" beschreibt diesen Vorgang zu ungenau. Herkömmliche Java-Lösungen *mappen* Datenbankobjekte, während Active Record diese *wrappt*. Das Wrappen stellt hier die bessere Abbildung der Datenobjekte dar.

himself. The bumbler will shoot himself in the foot anyway. You may save him from referring to variables in another module, but you can't save him from writing a badly designed program to solve the wrong problem, and taking forever to do it."

Fazit

Java ist Mitte der neunziger Jahre das gewesen, was Ruby heute ist: Eine aufkommende Sprache, hinter der ein gewisser Hype steht. Doch standen hinter Java damals eine Menge treibender kommerzieller Kräfte, so muss Ruby heute ohne ein großes Marketing auskommen. Ruby ist eine in jeder Hinsicht freie Sprache, die von Programmierern für Programmierer erstellt worden ist. Der Trend hin zu Web-Anwendungen im Zuge von Web2.0 baut stark auf dynamischen Sprachen auf. Javas Konzepte gehören hier der Vergangenheit an und sind nur begrenzt geeignet.

Javas Zukunft sieht mittlerweile nicht mehr so rosig aus wie vor 10 Jahren. Ruby hat inzwischen die technologische Vorreiterrolle übernommen und hat die besten Chancen, in den nächsten Jahren zu größerer Bedeutung zu kommen. Java stellt, der Meinung einiger Experten zufolge, auf lange Sicht eher eine Sackgasse in der Entwicklung der Programmiersprachen dar, deren Bedeutung zukünftig, aller Wahrscheinlichkeit nach, abnehmen wird.

Ruby profitiert derweil ungemein von seinem populären Rails-Framework, das Rubys Potential deutlich macht und deshalb im Folgenden kurz vorgestellt werden soll:

„Ruby allows Rails to implement convention over configuration at runtime, ..." – David Heinemeier Hansson [Tate-05]

5.3.1 Ruby on Rails

Ruby on Rails[78] ist ein enorm produktives Open-Source-Framework von David Heinemeier Hansson, welches auf der Sprache Ruby aufbaut und deren Stärken bestens nutzt. Es basiert auf den Frameworks *Active Record*[79] und *Action Pack*[80].

Das weit verbreitete Aufgabengebiet von Ruby on Rails sind Web-Anwendungen, die auf einer relationalen Datenbank basieren[81].

Eine der wesentlichen Stärken von Ruby on Rails ist *Convention over Configuration*. Dies beschreibt standardmäßig verwendete Vereinbarungen von Konventionen, die etwa in 90% der Fälle den Anforderungen einer Anwendung gerecht werden. Sollte dennoch eine Anpassung nötig sein, können alle Einstellungen bzw. Konventionen beliebig abgeändert werden. Dabei basiert die Konfiguration von Ruby on Rails nicht auf den bei Java üblichen, z.T. schlecht lesbaren, XML-Dateien. Es werden stattdessen Ruby-eigene Sprachmittel eingesetzt, um einen klaren Aufbau des recht geringen Konfigurationsaufwands umzusetzen.

Ruby in Rails besitzt einen verkürzten Feedback-Zyklus. Änderungen am Programm werden im Web-Browser direkt durch ein Neuladen der Seite umgesetzt; das Übersetzen des Codes entfällt hier. Aufgrund der Spracheigenschaften, die Ruby mitbringt, ist es sogar während der Laufzeit möglich, bestehenden Klassen neue Funktionalität hinzuzufügen.

[78] http://www.rubyonrails.org/
[79] Active Record verwaltet relationale Datenbanksysteme und stellt eine sehr gute Umsetzung einer DSL dar.
[80] Action Pack verarbeitet Web-Anfragen und implementiert das Model-View-Controller Design-Pattern (MVC). Nebenbei unterstützt es Logging und Caching.
[81] Dieser Bereich war früher der Fokus von Java, der C++-Anwendern den Weg ins Internet ebnete. Doch seitdem verlagert sich Javas Fokus immer weiter in Richtung Enterprise-Anwendungen, sodass der Forschritt bei den Web-Anwendungen vernachlässigt wird.

In der Regel wird mit der Kombination aus Ruby und dem Rails-Framework weniger Code als üblich erzeugt, der zudem einfacher gewartet werden kann. Die Ausführungsgeschwindigkeit einer Ruby on Rails Anwendung liegt mindestens gleichauf mit gängigen Java-Lösungen.

Zugänglichkeit

Die Installation von Ruby on Rails ist sehr einfach und mit wenigen Mausklicks erledigt. Mit einem kurzen Befehl ist ein Projekt angelegt, und die Arbeit kann beginnen.

Einzig eine Datenbank benötigt das Framework noch. Ein Webserver ist hingegen schon dabei, dieser kann aber auch durch einen *Apache*-Plugin ersetzt werden

Da Ruby on Rails auf Active Record basiert, kann es automatisch die Tabellen der Datenbank erkennen und die enthaltenen Daten auf Objekte abbilden. Die für die Objekte nötigen Attribute, Getter, Setter und Datenbank-Zugriffsmethoden werden für die entsprechenden Klassen dynamisch generiert. Diese Generierung geschieht zur Laufzeit, sodass der erzeugte Code nicht gewartet werden muss.

Durch die Konventionen von Ruby on Rails ist es möglich, eine kleine Weboberfläche innerhalb weniger Minuten zu erzeugen. Diese entspricht nicht unbedingt den ästhetischen Ansprüchen der meisten Webentwickler, interagiert aber schon perfekt mit der Datenbank. Sollte man nun Änderungen an den per Konvention festgelegten Eigenschaften vornehmen wollen, so ist es kein Problem, die Vorlagen von Ruby on Rails durch eigenen Code zu ersetzen.

Eine tiefergehende Beschreibung und eine Einweisung in Ruby on Rails und dessen Techniken finden sich bei Bruce Tate ([Tate-05], [031]) und bei Curt Hibbs ([032],[033]).

Ruby on Rails wird inzwischen für populäre sowie kommerzielle Internetauftritte eingesetzt und hat sich dort bewährt. In letzter Zeit erscheinen die ersten Bücher über Ruby und das Rails-Framework von namhaften Entwicklern (z.B. Dave Thomas), die zusätzlich im Internet darüber berichten.

Ruby on Rails wird derweil von über 4000 Entwicklern professionell eingesetzt. Dies zeigt auch, dass der Hype hinter Ruby on Rails nicht ganz unbegründet ist, obwohl Hypes immer mit einer gewissen Vorsicht zu genießen sind.

Fazit

Ruby on Rails erfreut sich inzwischen großer Aufmerksamkeit und dringt bis in die Java-Community vor. Das Rails-Framework kann inzwischen fast alles umsetzen, was mit Java und den entsprechenden Zusatzwerkzeugen möglich ist. Doch Ruby on Rails braucht dafür weniger Werkzeuge und erledigt vieles eleganter. Ein erwähnenswerter Vorteil ist die mögliche Produktivitätssteigerung gegenüber Java-Frameworks um den Faktor 10.

Auch das steigende Interesse an den Metaprogrammierungstechniken kann in Java, aus bereits erwähnten Gründen, nicht umgesetzt werden.

Das Rails-Framework ist ohne Zweifel ein Katalysator für die Sprache Ruby, der für einen immer größeren Bekanntheitsgrad sorgt. Es erlaubt einzelnen Ruby-Entwicklern produktiver als ganze Java-Teams zu sein. Die Bezeichnung Killer-Applikation trägt das Rails-Framework deshalb zu Recht, denn es definiert die nächste Stufe der Webentwicklung.

5.4 Language Oriented Programming

Language Oriented Programming (LOP) weist mit den Language Workbenches sehr interessante Sprachentwicklungswerkzeuge für domänenspezifische Sprachen aus. Die Language Workbenches befinden sich noch in der Entwicklung und werden wohl noch einige Jahre benötigen, bis sie die vollen Möglichkeiten des LOP nutzen können. Die benötigte Zeit hin zu marktreifen Produkten und deren Einsatz durch führende Entwickler ist laut Martin Fowler auch der größte Schwachpunkt dieser Werkzeuge. Dennoch versprechen die Language Workbenches eine umfangreiche Unterstützung der Arbeit mit domänennahen Sprachen. Sie wollen einerseits die Abstraktionsebene der Entwicklung verbessern und andererseits Fachexperten enger in die Entwicklung einbinden.

Language Oriented Programming ist bereits auf dem Markt präsent. Externe DSLs sind weit verbreitet und interne DSLs werden durch das steigende Interesse an Metaprogrammierung gefördert.

Das LOP wird klassische Programmiersprachen in absehbarer Zeit nicht vom Markt verdrängen können. In Softwaresystemen wird es immer Bereiche geben, in denen der Einsatz einer DSL einen zu hohen Aufwand bedeutet und nicht lohnenswert ist. Auf kürzere Sicht ist zum einen eine bessere Erweiterung von Sprachen wahrscheinlich, die etwa ausdrucksstarke Frameworks wie Ruby on Rails ermöglicht, zum anderen der Einsatz weniger externer DSLs für besonders geeignete Gebiete, der nicht zwangsweise eine Language Workbench erforderlich macht.

Das Paradigma des LOP weist die nötigen Mittel auf, die Programmierung der heutigen Zeit weiter zu beeinflussen, doch wird dies nicht von heute auf morgen geschehen. Richtungsweisend im Bereich des LOP ist die Abkehr von der klassischen, rein textuellen Programmierung hin zu mehreren bzw. unterschiedlichen Programm-Repräsentationsformen.

Ein besonders radikaler und exotischer Vertreter einiger mit dem LOP verbundenen Ideen ist Subtext. Jonathan Edwards hat Subtext als erstes Experiment implementiert, um eine völlig neue Art des Programmierens auszuprobieren und salonfähig zu machen. Die Ansätze von Subtext klingen viel versprechend, obwohl es noch in den Kinderschuhen steckt. Für dieses neue Programmierparadigma sind einige grundlegende Änderungen in der Denkweise der Programmierung notwendig, die den Umgang mit der neuen Programmiertechnik nicht gerade vereinfachen.

Subtext stellt auf absehbare Zeit einen Außenseiter in der Programmierung dar, der neue Ideen und Konzepte mit sich bringt, welche z.B. auf das LOP Einfluss nehmen könnten. Auf Sicht von mehreren Jahren könnte Subtext heranreifen und das zugehörige Paradigma könnte auf nicht absehbare Zeit zu größerer Bedeutung gelangen. Doch beides ist zum aktuellen Stand der Entwicklungen eher Spekulation als Realität.

Martin Fowler [002] hat in wenigen Sätzen, die dieses Unterkapitel abschließen sollen, die Position der Language Workbenches in der Softwareentwicklung sehr treffend beschreiben:

“This promise of bringing domain experts more directly into the development effort is perhaps the most tantalizing part of the language workbench promise. Time and time again we see that whatever tools we programmers use to boost our productivity there's a sense that we are optimizing the idle loop. On most projects I visit the biggest issue is the communication between the developers and the business. If that's working well, then you can make progress even with second rate technology. If that relationship is broken, then even Smalltalk won't save you.”

“If you try to solve a hard problem, the question is not whether you will use a powerful enough language, but whether you will (a) use a powerful language, (b) write a de facto interpreter for one, or (c) yourself become a human compiler for one.” – Paul Graham [055]

6 Fazit

Der Überblick auf die Situation im Bereich der Programmiersprachen zeigt, dass es in den letzten Jahren einige Veränderungen auf dem Gebiet der Sprachentwicklung gegeben hat. Die Entwicklung der Programmiersprachen als Ganzes schreitet langsam voran und bringt die Probleme einiger Sprachen ans Tageslicht. Nebenbei entstehen neue Sprachen und Programmierparadigmen, die für die Zukunft interessant zu sein scheinen. Welche Veränderung als nächstes ansteht, lässt sich, wie so oft, nicht genau sagen. Doch aufgrund einiger unumgänglicher Tatsachen lässt sich eine deutliche Tendenz erkennen:

Die Sprachentwicklung tendiert hin zu dynamischen und objektorientierten Open-Source-Sprachen, darunter im Speziellen zu Perl, Ruby und Python. Sprachen wie C++ und Java, ebenfalls objektorientiert, sind weit verbreitet, zeigten aber besonders im letzten Jahrzehnt, dass sie nicht uneingeschränkt eingesetzt werden können. Java weist mittlerweile massive Probleme auf, die nicht mehr so einfach zu ignorieren sind, sondern Entwicklern und Firmen die Programmierung deutlich erschweren. Bruce Tate erwähnt hier Javas Schwerfälligkeit, die nach 10 Jahren Entwicklung, bei einer Datenbankanbindung an eine Web-Oberfläche, noch vorhanden ist. Während Java mit seinen grundlegenden Designentscheidungen zu kämpfen hat, kommen innovative Ideen in anderen (Sprach-)Bereichen zum Vorschein:

Die aspektorientierte Programmierung verbreitet sich Stück für Stück und auch die Abstraktionsprobleme der Programmiersprachen werden mit dem Language Oriented Programming angegangen.

Im Bereich der Frameworks machen Seaside mit Continuation Servers und Active Record mit Ruby on Rails von sich reden. Diese basieren stark auf Metaprogrammierungstechniken und dynamischen Sprachen.

Auf lange Sicht könnte das revolutionäre Konzept von Subtext von Bedeutung sein. Jonathan Edwards beschreitet hier Neuland in der Sprachentwicklung und setzt neue Denkweisen um, die mit einigen bisher weit verbreiteten Vorstellungen brechen.

Die genannten Einflüsse stellen in der Entwicklung der Sprachen ernsthafte Alternativen dar.

Die aspektorientierte Programmierung weist schon einige erfolgreiche Umsetzungen auf. Noch fehlen hier grundlegende Standards, doch das Potential zu einer sinnvollen Ergänzung ist vorhanden.

Dynamische Programmier- bzw. Skriptsprachen haben inzwischen bewiesen, dass ihrem kommerziellen Einsatz grundsätzlich nichts im Wege steht. Sie versprechen, die nötige Fachkenntnis vorausgesetzt, eine Steigerung der Produktivität und Flexibilität bei der Umsetzung neuer Konzepte und Techniken. Ruby ist hier momentan der stärkste Vertreter dieser Sprachfraktion und bietet genügend Argumente für einen überzeugenden Einsatz.

Das Language Oriented Programming zeigt mit seinen Language Workbenches erste viel versprechende Ergebnisse in der Unterstützung domänenspezifischer Sprachen. Es wird aber wohl noch 5 bis 10 Jahre dauern, bis es größere Bedeutung erlangen kann.

Viele der Techniken, die aktuell wieder aufkommen, waren schon mit Sprachen wie Lisp oder Smalltalk möglich. Diese Sprachen stießen aber nicht auf breite Akzeptanz.

Die Akzeptanz ist ein großes Fragezeichen in Bezug auf Neuentwicklungen. Gerade neue Programmiersprachen haben es schwer, da Java weit verbreitet ist und viele Entwickler, so zeigt es die Erfahrung, den Umstieg auf Neuentwicklungen vermeiden möchten. Ruby hat hier durch seine Killer-Applikation Rails verhältnismäßig gute Chancen, recht zügig eine hohe Verbreitung zu erfahren.

Aufgrund dieser parallelen Entwicklungslinien und den Akzeptanzhürden ist es schwer zu sagen, welches *die nächste* Veränderung sein wird bzw. ob es diese überhaupt geben wird. Es ist ebenso möglich, dass diese Konzepte und Paradigmen gleichzeitig oder im Laufe der Zeit zu größerer Bedeutung kommen werden. Nach derzeitigem Stand bieten Ruby (on Rails) und die aspektorientierte Programmierung die größten Möglichkeiten, da das Language Oriented Programming noch nicht marktreif ist.

Vorsicht ist, wie bei vielen neuen Errungenschaften, vor dem Hype hinter Ruby, der AOP und dem LOP geboten. Dieser ist zwar nicht unbegründet, aber innerhalb der Communities durchaus erkennbar.

Ausblick

„If you look at these languages in order, Java, Perl, Python, Ruby, you notice an interesting pattern. At least, you notice this pattern if you are a Lisp hacker. Each one is progressively more like Lisp.” - Paul Graham [Graham-04]

Paul Graham und andere aufmerksame Beobachter haben erkannt, dass die Evolution der Sprachen inzwischen wieder in Richtung Lisp bzw. zu einer höheren Mächtigkeit und Open Source tendiert. Ruby stellt dafür den aktuellsten Beweis dar.

Im Zuge dieser Evolution der Sprachen sind auch die (mathematischen) Fähigkeiten der Entwickler wieder gefragter, die, nach Peter Gabriel [071], für das Verständnis der Abstraktionsmodelle von Programmiersprachen von Bedeutung sind.

Die Entwicklung bedeutet für Javas Perspektive früher oder später ein Ende. Doch in den nächsten Jahren wird sich an der Dominanz von Java voraussichtlich nicht viel ändern.

Eigenschaften wie die Performanz einer Sprache an sich spielen heutzutage keine übergeordnete Rolle mehr. Programmierer entwickeln performante Anwendungen eher durch intelligente Umsetzung von Algorithmen, als durch die Wahl der Sprache. Doch für einen guten Umgang mit der Sprache und deren Optimierung ist ein gutes Sprachdesign von immer größerer Bedeutung.

Jenseits konkreter Programmiersprachen stellen die Aspektorientierung und das Language Oriented Programming auf Sicht einiger Jahre wertvolle und ergänzende Techniken für die Programmierung dar.

Vor allem aber soll diese Arbeit zum kritischen Denken und zum Blick „über den Tellerrand hinaus“ anregen. Das Erlernen anderer Sprachen eröffnet Programmierern neue Perspektiven und bringt ihnen wichtige Techniken nahe, die das Verständnis vieler Konzepte erleichtern. Selbst bei selten eingesetzten Sprachen wie Lisp ist dies der Fall.

Da die Evolution von Programmiersprachen langsamer voranschreitet als die der Technologien, lässt die Zukunft noch wesentliche Möglichkeiten der (Sprach-)Entwicklung offen. Ein viel zitierte Aussage von Alan Kay unterstreicht dies:

“The real romance is out ahead and yet to come. The computer revolution hasn't started yet. Don't be misled by the enormous flow of money into bad defacto standards for unsophisticated buyers using poor adaptations of incomplete ideas.” [052]

Glossar

Aspektorientierte Programmierung (AOP)	Eine Erweiterung der objektorientierten Programmierung um Aspekte.
AspectJ	Erweiterung der Programmiersprache Java, die den Ansatz der aspektorientierten Programmierung implementiert. (Siehe auch Aspekt.)
Aspekt	Grundlage, um Crosscutting Concerns zu implementieren. Speziell in Java bzw. AspectJ eine Erweiterung des Klassen-Konzepts.
ByteCode	ByteCode stellt nicht maschinenspezifischen, übersetzten Programmcode (Maschinencode) dar. Er ist von einem virtuellen Computer interpretierbar (siehe auch virtuelle Maschine).
Cast / Typcast	Ein Cast ist eine Typumwandlung eines Objekts. Dabei wird das Objekt nach der Konvertierung mit dem Methodenprotokoll einer anderen Klasse angesprochen
Closure	Eine Closure beschreibt ein Stück Code, welches als Argument an Methoden übergeben wird und zeitversetzt ausgeführt wird. Dabei fängt die Closure mittels freier Variablen die lokale Umgebung ein.
Code	Auch als Quelltext bezeichnet. Quelltext ist Darstellung von Programmanweisungen. Dies geschieht in der Regel in Textform, ist aber nicht an diese gebunden (siehe DSL).
Crosscutting Concerns	C.C. beschreiben die querschneidenden Belange bzw. Bestandteile eines Systems, die sich über alle Klassen erstrecken.
Domain Specific Language (DSL)	Eine domänenspezifische Sprache ist eine, i.d.R. individuell ausgerichtete Sprache. Eine DSL ist auf ein spezielles Fachgebiet begrenzt und dort besonders ausdrucksstark.
Entwicklungsumgebung (IDE)	Eine grafische Benutzeroberfläche, die Entwickler als Unterstützung bei der Programmierung benutzen.
Generics	Generics sind die Java-typische Teil-Umsetzung der parametrischen Polymorphie.

Killer Applikation	Eine Anwendung, die als Katalysator eine Basistechnik, hier eine Programmiersprache, bekannt macht.
Language Workbench	Eine Art Entwicklungsumgebung des Language Oriented Programming für domänenspezifische Sprachen (siehe DSL).
Language Oriented Programming (LOP)	Ein Programmierparadigma, das die Objektorientierung ergänzt. Es basiert auf individuellen, auf ein Fachgebiet angepassten, Programmiersprachen.
Metaprogrammierung	Metaprogrammierung beschreibt die Veränderung bzw. Programmierung der verfügbaren Sprachmittel und Eigenschaften der Klassen.
Model Driven Architecture (MDA)	Die Model Driven Architecture ist ein Standard der OMG. Sie beschreibt verschiedene Modellierungsstufen für ein Softwaresystem.
Reflection	Reflection beschreibt eine Technik der Metaprogrammierung, bei der ein System Informationen über seine eigene Struktur erhält.
Refactoring	Refactoring beschreibt das Überarbeiten von Code, der aufgrund von Veränderungen im System diesem so nicht mehr gerecht wird. Es wird dabei kein neuer Code erzeugt und auch das Verhalten des Codes bleibt unverändert.
Ruby on Rails	Ein auf Ruby basierendes, produktives Web Framework.
Weaving	Ein Präprozessor verbindet bzw. verwebt die Aspekte mit dem (Komponenten-)Code eines Systems.

Tabellenverzeichnis

Abbildungsverzeichnis

Literaturverzeichnis

[Graham-04]: Paul Graham: Hackers and Painters
Sebastopol, O'Reilly Media, Inc., 0-596-00662-4, 2004

[Tate-05]: Bruce A. Tate: Beyond Java
Sebastopol, O'Reilly Media, Inc., 0-596-10094-9, 2005

[Hunt-99]: Andrew Hunt, David Thomas: The Pragmatic Programmer
Massachusetts, Addison Wesley Longman, Inc., 0-201-61622-X, 1999

[Gamma-95]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns
Massachusetts, Addison Wesley Professional, 0-201-63361-2, 1995

[001]: Martin Fowler: Articles
http://martinfowler.com/articles.html
-, 08.07.2006

[002]: Martin Fowler: Language Workbenches: The Killer-App for Domain Specific Languages?
http://martinfowler.com/articles/languageWorkbench.html
12.06.2005, 08.07.2006

[003]: Martin Fowler: A Language Workbench in Action - MPS
http://martinfowler.com/articles/mpsAgree.html
12.06.2005, 08.07.2006

[004]: Martin Fowler: Language Workbenches and Model Driven Architecture
http://martinfowler.com/articles/mdaLanguageWorkbench.html
12.06.2005, 08.07.2006

[005]: Martin Fowler: Generating Code for DSLs
http://martinfowler.com/articles/codeGenDsl.html
12.06.2005, 08.07.2006

[006]: Martin Fowler: Using the Rake Build Language
http://martinfowler.com/articles/rake.html
10.08.2005, 08.07.2006

[007]: Martin Fowler: How .NET's Custom Attributes Affect Design
http://martinfowler.com/ieeeSoftware/netAttributes.pdf
09/2002, 08.07.2006

[008]: Martin Fowler: Reducing Coupling
http://martinfowler.com/ieeeSoftware/coupling.pdf
07/2001, 08.07.2006

[009]: Martin Fowler: PostIntelliJ
http://martinfowler.com/bliki/PostIntelliJ.html
-, 08.07.2006

[010]: Martin Fowler: DomainSpecificLanguage
http://martinfowler.com/bliki/DomainSpecificLanguage.html
13.02.2004, 08.07.2006

[011]: Martin Fowler: Closure
http://martinfowler.com/bliki/Closure.html
08.09.2004, 08.07.2006

[012]: Markus Völter, Arno Haase: Textuelle DSLs – Programmiersprachen zum selbermachen
http://www.voelter.de/data/articles/TextuelleDSLs.pdf
21.09.2005, 08.07.2006

[013]: Sam Pullara: What should a domain specific language workbench look like?
http://www.javarants.com/C1464297901/E20050710165403/index.html
10.07.2005, 08.07.2006

[014]: empros GmbH, „nemo“: API-Entwurf ist DSL-Enwurf
http://www.empros.ch/nemoslog/apientwurfistdslentwurf.php
25.07.2005, 08.07.2006

[015]: Bruce Tate: Domain-specific languages in Active Record and Java programming
http://www-128.ibm.com/developerworks/java/library/j-cb04046.html
04.04.2006, 08.07.2006

[016]: Code Generation Network, Charles Simonyi Interview
http://www.codegeneration.net/tiki-read_article.php?articleId=61
14.07.2004, 08.07.2006

[017]: Cunningham & Cunningham, Inc.: Domain Specific Language
http://c2.com/cgi/wiki?DomainSpecificLanguage
28.04.2006, 08.07.2006

[018]: Arie van Deursen, Paul Klingt, Joost Visser: Domain-Specific Languages:
An Annotated Bibliography
http://homepages.cwi.nl/~arie/papers/dslbib/
09.02.2000, 08.07.2006

[019]: Guy Steele: bindings and assignments
http://article.gmane.org/gmane.comp.lang.lightweight/2274
21.08.2003, 08.07.2006

[020]: Sergey Dmitriev: Language Oriented Programming: The Next Programming Paradigm
http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf
2004, 08.07.2006

[021]: Alan Cameron Wills: Why not base domain specific languages on UML?
http://blogs.msdn.com/alan_cameron_wills/archive/2004/11/11/255831.aspx
11.11.2004, 08.07.2006

[022]: Wikipedia: Domain-specific programming language
http://en.wikipedia.org/wiki/Domain-specific_programming_language
25.06.2006, 08.07.2006

[023]: Code Generation Network: Dave Thomas Interview
http://www.codegeneration.net/tiki-read_article.php?articleId=9
02.04.2003, 08.07.2006

[024]: Code Generation Network: Sergey Dmitriev Interview about Fabrique and MPS
http://www.codegeneration.net/tiki-read_article.php?articleId=60
16.07.2004, 08.07.2006

[025]: Neal Ford: Language Oriented Programming
http://www.nealford.com/downloads/conferences/2006_nfjs_canonical/2006_nfjs_LOP_keynote.pdf
28.04.2006, 08.07.2006

[026]: Microsoft Corporation, Visual Studio 2005 Team System Modeling Strategy and FAQ
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvs05/html/vstsmodel.asp
05/2005, 08.07.2006

[027]: Markus Völter: Languages, Models, Factories
http://www.voelter.de/data/presentations/LanguagesModelsFactories.ppt
18.01.2006, 08.07.2006

[028]: Winfried Mueller, Ruby Tutorial: Eine Einladung in Ruby
http://www.wikidorf.de/reintechnisch/Inhalt/EinladungInRuby
13.07.2005, 08.07.2006

[029]: Edd Dumbill: Ruby on Rails: An Interview with David Heinemeier Hansson
http://www.oreillynet.com/lpt/a/6170
30.08.2005, 08.07.2006

[030]: Garrett Rooney: Extending Ruby with C
http://onlamp.com/lpt/a/5371
18.11.2004, 08.07.2006

[031]: Bruce Tate: Exploring Active Record
http://www-128.ibm.com/developerworks/java/library/j-cb03076/
07.03.2006, 08.07.2006

[032]: Curt Hibbs: Rolling with Ruby on Rails
http://onlamp.com/lpt/a/5546
20.01.2005, 08.07.2006

[033]: Curt Hibbs: Rolling with Ruby on Rails, Part 2
http://onlamp.com/lpt/a/5641
03.03.2005, 09.07.2006

[034]: Markus Völter: Modellgetriebene Softwareentwicklung
http://www.voelter.de/data/articles/MDSD.pdf
11.11.2004, 09.07.2006

[035]: Neal Ford: Using the Ruby Development Tools plug-in for Eclipse
http://www-128.ibm.com/developerworks/opensource/library/os-rubyeclipse/
11.10.2005, 09.07.2006

[036]: Yukihiro Matsumoto: What's Ruby
http://www.ruby-lang.org/en/20020101.html
15.01.2005, 09.07.2006

[037]: Wikipedia: Aspektorientierte Programmierung
http://de.wikipedia.org/wiki/AOP
09.07.2006, 09.07.2006

[038]: Rolf Dornberger, Andreas Reber, Alexandra Hochuli: Aspektorientierte Programmierung
http://www.fhso.ch/pdf/publikationen/dp05-03.pdf
02/2005, 09.07.2006

[039]: Oliver Böhm: Aspektorientierte Programmierung mit AspectJ5, Kapitel 14
http://www.dpunkt.de/leseproben/3-89864-330-1/Kapitel_14.pdf
12/2005, 09.07.2006

[040]: Oliver Böhm: Aspektorientierte Programmierung mit AspectJ5, Kapitel 1
http://www.dpunkt.de/leseproben/3-89864-330-1/Kapitel_1.pdf
12/2005, 09.07.2006

[041]: Oliver Böhm: Ist AOP reif für die Praxis?
http://www.agentes.de/documents/aop-ix-konferenz0603.pdf
03/2006, 09.07.2006

[042]: Keith Short: Modeling Languages for Distributed Applications
http://msdn.microsoft.com/vstudio/teamsystem/architect/default.aspx?pull=/library/en-us/dnvsent/html/vsent_ModelingLangs.asp
10/2003, 09.07.2006

[043]: Markus Völter, Martin Lippert: Die 5 Leben des AspectJ
http://www.voelter.de/data/articles/5LebenDesAspectJ.pdf
01.04.2004, 09.07.2006

[044]: Mira Mezini, Klaus Ostermann: CeasarJ Homepage
http://caesarj.org/
-, 11.07.2006

[045]: Glenn Vanderburg: Domain Specific Languages for Programmers
http://www.vanderburg.org/Speaking/Stuff/oscon05.pdf
01.08.2005, 13.07.2006

[046]: Hartmut Krasemann: Anforderungen an eine Programmiersprache
http://users.informatik.haw-hamburg.de/~raasch/AKOT/AnforderungenAnEinePS-3.pdf
05/2006, 09.07.2006

[047]: Sam Ruby: Beyond Java
http://intertwingly.net/blog/2005/11/01/Beyond-Java/
01.11.2005, 11.07.2006

[048]: David Heinemeier Hansson: Loud Thinking: Ruby is bad Python, but good Perl
http://www.loudthinking.com/arc/000551.html
19.12.2005, 11.07.2006

[049]: David Heinemeier Hansson: Ruby on Rails
http://www.rubyonrails.org/
-, 12.07.2006

[050]: Yukihiro Matsumoto: Yield to the Block: The Power of Blocks in Ruby
http://www.rubyist.net/~matz/slides/oscon2005/index.html
01.08.2005, 13.07.2006

[051]: Martin Fowler: ColectionClosureMethod
http://martinfowler.com/bliki/CollectionClosureMethod.html
01.08.2005, 13.07.2006

[052]: Wikiqoute: Alan Kay
http://en.wikiquote.org/wiki/Alan_Kay
17.06.2006, 14.07.2006

[053]: Wikiqoute: Edsger Wybe Dijkstra
http://en.wikiquote.org/wiki/Edsger_Wybe_Dijkstra
05.06.2006, 14.07.2006

[054]: Wikiqoute: Lisp Programming Language
http://en.wikiquote.org/wiki/Lisp_programming_language
20.03.2006, 14.07.2006

[055]: Wikiquote: Paul Graham
http://en.wikiquote.org/wiki/Paul_Graham
09.07.2006, 14.07.2006

[056]: Wikiqoute: Programming
http://en.wikiquote.org/wiki/Programming
13.07.2006, 14.07.2006

[057]: Wikiqoute: Python
http://en.wikiquote.org/wiki/Python
09.12.2005, 14.07.2006

[058]: SourceForge.net: Ruby Development Tool
http://sourceforge.net/projects/rubyeclipse
27.04.2006, 14.07.2006

[059]: Andreas Knauer, Christopher Kristes: Aspektorientierte Programmierung: AspectJ und HyperJ
http://www.gm.fh-koeln.de/~ehses/seminar/ergebnisse03/Aspektorientierte-Programmierung/ausarbeitung.pdf
17.02.2004, 09.07.2006

[060]: Martin Lau: Aspektorientierte Programmierung und interne Softwarequalität: Eine Untersuchung am Beispiel von AspectJ
Hochschule für Angewandte Wissenschaften Hamburg
20.06.2003, 09.07.2006

[061]: Olaf Spinczyk: AspectC++
http://www.aspectc.org/
25.05.2006, 14.07.2006q

[062]: - : AspectJ
http://www.eclipse.org/aspectj/
30.06.2006, 14.07.2006

[063]: Avi Bryant, Robert Feldt: AspectR
http://aspectr.sourceforge.net/
21.01.2002, 14.07.2006

[064]: Tom Sawyer: AspectOrientedRuby
http://wiki.rubygarden.org/Ruby/page/show/AspectOrientedRuby
25.06.2006, 14.07.2006

[065]: Jocelyn Frechot: Domain-Specific Languages - An Overview
http://compose.labri.fr/documentation/dsl/dsl_overview.php3
25.09.2003, 18.07.2006

[066]: Wikipedia: Ruby (Programmiersprache)
http://de.wikipedia.org/wiki/Ruby_%28Programmiersprache%29
08.07.2006, 18.07.2006

[067]: Lutz Prechelt: An empirical comparison of C, C++, Java, Perl, Python, Rexx and Tcl
http://page.mi.fu-berlin.de/~prechelt/Biblio/jccpprt_computer2000.pdf
14.03.2000, 27.07.2006

[068]: Jonathan Edwards: Subtext Demonstration Video, Part 1
http://subtextual.org/
18.10.2005, 12.08.2006

[069]: Jonathan Edwards: Subtext Demonstration Video, Part 2
http://subtextual.org/
18.10.2005, 12.08.2006

[070]: Why Smalltalk: Smalltalk Quotes
http://www.whysmalltalk.com/quotes/index.htm
-, 16.08.2006

[071]: Richard P. Gabriel: Patterns of Software, „The End of History and the Last Programming Language“
http://www.dreamsongs.com/NewFiles/PatternsOfSoftware.pdf
1996, 17.08.2006

[072]: Guy Steele, Gerald Sussman: Lambda the Ultimate Imperative
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-353.pdf
10.03.1976, 20.08.2006

[073]: Guy Steele: Lambda the Ultimate GOTO
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-443.pdf
10/1977, 20.08.2006

[074]: Tim O’Reilly: What is Web 2.0
http://www.oreillynet.com/lpt/a/6228
30.09.2005, 21.08.2006

[075]: Joe Walnes: The power of closures in C# 2.0
http://joe.truemesh.com/blog//000390.html
16.09.2004, 23.08.2006

[076]: Hartmut Krasemann: Messen und Schätzen
http://users.informatik.haw-hamburg.de/~raasch/Messen&Schaetzen.pdf
14.10.2002, 26.08.2006

[077]: Caspers Jones: Programming Languages Table
http://www.theadvisors.com/langcomparison.htm
03/1996, 26.08.2006

[078]: Wikipedia: Killerapplikation
http://de.wikipedia.org/wiki/Killer-Applikation
01.08.2006, 28.08.2006

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

_____ _____

Ort, Datum Unterschrift