Jenseits des Tellerrands

Softwareentwicklung ist weit mehr als Programmierung und Programmierung ist weit mehr als das bloße Beherrschen von Programmiersprachen und Werkzeugen. Die Artikel dieser neuen Serie möchten wichtige Themen, aktuelle Fragen und auch grundsätzliche Probleme aufgreifen, beleuchten und die gängigen Antworten immer wieder in Frage stellen. Subjektivität ist dabei keineswegs verpönt, sondern das notwendige Salz in der technischen Einheitssuppe.

Die Idee zu dieser Serie kristallisierte sich im Gespräch mit verschiedenen Freunden und Kollegen heraus, die alle über interessante Erfahrungen und weit reichendes Wissen verfügen – und so entstanden im ersten Anlauf sechs Artikel von neun Autoren, welche sukzessive im *Java Magazin* erscheinen werden. Diese Artikel, obgleich sie sich mit vielfältigen Aspekten beschäftigen, eint vor allem eines: die Neugierde an Softwareentwicklung und was wirklich dahinter steckt. Auf jeden einzelnen – Artikel und Autor – freue ich mich sehr.

In dieser Ausgabe beschäftigen wir uns mit der Frage, ob mit Java die Weiterentwicklung der Programmiersprachen ihr (glückliches?) Ende gefunden hat. Dies ist insbesondere deshalb interessant, weil immer häufiger der Vorwurf zu hören ist, Java sei „das Cobol von morgen", in anderen Worten: eine Sackgasse der Evolution, die uns jedoch wegen ihrer großen Verbreitung das Leben schwer machen wird. *Johannes Link*

Johannes Link ist Softwareentwickler und Projektleiter bei der andrena objects ag in Karlsruhe. Er ist Autor der Bücher „Unit Tests mit Java" (dpunkt, 2002) und „Unit Testing in Java" (Morgan Kaufmann, 2003) und war wiederholt Speaker auf der JAX. Auf zahlreiches Feedback zur Serie und Vorschläge für Themen wartet er gespannt unter johannes.link@andrena.de.

Jenseits des Tellerrands, Teil 1: Das Cobol von morgen?

Die letzte Programmiersprache?

VON EBERHARD WOLFF

Die einzige Möglichkeit, um die Zukunft zu prophezeien, ist das Lernen aus der Vergangenheit. Wenn man der Frage nachgehen will, wie die nächste Programmiersprache nach Java aussehen wird, dann sollte man einen Blick auf die Entwicklung werfen, die zu Java und seiner Dominanz geführt hat.

Der erste Grund für den Erfolg von Java ist, dass es von der damals dominierenden Sprache abgeleitet ist, nämlich C++, das wiederum von der davor dominierenden Sprache, nämlich C, abgeleitet war. Das erklärt, warum es zu dem Zeitpunkt, als Java erschien, mit Smalltalk [19] und Eiffel [18] auf der Ebene der Programmiersprachen mindestens gleichwertige Ansätze gab, die sich aber nicht durchgesetzt haben. Diese beiden Programmiersprachen besitzen nämlich eine deutlich andere Syntax als C, C++ und Java. Diese Hürde hatte C selber noch genommen, als es Cobol und Fortran ablöste. Bei dem Übergang zu Java war aber eine vermeint-

lich kleinere Einstiegshürde zu nehmen. Gleichzeitig war der Leidensdruck unter den C++-Entwicklern recht hoch, da sie sich mit einer ausgesprochen komplexen Sprache herumschlagen mussten. Java war wesentlich einfacher. Diese Kombination aus hohem Leidensdruck und leichtem Umlernen sind auf der Ebene der Programmiersprache die hauptsächlichen Ursachen für den Erfolg von Java.

Folglich dürfte eine neue Programmiersprache, die Java beerben soll, nur eine Evolution und keine Revolution sein, sodass ein sanfter Migrationspfad betreten wird. Gleichzeitig muss ein großer Leidensdruck existieren, der Entwickler tatsächlich dazu bewegt, die Plattform zu wechseln und damit einen großen Teil eigenen Wissens obsolet zu machen.

Wenn es stimmt, dass Entwickler erst dann migrieren, wenn sie massive Vorteile haben, dann kann C# nicht das Ziel der Migration sein.

Leidensdruck

Woher könnte so ein Leidensdruck kommen? Bevor wir dieser Frage nachgehen, sollte man erst einmal die Innovationen von Java gegenüber C++ näher beleuchten:

- Virtuelle Maschinen (VMs) als Ausführungsumgebung für Programme wurden eingeführt. Hauptmotivation war die Plattformunabhängigkeit. Interessanterweise sind VMs in Java eingeführt worden, um die Sicherheit und Portabilität auf heterogenen Set-Top-Boxen und anderen Clients zu gewährleisten. Mittlerweile ist das Konzept eher im Bereich Enterprise Computing von Interesse, wo Virtualisierung auch auf Hardware- und Betriebssystemebene schon länger eine Rolle spielt. JVMs kommen damit in das Problem, dass sie sich wie Betriebssysteme verhalten müssen – z.B. in Bezug auf Isolation und Scheduling von verschiedenen Prozessen. Diese Dinge werden im Rahmen des Java Community Process (JCP) seit einiger Zeit untersucht [20].
- Garbage Collection ist inzwischen allgemein akzeptiert. Wohl niemand wünscht sich die manuelle Speicherverwaltung von C++ zurück. Die anfänglichen Performance-Probleme sind längst beseitigt und angesichts der massiven Vorteile für die Entwicklung und Sicherheit der Anwendungen auch nur sekundär.

Hauptsächliche Vorteile von Java gegenüber C++ waren jedoch, dass Java in Bezug auf die Syntax und den Sprachumfang deutlich einfacher war. Genau dieser Vorteil schwindet mittlerweile: Java besteht heutzutage aus einer zum Teil unübersichtlichen Menge von Tools, APIs und Standards. Kaum ein Entwickler beherrscht dies wirklich noch. Gleichzeitig ist diese Plattform jedoch ausgesprochen mächtig und damit attraktiv. Wenn man C# als direkte Konkurrenz betrachtet, dann versucht Microsoft auch, eine solche Plattform anzubieten und es entstehen oft C#-Versionen von etablierten Java-Open-Source-Projekten (z.B. NUnit als Komplement zu JUnit).

Übrigens ist ein Ergebnis dieser Betrachtung, dass C# nicht das nächste Java sein kann: Wenn es stimmt, dass Entwickler erst dann migrieren, wenn sie massive Vorteile haben, dann kann C# nicht das Ziel der Migration sein. C# unterscheidet sich als Sprache nur wenig von Java und kann daher keine große Vorteile bieten. Auch die .NET-Plattform unterscheidet sich natürlich schon deutlich von Java oder J2EE, aber es gibt hier nur Unterschiede und eigentlich keinen eindeutig besseren. Ein Vorteil von C# ist natürlich, dass es von Microsoft kommt und daher für Entwickler von Visual Basic, Access oder Visual Basic for Applications eine ausgesprochen sinnvolle Alternative zu Java ist. C# hat auch sicherlich Einfluss auf Java, da sich die beiden Plattformen in einer Art Koevolution weiterentwickeln: Java erbt von C# zum Beispiel Autoboxing, C# von Java generische Datentypen.

Hier sieht man gleich, wo bei Java als Programmiersprache die Innovation stattfindet: auf Ebene des Typsystems. In Java 1.1 wurden Inner Classes oder in Java 5 die generischen Datentypen eingeführt. Allerdings ist dies eher ein Einholen als ein Überholen, wenn man als Vergleich Eiffel wählt, das es schon vor Java gab und diese Features bereits implementiert hatte.

Gibt es gar kein nächstes Java?

Wenn man also insbesondere das Zweigestirn aus C# und Java anschaut, muss man sagen, dass der Vorteil dieser Technologien nicht so sehr auf der Sprachebene zu finden ist, sondern eher auf der Ebene der Infrastruktur. Zum einen sind es J2EE und .NET als Laufzeitumgebungen, zum anderen sind es die zahlreichen Werkzeuge wie IDEs, Profiler, Debugger und natürlich die Libraries für die verschiedensten Zwecke.

Man ist dann erstmal versucht, vom Ende der Evolution der Programmiersprachen zu sprechen. Im Bereich der Politik hat Francis Fukuyama Anfang der 90er Jahre des vergangenen Jahrhunderts eine ähnliche These aufgestellt [1], nämlich dass nach dem Zusammenbruch des Ostblocks die ganze Welt auf den Endpunkt der Demokratie zustrebt. Die jüngere Geschichte hat ihn z.B. in Bezug auf den radikalen Islamismus eher nicht bestätigt. Es ist also gefährlich, von einem Punkt relativer Stabilität auf einen Endpunkt zu schließen.

Vielleicht spielen Programmiersprachen in Zukunft wirklich keine Rolle mehr. Immerhin gibt es die modellbasierten Ansätzen wie Model Driven Architecture (MDA) oder Model Driven Software Development (MDSD) [2], bei denen aus (UML-)Diagrammen oder Domänen-spezifischen Sprachen (Domain Specific Languages – DSL) Code erzeugt wird. Hauptziel ist, eine höhere Abstraktion zu schaffen und beispielsweise aus der Modellierung eines Workflow Code zu generieren. Ergänzungen mit manuell geschriebenem Code sind machbar. Dabei kann jedoch der generierte Code gleich die unangenehm zu programmierenden Teile der Infrastruktur kapseln und so ein eigenes Programmiermodell anbieten. Im Gegensatz zu den schon lange und auch sehr häufig eingesetzten „normalen" Generatoren wird bei diesen Ansätzen Generierung auf breiter Front und mit einer einheitlichen theoretischen Unterfütterung eingesetzt.

Durch MDA wird die zugrunde liegende Programmiersprache weniger wichtig, weil größere Teile der Software sowieso generiert werden. Gleichzeitig ist es von Bedeutung, eine Infrastruktur zu haben, mit der vor allem die nicht funktionalen Anforderungen abgedeckt werden können, wie es bei .NET oder J2EE der Fall ist.

Modellgetriebene Ansätze könnten also tatsächlich ein nächster Evolutionsschritt sein, denn sie bauen auf dem vorhandenen Java-Ansatz auf. Im Prinzip ändert sich nur die Art der Code-Erstellung, aber nicht zwangsläufig der Code an sich. Dies ist natürlich eine andere Evolution als von C++ zu Java: Es wird nicht eine Sprache durch eine bessere ersetzt, sondern es wird auf der vorhandenen aufgebaut.

Neue Aspekte

Ein anderer wichtiger Ansatz ist die aspektorientierte Programmierung (AOP). Dabei werden Cross Cutting Concerns (Querschnittsbelange) in Aspekte ausgelagert. Das klassische Beispiel ist das Logging: Jede Methode hat Code zum Loggen. Dies kann man zum Beispiel durch das Abfangen der Methodenaufrufe und das automatische Loggen so implementieren, dass es nur eine zentrale Stelle gibt, die sich mit Loggen beschäftigt.

Eine recht einfache Methode für die Implementierung von Aspekten hält zurzeit Einzug in Enterprise-Java-Umgebungen. Dazu gehört die Unterstützung von JBoss für Aspekte [3], Spring AOP [4] oder AspectWerkz [5]. Auch der zweite Public Draft der EJB 3-Spezifikation enthält nunmehr AOP-Elemente. Diesen Ansätzen ist gemeinsam, dass sie ein sehr einfaches Modell für Aspekte bieten, die insbesondere ohne Erweiterungen der Java-Syntax auskommen. Im Gegensatz dazu hat AspectJ [6] Syntaxerweiterungen und einen wesentlich mächtigeren Ansatz für die Implementierung von Aspekten (siehe auch das Interview mit dem AspectJ Project Lead Adrian Colyer auf Seite 21).

Wenn man heute die Verwendung von Aspekten in der Praxis betrachtet, fällt auf, dass Aspekte im Moment eher auf technische Belange begrenzt sind. Sie werden für triviale Probleme wie Logging verwendet oder für durchaus komplexere Themen wie Sicherheit und Transaktionen. Eine vollständige Dekomposition eines Systems in Aspekte ist jedoch recht selten, ein Beispiel ist das Content Management System (CMS), an dem Rickard Öberg [7] entwickelt. Dadurch wird die Implementierung des CMS anscheinend deutlich vereinfacht. Dennoch wird AOP noch nicht auf breiter Front verwendet.

Ein Teil des Problems ist dabei sicherlich, dass es im Bereich der Aspekte keine etablierten Verfahren für die Analyse gibt. So kann man ein Bonusprogramm als einen fachlichen Aspekt sehen, da zahlreiche fachliche Vorgänge wie ein Einkauf beeinflusst werden, weil dort dann Punkte für das Bonusprogramm gutgeschrieben werden müssen. Bei der Objektorientierung haben sich schon lange Verfahren in der Praxis bewährt, um fachliche Belange in Objekte zu unterteilen. Genau dies fehlt bei Aspekten. Dadurch ist es vor allem problematisch, fachliche Aspekte zu identifizieren.

Ingesamt bietet die aspektorientierte Programmierung also ein sehr interessantes, neues Programmierparadigma, dessen Vorteil vor allem ist, dass es evolutionär auf vorhandenen Sprachen wie Java aufbaut. Daher kann tatsächlich eine breite Nutzung entstehen, weil eben vorhandenes Wissen lediglich ergänzt und nicht vollständig neu gelernt werden muss. Außerdem kann man auf die etablierten und weit verbreiteten Plattformen aufsetzten.

Programm oder Skript?

Noch ein weiterer spannender Bereich sind Skriptsprachen. Ursprünglich sind Skriptsprachen entwickelt worden, um Routineaufgaben mit Skripten zu automatisieren. Entsprechend ist das wesentliche Merkmal der Skriptsprachen, dass sie einfach zu erlernen und zu handhaben sind. Skriptsprachen haben meistens ein einfaches Typsystem zum Beispiel mit dynamischer Typisierung (Typfehler werden erst zur Laufzeit erkannt) oder ohne explizite Deklaration von Variablen. Mittlerweile sind ganz viele Skriptsprachen auch objektorientiert, wie beispielsweise Python [8] und Ruby [9], sodass die Grenzen zu dynamisch typisierten objektorientierten Sprachen wie Smalltalk fließend sind.

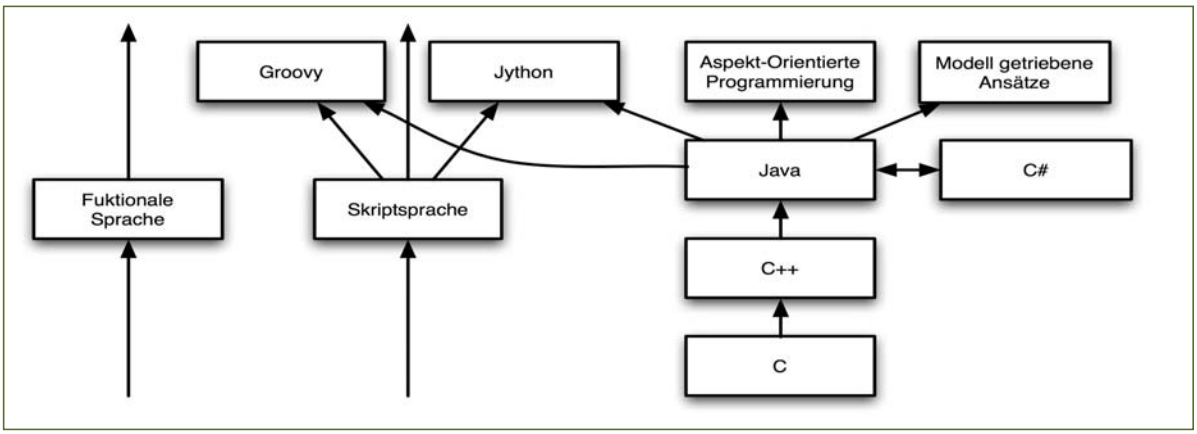

Abb. 1: Die Evolution von Java und die Konkurrenten

Üblicherweise assoziiert man Skriptsprachen mit einfachen Aufgaben, aber in der Praxis zeigt sich, dass zahlreiche komplexe Webanwendungen mit Skriptsprachen geschrieben werden. Das wirft nun natürlich die Frage auf, warum man nicht gleich alles in Skriptsprachen schreibt. Schließlich führen sie für sich eine höhere Produktivität ins Feld, die man überall brauchen kann. So gibt es ab und zu Vergleiche, wie beispielsweise jener zwischen Ruby on Rails und Java mit Spring und Hibernate [10]. Ruby on Rails ist ein Framework für die Erstellung von Webanwendungen, das in Ruby geschrieben ist. Spring und Hibernate sind eigentlich Java-Umgebungen mit einem einfachen und zugleich produktiven Programmiermodell, dennoch geht der Vergleich für Java nicht sehr gut aus.

Die Frage ist allerdings, ob die Entscheidung für oder gegen Skriptsprachen einzig von der Produktivität abhängig ist. Oft spielen Überlegungen wie strategische Entscheidungen oder das Vertrauen in eine Plattform eine Rolle. Außerdem gibt es bei Skriptsprachen keine gute Unterstützung durch IDEs, die wir bei Java schon lange gewöhnt sind.

Auswirkungen auf die Java-Plattform haben die Skriptsprachen jedoch auf jeden Fall: Zum einen gibt es mit Groovy [11], das gerade im JSR 241 standardisiert wird, oder Jython, das im Prinzip Python entspricht, Skriptsprachen, die auf einer Java Virtual Machine laufen und einfache Schnittstellen zu Java haben. Zum anderen werden mit Ansätzen wie JSP, JSP Tag Libraries und in letzter Zeit JSF (Java Server Faces) Werkzeuge auf der Java-Plattform etabliert, die wie Skriptsprachen auch eine hohe Produktivität und gute Unterstützung in Tools erlauben. Dies ist nicht zuletzt eine Antwort auf den enormen Konkurrenzdruck, der durch die Skriptsprachen entsteht. Auch hier sieht man, dass Java eben in erster Linie keine Sprache, sondern eine Plattform ist, die mit Skriptsprachen genutzt werden kann. Da Skriptsprachen schon vor Java existiert haben, ist es jedoch unwahrscheinlich, dass eine solche Sprache Java tatsächlich verdrängen kann. Aber auch bei Skriptsprachen findet viel Innovation statt, wie Ruby on Rails zeigt.

Function follows Function

Ein weiterer interessanter Bereich sind die funktionalen Programmiersprachen. Wie schon der Name verrät, verwenden diese Sprachen als zentrale Abstraktion Funktionen im mathematischen Sinne, als Abbildungen von Werten auf Werte. Daraus ergeben sich einige besondere Merkmale:

- Es gibt keinen Zustand. Man kann also nicht in einer Variablen einen Wert speichern, sondern man kann nur Funktionen für bestimmte Werte evaluieren lassen. Dies findet man auch zum Teil in Java-Anwendungen vor: So sollte zum Beispiel eine *get*-Methode einen Wert auslesen, ohne irgendetwas zu verändern.
- Statt Schleifen verwendet man rekursive Funktionsaufrufe.
- Funktionen werden als Datentypen behandelt. Dadurch entstehen Funktionen höherer Ordnung, die Funktionen als Parameter übernehmen oder als Ergebnis zurückgeben. So kann man zum Beispiel einer *map*-Funktion eine Liste mit Werten und eine Funktion übergeben. Als Ergebnis erhält man eine Liste mit den Ergebnissen der Funktion für die übergebenen Werte.
- Überhaupt unterstützen die funktionalen Sprachen Listen meistens sehr gut, so sind dort Mechanismen, wie Java 5 sie gerade mit *foreach* lernt, schon immer Standard gewesen.
- Einige Sprachen unterstützen Typ-Inferenz: Dabei kann die Sprache selbst ableiten, was für ein Typ ein Ausdruck hat bzw. erwartet. Dadurch kann man Typfehler finden, ohne dass man die Typen selbst deklariert hat. Es ist jedoch fraglich, ob dies einen echten Vorteil bei der Entwicklung mit einer solchen Sprache bringt.

Nun kann man sich natürlich fragen, was dies mit der Java-Nachfolge zu tun hat, immerhin fristen funktionale Sprachen eher ein Schattendasein. Es gibt ein bereits älteres Paper, das genau diese Frage beantwortet [13]: Funktionale Sprachen erlauben bessere Modularisierung, dadurch bessere Wiederverwendung und letztendlich kleinere und leichter zu schreibende Programme. Außerdem finden Wettbewerbe statt, in denen sich funktionale Programmiersprachen der Konkurrenz stellen und üblicherweise gewinnen [14]. Eine echte Erfolgsgeschichte liefert Paul Graham, der eine Shopping-Lösung in der funktionalen Programmiersprache Lisp geschrieben hat und sie für 49 Millionen Dollar an Yahoo verkauft hat. Er führt den Erfolg vor allem auf Lisp zurück [15]. Für Java-Entwickler ist vielleicht auch in-

teressant, dass die .NET-Jünger mit F# [16] bereits aktiv sind.

Der Hauptvorteil funktionaler Sprache ist, dass funktionale Programmiersprachen meistens eine sehr kompakte und gut verständliche Darstellung von Algorithmen erlauben, sodass recht schnell Lösungen entwickelt werden können. Die Gründe, warum sich funktionale Programmierung nicht in der Breite durchgesetzt hat, sind die hohen Ansprüche an funktionale Programmierer: Sie müssen den mathematischen Funktionsbegriff sehr gut beherrschen. Außerdem müssen sie funktionale Designs erstellen und können dabei nicht wie bei der Objektorientierung auf umfangreiche Literatur zurückgreifen. Die Umstellung auf dieses Paradigma ist gerade keine Evolution, sondern eine Revolution und dazu noch eine recht umfassende. Zudem steht gerade bei Geschäftsanwendungen die Datenhaltung im Mittelpunkt, während die Algorithmen nicht so sehr im Fokus sind, sodass funktionale Sprachen ihren Vorteil beim Implementieren algorithmisch komplexer Programme gar nicht ausspielen können.

Fazit

Wenn man also in die Zukunft von Java schauen will, ergibt sich folgendes Bild (Abb. 1):

- Es gibt zurzeit keine völlig andere Plattform, die sich anschickt, Java und J2EE zu ersetzen, wie dies bei der Ablösung von C++ durch Java der Fall war.
- Java wird durch AOP und modellgetriebene Ansätze ergänzt, ohne dass dabei die Sprache abgelöst wird. Dadurch wird eine einfache Migration zu den Konzepten ermöglicht.
- C# und Java setzen die gleichen Konzepte um, aber auf unterschiedlichen Plattformen. Die Sprachen entwickeln sich in einer Koevolution weiter: Fortschritte in Java werden in C# integriert und umgekehrt. Eine Ablösung ist nicht zu erwarten, da die beiden Sprachen sich nicht ausreichend voneinander absetzen können.
- Skriptsprachen und Java werden auch weiterhin nebeneinander existieren. Beide haben offensichtlich ihren Markt gefunden. Allerdings gibt es auch Skriptsprachen auf der Java-Plattform oder Skriptsprachen, die von Java beeinflusst werden.
- Funktionale Sprachen werden auch weiterhin ihr Nischendasein fristen, da es kaum ein Gegenstück zu objektorientierten Vorgehens- und Analysemodellen für diese Sprachen gibt. Außerdem ist nicht zu erwarten, dass das Wissen über die funktionale Programmierung nun plötzlich eine weite Verbreitung findet.

Es kann gut sein, dass uns Java noch weitere zehn Jahre erhalten bleibt. Vielleicht allerdings nicht mehr in der Form, wie wir es heute kennen. Genau das hat man allerdings bis 1989 auch von der Berliner Mauer gedacht.

Zum Schluss noch ein Tipp: Auf jeden Fall hilft es, wenn man sich an die Ideen aus dem „Pragmatischen Programmierer“ [17] hält und jedes Jahr eine neue Sprache lernt. Es erweitert den Horizont und bereitet einen auf den Ernstfall vor.

Eberhard Wolff ist als Chefarchitekt bei Saxonia Systems tätig. Hauptsächlich beschäftigt er sich dabei mit Java-Serveranwendungen.

Links & Literatur

[1] Francis Fukuyama: Das Ende der Geschichte, Kindler, 1992

[2] Thomas Stahl, Markus Völter: Modellgetriebene Softwareentwicklung, dpunkt, 2005

[3] Eberhard Wolff: Neue Aspekte von JBoss 4, in *Java Magazin* 12.2003

[4] Eberhard Wolff: Wunschlos glücklich. Aspektorientierte Programmierung mit dem Spring Framework, in *Java Magazin* 5.2005

[5] aspectwerkz.codehaus.org

[6] www.eclipse.org/aspectj/

[7] www.jroller.com/page/rickard/

[8] www.python.org

[9] www.ruby-lang.org

[10] www.rubyonrails.com

[11] groovy.codehaus.org

[12] www.jython.org

[13] www.md.chalmers.se/~rjmh/Papers/whyfp.html

[14] de.wikipedia.org/wiki/ICFP_Contest

[15] www.paulgraham.com/avg.html

[16] research.microsoft.com/projects/ilx/fsharp.aspx

[17] Andrew Hunt, David Thomas: Der Pragmatische Programmierer, Hanser, 2003

[18] www.eiffel.com

[19] www.smalltalk.org

[20] www.jcp.org/en/jsr/detail?id=121