Produktivität von Programmiersprachen

Tim Jammer

Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik, Arbeitsbereich Wissenschaftliches Rechnen
Seminar Softwareentwicklug in der Wissenschaft im SoSe 15
Betreuer: Dr. Hermann Lenhart

Zusammenfassung. In der vorliegenden Arbeit, wird es um die Produktivität von Programmiersprachen gehen.
Dazu werden zunächst die Faktoren betrachtet, die für die Produktivität entscheidend sind.
Anschließend werden beispielhaft die Ergebnisse einer Studie von Lutz Prechelt vorgestellt.
Diese werden in einem abschließenden Fazit bewertet.

Inhaltsverzeichnis

1 Überblick 3

Wie misst man die Produktivität von Programmiersprachen? 4

2 Performance 4

2.1 Laufzeit 4

2.2 Speicherbedarf 5

2.3 Stabitität 5

3 Entwicklungszeit 6

3.1 Kompilierungszeit 6

3.2 Wartbarkeit/Anpassbarkeit 6

3.3 Größe der Community 7

4 Sicherheit 7

5 Zusammenfassung 7

6 Produktivität hybrider Ansätze 8

Einige Programmiersprachen im Vergleich 9

7 Die zu vergleichende Problemstellung 9

8 Die Ergebnisse bezüglich Performance 9

9 Die Ergebnisse bezüglich Entwicklungszeit 12

10 Zusammenfassung 13

11 Fazit/Ausblick 14

1 Überblick

In der vorliegenden Arbeit wird die Produktivität von verschiedenen Programmiersprachen analysiert werden.
Dazu ist es zunächst vönnöten die Kriterien zu definieren, nach denen man die Sprachen beurteilen möchte. Der erste Abschnitt widmet sich daher den verschiedenen Kriterien, mit denen man die Produktivität einer Programmiersprache messen kann. Dabei wird auch darauf eingegangen, in welchen Anwendungsgebieten welche Faktoren von mesonderer Bedeutung sind.
Im zweiten Teil werden dann einige Sprachen anhand einiger Kriterien aus dem ersten Abschnitt miteinander verglichen, um festzustellen, welche Sprachen produktiver als andere sind.
Ein abschließendes Fazit rundet diese Arbeit ab.

Wie misst man die Produktivität von Programmiersprachen?

Dass die reine Messung der Anzahl der Code-Zeilen pro Stunde kein geeignetes Maß für die Messung der Produktivität einer Programmiersprache ist, leuchtet bei näherer Betrachtung schnell ein, ist sie doch insbesondere vom verwendeten Programmierstil abhängig. So lassen es viele Sprachen zu, auf Einrückungen und Zeilenumbrüche weitesgehend zu verzichten. Aus Sicht eines gut strukturierten und entsprechend wartbaren Quellcodes ist dies jedoch nicht sinvoll.
Daher stellt sich zunächst die Frage, wie man denn überhaupt die Produktivität einer Programmiersprache messen kann, welche Faktoren dafür wichtig sind und wie groß ihr jeweiliger Einfluss auf die gesamte Produktivität ist.
Wichtige Einflussfaktoren sind (vgl. [KK04]) :

2 Performance

Performance ist eigentlich ein Oberbegriff, der verschiedene Faktoren zusammenfasst. Diese werden wir im Folgenden näher erläutern.
In Abgrenung zur Entwicklungszeit, werden wir hierunter alle Faktoren fassen, die der Benutzer der fertigen Software selber beurteilen kann.

2.1 Laufzeit

Laufzeit bezeichnet die Zeit, die der fertige Code benötigt, um die Lösung des Problems zu berechnen.
Dieses Kriterium lässt sich vergleichsweise einfach messen. Neben der Messung der benötigten Zeit per Hand (z.B. Stoppuhr), stellen die meisten Betriebssysteme präzise Messmöglichkeiten bereit (z.B. Zählen der verwendeten CPU-Zyklen). Allerdings hängt die benötigte Zeit stark vom betrachteten Problem und dem zu deren Lösung verwendeten Algorithmus ab.
Man kann natürlich das selbe Problem mit dem gleichen Ansatz in zwei verschiedenen Sprachen lösen und dann deren Laufzeit vergleichen. Allerdings stellt sich auch dabei die Frage, inwiefern dieser Vergleich "fair" ist. Da ein bestimmter Ansatz vielleicht dem Verarbeitungsmodell der einen Sprache wiederspricht, hätte man in dieser Sprache einen anderen Ansatz gewählt. Wenn man nun aber für jede Sprache den Ansatz nimmt, der ihrem Verarbeitungsmodell am meisten entspricht, hat man wieder völlig unterschiedliche Ansätze zu vergleichen, was schon bei der theoretischen Analyse ohne irgendwelche Einflüsse von verwendeter Hardware oder Programmiersprache zu unterschiedlichen Ergebnissen führen kann.

Typischerweise sind Low-Level Sprachen -wie Assembler- hier im Vorteil, da

sie die Hardware der jeweiligen Maschine am besten ausnutzen.
Je öfter eine Anwendung ausgeführt wird, desto wichtiger wird dieser Faktor im Vergleich zu den anderen. Im Speziellen gilt dies für den Bereich des high performance computing.

In diesen Bereich fällt auch, wie gut sich Anwendungen parallelisieren lassen, um einen Performance-Gewinn zu erzielen, da dies die Zeit verringert, um die Problemlösung zu berechnen.

2.2 Speicherbedarf

Ein weiteres Kriterium, dass allgemein unter Performance fällt, ist der Speicherbedarf.

Er gibt zum einen an, welchen Speicherplatz das Programm auf der Festplatte benötigt.
Dieser Punkt ist aufgrund der hohen Verfügbarkeit von Speicher und der häufig viel größeren Größe der zu verarbeitenden Daten vernachlässigbar. Allerdings spielt er in Bereichen mit stark begrenzten Ressourcen (z.B. Microcontrollern) eine wichtige Rolle.

Zum anderen fällt hierunter die Größe des Speicherbedarfs, welchen das Programms zur Laufzeit hat.
Ähnlich zur Performance lässt er sich auch vergleichsweise einfach messen, hängt allerdings genauso stark vom betrachteten Problem und dem zu deren Lösung verwendeten Algorithmus ab.
Dabei geht es allgemein um den time-memory Tradeoff. Dieser hängt jedoch viel wesentlicher vom verwendeten Algorithmus als der verwendeten Sprache ab.
Gemeint ist bie diesem Punkt also im Wesentlichen, wie viel Overhead beispielsweise zum Verwalten der internen Variablen benötigt wird.

Im Bereich high performance computing kommt diesem Faktor jedoch eine besonders wichtige Rolle zu, da Hauptspeicher in diesem Bereich eine knappe Ressource darstellt. Es entsteht ein Flaschenhals beim Zugriff auf den Speicher, wenn mehrere Prozessoren gleichzeitig auf den vergleichsweise langsamen Speicher zugreifen wollen.

2.3 Stabitität

Stabilität bezeichnet, wie zuverlässig eine Sprache ist. Hierbei sind etablierte Sprachen im Vorteil, da sie meist wenig bis keine Bugs in ihrer Implementation besitzen.

Robustheit Robustheit bezeichnet das Verhalten der Anwendung im Fehlerfall. Hier kann es sich z.B. vorteilhaft auf die Entwicklungszeit auswirken, wenn eine

Fehlerbehandlung in der Sprache bereits vorgesehen ist. Andersherum kann es zu Performanceverlusten führen, wenn man annimmt, dass die zu verarbeitenden Daten fehlerfrei sind.

3 Entwicklungszeit

Im Gegensatz zur Performance bezeichnet Entwicklungszeit Faktoren, die meistens nicht vom Benutzer der fertigen Software beurteilt werden können.

Diese Zeit beinhaltet nun daher die Zeit, die von der Problemstellung bis zum fertigen Programm verstreicht. Man kann diese Zeit zwar ebenfalls sehr einfach messen, allerdings hängt sie von viel mehr Faktoren als nur der verwendeten Sprache ab. Beispielsweise auch von der Erfahrung der Programmierer und deren Verständnis vom Problem.

Domain-specific-languages sind hier oft im Vorteil. Man muss keine Strukturen entwerfen, um die Problemstrukturen abzubilden, da diese eingebaut sind.

3.1 Kompilierungszeit

Dies bezeichnet die Zeit, die das Programm zum Kompilieren benötigt. Auch diese lässt sich sehr einfach messen und hängt im Gegensatz zur Laufzeit viel stärker von der verwendeten Sprache als vom betrachteten Problem ab.
Eine große Kompilierungszeit kann das Testen und Debugging wesentlich verlängern und damit die Produktivität der Sprache vermindern.
Die Zeit, um das fertige Programm zum Schluss einmal zu kompilieren wird meist ignoriert, da man diese Zeit im Gegensatz zu dem häufigen kompilieren während des Debuggings, nur einmal abwarten muss.
Im Vorteil sind hier (Skript-)Sprachen, die nich kompiliert sondern mit einem Interpreter verarbeitet werden. Auch Sprachen, die sich partiell (z.B. paketweise) kompilieren lassen, sodass nicht nach jeder Änderung das komplette Programm für den nächsten Test neu kompiliert werden muss, schneiden in diesem Kriterium gut ab.

3.2 Wartbarkeit/Anpassbarkeit

Bei besonders langlebiger Software ist neben der Performance auch die Wartbarkeit ein wichtiger Aspekt, da es nötig sein kann Änderungen und Ergänzungen vorzunehmen.
Dieser Aspekt hängt allerdings auch wesentlich von der Disziplin der Programmierer ab, ein gut strukturiertes und dokumentiertes Programm zu schreiben.

Plattformunabhängigkeit Wenn die Anwendung auf vielen Plattformen genutzt werden soll, spielt dieser Punkt eine sehr wichtige Rolle, da es meistens unverhältnismäßig teuer ist, eine Anwendung für jede Plattform spezifisch zu entwickeln.
Hierbei geht dann die Zeit ein, die man benötigt, um das Programm auf eine andere Plattform anzupassen. Es ist daher ein Faktor, dem bei der Betrachtung der Anpassbarkeit eine besonders wichtige Rolle zufallen kann.
Weit verbreitete Sprachen, wie Java, sind hier im Vorteil, da sie von den meisten Plattformen unterstützt werden.

3.3 Größe der Community

Hierunter fallen viele schwer zu messende Faktoren, auch wenn im Allgemeinen gilt, dass eine größere Community hilfreicher ist, da man beispielsweise bei Problemen mehr Hilfestellung bekommen kann. Außerdem gibt es mehr (erfahrene) Programmierer, die man für sein Projekt anwerben kann. Nicht unterschätzen sollte man auch die Verfügbarkeit von Entwicklungsumgebungen und unterstützenden Tools für die jeweilige Sprache.
All diese Faktoren können sich auch prositiv auf die Entwicklungszeit eines Programms auswirken.

Vorhandensein von Librarys Häufig geht mit einer größeren Community eine größere Anzahl von vorhandenen Librarys einher. Hierbei sind im Allgemeinen nicht nur in der Sprache integierte Librarys gemeint, sondern auch benutzererstellte.
Das Benutzen bereits vorhandener Librarys kann den Entwicklungsprozess erheblich verkürzen, da die vorhandenen Funktionen weder entwickelt, noch getestet werden müssen.

4 Sicherheit

Wenn die verwendete Sprache viele Sicherheitslücken hat, muss in sicherheitskritischen Bereichen von ihrer Verwendung abgesehen werden.

5 Zusammenfassung

Generell werden obige Aspekte in der Messgröße Zeit gemessen, weshalb man verallgemeinert sagen kann, dass diejenige Sprache optimal ist, für die die benötigte Zeit vom Bekanntwerden des Problems zum Bekanntwerden der dazugehörigen Lösung minimal ist.

Die wesentliche Schwierigkeit bei Aussagen wie "Sprache A ist definitiv besser als Sprache B" ist, dass sich die Relevanz der Faktoren von Problemmfall zu

Problemfall ändert. Das ist (neben persönlichen Vorlieben von Programierern) auch der Hauptgrund, warum überhaupt verschiedene Sprachen existieren und für ihren jeweiligen Verwendungszweck die beste Wahl darstellen.

Herfür sei beispielsweise auf [VKD07] verwiesen, worin dargestellt wird, dass Fortran als domain spezific language im Bereich der wissenschaftlichen Anwendungen gegenüber Beispielsweise C++ überlegen ist. In anderen Bereichen wäre es aufgrund fehlender Funktionalität allerdings ein Nachteil Fortran zu benutzen.

6 Produktivität hybrider Ansätze

Hybride Ansätze, die die Stärke der verschiedenen Sprachen vereinen, sind oft produktiver, sofern die verschiedenen Sprachen sich gegenseitig unterstützen und technisch kompatibel sind. Allerding sollte man beachten, dass die Verwendung meherer Sprachen im einer Anwendung auch mit Produktivitätseinbußen verbunden ist, da beispielsweise nicht mehr alle Entwickler jeden Code verstehen können, man stärker für einheitliche Schnittstellen sorgen muss und auch die Performance leiden kann.
Hybride Ansätze werden oft als Kompromiss zwischen Performance und Entwicklungszeit eingesetzt. Dabei werden die performanceintensiven Teile in performancestarken und/oder sehr hardwarenahen Sprachen (wie z.B.CUDA für Nvidea Grafikprozessoren) geschrieben. Die anderen Teile werden dann meist in einfacheren Skriptsprachen geschrieben, um ihre Entwicklungsdauer zu reduzieren.

Einige Programmiersprachen im Vergleich

Im vorhergehenden Abschnitt haben wir uns mit den verschiedenen Faktoren beschäftigt, die für die Produktivität einer Programmiersprache wichtig sind.

In diesem Abschnitt sollen jetzt einige Programmiersprachen beispielhaft verglichen werden. Ausgangspunkt ist dabei eine Studie von Lutz Prechelt [Pre00], in der die Programmiersprachen Java, C, C++, Perl, Phyton, Rexx und Tcl vor allem bezüglich Performance verglichen wurden. Dabei wurden die Programme in Perl, Phyton, Rexx und Tcl unter dem Begriff Script zusammengefasst.

Verglichen wurden insgesamt 80 Implementationen des *phonecode-problem*. Daher können die im Folgenden präsentierten Ergebnisse nur für dieses eine Problem genutzt werden und lassen sich nicht einfach auf andere Problemstellungen übertragen.

7 Die zu vergleichende Problemstellung

Wir werden uns nun kurz mit der betrachteten Problemstellung befassen.
Das zu vergleichende Programm soll Telefonnummern[1] anhand eines bestimmten Schlüssels zu Wörtern dekodieren. Dabei darf es nur Wörter verwenden, die in einem gegebenen Wörterbuch zu finden sind. Der Kodierungsschlüssel (also welcher Buchstabe welcher Ziffer zugeordnet werden darf) ist vorher bereits bekannt.
In der Anwendung geht es dann darum, sich anstelle der Telefonnummer ein dazugehöriges Wort zu merken. Da man die Zuordung Buchstabe auf Ziffer kennt, kann man dann schnell die entsprechende Telefonnummer eintippen.

8 Die Ergebnisse bezüglich Performance

Die Performance wurde in 2 seperaten Phasen gemessen. Zuerst wurde nur die Phase gemessen, in der das Wörterbuch eingelesen wurde. Danach wurde die Phase gemessen, in der tatsächlich die Telefonnummernn dekodiert wurden. Die Abbildungen zeigen die Werte, die von den einzelnen Programmen erreicht wurden, als einzelne rote Punkte. Gelb markiert wurde die Reichweite der mittleren 50%. Der schwarze Punkt steht für den Median. Mit M und einer gestrichelte Linie werden Mittelwert und Standardabweichung aufgezeigt.
Man beachte in allen Grafiken die logarithmische Skala.
Das Wählen einer logarithmischen Skala ist angebracht, da diese die Unterschiede besser zeigt. Ein Laufzeitunterschied von 1 zu 2 Sekunden ist schließlich wesentlich schwerwiegender, als einer von 100 zu 101 Sekunden.

[1] d.h. Zufällige Abfolge von Zahlen

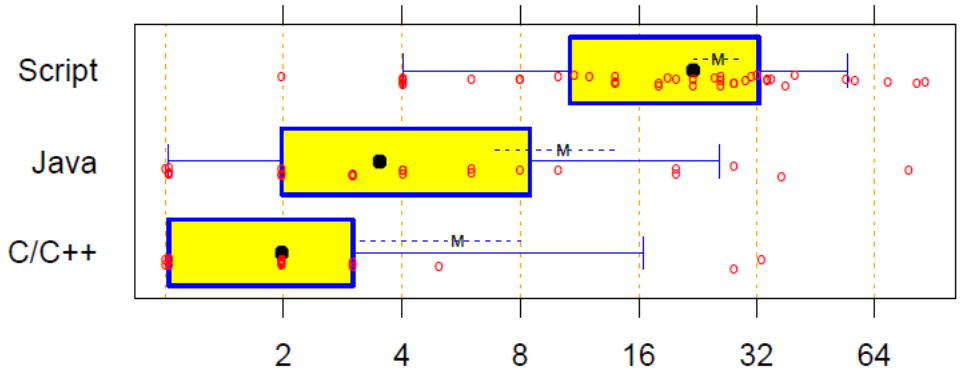

Abb. 1. Laufzeit zum Laden des Wörterbuches in Sekunden

Abb. 1 zeigt die Laufzeit der Programme, um das Wörterbuch einzulesen. Man erkennt, dass C/C++ Programme am schnellsten sind. Die Java-Programme benötigen im Mittel etwa 1.3 mal so viel Zeit wie C/C++ Programme. Die Programme der Skriptsprachen benötigen allerdings etwa 5.5 mal so viel Zeit, wie die C/C++ Programme.

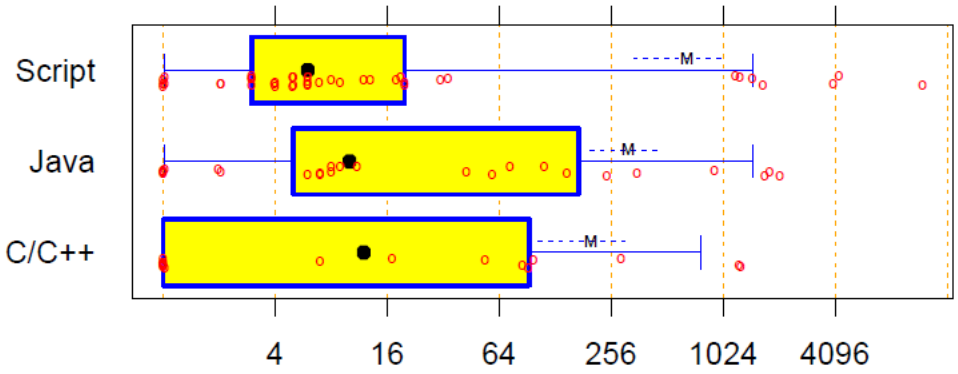

Abb. 2. Laufzeit dekodieren der Telefonnummern in Sekunden

Abb. 2 zeigt die Laufzeit für die Phase, in der Dekodierungen der Telefonnummern gesucht werden. Man erkennt, das in allen Sprachen sehr schnelle Programme geschrieben wurden. Im Mittel sind jedoch keine wirklich signifikanten Unterschiede festzustellen.

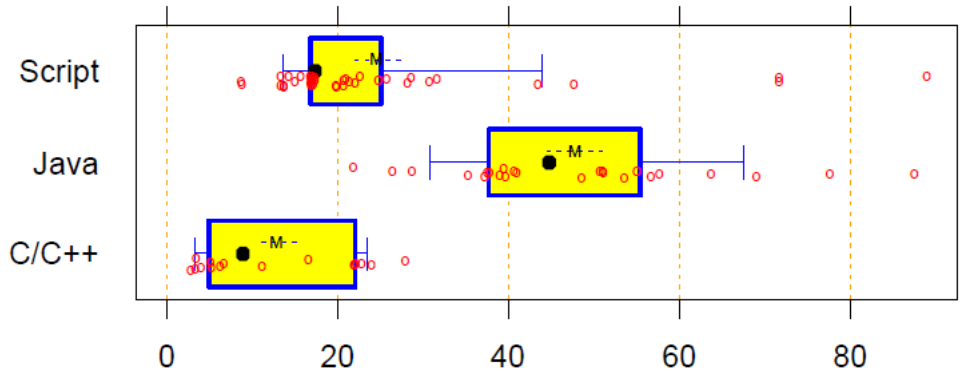

Abb. 3. Speicherverbrauch in MB

Abb. 3 zeigt nun den Speicherverbrauch der Programme. Man erkennt, das die Programme der Skriptsprachen im Mittel etwa 9MB Mehr Speicher als die C/C++ Programme benötigen. Allerdings benötigen die Java-Programme etwa 32MB mehr Speicher als die C/C++ Programme.

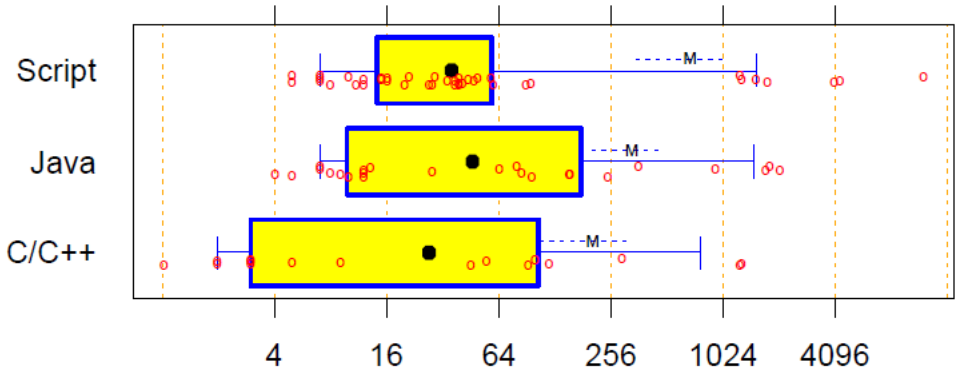

Abb. 4. Gesamtlaufzeit in Sekunden

Abb. 4 zeigt nun die Gesamtlaufzeit. Man erkennt zwar, das die schnellsten Programe in C/C++ geschrieben wurden. Allerdings unterscheiden sich die Laufzeiten im Mittel nicht wirklich signifikant.

9 Die Ergebnisse bezüglich Entwicklungszeit

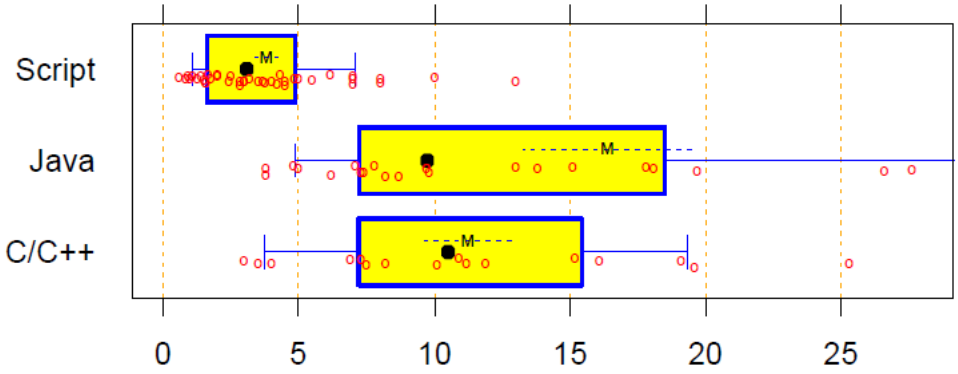

Abb. 5. Entwicklungszeit in Stunden

Abb. 5 zeigt die Zeit, die die Entwickler (nach eigenen Angaben) mit der Entwicklung der Programme verbracht haben. Man erkennt, dass die Entwicklung der Programme in einer Skriptsprache im Mittel nur etwa halb so viel Zeit kostet, wie Entwicklung in den anderen betrachteten Sprachen.
Allerdings ist dieses Ergebnis mit besonderer Vorsicht zu genießen, da nicht 100%ig sicher ist, welche Zeit die Programmierer jeweils gemessen haben[2]. Beispielsweise zählen nicht alle das Lesen der Anforderungen mit, und programmieren erst Tage, nachdem sie die Anforderungen kennen. In dieser Zwischenzeit haben sie sehr wahrscheinlich bereits über das Problem nachgedacht. Diese Zeit ist jedoch nur sehr schwer messbar. Die Programmierer, die nach dem Lesen der Anforderungen sofort mit ihrem Programm begonnen haben, sind dann im Vergleich ihrer Arbeitszeit im Nachteil. Sie haben die Zeit, in der sie überlegen, wie das Programm funktionieren soll, auch berücksichtigt.

[2] oder ob die angegebenen Zeit auf Schätzungen basieren

10 Zusammenfassung

In der Zusammenfassung lässt sich feststellen, dass die betrachteten Unterschiede zwischen den einzelnen Programmierern *innerhalb* ein und derselben Sprache signifikant größer sind, als die gemittelten Unterschiede *zwischen* den einzelnen Sprachen.
Dies legt die Vermutung nahe, dass es für die Produnktivität der Sprache viel entscheidender ist, wie gut sich der Programmierer in ihr auskennt.
Jede Sprache kann also nur so gut sein, wie der Programmierer, der ihr Leben einhaucht.

11 Fazit/Ausblick

Informatik-Studierende sind heute "multilingual". Das bedeutet, dass sie in der Regel bereits in ihrem Studium mehrere Programmiersprachen kennenlernen. Da die Produktivität viel stärker vom Programmierer als von der verwendeten Sprache abhängt, ist es wichtiger grundlegende Konzepte zu erlernen. Hierzu gehören insbesondere Herangehensweisen an komplexe Probleme z.B. design-patterns. Durch die Kentnisse dieser *sprachenübergreifenden* Konzepte, kann man seine Produktivität in allen Sprachen wesentlich erhöhen. Außerdem ist das Erlernen einer weiteren Sprache, wenn man die allgemeinen Konzepte des programmierens kennt, vergleichsweise einfach, sodass man sich sehr schnell auf jede Programmiersprache einstellen kann.

Diese These soll auch durch folgende Abbildung gestützt werden:

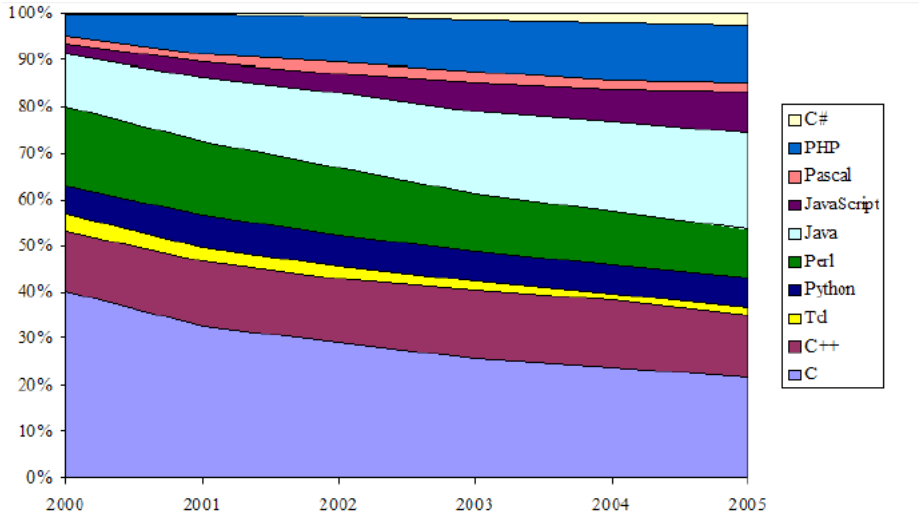

Abb. 6. Vergleich der Nutzung verschiedender Sprachen bei Open Source Projekten (aus: http://dml.cs.byu.edu/ cgc/pubs/WoPDaSD2007.pdf)

Die Abb. 6 zeigt beispielsweise die Verteilung von genutzten Sprachen bei Open Source Projekten. Man kann erkennen, dass sich im Trend von 2000 bis 2005 die Verteilung der benutzten Sprachen langsam einer Gleichverteilung annähert. Gemeint ist dabei, dass die Verteilung der Sprachennutzung 2005 mehr einer Gleichverteilung der Sprachen entspricht, als die Nutzung der Sprachen im Jahr 2000. Das unterstützt die These, dass die Wahl der benutzten Sprache im Gegensatz zur Erfahrung der Programmierer kein wesentlicher Aspekt für die

Produktivität darstellen kann. Andernfalls hätte man in diesen langen Zeitraum einen Trend zu den "produktiven" Sprachen feststellen können.

Literatur

[KK04] Charles Koelbel Ken Kennedy. Defining and measuring the productivity of programming languages. *International Journal of High Performance Computing Applications 18*, 2004.

[Pre00] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program. Technical report, Universität Karlsruhe, Fakultät für Informatik, March 2000.

[VKD07] Henry J Gardner Viktor K. Decyk, Charles D. Nortron. Why fortran. *Computing in sience & engineering*, 2007.