Institut für Nachrichtentechnik

Technische Universität Braunschweig
Institut für Nachrichtentechnik

Studienarbeit

Abstrakte Programmierung mittels Skripting

Untersuchung von geeigneten Implementierungen dynamischer Sprachen für die HSP

cand. inform. **Vitus Lorenz-Meyer**

Betreuer: Dipl.-Ing. Marius Spika

Braunschweig, den 20. September 2007

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Braunschweig, den 20. September 2007

____

Abstrakte Programmierung mittels Skripting

Vitus Lorenz-Meyer, Betreuer Marius Spika

Institut für Nachrichtentechnik
Technische Universität Braunschweig
Schleinitzstraße 22
38112 Braunschweig
emails {m.spika, v.lorenz-meyer}@tu-bs.de

Inhaltsverzeichnis

Tabellenverzeichnis v

Abbildungsverzeichnis vii

Quelltextverzeichnis ix

1. Aufgabenstellung xi
- 1.1. Gründe für diese Studienarbeit . . . xi

2. Einführung 1
- 2.1. Aufbau . . . 2

3. Vergangenheit von Skriptsprachen 3
- 3.1. Historische Einordnung von Skriptsprachen . . . 3
  - 3.1.1. Gleichnisse . . . 5
- 3.2. Geschichte . . . 6

4. Merkmale 9
- 4.1. Eigenschaften . . . 9
  - 4.1.1. Ousterhouts Dichotomie . . . 10
- 4.2. Typen und Beispiele . . . 10
  - 4.2.1. Javascript und Ecma . . . 11
- 4.3. Performance . . . 12
- 4.4. Sicherheit . . . 13
- 4.5. Zusammenfassung . . . 14

5. Sprachen 17
- 5.1. JACL . . . 17
- 5.2. JRuby . . . 17
- 5.3. Quercus . . . 18
- 5.4. Sleep . . . 19
- 5.5. Groovy . . . 19
- 5.6. Jython . . . 19
- 5.7. Pnuts . . . 19
- 5.8. BeanShell . . . 20
- 5.9. Rhino . . . 20
- 5.10. JavaFX . . . 20
- 5.11. FScript . . . 22

5.12. JudoScript 22
5.13. ObjectScript 22
5.14. Yoix 22
5.15. DynamicJava 23
5.16. Iava 23
5.17. FESI 23
5.18. iScript 24
5.19. Simkin 24

6. Evaluation **25**
6.1. Einleitung 25
6.2. Kriterien 27
6.3. Performance 28
6.3.1. Umgebung 28
6.3.2. Empty: Startup time 30
6.3.3. N-body: CPU lastig 30
6.3.4. IO: IO lastig 33
6.3.5. Hash: Hash performance 34
6.3.6. String: String Performance 35
6.4. Auswertung 37

7. Integration in die Handheld Software Platform (HSP) **39**
7.1. TestScriptXlet 39
7.2. JSShellXlet 39

8. Fazit **41**

9. Ausblick **43**

Literaturverzeichnis **47**

Abkürzungsverzeichnis **52**

Index **53**

A. Quelltexte **55**
A.1. Empty Quelltext 55
A.2. N-body Quelltext 55
A.3. IO Quelltext 57
A.4. Hash Quelltext 58
A.5. String Quelltext 58
A.6. HSP Implementierung 59
A.6.1. SimpleXlet 59
A.6.2. JSShellXlet 62

Tabellenverzeichnis

5.1. Evaluierte Systeme 18

6.1. Empty Skript Performance - Desktop Sun 1.6 JVMTI 30
6.2. IO Skript Performance in Java 1.6 (Desktop) und JVMTI 34
6.3. Hash Skript Performance Java 1.6 35
6.4. String Skript Performance Java 1.6 37

Abbildungsverzeichnis

6.1. Empty Skript Performance 31
6.2. N-body Skript Performance 2 32
6.3. N-body Skript Performance 32
6.4. IO Skript Performance 2 33
6.5. Hash Skript Performance 2 35
6.6. String Skript Performance 2 36

Quelltextverzeichnis

5.1. Java AWT “Hello World” - traditionelle Syntax 21
5.2. Java AWT “Hello World” - JavaFX Script deklarativ 21
5.3. Simkin XML Syntax 24
A.1. Java empty 55
A.2. Python N-body 55
A.3. Pnuts IO 57
A.4. JavaFX Hash 58
A.5. Iava Hash 58
A.6. SimpleXlet 59
A.7. JS TestScriptXlet 61
A.8. JSShellXlet 62
A.9. test.js 63

1. Aufgabenstellung

Die Aufgabe dieser Studienarbeit ist es, verschiedene Implementationen von Skripting Sprachen zu vergleichen, zu profilieren und am Ende eine Empfehlung auszusprechen, welche am besten für die HSP geeignet wäre. Dazu ist es nötig, sich sowohl einen Überblick über die Historie, die existierenden Techniken der Skriptsprachen und deren verschiedene Kategorien zu verschaffen, als auch Test-Skripte zu schreiben und auszuführen. Diese sollen nur einen Überblick über die Performance der Sprachen in den Bereichen CPU, I/O und Memory geben. Da der PDA auf dem die HSP läuft über die Java Virtual Machine (JVM) in der Connected Devices Configuration (CDC) verfügt, müssen die Interpreter der jeweiligen Sprachen sowohl in Java verfügbar sein, als auch auf dieser Version der JVM laufen. Neben Tests der Sprachen in der CDC auf einem Desktop PC, sollten die vielversprechendsten Sprachen auch direkt auf dem PDA getestet und profiliert werden. Ausserdem sollte in einem kurzen Beispiel demonstriert werden, wie sich Skripte in der HSP verwenden lassen.

Dabei soll die Studienarbeit auf Sicherheitsaspekte genauso eingehen, wie auf Ausblicke und zukünftige Entwicklungen der einzelnen Sprachen.

1.1. Gründe für diese Studienarbeit

Warum eine Skriptsprache für die HSP, wenn doch Java vorhanden ist, und sich ergo alles in Java programmieren ließe?

Diese Frage lässt sich anhand der in Kapitel 3 genannten Gründe beantworten: Skriptsprachen erleichtern die Programmierarbeit um ein vielfaches, denn sie verschnellern und vereinfachen das Codeschreiben durch ihren höheren Abstraktionsgrad. Dies spiegelt sich in einer höheren Ausdruckstärke wider, wofür es leider noch keinen Standard Index geschweige denn eine Übereinkunft gibt [7]. Die einzige Möglichkeit, relative Ausdrucksfähigkeiten zu beurteilen, besteht z.Zt. in dem Vergleich der *GZip* Größe in Bytes, das von [28] angezeigt wird. Demnach weisen weisen die folgenden vier Skriptsprachen die beste Ausdrucksfähigkeit mit 1,0 auf (dabei ist die Standardabweichung in Klammern mit angegeben). Ruby (1,18), Python (1,18), Perl (1,21) und JavaScript (1,23). Zum Vergleich: PHP Hypertext Preprocessor (PHP) fällt auf Platz 8 mit 1,3 (1,51) und Java auf Platz 29 mit 2,0 (2,37).

Die Ausdrucksfähigkeit hängt stark von der Syntax einer Sprache ab, denn wenn es kompaktere Befehle für häufig benutzte Aufgaben gibt, auch wenn diese etwas Flexibilität vermissen lassen (die nur in wenigen Fällen gebraucht wird), ist schon viel gespart. Skriptsprachen bieten z.B. meistens eine sehr kompakte Syntax für

sogenannte Hashes - assoziative Arrays - die im Vergleich zu Java viele Zeilen im Quelltext einspart. Die verlorene Flexibilität, die im Ernstfall durch eine größere Anzahl Befehle aufgefangen werden muss, macht in den meisten Fällen nichts, und insgesamt sind weniger Zeilen Quelltext in einer Skriptsprache vonnöten.

Das bedeutet für die Verwendung dieser Sprachen also, dass eine solche Sprache immer dann benutzt wird, wenn große Listen oder viele ungeordnete Daten (auch Strings) bearbeitet werden, oder sehr kleine Programme schnell geschrieben und debuggt werden sollen ("rapid prototyping"). Programme, die bei dieser Verarbeitung allerdings sehr stark auf Effizienz angewiesen sind, z.B. wegen der schieren Größe, dann empfiehlt sich keine Skriptsprache, da die Ausführungszeit von Skriptsprachen relativ zu System-level Sprachen überlinear wächst.

Viele alltägliche Funktionen stehen bereits effizient in Standard Bibliotheken der Skriptsprachen zur Verfügung und müssen nicht neu geschrieben werden. Wenn dann doch einmal etwas schief geht, kann in einer Skriptsprache der Fehler meist schneller gefunden werden, da Programme dort direkt interpretiert, d.h. ausgeführt werden können, ohne vorher erst kompiliert werden zu müssen, was u.U. sehr lange dauern kann. Darüber hinaus sind Skriptsprachen nicht nur toleranter was Fehler angeht, sondern bieten auch meistens informativere Fehlermeldungen, falls doch einmal einer auftritt.

Durch die Verknüpfung von Java und einer darauf aufbauenden Skriptsprache bekommt die HSP das Beste aus beiden Welten: die Einfachheit und abstrakte Welt der Skriptsprachen und die etwas kompliziertere aber dafür performantere Welt von Java. Große Software Projekte lassen sich durch die Benutzung zweier verschiedener Sprachen, einer low-(system-)level und einer high-(skript-)level Sprache, elegant aufteilen in 2 Domänen: die der performance-kritischen Bereiche tief im Program, wie z.B. mathematische Berechnungen; und die der nahe am Benutzer laufenden Teile, wie die Graphical User Interfaces (GUI) oder Plugins.

2. Einführung

Skriptsprachen verdanken ihren Namen ihrer Ähnlichkeit zu “Scripten” beim Theater oder Film. Ein Theater-script ist eine Abfolge von Befehlen, die dem Schauspieler gewisse Freiheiten bei der Umsetzung, den sogenannten “Interpretationsspielraum”, lässt. Skripte müssen von den Schauspielern also “interpretiert” werden, um sie in konkrete Bewegungen oder Mimik umzusetzen. Genauso ist ein Skript für eine Skriptsprache eine sehr abstrakte Abfolge von Befehlen, die dem Interpreter bei der Umsetzung in Maschinenbefehle einen gewissen Freiraum lässt, und vor der Ausführung “interpretiert” werden muss. Die Benennung kommt aus den Anfängen von Skriptsprachen, wo diese meistens noch als sogenannte *Batch-* oder *Job-control* Sprachen innerhalb von Unixsystemen ihre Arbeit verrichteten - dort hießen diese sehr kurzen Programme *scripts*. Um die Entwicklung der Skriptsprachen von einfachen Job-control zu starken, Turing kompletten Programmiersprachen widerzuspiegeln bevorzugen viele Entwickler heutzutage die Bezeichnung “dynamische Programmiersprache”. In dieser Studienarbeit werden beide Begriffe gleichbedeutend verwendet.

Skriptsprachen liegen eine Abstraktionsebene über den sogenannten “Hochsprachen” (C, C++ etc.) und sind deshalb schneller und einfacher zu schreiben. Dieses “Deuten” des Skriptes durch den Interpreter, im Gegensatz zum einfachen “kompilieren” durch einen Compiler - also ein umgangssprachliches “Anhäufen” der Befehle in Maschinensprache - ist natürlich aufwendiger und dauert länger. Auf der anderen Seite gewinnt man dadurch die Unabhängigkeit von einer bestimmten Plattform. Mittlerweile gibt es auch Mischformen, die diese Unterscheidungsgrenze verwischen. Java sowie Microsofts .NET kompilieren den Quelltext in einen Zwischencode (“bytecode”) und interpretieren diesen dann zur Laufzeit auf die jeweilige Zielplattform, das sogenannte Just-in-time Compilation (JIT). JIT ist von Vorteil, da beim Kompilieren nach Bytecode die meiste und längte Arbeit des parsens schon erledigt wird und eine kompaktere, maschinenverständlichere aber immer noch plattformunabhängige Syntax herauskommt. Dieser Bytecode kann dann zur Laufzeit schneller in auf die Zielplattform angepassten Maschinencode übersetzt werden, der dann die Möglichkeit des Cachens bietet, um weitere Ausführungen zu optimieren.

Darüber hinaus kann der Interpreter beim JIT'en mit Hilfe von Statistiken, die bei der Ausführung erzeugt werden, den Maschinencode weiter verbessern, indem er versucht, besonders häufig ausgeführten Code - sogenannte *HotSpots* - noch stärker zu optimieren. Dieses wird in der seit Java 1.3 zum Standard gewordenen Java Virtual Machine (JVM) eingesetzt - daher der Name HotSpot Virtual Machine (VM).

Man kann also grundsätzlich sagen, dass Skriptsprachen meistens interpretiert werden, wohingegen in Unterscheidung dazu, die sogenannten "System-level" Sprachen kompiliert werden. Obwohl die Unterscheidung zwischen *Kompilierung* und *Interpretierung* in der Praxis sehr schwierig, kaum vorhanden oder sehr verwaschen ist, sich also eher nur als generelles Label, denn als Unterscheidungsmerkmal eignet. Die Schritte der Kompilierung - lexikalische, syntaktische und semantische Analyse - sind bei interpretierten und kompilierten Sprachen notwendigerweise gleich, nur der Umfang und Zeitpunkt der einzelnen Schritte unterscheidet sich. Die Unterscheidungsgrenze ist nicht klar definiert, daher die Schwierigkeiten der Bezeichnung einzelner Sprachen.

2.1. Aufbau

Diese Studienarbeit gliedert sich grob in 2 Teile, der erste Teil befasst sich mit Literaturrecherche und der zweite Teil behandelt die Evaluation. Der Erste untergliedert sich wiederum in eine kurze Betrachtung der Entwicklung von Skriptsprachen, Merkmale von Skriptsprachen und daraus resultierende Klassifizierungen, sowie Typen und Beispiele. Als letztes befasst sich dieser Teil noch kurz mit der Thematik der Performance, auf die im zweiten Teil noch stärker eingegangen wird, sowie mit Sicherheit von Skriptsprachen.

Der zweite Teil stellt vorhandene Implementierungen von Skriptingsprachen für die Java Plattform vor, die in dieser Arbeit evaluiert wurden und betrachtet schließlich diejenigen näher, die mit Java in der Connected Devices Configuration (CDC) liefen und profiliert diese. Dazu wurden fünf kurze Skripte geschrieben, die die Performance verschiedener Aspekte der Implementierungen testen und vergleichen. Auf die Auswahl und Bedeutung dieser Skripte wird im zweiten Teil noch näher eingegangen.

Die letzten beiden Kapitel befassen sich mit der Auswertung und einem Ausblick für Skriptingsprachen im Allgemeinen. Appendix A liefert die Quelltexte zu den Skripten und Appendix B den Inhalt der beigelegten CD.

3. Vergangenheit von Skriptsprachen

In diesem Kapitel wird kurz die Vergangenheit der Skriptsprachen vorgestellt und eine historische Parallele zu dem Zwiespalt zwischen Assembler- und Hochsprachen gezogen.

3.1. Historische Einordnung von Skriptsprachen

Anhand des Zwiespaltes zwischen Assemblersprachen und Hochsprachen, der sich über 30 Jahre lang durch die Computergeschichte zog, lässt sich der Konflikt zwischen Hoch- (in diesem Zusammenhang auch "System-level-Sprachen" genannt) und Skript-Sprachen gut nachvollziehen.

Assembler bietet den Vorteil von Eindeutigkeit und daher die Notwendigkeit der Optimierung "von Hand". Allerdings muss jeder Maschinenbefehl einzeln geschrieben werden, was einerseits mehr Zeit kostet und andererseits die Fehleranfälligkeit sowie den Wartungsaufwand im Vergleich zu Hochsprachen erhöht. Generell bleibt die mittlere Anzahl Bugs pro Zeile gleich, d.h. Codedichte und Fehleranzahl sind umgekehrt proportional zueinander. Des weiteren schreibt ein Programmierer im Mittel die gleiche Anzahl Zeilen über einen Zeitraum - gleich in welcher Sprache [14, 30].

Hochsprachen sind für den Menschen einfacher zu verstehen, und daher auch schneller zu schreiben und besser zu warten. Ausserdem sind sie kompakter, daher also auch schneller zu erzeugen. Andererseits lassen Hochsprachen dem Compiler bei der Kompilierung mehr Freiheiten, z.B. bei der Resourcen (Register-) Zuweisung und Befehlsreihenfolge. Dieses schlägt sich in einer längeren Kompilierungszeit bei den Hochsprachen im Gegensatz zu Assemblersprachen nieder, da dort der Compiler mehr Analysearbeit hat. Da die Resourcenverteilung von Plattform zu Plattform verschieden ist, sind Assemblerprogramme meistens an die Plattform gebunden, auf der sie geschrieben und auf die sie optimiert wurden. Hochsprachenprogramme hingegen werden meistens erst beim Kompilieren vom Compiler an eine bestimmte

Plattform angepaßt/optimiert, und sind erst von dem Zeitpunkt an an diese Umgebung gebunden (also auch zur Laufzeit).

Compiler waren aber die längste Zeit nicht in der Lage, der "Hand-optimierung" ebenbürtigen Code zu liefern. Als dies nach über 30 Jahren schließlich gelang, wurden Assemblersprachen in eine Nische für sehr hardwarenahe (Treiber) und zeitkritische (Steuerungs-) Aufgaben verdrängt.

Die nächste Idee der Programmiersprachenentwickler waren die **Virtuellen Maschinen**. Eine Virtuelle Maschine stellt, wie der Name schon sagt, eine virtuelle also nicht reale Hardware Plattform dar. Virtuelle Maschinen unterscheiden sich in dem Umfang, inwieweit sie ein komplettes (reales) System nachbauen. Hardware-virtuelle Maschinen abstrahieren eine komplette Hardware-Plattform und erlauben es daher, in der virtuellen Maschine ein Betriebssystem zu installieren. Applikations-virtuelle Maschinen abstrahieren das Betriebssystem – und damit auch die Hardware – soweit, dass eine Software nur noch für diese VM geschrieben werden muss, und nicht mehr für jedes Betriebssystem. Von diesen VMs ist hier die Rede.

VMs bieten den zusätzlichen Vorteil der integrierten Fehlerbehandlung und Erkennung zur Ausführungszeit, sowie der automatischen Speicherverwaltung. Dies ermöglicht es einerseits, genau zu erkennen, wo und warum ein Fehler aufgetreten ist, sowie eventuell auftretende Unbekannte Fehler abzufangen und zu behandeln (dies bieten allerdings auch schon Hochsprachen). Die Speicherverwaltung erspart dem Entwickler, sich selber um Speicher für seine Objekte kümmern zu müssen und diesen auch nachher wieder freizugeben.

Ähnlich, wie Assembler zu Hochsprachen, verhält sich die Beziehung von Hochsprachen zu **Skriptsprachen**. Hochsprachen sind eindeutiger und länger als Skriptsprachen aber daher wiederum besser von Hand zu optimieren als Skriptsprachen. Skriptsprachen auf der anderen Seite sind abstrakter und prägnanter und deshalb einfacher zu verstehen und zu schreiben. Sie erfordern allerdings mehr Aufwand bei der Interpretierung, also zur Ausführungszeit, und sind daher etwas langsamer. Da Skriptsprachen meistens von einer bestimmten Plattform abstrahieren und die "Interpretierung" (d.h. Übersetzung der Befehle in Maschinencode) der Befehle on-the-fly, also zur Ausführungszeit, vornehmen, sind sie wiederum unabhängig von der Plattform. Compiler hingegen versuchen schon beim Kompilieren möglichst viele Optimierungen auf die jeweilige Hardware vorzunehmen und binden ihre Programme deshalb an die zum kompilieren benutzte Zielplattform.

Skriptsprachen erhalten durch die Ausführung in einer VM die Fehlerbehandlung und

Speicherverwaltung von diesen mit, sowie die Abstraktion vom darunterliegenden Betriebssystem. Hochsprachen bieten dies nicht, und sind meisten auf ein bestimmtes Betriebsystem abgestimmt.

Des weiteren eröffnet sich Skriptsprachen die Möglichkeit, interaktive Shells zu implementieren, da globales Wissen für die Ausführung eines Befehls nicht notwendig ist; wie es bei kompilierten Sprachen zur Optimierung und semantischen Analyse verwendet wird. Diese Shells ermöglichen es daher, sehr schnell einen Befehl oder einen kurzen Algorithmus einzutippen und auszuprobieren - was auch als "rapid prototyping" bekannt ist.

Im Gegensatz zu Assembler bieten Hochsprachen einen großen Umfang an wiederverwendbarem Code in Form von Opensource Bibliotheken. Sie müssen zwar erst installiert oder in die Laufzeitumgebung eingebunden werden, damit sie benutzt werden können, haben aber den großen Vorteil, dass das Rad in Hochsprachen nicht zweimal erfunden werden muss.

Skriptsprachen kommen darüber hinaus von Haus aus mit einer großen Auswahl von wiederverwendbarem Code. Dieser muss weder vom Entwickler noch vom Benutzer gesucht oder installiert werden. Dadurch kann sich der Programmierer einer Skriptsprache voll auf die Entwicklung neuer Funktionen durch das Zusammenfügen bereits bestehenden Codes konzentrieren (daher manchmal auch die Bezeichnung "glue-languages" - Klebesprachen).

3.1.1. Gleichnisse

Es gibt mehrere Gleichnisse im Internet, die diesen Umstand beschreiben.

3.1.1.1. Zeitmanagement

Dynamische Sprachen versuchen, dem Entwickler so wenig wie möglich im Weg zu sein, sodass dieser schnell mit der Implementierung fertig ist, und anfangen kann, sein Programm zu testen. Java oder andere Hochsprachen im Unterschied dazu, versuchen regelrecht, den Entwickler bei seiner Arbeit zu verlangsamen (zu seinem eigenen Vorteil), sodass am Ende, wenn er fertig ist, ein korrekteres (robusteres) System entstanden ist [29].

3.1.1.2. Hausbau

Skriptsprachen erlauben es dem Entwickler, Motorwerkzeuge und vorgefertigte komplette Teile für den Bau seines Hauses zu benutzen, anstatt die Axt selber herzustellen, damit das Holz zu fällen, mit dem die Schmelze betrieben wird, die das Eisen schmilzt, um die Basis zu schaffen, auf dem später einmal das Haus stehen wird und so fort. [3].

3.1.1.3. Hacker und Maler

Programmieren und Sketchen beim Malen sind sehr ähnlich - ein Programmierer sollte Programme beim Schreiben "herausfinden", genauso wie Architekten, Maler und Schriftsteller. Programmiersprachen sollten also dehn- und formbar sein, denn sie sind dazu da, neue Programme zu erfinden, nicht Programme auszudrücken, die schon fertig gedacht sind. Sie sollten Bleistifte sein, keine Kugelschreiber. Entwickler brauchen eine Sprache in der sie kritzeln, klecksen und schmieren können und keine, in der sie mit einer Teekanne voller Typen auf ihren Knien balancierend mit einem alten, strikten Kompiler freundlich ein Programm durchsprechen müssen. [15].

3.2. Geschichte

Erste Vertreter, die mit Recht Skriptsprache genannt werden können, waren als "batch-" oder "job control" Sprachen bekannt, die dazu dienten, auf großen Rechenanlagen mehrere Programme hintereinander auszuführen. Dieser Typ von Skriptsprache existiert heutzutage – etwas mächtiger – in Form der **Shell-Sprachen** immer noch. Das heißt, erste Vertreter von Skriptsprachen waren im Prinzip nur als Vorsteher oder Aufseher von "richtigen" - von Hand in Assembler - geschriebenen Programmen gedacht, ohne dabei selber gut genug für eine richtige Programmiersprache zu sein.

Dank des Umstandes, dass Skriptsprachen verschiedene andere Programme verknüpfen, werden sie auch manchmal als "glue" (Klebe-) Sprachen bezeichnet. Sie wurden nicht nur zwischen, also ausserhalb, abgeschlossener Programme, eingesetzt, sondern auch innerhalb, um interne Abläufe einfacher - in einer simpleren Sprache - modellieren und beschreiben zu können, sogenanntes "application scripting".

Es gibt Computer Experten, die argumentieren, dass jedes große Projekt aus mindestens zwei Programmiersprachen bestehen sollte - eine schnelle für low-level,

interne Abläufe, die schnell sein müssen, sowie eine high-level Skriptsprache, für nicht zeit-kritische Abläufe, wie die Graphical User Interfaces (GUI), Filter, Plugins, Mods und dergleichen [14]. Dieses lässt sich in professionellen Programmen und Spielen auch beobachten. Große Grafik- und 3d-Animationsprogramme (wie Adobe Photoshop oder 3dsMax) sowie Spiele (Halflife, Unreal, Doom usw.) spezifizieren ihre eigene Skriptsprache, oder bringen Schnittstellen (Application Programming Interfaces (APIs)) für extern kompilierte Module mit.

Nach und nach entwickelten sich auch die externen "Klebe-sprachen" weiter, und erhielten selber Strukturen und Merkmale einer richtigen Programmiersprache. Daraus entstanden dann schließlich die "general scripting languages", die nicht nur innerhalb von Programmen oder zum Verknüpfen ebenjener verwendet werden konnten, sondern für alles, wofür auch Hochsprachen eingesetzt werden. Das heißt, sie sind wie eine richtige Programmiersprache Turing-vollständig, können also für alles eingesetzt werden, was sich mit einer Turingmaschine modellieren lässt.

1987 veröffentlichte Larry Wall die erste Version seiner Sprache "Perl". Diese war ursprünglich als sogenannte "text processing language" gedacht, um awk [24] und sed [25], zwei Text-Werkzeuge für die Unix-Shell, zu ersetzen. Nach und nach wurde die Sprache jedoch immer mächtiger und betrat die Liga der "general purpose scripting languages", die damals noch von Sprachen wie MUMPS [13], Lisp [12] oder APL [1], sowie Smalltalk [26] und Tcl [27] angeführt wurde. Als Mitte der 90er Jahre das Internet verbreiteter wurde, und dynamische Webseiten langsam *en vogue* kamen, wurde zuerst meist Perl, später auch Sprachen wie Python, PHP Hypertext Preprocessor (PHP) und VisualBasic für server-seitig dynamische Webseiten eingesetzt.

Python [19] mit seinen vielen Bibliotheken, die z.B. auch Threading ermöglichen, war als ein Nachfolger von ABC [1] gedacht und eignet sich für alles, wo Einfachheit, Wartbarkeit und Lesbarkeit eine Rolle spielen, wie z.B. Shell-skripte für Packet Management Systeme (Portage in Gentoo) oder in 3d-Animationssoftware wie Maya und Blender.

PHP [17] erschien um 1994 herum als Ersatz für Rasmus Lerdorfs "Personal Home Page Tools" - ein Set aus einfachen Perl Skripten, die er für den Unterhalt seiner Webseite benutzte. PHP ist von Anfang an als reines Common Gateway Interface (CGI) für das server-seitige Skripten im Internet gedacht gewesen. Erst mit späteren Versionen kamen Fähigkeiten für den Zugriff auf die Konsole und Command Line Interface (CLI)-Paramter hinzu, sowie Klassen und weitere Merkmale von general purpose Skriptsprachen.

Ruby [21] wurde im Jahre 1995 von Yukihiro Matsumoto veröffentlicht, hauptsächlich weil er eine Sprache so mächtig wie Perl, aber mit weniger Unstimmigkeiten, wollte. In Ruby ist alles ein Objekt (sogar Codestücke) und daher lassen sich auch Codestücke an Funktionen übergeben, was z.B. sehr einfache Iteratoren erlaubt. Ruby hat mittlerweile durch die Verbreitung von "Ruby on Rails" [22] (vormals BaseCamp) einen hohen Bekanntheitsgrad gewonnen und wird immer häufiger für die Standardkonfiguration einer Webseite verwendet, also dynamische Webseiten mit Datanbank Backend, denn Rails abstrahiert die komplette Datanbankabwicklung (anlegen, verwenden, abfragen etc.).

4. Merkmale

4.1. Eigenschaften

Alle Skriptsprachen werden in einer (Laufzeit-) Umgebung ausgeführt, die es Ihnen erlaubt, einige praktische Erfindungen moderner Programmiertechnik zu benutzen - allen voran dynamische Speicherverwaltung (“garbage collection”). Ausserdem erlauben fast alle Skriptsprachen einen sehr laxen Umgang mit den Typen ihrer Variablen - auch bekannt als “weak” oder “duck” typing. Hochsprachen versuchen schon beim Kompilieren möglichst viel Arbeit zu erledigen, z.B. das Typ-checking. Dies kann aber nur gelingen, wenn der Typ einer Variable eindeutig deklariert wird.

Dieser Umgang besteht streng genommen aus zwei verschiedenen Dingen: Wie eine Variable bezeichnet wird, und wie sie behandelt wird. In einer “statically typed” Sprache wird ein Variablen Name sowohl an ein Objekt als auch an einen Typ gebunden, wohingegen eine “dynamically typed” (“latent”) Sprache einen Variablennamen nur an ein Objekt bindet. In einer “strongly typed” Sprache müssen nicht verwandte Variablen explizit gecastet werden, bevor sie in einem Befehl zusammen verwendet werden können. Verwandte Variablen sind zum Beispiel Integers und Floats. In einer “weakly typed” Sprache können Variablen hingegen dazu “genötigt” werden mit einer anderen zusammen zu arbeiten, obwohl sie nicht verwandt sind - der Interpreter “rät” dann, welchen Effekt der Programmierer damit erreichen wollte - z.B. Integers und Strings.

Anders als Hochsprachen, scheren sich Skriptsprachen erst um den Typ einer Variablen, wenn sie bei der Interpretierung an der entsprechenden Stelle im Code angelangt sind und erraten den Typ aus dem Kontext, oder behandeln, wie einige Interpreter, jede Variable als String - die dann bei Bedarf gecastet wird. Manche Skriptsprachen, wie Python, behandeln einfach jedes Wort, was kein Schlüsselwort ist, als Variable und initialisieren neue Variablen einfach mit `null`. Sie müssen aber zuerst mit einer Zuweisung initialisiert werden.

Bei PHP geht das sogar, ohne dass der Name der Variable vorher mit einer Zu-

weisung bekannt gegeben wurde, das heißt PHP behandelt einfach alles, was kein Schlüsselwort ist erst einmal als Variable. Dabei kommt dem Interpreter allerdings zugute, dass viele Skriptsprachen für ihre Variablen ein sogenanntes "Sigil" vorschreiben, so etwas wie ein Prefix - in PHP und Perl z.B. ein Dollarzeichen ("$").

Python benutzt keine Sigils und erlaubt daher keine Verwendung von Variablen auf der rechten Seite, bevor sie nicht mit einer Zuweisung auf der linken Seite bekannt gemacht und initialisiert wurden.

4.1.1. Ousterhouts Dichotomie

Weder die Eigenschaft der `garbage collection` noch `dynamic typing` eignet sich aber zur Klassifizierung von Programmiersprachen, da weder die Syntax noch die Semantik einer Sprache von einem dieser Merkmale abhängt. Dieses ist auch als "Ousterhouts Dilemma" bekannt, weil Ousterhout genau dieses behauptet hatte [14].

Dies zeigt sich zum Beispiel an Java und der .NET Laufzeitumgebung (JVM bzw. Common Language Runtime (CLR)), die beide "garbage collection" implementieren aber nichtsdestotrotz stark typisiert sind. Beide kompilieren den Quellcode in eine "Bytecode" genannte Zwischendarstellung (Java Bytecode für Java bzw. Common Intermediate Language (CIL), früher Microsoft Intermediate Language (MSIL), für .NET), die länger und schon etwas eindeutiger ist, also Resourcenzuweisung usw. festlegt - soweit das ohne Festlegung auf eine Plattform möglich ist - die dann schließlich zur Laufzeit für die jeweilige Plattform interpretiert wird. Dieses geht meist aber schneller als das komplette Interpretieren von Grund auf.

4.2. Typen und Beispiele

Es gibt verschiedene Arten von Skriptsprachen, je nachdem welche Aufgabe sie übernehmen sollen (siehe dazu auch [9]). Wie schon erwähnt, sind Program-"interne" Sprachen - also Sprachen, die im Inneren eines größeren Ganzen werkeln, um z.B. kleine alltägliche Aufgaben zu übernehmen oder neue Funktionen hinzuzufügen, sehr verbreitet. Dazu zählen **Applikations-skripting** Sprachen, wie *VBA*, *hypertalk*, *maya embed lang*, *actionscript* aber natürlich auch Spiele-interne Sprachen, die hauptsächlich fürs Modding benutzt werden, wie z.B. *UnrealScript*, *QuakeC* usw.

Weiterhin lassen sich im gewissen Sinne auch **Job-Control** und **Shell-Sprachen**,

wie *applescript*, *bash*, *DOS-batch* und *Windows Scripting Host (WSH)* zu den Applikations-skripting Sprachen zählen, da sie innerhalb einer Shell oder eines OS ausgeführt werden, um eine bestimmte Aufgabe zu übernehmen oder zu erleichtern. Aus Sicht eines Webservers gilt das genauso für sogenannte **Web-programming** Sprachen, wie *coldfusion*, *Java Servlet Pages (JSP)* und PHP (server-side) und aus der Sicht des Browsers für client-side Sprachen, wie *J(ava)-*, *ECMA Script* und *VBScript*.

Die Kategorie der **extension/embeddable** Sprachen versucht, diesen Vorteil noch etwas weiter zu treiben, und Sprachen zu entwickeln, die sich generell in jeder Anwendung einsetzen lassen, wenn der Programmierer dieser Anwendung gewisse Stellen ("Hooks") vorsieht, wo die (Skript-) Sprache ansetzen kann. Das ist das "plugin" Prinzip, die sich prinzipiell in jeder Sprache schreiben lassen, also auch Skriptsprachen. Beispiele dieser Sprachen wären also alle generellen Skript- sowie Hochsprachen und speziell *Guile*, *Lua*, *Pawn*, *Python*, *Squirrel* und natürlich auch *ECMA* (J(ava)Script).

Desweiteren existieren noch **GUI scripting** Sprachen, für die Erstellung und Benutzung von graphischen Oberflächen, wozu sich *XUL* (Mozilla) - die sich allerdings aus XML und JavaScript zusammensetzt - *Expect* und *Automator* zählen lassen.

Die schon erwähnten **text processing** Sprachen sind zum schnellen extrahieren und analysieren von Informationen gedacht; *AWK*, *sed* und *Perl* sowie *XSLT* zählen zu ihren Vertretern.

Als letztes zählen die meisten der großen Skriptsprachen (*Perl*, *PHP*, *Python*, *Ruby*, *Tcl* etc.) genauso zu der Kategorie der sogenannten **general purpose** Skriptsprachen, also Sprachen, die nicht nur für einen bestimmten Zweck sondern für alles mögliche eingesetzt werden können - wie eben "richtige" Hochsprachen auch.

4.2.1. Javascript und Ecma

Javascript wurde im Jahre 1995 vom damaligen Netscape erfunden und in die Version 2.0B3 des Browsers eingebaut, zuerst als *Mocha* und später als *LiveScript*. Die Namensänderung zu JavaScript geschah ungefähr zu der Zeit, als Netscape Sun's Java in ihrem Browser unterstützte, aber eigentlich hat Javascript nicht viel mit Java gemein, ausser den gemeinsamen syntaktischen Wurzeln in C. Durch die Möglichkeit, Webseiten mit Hilfe von JavaScript dynamisch zu gestalten - dies wurde später bekannt als Dynamic HTML (DHTML) und mittlerweile auch Ajax - sah sich der immer beliebter werdendere Browser *Internet Explorer* von Microsoft bald dazu gezwun-

gen, auch JavaScript zu implementieren. Dies geschah mit einer kompatiblen aber einem leicht abweichendem Model folgenden Sprache namens JScript.

Um diese Inkompatibilitäten aus der Welt zu schaffen, einigten sich die beiden Browserhersteller später auf einen gemeinsamen Standard - EcmaScript, der ursprünglich von Microsoft eingereicht und später auch von JavaScript implementiert wurde. Auch einige andere Sprachen implementieren inzwischen EcmaScript, darunter ActionScript, die Skriptsprache für Macromedia Flash.

Im Gegensatz zu JScript, das einen Klassen-Ansatz verfolgt, ist JavaScript eine *prototype* basierte Sprache, im Endeffekt also Klassenlos. In Prototyp Sprachen gibt es keine Unterscheidung zwischen Klassen und Instanzen dieser Klassen, die auf eine Differenzierung in *Verhalten* (behaviour) und *Zustand* (state) hinausläuft. Stattdessen werden Objekte in diesem Ansatz nur geklont und erhalten automatisch alle Eigenschaften und Zustände ihres Vorgängers. Diese Objekte können zur Laufzeit verändert und einfach Methoden und Variablen hinzugefügt werden, was sich bei den meisten Implementationen dann auch sofort auf etwaige “Kinder” auswirkt. Beim Funktions-*dispatch* muss nur die Kette aller Objekte von Vater zu Kindern so lange verfolgt werden bis ein passendes Objekt gefunden wurde.

Die Kritik an diesem Ansatz verläuft meistens entlang der gleichen Linien, wie die Kritik der stark-typisierten (Hoch-)Sprachen Verfechter an den schwach typisierten (Skript-) Sprachen, nämlich dass er weder Sicherheit, Vorhersagbarkeit, Korrektheit noch Effizienz garantiert.

4.3. Performance

Nach [18] sind, wie zu erwarten war, kompilierte Sprachen im Mittel am schnellsten bei der Ausführung datenintensiver Programme. Skriptsprachen folgen mit einigem Abstand dahinter, sind aber dank immer schneller werdenderer CPUs am Aufholen. Dazwischen finden sich die Mischsprachen, die einen Zwischencode benutzen, also Java und .NET, sowie kompiliertes Python.

Bei rechenintensiven Programmen (große Hashtables und Maps) profitieren Skriptsprachen von der effizienten Implementierung ihrer internen Datenstrukturen - hauptsächlich assoziativen Arrays - und liegen im Schnitt etwas vor den Zwischensprachen, die ca. um den Faktor 2 langsamer sind als Hochsprachen. Die Laufzeitdifferenzen zwischen diesen drei Gruppen fallen jedoch meist weniger ins Gewicht, als die Unterschiede, die sich aus der Anwendung verschiedener Algorithmen oder

durch verschiedene Programmierstile ergibt.

Skriptsprachen verbrauchen im Durchschnitt etwas mehr Arbeitsspeicher als Hochsprachen, und Java erkauft sich seinen Geschwindigkeitsvorteil gegenüber Skriptsprachen mit erheblich mehr Arbeitsspeicher, was sich wohl durch die Benutzung von JIT erklären lässt.

Weiterhin wird aus dem Artikel ersichtlich, dass das die Zeit für das Entwickeln und Testen eines Skriptes im Schnitt weniger als die Hälfte der Zeit verbraucht, die das Entwickeln des gleichen Programms in einer Hochsprache in Anspruch nimmt. Dafür nimmt die Korrektheit der Skripte im Vergleich zu Hochsprachen Programmen nicht ab.

Das heißt, Skriptsprachen eignen sich immer dann wenn der sogenannte "edit-interpret-debug cycle" kürzer ist, als der "edit-compile-run-debug cycle". Auf deutsch also immer dann, wenn ein Skriptprogrammierer in der Zeit, die bei einem Hochsprachler fürs Kompilieren und Feststellung eines Bugs vergeht, den Bug schon lange gefunden und beseitigt hat.

Der vergrößerte Aufwand für den Interpreter bei der Ausführung von Skriptsprachen fällt mit zunehmend schneller werdenden CPUs immer weniger ins Detail, so dass es sich lohnt, Zeit für die Entwicklung und Ausführung eines gegebenen Programms vom Programmierer zum Computer zu verschieben. Dynamische Sprachen sind für den Menschen einfacher und vor allen Dingen schneller zu programmieren, dies allerdings auf Kosten der Ausführungszeit.

4.4. Sicherheit

Die Sicherheit einer Sprache steht und fällt mit den Programmierern, die sie benutzen. Eine Sprache ist nicht von selber unsicher, sondern wird es erst durch schlecht geschriebene Programme von unerfahrenen Entwicklern, mit Ausnahme einiger weniger Bugs in den Interpretern selber. Skriptsprachen sind deshalb anfälliger für Fehler von unerfahrenen Programmierern, weil das Einstiegslevel dieser Sprachen sehr viel niedriger ist. Das Programmieren in Skriptsprachen geht schneller von der Hand und mehr Flüchtigkeits- oder Faulheitsfehler treten auf. Ausserdem nehmen Skriptsprachen es dem Programmierer meist nicht übel, wenn eine Variable nicht den deklarierten Typ hat, daher schleichen sich Fehler dort leichter ein. Hochsprachen zwingen einen Programmierer dazu, sich mehr Gedanken über seinen Code zu machen, was generell in saubererem Code resultiert.

Andererseits sind Skriptsprachenprogramme kürzer, mithin also besser zu verstehen und einfacher zu schreiben, weshalb weniger Fehler erst entstehen. Ausserdem gibt es mehr Code, der sich einfach wiederverwenden läßt, z.B. für häufig benutzte Algorithmen, wie einen Netzwerkserver, was wiederum die Wahrscheinlichkeit, dort einen Bug einzubauen, verringert.

Dennoch kommt es in Skriptsprachen schnell zu den beiden am häufigsten auftretenden Sicherheitslücken in der Webprogrammierung, das Cross-Site Scripting (XSS) und die *SQL injection*. Das implizite Vertrauen in vom User eingegebene Werte, wenn diese ungeprüft an eine Funktion (z.B. Structured Query Language (SQL) oder Shell-command) weitergegeben werden, ist ein Problem in jeder Sprache, in Skriptsprachen aber besonders verbreitet, da diese traditionell eingabelastigen Webseiten zum Leben verhelfen.

SQL injection passiert bei zu leichtsinnigem Umgang mit vom Benutzer eingegebenen Werten, wohingegen XSS auf ein zu niedriges Sicherheitslevel der Browser bei der Interpretierung setzt. XSS setzt fast ausschließlich auf J(ava)Script und beschreibt grundsätzlich alle Techniken, die auf dem Austausch von Daten zwischen verschiedenen Seiten in einem einzigen Browser-Fenster, oder verschiedenen Seiten in verschiedenen Browser-Fenstern (Tabs) beruhen. Dies kann nicht nur räumlich (gleichzeitig verschiedene Fenster) sondern auch zeitlich (verschiedene Seiten nacheinander oder im Hintergrund im gleichen Tab) geschehen.

Weitere Sicherheitslücken oder Crash-Möglichkeiten, wie z.B. Memory-leaks sind für diese Studienarbeit nicht relevant, da alle Interpreter in der JVM ausgeführt werden, die eine “Sandboxing” Umgebung darstellt.

4.5. Zusammenfassung

Generell läßt sich also sagen, dass Skriptsprachen für eine Vielzahl von Aufgaben geeignet sind, bei denen es nicht so sehr auf Geschwindigkeit ankommt. Da es bei vielen großen Firmen mehr auf die “time-to-market”, also die Zeit von der Entwicklung bis zum Verkauf ankommt, und auf die Programmiererstunden - denn die sind teuer - als auf eine bis aufs Letzte optimierte Leistung, lohnt es sich, Programmiersprachen einzusetzen, die Zeit von der Entwicklung auf die Ausführung verschieben. Dieser Unterschied fällt auch nicht zuletzt dank immer leistungsfähigeren CPUs zunehmend weniger ins Gewicht, der Vorteil der schnelleren Entwicklung und des schnellen *mal eben Ausprobierens* in Code bleibt erhalten.

Es ist jetzt schon teilweise schwierig eine Programmiersprache einem der beiden Lager (dynamisch oder Hochsprache) zuzuordnen. In Zukunft wird es noch mehr Sprachen geben, die diese Grenze weiter verwischen, und die Bezeichnung damit auf eine reine nominelle Anwendung verbannen.

5. Sprachen

In diesem Kapitel werden kurz die verschiedenen Sprachen und ihre Interpreter für Java vorgestellt, die für diese Studienarbeit evaluiert wurden und dann auf die näher eingegangen, die mit der Java CDC liefen (in den Implementationen von Sun, IBM Webspheres J9 und Esmertec).

Dabei sei vorweg gesagt, dass davon nicht alle in Java mit der CDC liefen, die meisten davon wegen des Fehlens der Klasse `java.nio.Charset`. Wohingegen JavaFX Script nicht läuft, weil es die falsche ClassVersion (die MajorMinor) Version hat. Dieses bildet aber ohnehin eine Ausnahme, da es im Zuge der JavaFX Initiative von Sun auch bald für CDC spezifiziert und implementiert werden wird. Daher wird es hier zusätzlich profiliert. Diejenigen, die mit der CDC liefen sind in der Tabelle gekennzeichnet.

Nun werden jetzt alle oben in der Tabelle aufgeführten Sprachen und dazugehörigen Programme kurz angerissen und mit Vor- und Nachteilen näher vorgestellt. Des Weiteren wird kurz darauf hingewiesen, wenn und wie sich die Sprache zum Benutzen von Java Objekten und umgekehrt, ob sie sich zum Verwenden aus Java heraus eignet.

5.1. JACL

JACL interpretiert die Skriptsprache Tcl [27] und ist in Java geschrieben. Es erlaubt sowohl den Aufruf des Interpreters aus Java heraus, als auch die Benutzung von Java Objekten in Tcl. Leider fehlt die Klasse `java.nio.charset` in CDC, sodass sich JACL nicht ausführen lässt.

5.2. JRuby

JRuby ist ein Interpreter für Ruby [21] in Java. Es erlaubt auch die Interaktion mit Java in beiden Richtungen, stößt sich aber leider auch an dem Fehlen der Klasse

Tabelle 5.1.: Evaluierte Systeme

Name	Syntax	CDC?	Webseite
JACL	Tcl/Tk		http://tcljava.sourceforge.net/
Jython	Python	X	http://jython.org/
JRuby	Ruby		http://jruby.codehaus.org/
Quercus	PHP		http://www.caucho.com/resin-3.0/quercus/
Pnuts	Java-like	X	https://pnuts.dev.java.net/
BeanShell	Java	X	http://www.beanshell.org/
Rhino	JavaScript	X	http://www.mozilla.org/rhino/
Sleep	perl-like		http://sleep.hick.org/
Groovy	perl-like		http://groovy.codehaus.org/
JavaFX Script	Java-like		http://openjfx.dev.java.net/
FScript	Java-like		http://fscript.sourceforge.net/
JudoScript	JS-like		http://www.judoscript.com
ObjectScript	JS-like		http://objectscript.sourceforge.net/
Yoix	JS-like		http://www.yoix.org/
Simkin	XML		http://www.simkin.co.uk/
DynamicJava	Java	X	http://koala.ilog.fr/djava/
Iava	Java	X	http://members.tripod.com/mathias/
iScript	BASIC		http://www.servertec.com/products/iscript/

`java.nio.charset` in CDC.

5.3. Quercus

Quercus ist eine Implementation der Skriptsprache PHP [17] geschrieben für Cauchos httpd-server *Resin*. Dieser ist vollständig in Java geschrieben, und damit auch Quercus. Quercus erlaubt das Benutzen von in Java geschriebenen Modulen aus PHP heraus. Da PHP allerdings nur innerhalb eines Webservers Sinn macht aufgrund der Historie der Sprache, ist Quercus nicht als Stand-alone Shell Interpreter benutzbar. Da dafür also der Webserver gestartet werden muss, was einerseits sehr resourcenaufwending ist, und dieser andererseits sowieso nicht auf der CDC läuft ("unsupported MajorMinor Version 49.0") wird Quercus hier nicht weiter betrachtet.

5.4. Sleep

Sleep ist sehr genügsam und kommt in der Form einer einzigen 218KB großen .jar Datei daher. Diese startet die Sleep Konsole und nimmt auch Skripte direkt entgegen. Sleep erlaubt auch die Interaktion in beiden Richtungen mit Java. Wiederum scheitert Sleep in CDC an dem Fehlen von `java.nio.charset`.

5.5. Groovy

Groovy ist eine eigens für Java geschriebene Skriptsprache, die eine Syntax sehr nahe an Java aufweißt, erweitert um einige sinnvolle Elemente aus Python, Ruby und Smalltalk. Sie erlaubt die Interaktion in beide Richtungen. Groovy wird mittlerweile in dem Java Specification Request (JSR)-241 spezifiziert. Groovy stößt sich, im Gegensatz zu allen anderen Interpretern, daran, dass die Implementation von `String` in CDC keine Funktion `split` aufweist. Groovy Skripte lassen sich als einzige neben Jython auch direkt in Java Bytecode (also ".class" files) kompilieren, und danach wie normaler Java Code behandeln.

5.6. Jython

Jython ist ein Interpreter für Python [19] geschrieben in Java. Damit erlaubt es nicht nur den Zugriff aus Java auf alle in Python geschriebenen Module (und dessen umfangreiche Standard Bibliothek), sondern auch die Erweiterung von Java Klassen in Jython und umgekehrt. Ausserdem erlaubt Jython das statische Kompilieren von Python-code nach Java Bytecode. Jython läuft in CDC.

5.7. Pnuts

Pnuts ist eine Scripting Sprache, die eigens für die JVM geschrieben wurde. Es war zuerst als Tool zum schnellen Testen von Java Klassen gedacht und wurde dann später zu einer kompletten Skriptsprache erweitert. Pnuts erlaubt auch die Interaktion in beide Richtungen mit der JVM und richtet sich ausserdem nach dem JVM Scripting API (JSR-223) Die Syntax ist verwandt mit der JavaScript Syntax. An den

Performance Messungen zeigt sich aber, dass Pnuts nicht gut mit großen Zahlen umgehen kann, da die Performance beim N-body Skript mit Abstand die Schlechteste im Feld ist – sogar noch weit hinter JavaFX.

5.8. BeanShell

BeanShell ist in der JSR-274 spezifiert, die schon den sogennanten "voting process" überstanden hat, d.h. dass nun eine *BeanShell Expert Group* formiert wird, die eine Sprachen-Spezifikation für BeanShell schreibt, welche vielleicht in eine zukünftigen Version der J2SE integriert wird. BeanShell wird in nur einem 276KB großen .jar File ausgeliefert. Dieses erzeugt beim normalen Starten ein Editor Fenster, in dem sich mehrere bsh.shell sessions starten lassen und das komfortables Skripten erlaubt. Dieser Editor bietet sogar einen integrierten Class-Browser. Ausserdem läßt sich BeanShell in einer Konsole als Interpreter starten. BeanShell kombiniert Standard Java Syntax mit der sogenannten "loosely typed" Java Syntax; diese ist dynamisch typisiert, sieht aus wie normales Java, und erlaubt es, Variablen zu benutzen, ohne sie explizit deklarieren zu müssen. Bsh erlaubt wiederum die Interaktivität in beide Richtungen.

5.9. Rhino

Rhino ist in die Mozilla Suite integrierter Interpreter, der die Javascript Unterstützung der Browser der Mozilla Familie sicherstellt. Dieser wird als eigenes Paket von Mozilla zur Verfügung gestellt. Es besteht aus einer einzelnen 693KB großen .jar File und weiteren Skripten, sowie Demos und vielem mehr. Dieses .jar startet eine interaktive JavaScript Shell. Wiederum unterstützt Rhino die beidseitige Interaktivität.

5.10. JavaFX

JavaFX Script ist Teil der JavaFX Familie, wozu im Moment noch JavaFX Mobile gehört. JavaFX Script ist eine dynamische aber dennoch stark typisierte Skriptsprache für Java, die direkt in Java-byte-code kompiliert wird. Die Syntax ist noch ähnlicher zu klassischem Java als bei den anderen Skriptsprachen, bis auf eine neue deklarative Syntax für die Initialisierung von großen Klassen. Diese ist an die mit

Ajax eingeführte “Array” Syntax in Javascript angelehnt und erlaubt das sehr platzsparende Initialisieren von GUI Elementen und großen Objekten, das in Java ja normalerweise viel Code in Anspruch nimmt.

Ein kurzes Beispiel: Dies ist der Code für Java AWT für ein “Hello World” Program:

Quelltext 5.1: Java AWT “Hello World” - traditionelle Syntax

```
var win = new Frame();
win.title = "Hello World JavaFX";
win.width = 200;
var label = new Label();
label.text = "Hello World";
win.content = label;
win.visible = true;
```

Man sieht, dass alle Felder der GUI Klassen einzeln initialisiert werden müssen, was bei vielen Elementen und Feldern sehr schnell unübersichtlich wird.

Die deklarative Syntax in JavaFX Script für das gleiche Program sieht hingegen so aus:

Quelltext 5.2: Java AWT “Hello World” - JavaFX Script deklarativ

```
class HelloWorldModel {
        attribute saying: String;
    }
    var model = HelloWorldModel {
        saying: "Hello World"
    };
    var win = Frame {
        title: "Hello World JavaFX"
        width: 200
        content: Label {
            text: bind model.saying
        }
        visible: true
    };
```

In der deklarativen Syntax ist besser zu sehen, welche Felder zu welchen Elementen gehören und wozu sie gut sind. Bei diesem einfachen Beispiel ist die deklarative Syntax etwas länger, aber für komplizierte GUIs lohnt sich die kompaktere Syntax durchaus.

Da JavaFX von Sun selber vorangetrieben wird, ist es zu erwarten, dass auch CDC bald JavaFX Script enthalten wird. JavaFX läuft allerdings noch nicht mit CDC zusammen, wegen einer falschen MajorMinor Version. Es wird hier aber nichtsdestotrotz profiliert, obwohl einige Schwierigkeiten bestanden, das Test-skript “n-body” für JavaFX umzuschreiben, nicht zuletzt wegen der untypischen Syntax - z.B. für Arrays, die als Klassenattribut anders definiert werden müssen, als als normale Variable. Nachdem N-Body schließlich als JavaFX Skript lief, verrechnete es sich aber trotzdem noch.

5.11. FScript

FScript ist eine sehr einfache Automatisierungssprache für das Skripten von Java. Es ist zwar möglich, Java Objekte zu erzeugen und zu benutzen, jedoch besitzt FScript keine eigenen Standard Libraries, so wie andere Sprachen, die z.B. File I/O u.Ä. ermöglichen, sodass es sich nur für sehr einfache Anwendungen eignet. Ausserdem bietet FScript keine interaktive Shell und damit auch keine Möglichkeit, Skripte von der Konsole zu starten. Es lässt sich nur aus Java heraus der Interpreter laden, um damit Skripte auszuführen. Daher wird FScript hier nicht weiter betrachtet.

5.12. JudoScript

Judo ist eine in Java geschriebene multi-purpose Skripting Language, was sie nicht nur für das Skripten von Java sondern auch für andere skripting Aufgaben, wie z.B. Shell, Java Database Connection (JDBC), eXtensible Markup Language (XML) und SGML, oder auch ActiveX. Das macht JudoScript besonders praktisch für alle erdenklichen Anwendungen des Skriptens, nicht aber für unsere Zwecke, da JudoScript auf viele Klassen in Java angewiesen ist, die CDC nicht aufweist, z.B. `java.sql`.

5.13. ObjectScript

ObjectScript hat eine Syntax dem JavaScript sehr ähnlich und bietet auch sonst alle Features der anderen Sprachen. Darüber hinaus liefert es eine Integrated Development Platform (IDE) mit, die neben einem komfortablen Editor mit Syntaxhighlighting und -checking, Debugger und Klassen-Explorer noch einiges mehr bietet, natürlich alles in Java. Leider läuft auch dieser Interpreter nicht mit CDC wegen einer falschen Major/Minor Version.

5.14. Yoix

Yoix ist eine general-purpose Skript Sprache von den AT&T Research Labs. Sie bietet eine Menge Standard Libraries für alle erdenklichen Aufgaben und ausserdem eine Schnittstelle von und zu Java Code. Entgegen der Java Philosophie enthält Yoix Zeiger, die allerdings durch das darunter liegende Java ihre Gefährlichkeit verlieren. Leider bietet Yoix keine interaktive Shell sondern nur die Möglichkeit Skripte direkt

an den Interpreter per Kommandozeile zu übergeben. Yoix läuft nicht unter CDC wegen des Fehlens der Klasse `java.awt.datatransfer.ClipboardOwner`.

5.15. DynamicJava

DynamicJava ist eine Skriptsprache, die Standard Java Code mit eingestreuten dynamischen Elementen interpretiert, ähnlich wie BeanShell. Dabei versucht DJava, im Gegensatz zu BeanShell so nahe, wie möglich an der ursprünglichen Java Syntax zu bleiben. Daher ist DJava in der Lage, Java Code mit sehr wenig bis gar keinen Änderungen zu interpretieren. Sie bietet einen Kommandozeilen Interpreter, der allerdings nicht interaktiv arbeitet, sowie eine einfache Swing GUI, die in der Lage ist, ihre Buffer mit Hilfe des Interpreters auszuführen. DynamicJava läuft mit der CDC zusammen. Merkwürdigerweise verstehen sich die anderen 5 Sprachen sowohl mit Suns CDC Implementierung als auch mit IBMs J9, ausser DynamicJava, denn das funktioniert nicht mit der J9.

5.16. Iava

Iava ist eine interpretierte Sprache mit Java Syntax, allerdings ohne die Möglichkeit, Klassen zu definieren. Vier Skripte, bis auf N-Body - das wäre ohne Klassen zu umständlich geworden, wurden für diese Studienarbeit in Iava geschrieben und profiliert. Iava läuft auch innerhalb von der CDC.

5.17. FESI

FESI ist der “Free European Computer Manufacturers Association (Ecma)Script Interpreter” für Java. FESI interpretiert EcmaScript nach dem Standard Ecma-262 [5] (Stand Juni 1997). Damit interpretiert FESI JavaScript Version 1.1 (oder JScript 1.1), ohne die Browser Zusätze. Leider läuft FESI nicht innerhalb der CDC wegen des Fehlens des Namenspaces `java.sql`.

5.18. iScript

iScript ist eine dynamische Sprache, deren Anweisungen innerhalb von Hypertext Markup Language (HTML) Tags geschrieben werden, die daher also, ähnlich wie PHP, nur innerhalb eines Webservers läuft. Der Interpreter dazu ist in Java geschrieben, und kann natürlich auch Skripte von der Kommandozeile entgegennehmen. Allerdings benutzt dieser Interpreter Klassen aus dem Namespace `javax.servlet.http` die nicht einmal im Standard J2SE vorhanden sind.

5.19. Simkin

Simkin ist eine interpretierte Sprache mit XML oder sogenannter “Tree-Node” Syntax. Das bedeuetet, das sich aller Simkin code in XML oder Datenbank (SQL) Code einbetten lässt. Simkin bietet leider keine interaktive Shell und kann auch keine Skripte direkt ausführen, dafür muss der Interpreter immer innerhalb eines Java Programms geladen und dann dazu benutzt werden, ein Skript auszuführen.

Der Autor stellt eine Version bereit, die innerhalb der Mobile Information Device Profile (MIDP) Umgebung laufen soll, allerdings habe ich nach etlichen Versuchen aufgegeben, diese zum Laufen zu überreden.

Hier ist ein kurzes Beispiel das die XML Syntax besser verdeutlicht, es deklariert einige Variablen und gibt diese dann in der “main” Funktion aus.

Quelltext 5.3: Simkin XML Syntax

```
1 <myobject>
  <Personal Debit="200.564" BankAssets="129.04">
  <Name>Simon Whiteside</Name>
  <Address>19 Allenby Road</Address>
  <City>London</City>
6 </Personal>
  <function name="main">
   trace("My name is " # Personal.Name # " and I live at " # Personal.Address # " in " # Personal.City);
   trace("The balance of my assets and debits is: \$" # (Personal:BankAssets-Personal:Debit));
  </function>
11 </myobject>
```

6. Evaluation

Dieses Kapitel bildet den zweiten Teil der Studienarbeit und befasst sich mit Performance Messungen verschiedener Implementierungen von Skriptsprachen für die Java Plattform. Dazu beleuchtet die Einleitung erst einmal die Frage, warum das sinnvoll ist, und wonach dabei gesucht wird.

6.1. Einleitung

Die Aufgabe dieser Studienarbeit ist, verschiedene Implementationen von Skriptsprachen zu suchen, die mit der Handheld Software Platform (HSP) laufen könnten, sie zu vergleichen und zu testen, in wie weit sie für dieses Gerät geeignet sind. Ein PDA auf dem die HSP läuft ist ein Handheld Gerät, das über Java in der CDC Version verfügt (momentan ist die *Esmertec* JVM [6] in der Version CDC 1.0/Personal Basis Profile (PBP) 1.0 installiert) und nur über einen beschränkten Speicher verfügt. Die Java CDC Version ist eine Konfiguration von Java, die speziell für mobile Geräte, wie PDAs gedacht ist, sie enthält die Kernklassen sowie eine kleine Auswahl an weiteren Klassen, die für mobile Geräte relevant sind. Des weiteren gibt es noch die Connected Limited Devices Configuration (CLDC), die für noch schwachbrüstigere Geräte, wie z.B. normale Handys, spezifiziert ist. Für diese Konfigurationen gibt es verschiedene Profile, die Zusatzklassen enthalten, wenn noch weitere gebraucht werden. PBP ist das Basisprofil, also die Grundkonfiguration, darauf aufbauend gibt es weitere, wie z.B. GUI Profile u.Ä. Diese Konfigurationen wurden bei ihrer Einführung MIDP genannt.

HSP ist eine OpenSource Implementierung des Mobile Home Plattform (MHP) Standards vom Institut für Nachrichtentechnik der TU-Braunschweig.

Daher werden die Implementationen in dieser Arbeit auf der Esmertec JVM getestet und auch hinsichtlich ihres Speicherverbrauches untersucht. Ausserdem interessiert natürlich die allgemeine Performance der Implementierungen, also ihre Effizienz hinsichtlich CPU intensiver Berechnungen sowie ihrer I/O Performance. Und es interes-

siert auch die Ladezeit des Interpreters selber, also wie lange es dauert, bis dieser mit der Interpretierung eines Skriptes anfängt, denn das trägt zur allgemeinen Performance bei. Skriptsprachen weisen ihre Stärken im Umgang mit Strings, sowie in der Einfachheit der Benutzung von "hashes" (assoziativen Arrays) auf. Daher werden sie häufig für das sogenannte "text processing" eingesetzt.

Als Konsequenz wurden für diese Studienarbeit fünf Skripte geschrieben, die die oben beschriebenen Eigenschaften der Interpreter Implementationen testen, dies sind:

1. *Ladazeit ("Startup time")*: Empty,
2. *CPU Performance*: N-body,
3. *I/O Performance*: IO,
4. *Hash Performance*: Hash, und
5. *String Performance*: String

Natürlich kann dieser Vergleich der Interpreter Implementationen untereinander kaum einen Hinweis auf die Performance derselben Interpreter auf anderen Maschinen oder sogar nur mit anderen Skripten geben. Nach [10] sind Skriptsprachen Interpreter in modernen Betriebsystemen so vielen Unwägbarkeiten unterworfen, z.B. *Memory Hierarchy* (caching), JIT (da hier Java betrachtet wird) und Parallel laufende Prozesse - um nur einige zu nennen, dass es unmöglich ist, aus einem Experiment das Resultat eines Anderen vorherzusagen. In dem Paper wurden die folgenden Gebiete der Programmierung von Skriptsprachen betrachtetet und zu jedem mehrere Skripte geschrieben:

1. *Grundlegende Funktionen*: mathematische Berechnungen und Funktionsaufrufe;
2. *Arrays und Strings*: indiziert, assoziativ und String Manipulationen;
3. *I/O*: Dateien kopieren und umkehren, Wörter zählen und Summe bilde; sowie
4. *Grafische Oberflächen*: Knöpfe und Linien in einem Rad.

Es wird kein Anspruch auf Vollständigkeit oder Repräsentativität der Auswahl von Skripten erhoben, die Autoren glauben jedoch, dass ihre Wahl wenigstens fair für

alle Sprachen ist. Daher wurde eine ähnliche Auswahl an Tests auch für diese Studienarbeit verwendet - bis auf die GUI Aufgaben, denn die werden in der HSP aller Wahrscheinlichkeit nach in einer Beschreibungssprache, wie MPEG LaSER [11] implementiert werden.

Da mit der Esmertec JVM auf dem PDA keine Ermittlung des Speicherverbrauches möglich war, der Speicherverbrauch auf einem Desktop-PC jedoch kaum Korrelation mit dem auf einem PDA aufweist, wird dem auf dem Desktop-PC ermittelten Wert hier keine große Bedeutung beigemessen. Interessanter sind jedoch die Zeiten der Messungen. Dabei sind verständlicherweise die Interpreter besser, die kürzere Ausführungszeiten aufweisen. Es wurde darauf geachtet, dass die Skripte annähernd linear in n sind, denn sie wurden mit wachsendem n ausgeführt und danach geplottet. Da die Skripte eine lineare Komplexität aufweisen, sollte die Ausführungszeit linear mit n wachsen. Tut sie das nicht, lohnt ein genauerer Blick, denn das könnte einen Performance-Engpass in dieser Disziplin und sich daraus ergebende Probleme bedeuten.

Die Evaluation von verschiedenen Skriptsprachen-Interpreterimplementation für die JVM in dieser Studienarbeit konzentriert sich also sehr stark auf die Performance.

Dabei gäbe es natürlich weitere Eigenschaften von Skriptsprachen sowie von ihren Implementationen, die sich messen und vergleichen ließen, aus denen sich Schlüsse über ihre Verwendbarkeit für die HSP ziehen ließen und es daher von Interesse wäre, sie zu betrachten. Dazu würden unter Anderem die Ausdrucksstärke der Skriptsprachen - wie vielen C Befehlen ein Befehl in dieser Sprache etwa entspricht), wie schnell sich in einer Sprache entwickeln lässt - die Arbeitsgeschwindigkeit einer Sprache, ihre Erweiterungsfähigkeit - und vieles mehr, zählen.

6.2. Kriterien

Die in dieser Arbeit ausgesprochene Empfehlung für die Auswahl einer Skriptsprachen-Implementation für die HSP stützt sich neben der Performance Messung auf die folgenden Kriterien:

1. *Standardbibliothek*: eine große und umfangreiche Standardbibliothek,
2. *Community*: eine aktive Community,

3. *Syntax*: eine relativ einfach zu erlernende Syntax,

Die bei einer Implementation mitgelieferte Standardbibliothek enthält Code für die verschiedensten Aufgaben, wie z.B. HTTP Zugriffe und dergleichen. Bis auf Python bringt keine der hier getesteten Sprachen eine eigene Standardbibliothek mit, da sich alle auf die in Java schon vorhandenen Funktionen verlassen.

Eine große und aktive Community, sowie die Tatsache, dass eine Sprache benutzt wird, sorgt für das Vorhandensein von viel Beispielcode im Internet. Das erleichtert die Entwicklungsarbeit, falls einmal Probleme auftreten sollten.

Eine Syntax, die nahe an Java ist, erleichtert das Erlernen für Entwickler, die schon Java beherrschen. Eine einfache Syntax enthält keine Inkonsistenzen und ist auch für relativ unerfahrene Entwickler schnell beherrschbar.

6.3. Performance

In diesem Teil werden die Ergebnisse der Performance Messungen widergegeben, die mit insgesamt acht Implementierungen durchgeführt wurden. Dies waren:

1. *Java*: “plain” Java ohne einen Skriptsprachen Interpreter,
2. *Rhino/js*: JavaScript,
3. *DJava*: DynamicJava,
4. *Iava*
5. *bsh*: BeanShell,
6. *Pnuts*
7. *Jython*: Python,
8. *JavaFX*: JavaFX Script.

6.3.1. Umgebung

Um einen Eindruck von der Performance direkt auf dem PDA mit der HSP zu gewinnen, wurden die Skripte, allerdings ohne die genaue Zeitmessung des ThreadMXBeans, auf ebenjener mit der Esmertec JVM ausgeführt und die Zeiten gemessen. Dazu

wurden die jars und die Skripte auf den PDA übertragen und direkt dort mit Hilfe einer Windows Batch Datei ausgeführt.

Des Weiteren wurden jeweils die Ausführungszeiten und der maximale, sowie endgültige RAM Verbrauch mit Suns J2SE JVM 1.6 in Netbeans 5.5 gemessen. Das geht mit JVM Toolkit Interface (JVMTI) in Java 1.6 und dem zu Netbeans gehörenden *Profiling Agent* sehr einfach durch Angabe eines Agent mit:

```
java -agentpath:"D:/progs/NB5.5/profiler/lib/deployed/jdk15/windows/
profilerinterface.dll"="\"D:/progs/NB5.5/profiler/lib/"",5140.
```

Allerdings ist dabei zu beachten, dass die gesammelten und in Netbeans angezeigten Zeiten *inklusiv* sind, das heißt sie geben die gesamte in einer Funktion verbrachte Zeit an, zuzüglich die aller aufgerufenen Methoden. Das spielt allerdings keine Rolle, da ich hier nur die gesamte Laufzeit betrachte. Zu beachten ist jedoch, dass Netbeans dabei die Zeiten aller Threads zusammenzählt, also auch die der anderen Interpreter Threads sowie der Garbage Collection.

Um das einmal aus einem anderen Blickwinkel zu betrachten, wird in einem 2. Versuch noch einmal mit den Java Bordmitteln nur die CPU Zeit des aktuellen Threads und die gesamte verstrichene Systemzeit gemessen. Dazu gibt es das `ThreadMXBean` in dem Paket `java.lang.management`, das eine Funktion `getCurrentThreadCPUTime()` anbietet, allerdings nicht in der CDC. Die verstrichene Systemzeit lässt sich mit der Funktion `System.currentTimeMillis()` ermitteln, die es auch in der CDC gibt. Durch das Verhältnis dieser beiden Zeiten lässt sich die CPU Nutzung eine Programms erahnen, da die CPU Zeit eines Programms nicht weiterläuft wenn es z.B. auf ein I/O Ereignis wartet, wohl aber die Systemzeit. Manchmal wird die CPU Zeit allerdings durch die Nutzung mehrerer Kerne sowie durch Messungenauigkeiten auf dem Desktop PC verfälscht, was zu Auslastungen über 100% führen kann. Dass die bei diesem zweiten Versuch gemessenen Zeiten weit unter denen des 1. Versuchs liegen ist nur durch die Nutzung mehrerer Kerne sowie durch Optimierungen des JIT erklären.

Diese Methode wurde sowohl zur Messung in Suns JVM, IBM Webspheres J9 als auch der Esmertec JVM auf dem PDA verwendet. Die J9 JVM erwartet in ihrer Kommandozeile beim Aufruf eine Angabe, welches Profil benutzt werden soll, sowie den Pfad zu den Java Standard Klassen, die beim Booten geladen werden sollen. Die Kommandozeile sah dann so aus:

```
j92 -jcl:foun -Xbootclasspath/a:"C:/Programme/IBM/ive-2.1/lib/
jclFoundation/classes.zip".
```

Als Vergleich wurden bei allen Versuchen auch die Zeiten bei der Ausführung mit der Standard J2SE (also in "normalem" Java) gemessen.

6.3.2. Empty: Startup time

Dieses Skript mit dem Namen "empty" ist eine leere Datei in jeder Sprache. Es misst die Startzeit der Interpreter Umgebung, sowie den initialen RAM Verbrauch.

Jedes Skript wurde drei mal ausgeführt, und die Werte nur in der Tabellen notiert, falls sie stark von den anderen abwichen. Die Tabellen enthalten die Ausführungszeiten für das die Skripte in allen fünf Sprachen, die in CDC laufen, sowie den maximalen und endgültigen Speicherverbrauch. Es wurde mit Suns JVM Version 1.6 ausgeführt und mit Hilfe von JVMTI und Netbeans gemessen. Da es dabei keinen Sinn macht, die Zeit mit den Java Bordmittel zu messen, da ab dann die Interpretation des Skriptes schon angefangen hat, hier kein zweiter Versuch.

JVMTI war nicht in der Lage, den Speicherverbrauch der JVM zu ermitteln, wenn kein Interpreter gestartet wurden, daher das Fragezeichen an dieser Stelle.

Tabelle 6.1.: Empty Skript Performance - Desktop Sun 1.6 JVMTI

Name	runtime/ms	Max Mem/KB	Mem Ende/KB
java	0,554/0,551	?	?
Rhino/js	0,605	2.776	1.471
DJava	0,912/0,750	3.621	1.670
Iava	36/0,595	1.802	1.609
bsh	51,5/55,9/0,65	2.800	1.700
Pnuts	245/205	3.825	3.825
Jython	1.080/952	4.846	4.204
JavaFX	3.505/2.252/423	3.699/4.307	2.308/4.307

6.3.3. N-body: CPU lastig

Dieses Skript ist eine doppeltgenaue N-body (Sternenkörper-) Simulation und stammt von [28]. Einige der Skripte mussten etwas angepasst werden, z.B. für Pnuts und für JavaFX. n gibt jeweils die Anzahl der Iterationen an, die das Skript berechnen musste. Die Quelltexte sind in Appendix A zu finden.

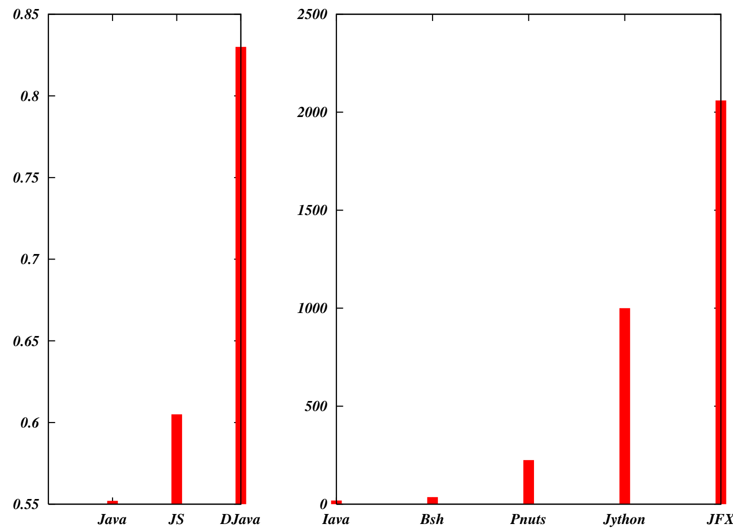

Abbildung 6.1.: Durchschnittliche Performance der Interpreter für das Empty Skript in Java 1.6 (Desktop) mit JVMTI

Die folgenden Daten stammen vom 2. Versuch. Wenn die Auslastung über 100% lag, wurde die Zahl der Iterationen erhöht, bis sich das änderte. Dass die Auslastung sinkt, könnte allerdings auch nur daran liegen, dass durch die längere Ausführungszeit der CPU Verbrauch anderer gleichzeitig laufender Programme stärker ins Gewicht fällt. Die Versuche mit Suns und J9s JVM wurden auf einem Desktop PC ausgefüht, die mit der Esmertec direkt auf dem PDA - daher auch die niedrigeren Werte für n. Jython verträgt sich nicht mit der Esmertec JVM, und Pnuts und JavaFX waren nicht vielversprechend genug, um sie noch auf dem PDA zu testen. Dort wurden nur die drei Besten aufeinander losgelassen - Rhino, DJava und BeanShell.

Rhino/JS erzeugt bei der Ausführung mit J9 und $n > 900$ eine `IllegalStateException`, deshalb hier nur der Wert für $n = 900$.

Dieses Skript wurde nicht in Iava implementiert, da das Fehlen von Klassen in dieser Sprache einen zu hohen Aufwand nach sich gezogen hätte.

Dabei sei angemerkt, dass Pnuts nur 100 Iterationen berechnen musste, während alle anderen jeweils 2.000 berechneten. Ausserdem überleben bei den 4 anderen Implementationen am Ende immer nur weniger als 20 Objekte während es bei Pnuts nach 100 Iterationen schon fast 400 sind. Des Weiteren musste die VM für Pnuts die Cache Größe von am Anfang 5 Megabyte auf am Ende mehr als 7 MB anpassen. Mit knapp 43% der gesamten Rechenzeit ist `math.MutableInteger.divide` der

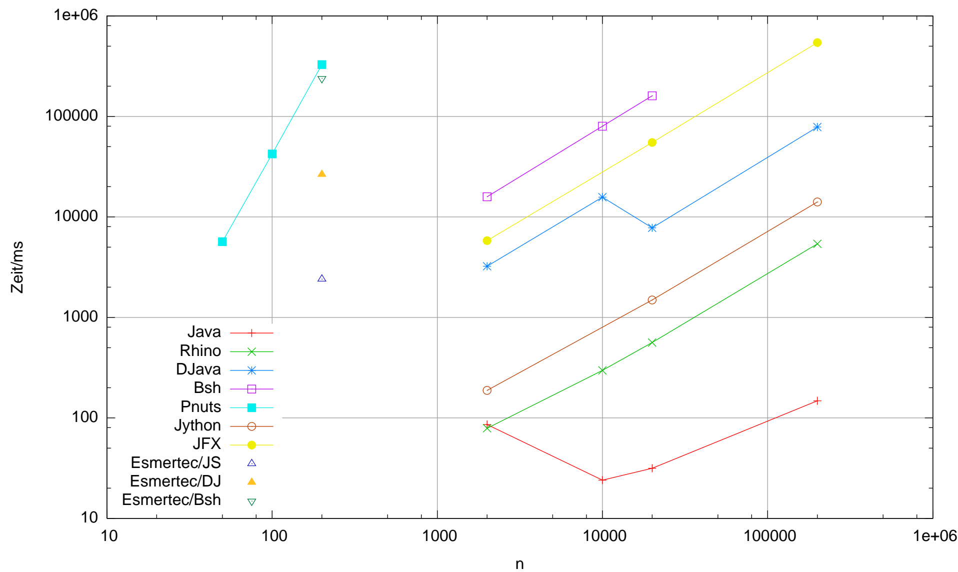

Abbildung 6.2.: Performance der Interpreter für das N-body Skript in Java 1.6 (Desktop) und Esmertec

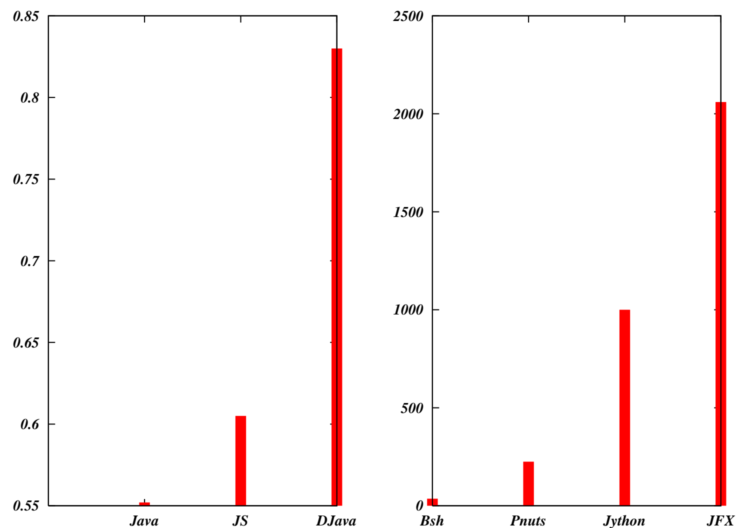

Abbildung 6.3.: Durchschnittliche Performance der Interpreter für das N-body Skript in Java 1.6 (Desktop) mit JVMTI

größte Hotspot in diesem Skript. Da alle anderen Interpreter die meiste Zeit beim Durchlaufen der Arrays verbrachten (in `HashMap.interator`) liegt die Vermutung nahe, dass Pnuts nicht gut mit großen Zahlen umgehen kann.

JavaFX hingegen hat den Hotspot merkwürdigerweise in `sun.awt.windows.WToolkit.run` obwohl das Skript gar keine Fenster erzeugt. Vielleicht liegt das aber auch nur daran, dass in dieser Funktion die main program loop des Skriptes ausgeführt wird.

Ausserdem scheint es bei einigen Sprachen bei mehrmaligen, hintereinander liegenden Versuchen zu Cache Effekten oder zu Optimierungen der JIT zu kommen, was die kürzeren Ausführungszeiten bei Folgeversuchen erklären würde.

6.3.4. IO: IO lastig

Dieses Script schreibt in einer kurzen Schleife den aktuellen Wert der Schleifenvariablen als String in eine Datei. Die Idee dazu stammt von [20]. Quelltexte wiederum in Appendix A. In der Tabelle sind Ausführungszeiten, sowie RAM Verbrauch und die Anzahl der Iterationen vermerkt.

Es ist deutlich ersichtlich, dass bei diesem Test die Unterschiede weit weniger groß ausfallen, von JavaFX einmal abgesehen. Hierbei hängt die Performance offensichtlich vom Filesystem ab, und nicht so stark von den einzelnen Interpretern, die die Befehle sowieso nur an Java weiterreichen.

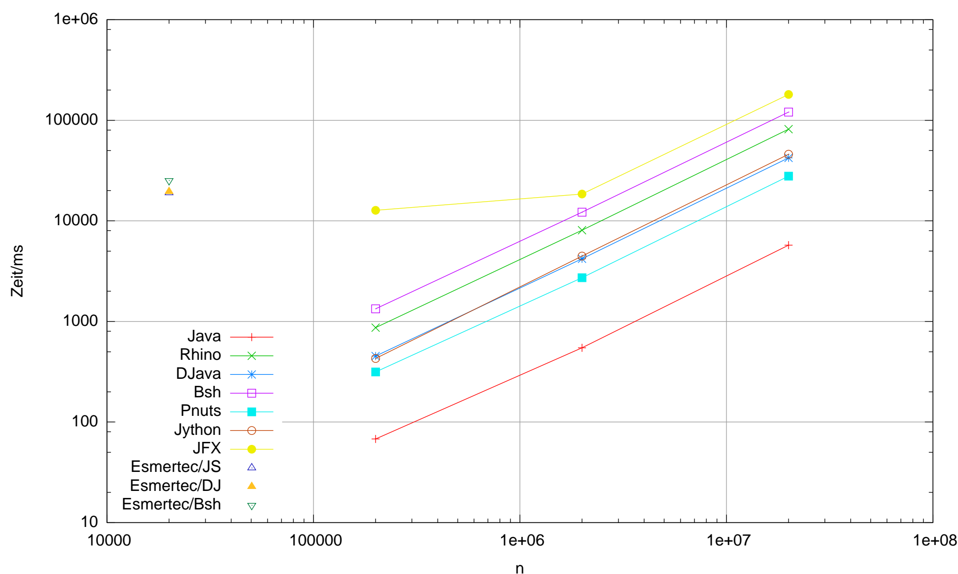

Abbildung 6.4.: Performance der Interpreter für das IO Skript in Java 1.6 (Desktop) und Esmertec (PDA)

Jython benutzt in seiner eigenen Implementation der File-I/O Bibliotheken von

Python die Funktion `String.getBytes`, die in CDC nicht vorhanden ist, daher ist kein zweiter Versuch mit Jython an dieser Stelle möglich.

Tabelle 6.2.: IO Skript Performance in Java 1.6 (Desktop) und JVMTI

Name	n	runtime/ms	Max Mem/KB	Mem Ende/KB
java	200.000	0,537	?	?
Rhino/js	200.000	5.836/6.073	4.085	3.223/3.569
DJava	200.000	0,663	3.912/3.731	3.767
Iava	200.000	0,651	3.902	3.443
bsh	200.000	0,743/9.489	4.097	3.308
Pnuts	200.000	2.431/2.625	4.207	3.977
Jython	200.000	3.062/2.724	4.378/3.646	3.370
JavaFX	200.000	69.430/68.971	4.336	2.886/3.028

6.3.5. Hash: Hash performance

Hashes, auch bekannt als assoziative Arrays, sind von sehr großer Bedeutung in einer Skriptsprache, weil sie sehr häufig Verwendung finden. Daher ist die Hash Performance, die in diesem Test gemessen wird, sehr wichtig für die Gesamtperformance einer Sprache.

Dieses Skript erzeugt in zwei ineinander verschachtelten Schleifen zwei Zahlen und fügt diese in ein Hash ein, die erste als *Schlüssel*, die zweite als zugehöriger *Wert*. Dann wird der gerade eingefügte Wert mit Hilfe des Schlüssels wieder abgerufen. Dabei läuft die äußere Schleife von 0 bis 10000, die Innere von 0 bis n. Die Idee stammt wieder von [20].

Der Code dazu in Appendix A.

Bei diesem Skript ist anzumerken, dass nur Jython und Javascript native Implementationen für Hashes aufweisen, die in diesem Skript benutzt werden konnten. Die anderen Sprachen unterstützen natürlich auch Arrays, allerdings keine dynamischen, die für diesen Versuch nötig wären. Also mussten die anderen Interpreter mit der Klasse `java.utils.HashMap` von Java auskommen, die verständlicherweise etwas langsamer ist.

Der erste Test Durchlauf, der im Folgenden widergegeben ist, wurde noch mit einer älteren Version des Skriptes durchgeführt, die quadratische Komplexität aufweist.

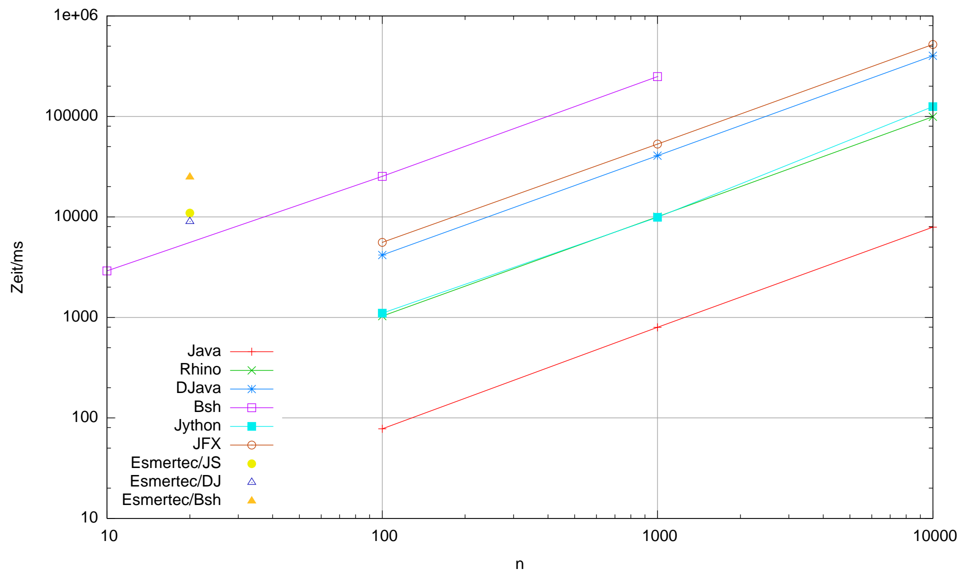

Abbildung 6.5.: Performance der Interpreter für das Hash Skript in Java 1.6 (Desktop) und Esmertec (PDA)

Tabelle 6.3.: Hash Skript Performance Java 1.6 (Desktop) und JVMTI - quadratische Komplexität

Name	n	runtime/ms	Max Mem/KB	Mem Ende/KB
java	100	0,571	?	?
Rhino/JS	100	2.800/3.479/2.939	3.966	3.532
DJava	100	0,775	4.067	3.856
Iava	100	49.790/52.798	4.980	4.358
bsh	100	21.016	4.133	3.294
Pnuts	100	3.102/2.905	4.121	3.472
Jython	100	2.000/1.821	4.792	4.099
JavaFX	100	63.127/64.100	3.893/4.416	3.066/2.844

6.3.6. String: String Performance

Text Processing, dessen Performance mit Hilfe dieses Skriptes gemessen wird, ist wahrscheinlich die zweitwichtigste Aufgabe einer jeden Skriptsprache. Besonders Logfiles und dergleichen werden sehr häufig von (Text processing-) Skriptsprachen analysiert. Daher diese Performance Messung.

In diesem Skript wird ein String in zwei ineinander verschachtelten Schleifen aufgebaut, die die laufende Schleifennummer abwechselnd als ASCII Charakter und als Nummer an den String anhängen. Wenn die beiden Variablen zusammen eine gerade Zahl ergeben dann wird zuerst der ASCII Charakter angehängt und dann die Zahl, sonst umgekehrt. Beim ersten Versuch liefen beide Schleifen bis 2^n, also exponentielle Komplexität, daher wurden sie für den zweiten Versuch auf $10 * n$ und 12 angepasst um linear in n zu sein. Am Ende wird noch das Pattern “a1” mit Hilfe einer `Regular Expression` in dem String gesucht um auch diese Performance zu berücksichtigen - allerdings nicht in der PDA Version, da in der CDC die Funktion `String.matches()` nicht existiert. Der Code dazu wieder in Appendix A.

JavaFX war nicht dazu zu überreden, eine Integer Zahl in einen Character umzuwandeln und an einen String anzuhängen, daher fehlt JavaFX in diesem Test.

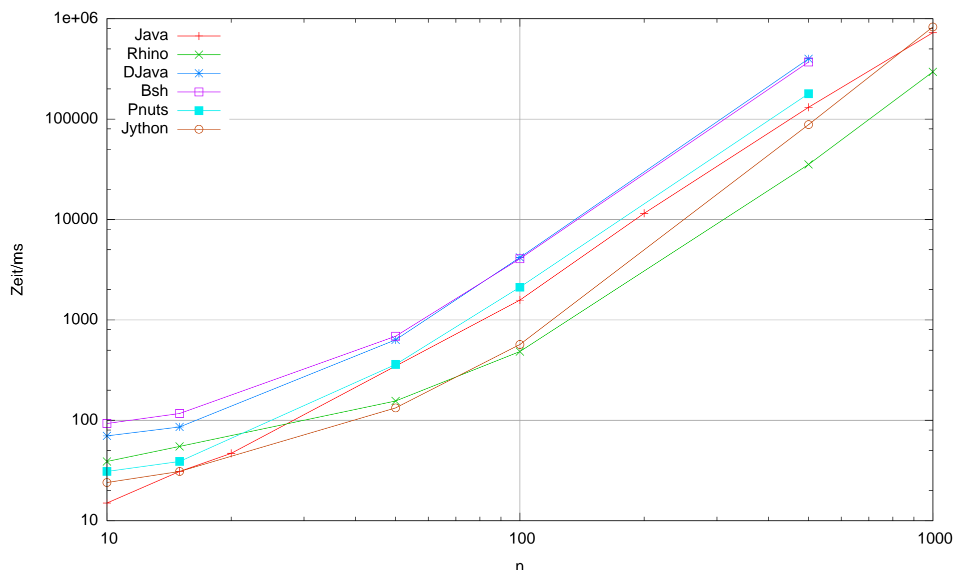

Abbildung 6.6.: Performance der Interpreter für das String Skript in Java 1.6 (Desktop)

Die erste Version des Skriptes wieß unglücklicherweise noch exponentielle Komplexität auf, daher sind hier nur die Werte in der folgenden Tabelle widergegeben.

Tabelle 6.4.: Hash Skript Performance Java 1.6 (Desktop) und JVMTI - exponentielle Komplexität

Name	n	runtime/ms	Max Mem/KB	Mem Ende/KB
java	6	0,626/0,586/0,561	?	?
Rhino/JS	6	327/305/395	3.237/3.836	3.237/1.767
DJava	6	0,753/0,654	3.942	3.942
Iava	6	2.948/0,661	4.128/4.025	3.415/4.025
bsh	6	861	3.988/3.831	2.273/3.831
Pnuts	6	445	3.137	3.076
Jython	6	1.155	4.342/4.216	4.060/4.216

6.4. Auswertung

DynamicJava scheint insgesamt eine sehr schnelle Interpretation zu gelingen mit einem arithmetischen Durchschnitt von 0,754ms, wohingegen JavaFX Script mit durchschnittlich 80.646ms insgesamt am Schlechtesten abschneidet. Damit liegt DJava sogar noch vor normalen Java mit 11,676ms, das sich durch Ausreisser bei N-Body (86ms und 26ms), wohl wegen des nicht optimalen Umgangs mit großen Zahlen, seinen Durchschnitt verhagelt.

DJava liegt bei allen Tests mit Java gleich auf oder ist nur geringfügig langsamer, nur bei N-Body ist Djava im Durschnitt sogar schneller, als Java. Java erkämpft sich dabei von drei Versuchen nur eine Zeit unter 1 Millisekunde, DJava schafft dieses hingegen drei mal. Es wäre durchaus interessant, diesem Phänomen nachzugehen, also den Quellcode von DJava zu betrachten, und Java und DJava nebeneinander zu profilieren. Dazu hat in dieser Studienarbeit allerdings die Zeit gefehlt.

Dahinter liegen Jython mit 1.947ms, Rhino/JS mit 1.948ms und bsh mit 7.886ms. JavaFX kann man jedoch zu Gute halten, dass es noch eine erste Beta Version ist, und daher wahrscheinlich noch an Performance zulegt (man erinnere sich nur an die Performance der ersten Java Implementationen von Sun - ohne HotSpot).

7. Integration in die HSP

7.1. TestScriptXlet

Für die Implementierung auf der HSP wurde zuerst ein vorhandenes Beispiel Xlet, das in Java geschrieben ist, und als `.class` Datei vorliegen muss, in JavaScript neugeschrieben und mit dem beiliegenden JSC (JavaScript Compiler) kompiliert. Dieses kann dann innerhalb der HSP ausgeführt werden, und verhält sich, wie das in Java geschriebene. Die beiden Quelltexte dazu sind in Appendix A zu finden.

Das Xlet Skript muss mit JSC folgendermaßen kompiliert werden:

```
jsc -cp ".;js.jar"org.mozilla.javascript.tool.jsc.Main -implements
xjavax.tv.xlet.Xlet TestScriptXlet.js.
```

Der JSC braucht eine Angabe darüber, dass er eine Klasse erstellen soll, die das Interface `Xlet` implementiert, daher das -implements switch. Ein anderes Objekt, das dann das Interface `KeyListener` implementiert landet in der zweiten Klasse. Daraus entstehen zwei `class` Dateien (TestScriptXlet.class und TestScriptXlet1.class), die beide in dem /bin Verzeichnis der Xlets vorhanden sein müssen. Nun muss dem Classpath des xletviewer noch “js.jar” hinzugefügt werden, das Xlet als `TestScriptXlet` in die Applications.xml eingetragen, und die HSP gestartet werden.

7.2. JSShellXlet

Als nächster Versuch wurde ein Xlet in JavaScript geschrieben und in eine .class Datei kompiliert, das in der Lage ist, JavaScript Code aus einer Datei zu lesen und mit Hilfe von Rhino zu interpretieren. Die Standardausgabe des Programms wird ausgelesen und in der Shell der HSP ausgegeben. Die `HScene`, also die Haupt-Graphikplane der HSP, wird als Variable “S” in dem Kontext des interpretierten Skripts registriert. Dadurch erhält das Skript die Möglichkeit, auf die HSP und deren Bildschirmausgabe zuzugreifen. Um das zu testen, wurde ein JavaScript Skript geschrieben, das auf der

HScene ein TU Braunschweig Logo mit zufälliger Größe und Position zeichnet. Dazu wird innerhalb des Skriptes ein Objekt “o” erstellt, das von `java.awt.Component` abgeleitet ist und nur die zum Zeichnen des Components und Setzen der zufälligen Parameter wichtigen Funktionen enthält.

Der Quelltext der beiden Skripte befindet sich wie immer in Appendix A.

8. Fazit

Wenn es allein auf Geschwindigkeit ankommt, dann ist nach den Performance Messungen DynamicJava eindeutig die beste Wahl, mit einer Java ebenbürtigen Ausführungsgeschwindigkeit und teilweise sogar besseren Zeiten als Java! Dabei sollte man natürlich beachten, dass Benchmarks immer nur "Laborwerte" darstellen, also nie echte Anwendungssituationen widerspiegeln können.

DynamicJava bietet allerdings ähnlich wie BeanShell nur die Möglichkeit, Standard Java Syntax mit Skriptingzusätzen - also lose typisierte Variablen - anzureichern. Es bietet im Gegensatz zu der Python und JavaScript Syntax keine eigenen *Language Features*, also Sprachelemente, die Java nicht bietet. Andererseits ist es durch die Java-ähnliche Syntax in der Lage, Java-Code direkt zu benutzen, also Skript- und Java-Syntax zu mischen. Das versetzt sie in die Lage, auf die große Codebasis von Java Programmen zurückzugreifen. Auf der anderen Seite können auch Python und JavaScript Java-Code mit nur geringen Veränderungen verwenden, und gleichzeitig auf Code aus ihrer jeweiligen Community zugreifen, welcher wiederum DJava und Bsh nicht zur Verfügung steht.

Da es aber den Kriterien zufolge auf den Gesamteindruck ankommt - also einen gesunden Mix aus Geschwindigkeit, Syntax, Anwenderfreundlichkeit und Standard-Bibliothek Umfang - würde die nächste Wahl auf Jython fallen, wenn es denn in Esmertecs JVM laufen würde. Es läuft sowohl in Suns CDC Emulator, als auch mit der J9 JVM zusammen, nur die Esmertec JVM meldet einen `UnsupportedClassVersionError`.

Direkt nach Python kommt Rhino/JavaScript, welches Dank großer Verbreitung der Sprache in jedem Browser über eine sehr große Codebasis sowie eine weithin bekannte Syntax verfügt. Darüber hinaus platziert sich Rhino nur knapp hinter Jython auf einen guten dritten Platz und bietet in etwa den gleichen gesunden Mix aus den drei genannten Kriterien wie Python. Ausserdem weist JavaScript native Syntax für Regular Expressions und Hashes auf, welches von den anderen hier getesteten Sprachen nur noch für Python (Jython), Perl (Sleep) und Ruby (JRuby) gilt. Dies erspart das umständliche Benutzen der Java Objekte `HashMap` statt assoziativen

Arrays und `Pattern` statt Regular Expressions.

Als Drittes wäre noch BeanShell zu empfehlen, welches auf Grund der Verschmelzung von Standard Java und "loosely typed" Java Syntax nicht minder einfach zu programmieren ist als normales Java. Ausserdem stehen die Chancen für Beanshell nicht schlecht, über die JSR-274 an einem Punkt in der Zukunft in die J2SE aufgenommen zu werden, falls nichts dazwischenkommt (z.B. JavaFX Script). BeanShell verfügt dank der Verwendung in jEdit, NetBeans und JMeter sowie EclipseShell und OpenOffice über eine Community und daher auch einigen Beispielcode.

9. Ausblick

Skriptsprachen werden immer wichtiger [3, 8] und immer mehr auch große Softwareprojekte (oder Teile davon) werden zunehmend in Skriptsprachen umgesetzt [16]. Dieses wird noch durch den Trend verstärkt, Applikationslogik vom Client auf einen Server zu verlagern, und diese dann nur noch per Browser-Interface zugänglich zu machen, was die Mobilität und Konsolidierung von Software erleichtert. Da das Internet traditionell eine Domäne der Skriptsprachen war, werden diese sogenannten "Webservices" zum großen Teil in einer solchen umgesetzt.

Darüber hinaus werden sich in Zukunft weitere Sprachenspezifizierungs-Gremien dazu durchringen müssen, für ihre Hochsprachen Skriptsprachen-Zusätze oder zumindest Schnittstellen zu definieren und damit Suns Beispiel zu folgen. Denn mit Groovy, das in dem JSR-241 beschrieben wird, BeanShell, das in JSR-274 weitergeführt wird, und JavaFX Script, sowie den beiden Scripting Zusätzen zum Kern der Java Sprache – dem Scripting-API JSR-223 und dem JSR-292, das einen neuen Bytecode für "loosely typed" Variablen vorschlägt – bestehen schon fünf Ansätze, Java skriptingfreundlicher werden zu lassen.

Skriptsprachen werden den Ruf der "quick&dirty" oder "proof of concept" (POC) Sprachen langsam verlieren und sich aus der Ecke der Logfile-Auswertung und Hacker-Skripte aufmachen, die *Hochsprachen* – wie C und C++ – von ihrem Thron der "general-purpose" Programmierung in performance-abhängige Nischen, wie Treiber, Spiele, grafikintensive Programme und der Gleichen zurückzudrängen. Allerdings ist die Möglichkeit des "Sketchens" von Skriptsprachen, also des schnellen Ausprobierens neuer Gedanken in Code einer der Vorteile, die Skriptsprachen dazu verhelfen wird, auch noch in Zukunft an vorderster Front eingesetzt zu werden. Skriptsprachen werden auch vielfach von Leuten eingesetzt, die neue Methoden der Programmierung oder Entwicklung erfinden (wie z.B. *Extreme Programming (XP)* oder *Aspect Oriented Programming (AOP)*), da diese häufig die Mentalität besitzen, nicht mit dem Strom zu schwimmen und sich daher nicht-Mainstream ("Skript"-) Sprachen aussuchen [2].

Einhergehend damit wird auch die Problematik der Sicherheit von in Skriptsprachen

geschriebenem Code an Bedeutung gewinnen, was bedeutet, dass sich Programmierer über kurz oder lang darüber Gedanken werden machen müssen, wie dem abzuhelfen sein wird. Das wird naturgemäß einen Kompromiss zwischen Sicherheit auf der einen, und Programmierfreundlichkeit auf der anderen Seite darstellen. Relative Programmier-unfreundlichkeit ist allerdings nicht gleichbedeutend mit gleichmäßig hohem Sicherheitsstandard, wie sich an jetziger in C/C++ geschriebener Software und Mailinglisten wie Bugtraq ablesen lässt.

Die Zukunft der Programmiersprachen wird zweifellos denjenigen Sprachen gehören, die es schaffen, starke Typisierung mit schwacher ("soft typing") zu verbinden (wie BeanShell), sowie gleichzeitig *type inferencing* in einigen Teilen und explizites Casten in anderen Teilen zu verwenden. Diese Sprachen sind dann in der Lage, performance-kritische Teile des Projektes zu "kompilieren", d.h die dort im Quelltext vorhandenen zusätzlichen Informationen in Optimierungen und Verifizierung der Korrektheit umzusetzen, sowie andere Teile "nur" zu interpretieren - also dem Entwickler alle Vorteile der dynamischen Programmierung an die Hand zu geben. Es ist sicherlich richtig, dass Unit Tests in einer dynamischen Sprache schneller geschrieben werden können, daher würden auch die performance-kritischen Teile eines Projektes davon profitieren.

Allerdings ist die Korrektheit im Falle von Skripten schwieriger formal zu verifizieren, da weniger Informationen vorhanden sind. Die Korrektheit ist vor allem für große Projekte wichtig. Verifizierung der formalen Korrektheit kann jedoch niemals die (Unit-) Tests ersetzen, die von den Programmierern geschrieben werden, die Software unter Berücksichtigung der späteren Verwendung auf ihre Funktion hin zu testen. Verifizierung kann nur die korrekte Verwendung eines vom Programmierer angegebenen Typs checken. Nur beides zusammen gibt die höchste Garantie dadrauf, dass die Software später im Einsatz funktionieren wird, obwohl 100%-tige Sicherheit nie gegeben sein kann.

Manche vertreten die Ansicht, JavaScript würde die wichtigste dynamische Sprache des nächsten Jahrzehnts werden [23], da jeder, der auch nur etwas Dynamik in seine Webseiten integrieren will, um JavaScript nicht herumkommt, und damit die Infrastruktur für JavaScript geschaffen ist. Darüber hinaus bietet JavaScript auch noch eingebaute Syntax für Regular Expressions (wie Perl) und einen sehr einfachen und entspannten Umgang mit Objekt-orientierter Programmierung. Und als letztes wird JavaScript von Ecma standarisiert, was einen Vorteil hinsichtlich Zuverlässigkeit und Seriosität darstellt. Es fehlt nur noch an Kommandozeilen Interpretern und dass diese mit den Betriebssystemen mitgeliefert werden. Auf der anderen Seite stehen

einige, die die Ansicht vertreten [2][1], dass JavaScript überstandarisiert (dank Ecma und des Browserkrieges zwischen Microsoft und Netscpae), unnötig Sicherheitsorientiert (dank sei den Browsern und dem unsicheren Internet) und unintuitiv im Umgang mit Fehlern sei - denn es ist für eine Websprache besser, im Angesicht von Fehler stillschweigend zu versagen. Daher entwickelt sich JavaScript praktisch nicht mehr weiter, ist mit anderen Worten also so gut wie tot.

[1]Diese Ansicht ist allerdings nicht ganz unparteiisch, da Activestate stark in Perl involviert ist.

Literaturverzeichnis

[1] APL: *http://en.wikipedia.org/wiki/APL_(programming_language)*. – Valid on 7.6.2007

[2] Ascher, David: Dynamic Languages — ready for the next challenges, by design. (2004). – URL http://www.activestate.com/Company/NewsRoom/whitepapers.plex

[3] DevSource - Scripting's Future: *http://www.devsource.com/article2/0,1895,1778141,00.asp*. – Valid on 22.8.2007

[4] Ecma: *http://www.ecma-international.org/*. – Valid on 27.6.2007

[5] Ecma-262: *http://www.ecma-international.org/publications/standards/Ecma-262.htm*. – Valid on 27.6.2007

[6] esmertec - wireless software solutions for mass market mobile phones and embedded devices: *http://www.esmertec.com/*. – Valid on 18.7.2007

[7] Felleisen, Matthias: On the Expressive Power of Programming Languages. In: Jones, Neil D. (Hrsg.): *ESOP '90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings* Bd. 432. New York, N.Y. : Springer-Verlag, Mai 1990, S. 134–151. – URL citeseer.ist.psu.edu/felleisen90expressive.html. – ISBN 3-540-52592-0

[8] ITworld.com - Bob Martin Interview about Scripting Languages: *http://www.itworld.com/AppDev/1262/itw-0314-rcmappdevint/*. – Valid on 22.8.2007

[9] Kanavin, Alexander: An overview of scripting languages / Lappeenranta University of Technology, Finland. 2002. – Forschungsbericht

[10] Kernighan, Brian W. ; Wyk, Christopher J. V.: Timing trials, or the trials of timing: experiments with scripting and user-interface languages. In: *Softw.*

Pract. Exper. 28 (1998), Nr. 8, S. 819–843. – URL citeseer.ist.psu.edu/89435.html. – ISSN 0038-0644

[11] LASeR: *http://www.mpeg-laser.org/*. – Valid on 25.7.2007

[12] LISP: *http://en.wikipedia.org/wiki/Lisp_programming_language*. – Valid on 7.6.2007

[13] MUMPS: *http://en.wikipedia.org/wiki/MUMPS*. – Valid on 7.6.2007

[14] Ousterhout, John K.: Scripting: Higher-Level Programming for the 21st Century. In: *IEEE Computer* 31 (1998), Nr. 3, S. 23–30. – URL citeseer.ist.psu.edu/ousterhout97scripting.html

[15] Paul Graham: Painter and Hackers: *http://www.paulgraham.com/hp.html*. – Valid on 22.8.2007

[16] Perl Success Stories: *http://perl.apache.org/outstanding/success_stories/* und *http://www.oreillynet.com/pub/a/oreilly/perl/news/success_stories.html*. – Valid on 22.8.2007

[17] PHP: *http://php.net*. – Valid on 7.6.2007

[18] Prechelt, Lutz: An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl. In: *IEEE Computer* 33 (2000), Nr. 10, S. 23–29. – URL citeseer.ist.psu.edu/article/prechelt00empirical.html

[19] Python: *http://python.org*. – Valid on 7.6.2007

[20] Python vs. Perl vs. Java vs. C++ runtimes: *http://furryland.org/~mikec/bench/*. – Valid on 18.6.2007

[21] ruby: *http://www.ruby-lang.org/*. – Valid on 7.6.2007

[22] Ruby on Rails: *http://rubyonrails.org/*. – Valid on 22.8.2007

[23] Scripting Languages: Into the future: *http://www.ddj.com/web-development/184407823*. – Valid on 22.8.2007

[24] sed: *http://de.wikipedia.org/wiki/Awk*. – Valid on 7.6.2007

[25] sed: *http://www.tty1.net/sed-tutorium/html/*. – Valid on 7.6.2007

[26] Smalltalk: *http://en.wikipedia.org/wiki/Smalltalk*. – Valid on 7.6.2007

[27] Tcl: *http://en.wikipedia.org/wiki/Tcl*. – Valid on 7.6.2007

[28] The Computer Language Benchmark Game: *http://shootout.alioth.debian.org/*. – Valid on 18.6.2007

[29] Using the best tool for the job: *http://www.artima.com/commentary/langtool.html*. – Valid on 22.8.2007

[30] Wiger, Ulf: Four-fold increase in productivity and quality – industrial-strength functional programming in telecom-class products. In: *Proceedings of the Third Workshop on Formal Design of Safety Critical Embedded Systems FEmSys '01*, URL citeseer.ist.psu.edu/wiger01fourfold.html, März 2001

Alle Referenzen enthalten Links zu der entsprechenden *citeseer* Webseite (http://citeseer.ist.psu.edu/), falls diese existiert. Darüber hinaus kennzeichnen Internet Links das Datum des letzten Zugriffs ("date of validity").

Abkürzungsverzeichnis

A

API Application Programming Interface

C

CDC Connected Devices Configuration

CLDC Connected Limited Devices Configuration

CGI Common Gateway Interface

CIL Common Intermediate Language

CLI Command Line Interface

CLR Common Language Runtime

D

DHTML Dynamic HTML

E

Ecma European Computer Manufacturers Association [4]

G

GUI Graphical User Interfaces

H

HSP Handheld Software Platform

HTML Hypertext Markup Language

I

IDE Integrated Development Platform

J

JDBC Java Database Connection

JIT Just-in-time Compilation

JVM Java Virtual Machine

JVMPI JVM Profiling Interface (Java 1.5)

JVMTI JVM Toolkit Interface (Java 1.6)

JSP Java Servlet Pages

JSR Java Specification Request

M

MHP Mobile Home Plattform

MIDP Mobile Information Device Profile

MSIL Microsoft Intermediate Language

P

PBP Personal Basis Profile

PHP PHP Hypertext Preprocessor

S

SQL Structured Query Language

V

VM Virtual Machine

W

WSH Windows Scripting Host

X

XML eXtensible Markup Language

XSS Cross-Site Scripting

Index

actionscript, 10
Ajax, 11
applescript, 11
application scripting, 6
Applikations-skripting Sprachen, 10
Assembler, 3
Ausdruckstärke, xi
AWK, 11

bash, 11
batch, job control, 6

coldfusion, 11

deklarative Syntax, 20
DHTML, 11
dynamically typed, 9

EcmaScript, 12
edit-compile-run-debug cycle, 13
edit-interpret-debug cycle, 13
extension, embeddable Sprachen, 11

garbage collection, 9
general purpose Sprachen, 11
general scripting languages, 7
Geschichte, 6
Glue-Sprachen, 6
GUI Skripting Sprachen, 11
Guile, 11

Hash, xi
Hochsprachen, 1, 3
HotSpot, 2
HSP, 25
hypertalk, 10

Java CDC, 25
Java-, ECMA Script, 11
JavaScript, 11
JIT, 1
JScript, 12
JSP, 11
JSR-223, 43
JSR-241, 43
JSR-274, 43
JSR-292, 43

LiveScript, 11
Lua, 11

maya embed lang, 10
MIDP, 25
Mocha, 11

Ousterhouts Dilemma, 10

Pawn, 11
PBP, 25
Perl, 7
PHP, 7, 11
Plugins, 11
proof-of-concept (POC), 43
prototype basierte Sprache, 12
Python, 7, 11

QuakeC, 10

quick&dirty, 43

rapid prototyping, 5
Ruby, 7

sed, 11
Shell-Sprachen, 6
Sigil, 10
Skriptsprachen, 1
SQL injection, 14
Squirrel, 11
Standard Bibliotheken, xii
statically typed, 9
strongly typed, 9
System-Level Sprachen, 2

text processing language, 7
text processing Sprachen, 11

UnrealScript, 10

VBA, 10
VBScript, 11
Virtuelle Maschine, 4

weak, duck typing, 9
weakly typed, 9
Web-programming Sprachen, 11
WSH, 11

XML Syntax, 24
XSLT, 11
XSS, 14
XUL, 11

A. Quelltexte

A.1. Empty Quelltext

Hier ist die leere Version des Programmes für Java, da diese noch am interessantesten ist. Alle anderen Skripte bestehen nämlich nur aus einer leeren Datei.

Quelltext A.1: Java empty

```
public class empty {
    public static void main(String[] args) {
    }
}
```

A.2. N-body Quelltext

Aufgrund der Ähnlichkeit der Quelltexte zueinander (sie unterscheiden sich nur in einigen wenigen Details der Syntax), werde ich im Folgenden nur jeweils eine Version des Programms aufführen. Im Falle von N-Body ist das Skript in Python geschrieben.

Quelltext A.2: Python N-body

```
# The Computer Language Shootout
# http://shootout.alioth.debian.org/
#
# submitted by Ian Osgood
# modified by Sokolov Yura
# modified by bearophile
import sys
#from java.lang.management import *
from java.lang import System;

#mx = ManagementFactory.getThreadMXBean()
start = System.currentTimeMillis()

pi = 3.14159265358979323
solar_mass = 4 * pi * pi
days_per_year = 365.24

class body :
    pass

def advance(bodies, dt) :
    for i in xrange(len(bodies)) :
        b = bodies[i]

        for j in xrange(i + 1, len(bodies)) :
```

```
26              b2 = bodies[j]

                dx = b.x - b2.x
                dy = b.y - b2.y
                dz = b.z - b2.z
31              distance = (dx**2 + dy**2 + dz**2)**0.5

                b_mass_x_mag = dt * b.mass / distance**3
                b2_mass_x_mag = dt * b2.mass / distance**3

36              b.vx -= dx * b2_mass_x_mag
                b.vy -= dy * b2_mass_x_mag
                b.vz -= dz * b2_mass_x_mag
                b2.vx += dx * b_mass_x_mag
                b2.vy += dy * b_mass_x_mag
41              b2.vz += dz * b_mass_x_mag

        for b in bodies :
            b.x += dt * b.vx
            b.y += dt * b.vy
46          b.z += dt * b.vz

    def energy(bodies) :
        e = 0.0
        for i in xrange(len(bodies)) :
51          b = bodies[i]
            e += 0.5 * b.mass * (b.vx**2 + b.vy**2 + b.vz**2)

            for j in xrange(i + 1, len(bodies)) :
                b2 = bodies[j]
56
                dx = b.x - b2.x
                dy = b.y - b2.y
                dz = b.z - b2.z
                distance = (dx**2 + dy**2 + dz**2)**0.5
61
                e -= (b.mass * b2.mass) / distance

        return e

66  def offset_momentum(bodies) :
        global sun
        px = py = pz = 0.0

        for b in bodies :
71          px += b.vx * b.mass
            py += b.vy * b.mass
            pz += b.vz * b.mass

        sun.vx = - px / solar_mass
76      sun.vy = - py / solar_mass
        sun.vz = - pz / solar_mass

    sun = body()
    sun.x = sun.y = sun.z = sun.vx = sun.vy = sun.vz = 0.0
81  sun.mass = solar_mass

    jupiter = body()
    jupiter.x = 4.84143144246472090e+00
    jupiter.y = -1.16032004402742839e+00
86  jupiter.z = -1.03622044471123109e-01
    jupiter.vx = 1.66007664274403694e-03 * days_per_year
    jupiter.vy = 7.69901118419740425e-03 * days_per_year
    jupiter.vz = -6.90460016972063023e-05 * days_per_year
    jupiter.mass = 9.54791938424326609e-04 * solar_mass
91
    saturn = body()
    saturn.x = 8.34336671824457987e+00
    saturn.y = 4.12479856412430479e+00
    saturn.z = -4.03523417114321381e-01
96  saturn.vx = -2.76742510726862411e-03 * days_per_year
    saturn.vy = 4.99852801234917238e-03 * days_per_year
    saturn.vz = 2.30417297573763929e-05 * days_per_year
    saturn.mass = 2.85885980666130812e-04 * solar_mass

101 uranus = body()
```

```
uranus.x = 1.28943695621391310e+01
uranus.y = -1.51111514016986312e+01
uranus.z = -2.23307578892655734e-01
uranus.vx = 2.96460137564761618e-03 * days_per_year
106 uranus.vy = 2.37847173959480950e-03 * days_per_year
uranus.vz = -2.96589568540237556e-05 * days_per_year
uranus.mass = 4.36624404335156298e-05 * solar_mass

neptune = body()
111 neptune.x = 1.53796971148509165e+01
neptune.y = -2.59193146099879641e+01
neptune.z = 1.79258772950371181e-01
neptune.vx = 2.68067772490389322e-03 * days_per_year
neptune.vy = 1.62824170038242295e-03 * days_per_year
116 neptune.vz = -9.51592254519715870e-05 * days_per_year
neptune.mass = 5.15138902046611451e-05 * solar_mass

def main() :
    try :
121         n = int(sys.argv[1])
    except :
        print "Usage: %s <N>" % sys.argv[0]

    bodies = [sun, jupiter, saturn, uranus, neptune]
126
    offset_momentum(bodies)

    print "%.9f" % energy(bodies)

131     for i in xrange(n) :
        advance(bodies, 0.01)

    print "%.9f" % energy(bodies)

136 main()

#cpu = mx.getCurrentThreadCpuTime()/1000000
stop = System.currentTimeMillis()
end = stop-start
141 #print "cputime/ms:",cpu
print "tottime/ms:",end
#print "cpu util:",( (float(cpu)/end) * 100)
```

A.3. IO Quelltext

Für das IO Skript ist der folgende Quelltext in Pnuts.

Quelltext A.3: Pnuts IO

```
import ("java.io.*");
2 //import ("java.lang.management.*");
//mx = ManagementFactory.getThreadMXBean();
start = System.currentTimeMillis();


7 n = 20000000

    try {
        f = new File("C:\\Documents and Settings\\derDoc\\Local Settings\\Temp\\scratch");
                        pw= new PrintWriter(
12                              new BufferedWriter(
                                    new FileWriter(f)));

        for (i = 0; i < n; i++) {
            pw.print(i);
17      }
        pw.close();
    }
```

```
        catch(IOException ioe) {
            ioe.printStackTrace();
22      }

    //cpu = mx.getCurrentThreadCpuTime()/1000000;
    stop = System.currentTimeMillis();
    end = stop-start;
27  //System.out.println("cputime/ms:"+cpu);
    System.out.println("tottime/ms:"+end);
    //System.out.println("cpu util:"+( ((double)cpu/end) * 100));
```

A.4. Hash Quelltext

Der Code für das Hash Skript ist diesmal in JavaFX Script.

Quelltext A.4: JavaFX Hash

```
1   import java.util.HashMap;
    import java.lang.System;
    //import java.lang.management.*;

    //var mx = ManagementFactory.getThreadMXBean();
6   var start = System.currentTimeMillis();

    var n = 10000;

    for (i in [0..10000]){
11                              //var x = new HashMap();
                                var x = [];
          for (j in [0..n]){
                                        //var I=new Float(i);
                                        //var J=new java.math.BigInteger(j);
16          //x.put(i,j);
            //x.get(i);
            x[i] = j;
            x[i];
          }
21  }

    //var cpu = mx.getCurrentThreadCpuTime()/1000000;
    var stop = System.currentTimeMillis();
    var end = stop-start;
26  //System.out.print("cputime/ms:");System.out.println(cpu);
    System.out.print("tottime/ms:");System.out.println(end);
    //System.out.print("cpu util:");System.out.println( (cpu / end) * 100);
```

A.5. String Quelltext

Der Code für das String Skript ist in Iava.

Quelltext A.5: Iava Hash

```
    public void test(int n){
2       String t = "";

        for (byte i = 0; i < 10*n; i++){
                for (byte j = 0; j < 100; j++){

7                       if ( ((i+j) % 2) == 0){
                                t = t + String.valueOf((char)(i+32));
```

```
                        t = t + String.valueOf((int)(j));
                    }else{
                        t = t + String.valueOf((char)(j+32));
12                      t = t + String.valueOf((int)(i));
                    }
                    //System.out.println(t);
                }
            }
17          String re = "a1";
            System.out.println(t.matches(re));
    }

    int n = 6;
22  test(n);
```

A.6. HSP Implementierung

A.6.1. SimpleXlet

Dies Quelltext des originalen SimpleXlet in Java. Es erzeugt 2 Knöpfe, die beim Drücken von den Pfeiltasten sowie Enter die Farbe wechseln.

Quelltext A.6: SimpleXlet

```
   import java.awt.Color;
   import java.awt.Font;
 3 import java.awt.event.KeyEvent;
   import java.awt.event.KeyListener;

   import java.io.File;
   import java.io.FileInputStream;
 8 import java.io.FileNotFoundException;
   import java.io.FileOutputStream;
   import java.io.PrintStream;
   import javax.tv.xlet.Xlet;
   import javax.tv.xlet.XletContext;
13 import javax.tv.xlet.XletStateChangeException;

   import org.havi.ui.HDefaultTextLayoutManager;
   import org.havi.ui.HScene;
   import org.havi.ui.HSceneFactory;
18 import org.havi.ui.HScreen;
   import org.havi.ui.HStaticText;

   /**
   * A simple Xlet with a HStaticText label that changes background color
23 * when a key is pressed.
   */
   public class SimpleXlet implements Xlet, KeyListener {

           private XletContext context;
28         private HScene scene;
           private HStaticText[] labels;
           private Color[] colors = { Color.black, Color.red, Color.blue };
           private int intColor;
           private int listpos;
33
           public SimpleXlet() {
           }

           public void initXlet(XletContext xletContext) throws XletStateChangeException {
38                 System.out.println("begin initXlet");
                   context = xletContext;
           }
```

```
        public void startXlet() throws XletStateChangeException {
43              System.out.println("begin␣startXlet");
                HSceneFactory hsceneFactory = HSceneFactory.getInstance();
                scene = hsceneFactory.getFullScreenScene(HScreen.getDefaultHScreen().getDefaultHGraphicsDevice());
                labels = new HStaticText[2];
                listpos = 0;
48
                scene.setSize(160, 200);
                scene.setLayout(null);
                scene.addKeyListener(this);

53              labels[0] = new HStaticText("IFN␣MHP␣Mobil", 10, 20, 160, 80, new Font("Tiresias", Font.BOLD, 20),
                        Color.yellow, Color.gray, new HDefaultTextLayoutManager());
                scene.add(labels[0]);
                labels[1] = new HStaticText("[Close]", 100, 150, 30, 15, new Font("Tiresias", Font.BOLD, 14),
                        Color.yellow, Color.black, new HDefaultTextLayoutManager());
58              scene.add(labels[1]);

                scene.setVisible(true);
                scene.requestFocus();
        }
63
        public void pauseXlet() {
        }

        public void destroyXlet(boolean flag) throws XletStateChangeException {
68              System.out.println("destroyXlet");
                if (scene != null) {
                        scene.setVisible(false);
                        scene.removeAll();
                        scene = null;
73              }
                context.notifyDestroyed();
        }
        public void keyTyped(KeyEvent e) {
        }
78
        public void keyReleased(KeyEvent e) {
        }

    public void keyPressed(KeyEvent e) {
83              int keycode = e.getKeyCode();
                if(keycode==java.awt.event.KeyEvent.VK_UP ||
                        keycode==java.awt.event.KeyEvent.VK_DOWN){
                        listpos = (listpos+1)%2;
                        HStaticText label = (HStaticText) labels[listpos];
88                      HStaticText labelold = (HStaticText) labels[(listpos+1)%2];
                        labelold.setBackground(Color.black);
                        label.setBackground(Color.gray);
                        labelold.repaint();
                        label.repaint();
93              }
                else if(keycode==java.awt.event.KeyEvent.VK_ENTER && listpos==1){
                        try{
                                destroyXlet(false);
                        }
98                      catch(XletStateChangeException err){
                                err.printStackTrace();
                        }
                }
                else if(listpos==0){
103 //                  try{
    //                      PrintStream out = new PrintStream(new FileOutputStream(
    //                              new File("test.txt")));
    //                      out.print("TADA");
    //                      out.close();
108 //                  }catch(FileNotFoundException exc){exc.printStackTrace();}

                        intColor++;
                        if (intColor == colors.length) {
                                intColor = 0;
113                     }
                        labels[listpos].setBackground(colors[intColor]);
                        labels[listpos].repaint();
                }
```

```
        }
}
```

Nun das Gleiche in Rhino/JavaScript. Dabei fällt auf, dass Funktionen eines Interfaces, die nicht implementiert werden, weggelassen werden können, sowie dass es Syntaxmäßig relativ einfach ist, das KeyListener Interface zu implementieren und registrieren.

Quelltext A.7: JS TestScriptXlet

```
importPackage(java.awt)
importPackage(java.io)
importPackage(Packages.xjavax.tv.xlet)
importPackage(Packages.org.havi.ui)

/**
* Simple Scripted Xlet that behaves like the SimpleXlet
* @author derDoc
*/
        var context;
        var scene;
        var labels = Array();
        var colors = Array(Color.black,Color.red,Color.blue);

        var intColor=0;
        var listpos=0;

        function initXlet(xletContext){
                java.lang.System.out.println("begin initXlet");
                context = xletContext;
        }

        function startXlet(){

                java.lang.System.out.println("begin startXlet");

                var hsceneFactory = HSceneFactory.getInstance();
                scene = hsceneFactory.getFullScreenScene(HScreen.getDefaultHScreen().getDefaultHGraphicsDevice());

                scene.setSize(160, 200);
                scene.setLayout(null);

                labels[0] = new HStaticText("IFN MHP Mobil", 10, 20, 160, 80, new Font("Tiresias", Font.BOLD, 20),
                        Color.yellow, Color.gray, new HDefaultTextLayoutManager());
                scene.add(labels[0]);
                labels[1] = new HStaticText("[Close]", 100, 150, 30, 15, new Font("Tiresias", Font.BOLD, 14),
                        Color.yellow, Color.black, new HDefaultTextLayoutManager());
                scene.add(labels[1]);

                scene.addKeyListener(function keyPressed(e){
                        var keycode = e.getKeyCode();

                        if(keycode==java.awt.event.KeyEvent.VK_UP ||
                           keycode==java.awt.event.KeyEvent.VK_DOWN){

                                listpos = (listpos+1)%2;
                                var label = labels[listpos];
                                var labelold =  labels[(listpos+1)%2];
                                labelold.setBackground(Color.black);
                                label.setBackground(Color.gray);
                                labelold.repaint();
                                label.repaint();
                        }
                        else if(keycode==java.awt.event.KeyEvent.VK_ENTER && listpos==1){
                                try{
                                        destroyXlet(false);
                                }
                                catch(err){
                                        err.printStackTrace();
                                }
```

```
                        }
                        else if(listpos==0){
                                intColor++;
                                if (intColor == colors.length) {
65                                      intColor = 0;
                                }
                                labels[listpos].setBackground(colors[intColor]);
                                labels[listpos].repaint();
                        }
70
                });
                scene.setVisible(true);
                scene.requestFocus();
        }
75
        function destroyXlet(flag){
                java.lang.System.out.println("destroyXlet");

                if (scene != null) {
80                      scene.setVisible(false);
                        scene.removeAll();
                        scene = null;
                }
                context.notifyDestroyed();
85      }
```

A.6.2. JSShellXlet

Das ist der Code des Xlet, das JavaScript mit Hilfe von Rhino interpretiert. Es ist in JavaScript geschrieben.

Quelltext A.8: JSShellXlet

```
importPackage(java.awt)
importPackage(java.io)
importPackage(java.lang)
importPackage(Packages.javax.tv.xlet)
5 importPackage(Packages.org.havi.ui)
importPackage(Packages.org.mozilla.javascript)


/**
10 * Simple JS Script Interpreter reading from file test.js
* @author derDoc
*/
        var context;
        var scene;
15      var cx;
        var scope;


        function initXlet(xletContext){
20              System.out.println("begin␣initXlet");
                context = xletContext;
        }

        function startXlet(){
25
                System.out.println("begin␣startXlet");
                var hsceneFactory = HSceneFactory.getInstance();
                scene = hsceneFactory.getFullScreenScene(HScreen.getDefaultHScreen().getDefaultHGraphicsDevice());

30              scene.setSize(240, 240);
                scene.setLayout(null);

                scene.setVisible(true);
                scene.requestFocus();
35
                //set up context and register HScene as S in ScriptContext
```

```
                cx = Context.enter();
                scope = cx.initStandardObjects();

40              var wrappedS = Context.javaToJS(scene, scope);
                ScriptableObject.putProperty(scope, "S", wrappedS);
                interpret("test.js");

        }
45

        function destroyXlet(flag){
                System.out.println("destroyXlet");
                if (scene != null) {
50                      scene.setVisible(false);
                        scene.removeAll();
                        scene = null;
                }
                context.notifyDestroyed();
55              Context.exit();
        }

        function interpret(str){
                System.out.println("interpreting␣"+str);
60       try {

                var t = '';
                var br = new java.io.BufferedReader(
                        new java.io.FileReader(str));
65              while (s = br.readLine())
                        t += s + "\n";

                var result = cx.evaluateString(scope, t, "<interpreted␣code>", 1, null);
                var res = Context.toString(result);
70              System.out.println("Result:␣"+res);

          }catch(exc){
                System.out.println("Exception:␣"+exc);
          }
75      }
```

Nun der Code des Test-Skriptes, das das TUBS Logo zeichnet.

Quelltext A.9: test.js

```
var o = {  image: null,
        x: 0, y: 0, h: 0, w: 0,
3
        setImage: function(img) {
                image = img;
                if(image!=null){
                        image.flush();
8                       //repaint();
                }
                java.lang.System.out.println("Image␣set␣to:␣"+img);
        },
        setMySize: function(x,y,wid,hei) {
13              o.x = x;
                o.y = y;
                o.w = wid;
                o.h = hei;
                //java.lang.System.out.println("Size set to: "+o.x+","+o.y+","+o.w+","+o.h+",");
18      },
        update: function(g){
                paint(g);
        },
        paint: function(g) {
23              g.drawImage(image, o.x, o.y, o.w, o.h,
                        null);
                //java.lang.System.out.println("Done drawing w/: "+o.x+","+o.y+","+o.w+","+o.h+",");
        }
}
28
var bg = new JavaAdapter(java.awt.Component, o);
```

```
   var MaxSize = 240;
33 var MinSize = 1;

   var bg_img = loadImage("/hsp_svg/bin/tu-logo.gif", S);
   bg.setImage(bg_img);
   bg.setSize(240,240);
38 bg.setMySize(120-(99/2),120-(116/2), 99, 116);
   S.add(bg);
   S.requestFocus();

   S.addKeyListener(function keyTyped(e){
43         if (e.getID() == java.awt.event.KeyEvent.KEY_TYPED){
                   x = nextInt(MinSize,MaxSize);
                   y = nextInt(MinSize,MaxSize);
                   w = nextInt(MinSize,MaxSize);
                   h = nextInt(MinSize,MaxSize);
48                 java.lang.System.out.println("repainting logo w/: "+x+","+y+","+w+","+h);
                   bg.setMySize(x,y,w,h);
                   bg.repaint();
           }
   });
53

   function loadImage(f, component) {
                   try {
                           var url = new java.net.URL("file://"+f);
58
                           var mediatracker = new java.awt.MediaTracker(component);
                           var toolkit = java.awt.Toolkit.getDefaultToolkit();
                           var image = null;
                           if (f != null) {
63                                 image = toolkit.getImage(f);
                           }
                           mediatracker.addImage(image, 0);
                           try {
                                   mediatracker.waitForID(0);
68                         } catch (ex) {
                                   java.lang.System.out.println("Exception in waiting4loaded Image: "+ex);
                           }
                   return image;
                   } catch (e) {
73                         java.lang.System.out.println("Exception in loadImage: "+e);
                   }
           }


78 function nextInt(min, max) {
           return java.lang.Math.floor(min+java.lang.Math.random()*(max-min+1));
   }
```