Wenn Java die Antwort ist, was war dann die Frage?

Erschienen in der Fachzeitschrift „Computer" der IEEE Computer Society, März 1997. Übersetzt von Walter Riemer. Der Autor, Ted Lewis, ist an der Naval Postgraduate School, Monterey, California, tätig (*tedglewis@aol.com*, *http://www.spindoczine.com*).

Übersetzt von Walter Riemer

Innerhalb ganz weniger Jahre wird Java, Java Beans und alles mit Java Zusammenhängende unausweichlich sein. Mit nur wenigen anderen Innovationen erfreut sich Java der schnellsten Akzeptanz, ungefähr wie Videospiele und fast so wie Multimedia-PCs. Java wird hell brennen und dann ausbrennen; vor dem Letzeren allerdings wird Java so gebräuchlich sein wie ein Staubtuch im Haushalt.

Euphorie über ein Produkt gehört zur Computerindustrie wie zu Hollywood. Erstklassigkeit wird oft ein Opfer der Public Relations-Aktivität. Im Fall Java ist es besonders schwierig, PR von der Realität zu unterscheiden. Wer kann einem die Wahrheit über Java sagen?

Die erste Frage könnte sein: ist Java wirklich eine Verbesserung? Schlicht gesagt: nein. Wenn die heutigen Sprachen den heutigen Herausforderungen der Softwareerstellung nicht genügen, gilt das auch für Java. Denken Sie daran, daß Java aufgewärmtes C/C++ ist. Obwohl es so gefeiert wird, fehlen ihm viele Eigenschaften, welche das elende Gewerbe des Software Engineering erleichtern würden, genauso wie seinen Vorgängern. Wie so, fragen Sie? Hier ist meine Analyse - so sachlich wie möglich - der Pros und Kontras von Java.

Java als Wählton

Mein erstes Programm schrieb ich für einen Elektronenröhren-Rechner mit einem grandiosen 32 kB-Trommelspeicher und Unmengen von Lochstreifen. Auf dieser erstaunlichen Maschine lief 32-Bit-Software, mehr als man von vielen WIN95-Applikationen behaupten kann. Bald stieg ich auf zu Transistoren, Algol, Fortran, Pascal, Ada und C++. Mit einem derartig steilen Fortschritt hätte ich zufrieden sein können, aber ich leide unter chronischer Sucht, mich zu beklagen. Ich wollte mehr, ich wollte eine universelle Computersprache, die auf jedem Computer, unter jedem Betriebssystem und von irgendwo her arbeitet.

Eine Meute Programmierer, die im Computerzentrum herumlungerten, teilten diesen Traum mit mir. Es waren gar nicht wenige. Dieser verbreitete Traum hatte sogar einen Namen: UNCOL (Universal Common Language). Kommt Ihnen das bekannt vor? Der Name wurde 1963 erfunden, als jedermanns Augen auf Algol gerichtet waren, weil es versprach, das UNCOL-Problem zu lösen. Aber es konnte es nicht.

Dann passierte Pascal. Pascal fand enorme Akzeptanz, weil PL/I zu groß, zu umständlich und zu inkonsistent in den Regeln war (ähnlich wie heute C++). Das UNCOL-Problem löste es ebensowenig wie Algol. Dann passierte Ada, als besseres Pascal gedacht, aber es versagte weil der Kapitalismus Monopole ablehnt. Schließlich passierte C++; Sie erkennen, worauf ich hinaus will: im Laufe der Jahre sind wir daraufgekommen, daß Software nicht mit einer standardisierten, universellen und plattformneutralen Sprache konstruiert werden kann.

Je mehr sich die Dinge ändern, desto mehr bleiben sie gleich: im Augenblick ist es Java, welches das UNCOL-Problem lösen soll. Man sagt: Java wird der Wählton des einundzwanzigsten Jahrhunderts sein. Wenn die Vergangenheit ihre Funktion als Prolog zur Gegenwart wahrnimmt, hat Java als UNCOL eine zweifelhafte Zukunft.

Geisel von ererbtem Code („Legacy Code")

Inzwischen wird Java seine 15 ruhmreichen Minuten haben; was aber danach? Die wirklich wichtige Frage ist: welche Verwüstungen wird Java hinterlassen, wenn es abtritt?

Aufgrund von Javas Akzeptanzrate und seiner zu erwartenden Lebensdauer werden heutige Java-Applikationen innerhalb eines Jahrzehnts ersetzt oder zumindest modifiziert werden. Während Java eine neue Ära einleitet, ist es in unserer Verantwortung, dafür zu sorgen, daß Java-Applikationen gutartige („well-behaved") Systeme werden.

Aus dieser Sicht wird das Problem Java als UNCOL ernst. Eine Studie hat zum Beispiel geschätzt, daß das US-Verteidigungsministerium nur für das Jahr-2000-Problem 30 Milliarden US-$ ausgeben wird. Ererbte Cobol-Systeme zu warten ist heute eine bedeutende Industrie, wodurch viele Organisationen Geiseln der Softwaretechnik der Siebzigerjahre sind. Legacy Code beschäftigt IBM, Andersen Consulting, EDS und viele Fortune 500-Firmen. Der Schwanz wedelt mit dem Hund. Im Jahre 2010 wird Java der Wartungsschwanz sein, der mit dem Softwarehund wedelt, und nicht mehr so sehr Cobol.

Verdammt dazu, die Geschichte zu wiederholen

Eines der Ziele von Java ist eleganter Minimalismus. So wie Pascal ist es eine einfache Sprache, und zwar mit Absicht. Aber wir wissen aus der Geschichte, daß Minimalismus selten erfolgreich ist. Das sichere, verläßliche, minimalistische Pascal setzte sich nicht durch, weil der Markt Minimalismus als „unvollständig" vorverurteilt: Pascal wurde sofort als Spielzeugsprache klassifiziert.

Auch die Universalsprache Java ist unvollständig: I/O, eingebaute Funktionen, APIs zu einem Betriebssystem und andere Eigenschaften sind nicht vorhanden. Da es viele minimalistische Eigenschaften von Pascal übernahm (Bytecodes, strenge Typisierung, eingeschränkte Zeiger, sprechende Schlüsselwörter statt kryptischer Symbole), hat Java offensichtlich den von Pascal, der ersten portablen Sprache, vorgegebenen Weg neuerlich beschritten.

Trotz vieler und jahrelanger Plädoyers für Einfachheit hat Minimalsmus im Bereich der Sprachen versagt, wie auch in fast allen anderen Bereichen der Softwareindustrie. Java mag als verbessertes C++ seine Verdienste haben, aber auf den Märkten reüssieren selten Techniken im Sinne einer „besten Lösung". Warum also sollte Java reüssieren?

Was war die Frage?

Die Softwareindustrie benötigt ganz offensichtlich ein neues Paradigma: ist Java das neue Paradigma? Auf Java angewendet ist das Wort Paradigma vielleicht viel zu stark. Das meiste von Java ist verwässertes C++, und dazu ein teilweises Wiederaufleben von Pascal (strenge Typprüfung, sprechende Schlüsselwörter, Packages vom USCD-P-Code und portabler Byte-Code). Java Threads sind bedeutende Verbesserungen gegenüber Unix Tasking, sind aber eigentlich nichts neues.

So ziemlich das einzige, worin Java sich vielleicht die Klassifizierung als neues Paradigma verdienen könnte ist das „Tagged

Applet". Applets und Servlets sind wahrhaftig neu, aber in Wirklichkeit ist es keineswegs zwingend, sie in Java zu schreiben.

Das macht mich darüber nachdenken: Wenn Java die Antwort ist, was war dann die Frage? Welches Problem löst Java, welches vor Java unlösbar war? Ich gehe dieser Frage zunächst mit einigen allgemeinen Bemerkungen nach und später mit einigen speziellen, wobei Java mit den Alternativen verglichen wird.

Hundefutter fürs Denken

Mit welchen Problemen muß der zukünftige Java-Programmierer rechnen? Erstens, Java hat noch immer zu viele der fehlerträchtigen Eingenschaften von C++ beibehalten. Es gibt eine Fülle syntaktischer Probleme, von denen ich nur einige erwähne als Beispiele für Hundefutter für Java-Programmierer.

Man denke nur einmal an die schlechte C-Gewohnheit des Inkrementierens und Dekrementierens. Was bedeutet

```
int i = ++i-;
```

in Java? Wie in der elenden C-Tradition startet Java immer ab Null anstelle von Eins. Wieviele Programmierer werden noch Stunden brauchen um Fehler um Plus/Minus Eins aufzuspüren? Was spricht eigentlich gegen die Zahl Eins?

Javas ungeregelte Gültigkeitsbereichsregeln beleidigen sogar noch die diesbezüglichen Gemeinheiten in C++. Gültigkeit, Bereiche und Synchronisierungskonstruktionen sind überkompliziert, manchmal widersprüchlich und meist nicht gut durchdacht. Nicht weniger als zehn Objektmodifizierer gibt es in Java (schon wieder Hundefutter): public, private, protected, static, final, native, synchronized, abstract, threadsafe und transient. Der sakrosankte public-Modifizierer zerstört die Kapselung komplett. Static, final und public widersprechen einander, stiften Verwirrung und ihre Notwendigkeit ist schwer einzusehen. Native ist überhaupt nicht zu rechtfertigen in einer Sprache, deren Anspruch Plattform-Neutralität ist.

Ebenso wie Pascal geht Java I/O aus dem Weg. Pascal wie auch Java überlassen I/O Bibliotheken, die oft schlecht definiert und unvollständig sind. Als Konsequenz daraus sprießen I/O-Bibliotheken aus dem Boden, die diverse Java-Dialekte entstehen lassen. Das restlose Standardisieren der Sprache war es gerade, dessen Fehlen Pascal zur Strecke brachte - Microsoft wirft sich auf genau diesen Mangel von Java und ist schon dabei, Java umzubringen.

Java ohne Threads

Die Leute, welche die Threads entworfen haben, Javas meistgelobte Eigenschaft, haben eine einzigartige Gelegenheit versäumt, zukünftige Legacy-Systeme verläßlicher zu machen.

Java-Schreiber erhielten leichtgewichtige Threading-Rechte: im großen und ganzen sind sie eine Verbesserung gegenüber dem schwergewichtigen Threading-Tasking in Unix. Es eröffnet viele Möglichkeiten, im Guten wie im Bösen. Donald Knuths Atomprozeduren erfahren in Java eine Reinkarnation um gegenseitiges Ausschließen sicherzustellen, aber Java geht nicht genügend weit. In der Tat kann die Kombination leichtgewichtiger Threads mit den Java Atomprozeduren einen Programmierer in den Ruin führen.

Angenommen ein Programm speichert Daten in einem Doppelpuffer L1 und L2 und erzeugt dazu zwei Threads. Der erste Thread kopiert Daten von L2 nach L1 und letzlich in den ersten laufenden Thread T1. Ein anderer Thread tut genau das Gegenteil: er kopiert Daten von L1 nach L2 und letzlich in den zweiten laufenden Thread T2. T1 und T2 haben also gleichzeitig Zugriff auf L1 und L2.

In Java benützt ein Programmierer Atomprozeduren um das gegenseitige Ausschließen sicherzustellen, zum Beispiel mit dem Modifizierer `synchronized`. Synchronisierte Funktionen sichern gegenseitiges Ausschließen, indem sie innerhalb der Funktion nur einem Thread zu einer bestimmten Zeit Aktivität erlauben. Ein Programmierer könnte unschuldigerweise eine synchronisierte Java-Funktion wie die nachstehende, etwas vereinfachte, schreiben. Darin wird eine get access-Funktion innerhalb einer List-Klasse deklariert und dann zweimal zu einer Instanz gemacht, einmal für L1 und einmal für L2.

```
class List   {
  synchronized public get(List L; char c)
  {...     }     ...
  L1=new LIST(...);
  L2=new LIST(...);
```

Angenommen Thread T1 verwendet die folgenden Objekte in der gegebenen Reihenfolge:

```
   get(L1...);   get(L2...);
```

und Thread T2 wendet sie unglücklicherweise und vielleicht unabsichtlich in umgekehrter Reihenfolge an:

```
   get(L2...);   get(L1...);
```

Dieses Programm mit zwei Threads funktioniert meistens, aber nicht immer. Meistens erfolgen die Zugriffe in der vom Programmierer festgelegten Reihenfolge. Gelegentlich aber (und abhängig vom Time-Slicing-Algorithmus des Betriebssytems) wird T1 gerade unmittelbar, nachdem es den ausschließlichen Zugriff auf L1 erhalten hat, unterbrochen und T2 erhält den ausschließlichen Zugriff auf L2. Sobald T1 wieder die Kontrolle erhält, kann T1 von T2 blockiert sein und sobald T2 wieder die Kontrolle erhält, kann T2 von T1 blockiert sein. Das System zweier Threads hängt!

Dies ist ein besonders böser Fehler, weil er unvorhersehbar auftritt, fast wie ein zufälliger Hardwarefehler. Mit normalen Debugging-Techniken kann man kaum dahinter kommen. Aus Sicht der Programmierer ist der Code in Ordnung: der Fehler muß in der Hardware liegen! Tausende zukünftige Programmierer werden hunderte Arbeitsstunden aufwenden müssen, um solche Fehler aufzuspüren.

Eine bessere Lösung, Threads zu steuern wäre es gewesen, Path-Ausdrücke aus Path Pascal zu übernehmen. Die gibt es, wie fast alles in Java, auch schon seit zwanzig Jahren. Sie können in den Interface-Teil einer Klasse gestellt werden und können daher vom Compiler zum Überprüfen auf Synchronisierungsprobleme benützt werden. Anders ausgedrückt: Synchronisierung sollte eine Interface-Spezifikation sein, nicht eine Codierungsmethode.

Eine bessere Mausefalle

Dieses Beispiel zeigt nur einige in Java schlummernde Probleme - Probleme, welche zukünftige Programmierer verfluchen werden, wenn sie die Tonnen von Java-Code warten, die im kommenden Jahrzehnt anfallen werden.

In der Tat gelingt es Java nicht, einigen Grundproblemen beizukommen, welche das Software-Engineering in den nächsten Jahren weiterhin plagen werden. Wenn Java eine bessere Mausefalle werden will, muß es sich den Kernproblemen der Softwareentwicklung stellen:

Anforderungen: Festschreibung von Anforderungen und Spezifikation hat sich als zu weit gespannter Schritt herausgestellt. Java versucht nicht einmal, sich diesem Problem zu stellen, und der Java-Enthusiasmus birgt in sich die Gefahr, den auf diesem Gebiet schwer errungenen Fortschritt zu behindern.

Fehlerbeseitigung: In den letzten 30 Jahren kann fast der gesamte Fortschritt in der Softwareentwicklung auf frühzeitige Fehlerbeseitigung zurückgeführt werden. Positiv zu vermerken ist, daß Java strenge Typisierung, Einschränkungen bei Zeigern und nur einfaches Vererben besitzt. Als negativ ist zu erwähnen, daß Java noch immer C/C++-Probleme weiterschleppt, die APIs nicht standardisiert sind, das gebrechliche Basisklassensystem, in dem Änderungen im Interface abstrakter Klassen ganze Klassenhierarchien ruinieren können, sowie das ebenfalls empfindliche Interface-Problem, in dem eine Änderung

in einer Interface-Spezifikation sich an allen Punkten eines großen Programms, wo dieses Interface benützt wird, auswirken kann (hier nur einige Negativa angeführt). Mit Java wurde nur ein kleiner Schritt in Richtung früher Fehlerbeseitigung gegangen.

Component-Konzept: Es ist bekannt, daß Software-Entwicklungskosten je Funktion exponentiell mit der Codegröße steigen: es kostet mehr als doppelt so viel, doppelt so viel Code zu schreiben. Die Java Bean-Technik mag hier eine Verbesserung sein. Nichtsdestoweniger versprechen andere Techniken wie ActiveX, CORBA und OpenDoc ähnliche Verbesserungen; die Java Lösung an sich ist um nichts besser als jene.

Lebenszykluszeit: Applikationen müssen sich synchron mit dem Internet ändern, also ungefähr innerhalb von 18 Monaten. Dies begrenzt die Größe eines neuen Systems, seine Funktionalität oder auch beides. Java hilft wenig im Sinne einer Anpassung an Internet-Zeitabläufe. Kombiniert mit RAD (Rapid Application Development) kann Java allerdings dank seiner Plattform-Neutralität den Aufbau von Systemen innerhalb des Internet-Zyklus ermöglichen - Systemen, die ohne RAD nicht möglich wären.

Komplexität: Heutige Applikationen sind bedeutend komplexer als frühere. Applikationen von morgen werden noch komplexer sein und noch größere Herausforderungen sein. Ganz allgemein steigen die Erwartungen der Öffentlichkeit schneller als die Softwareentwicklung dem nachkommen kann. Java scheint auch kein stärkerer intellektueller Hebel zu sein als Ada, Pascal und so weiter. Noch schlimmer, Java schiebt den intellektuellen Horizont nicht hinaus. So manche andere Sprachen mögen Verbesserungen sein oder auch nicht; sie haben jedoch nicht die gleiche Gelegenheit bekommen, sich zu beweisen, wie Java.

In Java gibt es einiges, dem Beifall zu zollen ist: die Fehlerbehandlung, einfache Vererbung, Interface-Spezifikations-Konstrukte. Außerdem hat Java klugerweise einige wertvolle Eigenschaften von Programmiersprachen vor C/C++ übernommen und als Innovation die Idee des HTML-tagged Applet kultiviert. Fortschritt wurde gemacht (wenn er auch 20 Jahre gebraucht hat). Das aber ist viel zu wenig Fortschritt in einer erwachsen gewordenen Industrie! Java enttäuscht im derzeitigen Entwicklungsstand einfach noch zu sehr.

Wohl besteht Hoffnung daß Java zuletzt reifen, daß wir unsere Zukunft ganz auf Java aufbauen, zumindest noch nicht!

VBScript

Hans Blocher

VBScript – Visual Basic Scripting Edition – ist eine Untermenge der bekannten Programmiersprache Visual Basic von Microsoft. VBScript wurde als Sprache für WWW-Dokumente entwickelt und sollte leicht, schnell und sicher sein. VBScript erfüllt die beiden ersten Ansprüche, in den Expertenstreit über die Sicherheit von VBScript soll an dieser Stelle aber nicht eingegangen werden.

Zum Leistungsumfang von VBScript gehören Möglichkeiten wie dynamische Generierung von Webseiten oder Überprüfung von Formulareingaben.

Serverseitige Datenverarbeitung

Bei serverseitiger Datenverarbeitung werden von der (Internet-) Station Instruktionen und Daten über das Internet an den Server geschickt. Dort werden diese Informationen verarbeitet und die Ergebnisse wieder an die Station zurückgesandt, wo sie interpretiert und im Browser angezeigt werden.

Zur Zeit wird im Internet die meiste Datenverarbeitung an den Servern durchgeführt, wo Programme auf Basis von CGI, Java oder Perl ablaufen.

Clientseitige Datenverarbeitung

Bei diese Art der Datenverarbeitung ist der Browser selbst in der Lage, Instruktionen und Daten zu interpretieren, die sonst an den Server gesandt werden. Die Hauptvorteile sind dabei die Schnelligkeit der Durchführung (es fällt sowohl die langsame Datenübertragung über das Internet weg, als auch die zeitaufwendigen Prozesse, die in den oft überlasteten Servern ablaufen müssen) und die Einsparung an Übertragungsbandbreite.

VBScript ist nun ein Hilfsmittel, mit dem diese Art der Datenverarbeitung unterstützt wird. Es wird als Teil des HTML-Stroms als ASCII-Text zum Browser übertragen, dort mit Hilfe der VBScript Engine compiliert und im Speicher bereitgehalten. Teile des Programmes können nun auf zwei Arten aufgerufen werden:

- Ereignisse des Browser-Fensters (wie z.B. Anklicken eines Buttons) rufen die dafür definierten Teile auf.
- Bestimmte Programmteile (Funktionen) können von anderen Programmteilen aufgerufen werden.

Dieses Programmbeispiel ist dem englischen Buch "Learning VBScript" von Paul Lomax, Verlag O´Reilly, inkl. CD-Rom mit über 170 Beispielen, entnommen. Dieses Buch ist ein sehr gute Einführung in VBScript, das die wesentlichen Möglichkeiten dieser Script-Sprache bis zu ActiveX-Techniken, dynamische Erzeugung von Webseiten und Überprüfung von Benutzereingaben, abdeckt. Als Zielgruppe