# Java - Ein Schnappschuß

**Klaus Berg** [1], **Briktius Marek** [1],
**Gustav Pomberger** [2] , **Johannes Sametinger** [2] , **Alois Stritzinger** [2]
[1] Siemens AG, ZT SW 1, Otto-Hahn-Ring 6, D-81730 München
{Klaus.Berg | Brix.Marek}@mchp.siemens.de
[2] CD-Labor für Software Engineering, Johannes Kepler Universität, A-4040 Linz
{pomberger | sametinger | stritzinger}@swe.uni-linz.ac.at

In diesem Artikel werden ein kurzer Überblick über Java und dessen Einsatz, sowie eine Diskussion der Stärken und Schwächen aufgezeigt und beurteilt. Dadurch soll eine Evaluierung von Java im Hinblick auf die praktische Einsetzbarkeit in der Softwareentwicklung ermöglicht werden. Zusätzlich werden Punkte erläutert, die vor einem Einsatz von Java berücksichtigt werden sollen. Durch das rasante Voranschreiten der Entwicklungen um Java kann nur ein aktuelles Bild der Situation skizziert werden. Zwar enthält der Artikel viele allgemein gültige Aussagen, insbesondere über Sprache und deren Eigenschaften. Die Verfügbarkeit sowie die Mächtigkeit von Klassenbibliotheken und Werkzeugen ist aber einem ständigen Wandel unterzogen. In diesem Bereich kann nur ein aktueller Schnappschuß geboten werden. Das Komponentenmodell JavaBeans wird ebenfalls kurz vorgestellt und einer Beurteilung unterzogen.

## 1. Einleitung

Im Jahre 1995 wurde von Sun Microsystems die objektorientierte Programmiersprache Java der Öffentlichkeit vorgestellt [Sun97d]. Die Sprache war Ergebnis mehrjähriger Forschungsarbeiten bei Sun und zeichnet sich insbesondere durch ihre Unterstützung der Programmierung von Internet-Anwendungen aus. Zu diesem Zeitpunkt verzeichnete die praktische Bedeutung des Internet, insbesondere wegen des *World Wide Web* (WWW), einen raschen Anstieg. Gleichzeitig wurde *HotJava* vorgestellt, ein in Java implementierter Web-Browser mit integriertem Java-Interpreter. Damit war es möglich, Programme in WWW-Seiten einzubinden und automatisch über das Internet zu laden und auszuführen. Die Bandbreite möglicher Anwendungen des WWW vergrößerte sich sprunghaft. Zusätzlich gelang es den Entwicklern von Java durch die Anlehnung an C und C++, sowie der Ausmerzung von Unzulänglichkeiten dieser Sprachen, eine hohe Akzeptanz zu erzielen. Java erfuhr somit über das Internet weltweit eine einzigartig rasche Verbreitung und Etablierung.

Java ist nicht nur eine weitere Programmiersprache. Ihr Erfolg basiert nicht zuletzt darauf, daß durch die Definition einer *virtuellen Maschine* Java-Programme plattformunabhängig erstellt werden können. Jedes Java-Programm wird vom Compiler nicht wie sonst üblich in Maschinencode (*object code*), sondern in einen maschinenunabhängigen Bytecode übersetzt, der dann auf verschiedenen Plattformen, auf denen eine virtuelle Java-Maschine verfügbar ist, ausgeführt, d.h. interpretiert werden kann. Um Effizienzeinbußen durch die Interpretation zu verringern, wird der Bytecode oft unmittelbar vor der Ausführung, also zur Laufzeit, in echten Maschinencode übersetzt. Man spricht in diesem Fall von einem *Just-in-Time*-Compiler (JIT). Neben der Programmiersprache, der virtuellen Maschine und dem JIT-Compiler spielen auch Klassenbibliotheken für die Programmierung eine wichtige Rolle. Solche werden ebenfalls auf allen Plattformen mit identischen Schnitt-

stellen angeboten, um die Portabilität von Java-Programmen zu gewährleisten. Dies gilt auch für Klassen zur Programmierung von graphischen Benutzerschnittstellen.

Mit Java können nicht nur Applets, sondern, wie mit anderen Programmiersprachen auch, vollständige Applikationen entwickelt werden. Da Applets über das Internet geladen werden, ist der Sicherheitsaspekt besonders wichtig und in der Sprache auch berücksichtigt.

Neben Applikationen und Applets gibt es auch Java-Komponenten, *JavaBeans* genannt. JavaBeans bilden eine plattformunabhängige Komponentenarchitektur und sollen eine komponentenbasierte Softwareentwicklung ermöglichen. Komponenten sind unabhängiger voneinander als Objekte und sollten einfacher zu verbinden sein. Dadurch werden Komponenten besser wiederverwendbar und können helfen, die Produktivität bei der Softwareentwicklung zu steigern, siehe z.B. [Ber98, Ste98a, Ste98b].

Java bietet sogenannte *Threads* (parallele Prozesse), dynamisches Laden von Klassen und eine umfangreiche Klassenbibliothek. Trotz der Leistungsbreite ist ein überschaubarer Sprachumfang gegeben. Zusätzlich ist Java durch die Anlehnung an die Sprachen C und C++ von vielen Programmierern leicht zu erlernen.

# 2. Einsatzszenarien

Für den Einsatz von Java genügt grundsätzlich ein Java-Compiler und ein Java-Interpreter für den Entwickler, sowie ein Java-Interpreter für den Java-Anwender. Mit herkömmlichen Texteditoren kann der Quelltext erstellt werden. Mit dem Compiler wird der plattformunabhängige Bytecode erzeugt, der dann mit dem Interpreter ausgeführt, d.h. interpretiert wird. Neben dem eigentlichen Interpreter müssen dazu immer auch die Kernpakete der Bibliothek für eine Plattform vorhanden sein. Für Entwicklungen größeren Umfangs ist zusätzliche Werkzeugunterstützung zweckmäßig, z.B. Klassenbrowser, GUI Builder, Debugger, Konfigurations- und Versionsmanagement. Programmierumgebungen und Klassenbibliotheken werden mittlerweile von mehreren Herstellern angeboten.

## 2.1 Applets

Mit Java können Programme entwickelt werden, die sich in Seiten des *World Wide Web* integrieren lassen und dann über das Internet geladen und lokal ausgeführt werden können (sogenannte Applets). In diesem Zusammenhang kommt Sicherheitsaspekten große Bedeutung zu, da sich niemand Applets über das Netz laden will, die beispielsweise Informationen zerstören oder an Dritte weitergeben. Um solche "Angriffe" zu vermeiden, sind Applets in ihren Rechten beschränkt und haben nur einen eingeschränkten Aktionsradius (*Sandbox Model*). Applets dürfen beispielsweise nicht auf die lokale Platte zugreifen oder Netzwerkverbindungen zu anderen Servern als dem aufbauen, von dem sie geladen wurden.

Der Vorteil des Einsatzes von Applets ist, daß die reine Benutzerschnittstelle von der Funktionalität entkoppelt wird und separat in verschiedenen Web-Browsern auf unterschiedlichen Maschinen ablauffähig ist. Java-Applets kommunizieren über ein Netz (z.B. Internet oder firmeneigenes Intranet) mit dem Server (z.B. Datenbank). Speziell wenn Applets über das Internet geladen werden, sind Sicherheitsvorkehrungen von großer Bedeutung.

Die Verwendung von Applets bedeutet naturgemäß eine gewisse Netzbelastung, da sie, falls nicht lokal vorhanden, zuerst geladen werden müssen, und dann oft auf den Server über das Netz zugreifen. Eine hohe Verfügbarkeit des Netzes ist in diesem Fall wichtig.

## 2.2 Applikationen

Java wird vielfach als Sprache für das Internet gesehen. Dies ist jedoch nur ein Aspekt der Sprache. Grundsätzlich ist Java eine anwendungsunabhängige objektorientierte Programmiersprache, mit der Softwaresysteme verschiedener Art erstellt werden können. Java wird auch längst nicht mehr nur für kleine Applets und Miniapplikationen verwendet. Durch ihre Robustheit ist Java eine überlegenswerte Alternative zu Sprachen wie C und C++. Die Plattformunabhängigkeit ist für eigenständige Applikationen ebenso interessant. Für Realtime-Anwendungen ist der Einsatz von Java jedoch noch nicht geeignet. Um eine Applikation auf einer Plattform verwenden zu können, muß auf dieser Plattform eine *Java Virtual Machine* verfügbar sein. Außerdem müssen die von der Applikation verwendeten Bibliotheken auf der Plattform vorhanden sein, was bei den Kernbibliotheken aber ohnehin der Fall sein muß.

Obwohl sich viele Firmen auf Java eingeschworen haben, gibt es derzeit erst wenige umfangreiche, in Java implementierte Produkte. Beispiele für in Java implementierte Applikationen sind die als *100% Pure Java* zertifizierten Produkte, siehe [Jav98].

## 2.3 Client/Server

Durch die Plattformunabhängigkeit und die leichte Einbindung in ein Netzwerk können mit Java elegant verteilte Lösungen entwickelt werden. Diese müssen nicht notwendigerweise als Applets realisiert werden. Klassische Client/Server-Architekturen sind ebenso möglich. Ob es günstiger ist, eine monolithische Applikation, eine Client/Server- oder eine Applet-Lösung zu entwickeln, mag von Fall zu Fall unterschiedlich sein. Es läßt sich allerdings ein eindeutiger Trend weg von monolithischen Applikationen und hin zu kleinen Clients erkennen. Grund dafür ist nicht zuletzt eine Verringerung des Administrationsaufwandes.

Gegenüber traditionellen Client/Server-Lösungen haben Applets den Vorteil, daß sie in World-Wide-Web-Seiten integriert und plattformunabhängig verwendet werden können und daß der Benutzer, falls gewünscht, immer automatisch mit der neuesten Version arbeitet. Neben der eigentlichen Applikation können in diesem Szenario auch sämtliche Benutzerdaten auf dem Server abgelegt werden. Dadurch ergeben sich wesentliche Vereinfachungen für systematische Datensicherungen. Außerdem kann ein Benutzer von unterschiedlichen Rechnern auf seine gewohnten Daten und seine persönliche Arbeitsumgebung zugreifen [Bro97].

## 2.4 Netzcomputer

Nach der Ära der Großrechner haben sich PCs durchgesetzt und wurden dann sukzessive vernetzt. Zunächst wurde jedes Gerät als eigenständige Einheit konfiguriert, so daß auf allen hohe Leistung und Kapazität erforderlich war. Die notwendigen Investitionen dafür sind hoch, ebenso der Administrationsaufwand. Leistungsfähige Server wirken diesem Trend entgegen, da der Endbenutzer nur ein einfaches Gerät benutzt, das vernetzt ist und über einen oder mehrere Server zu einem leistungsfähigen Werkzeug wird. Das ist die Idee der *Netzcomputer*. Im Unterschied zu den *Terminals* von damals sind Netzcomputer mit graphischen Benutzerschnittstellen ausgestattet und können komplexe Anwendungen

ausführen. Java-Applets eignen sich besonders gut für diesen Zweck, da dazu auf Seite des Anwenders eine virtuelle Java-Maschine mit nur wenig Ressourcen genügt. Zum Beispiel hat Sun angekündigt, für alte PCs eine billige Umrüstung zu Netzcomputern anzubieten. Dies wurde zunächst für Herbst 1997 angekündigt [Sun97g], wird nun aber nicht vor Mitte 1998 verfügbar sein [Sun98c]. Netzcomputer haben aber auch gegenüber den sog. *X-Terminals* Vorteile, da die Netzbelastung geringer ist und die Server weniger leistungsstark sein müssen.

## 2.5 Verteilung von Software

Konventionelle Lösungen resultieren vielfach in monolithischen Applikationen. Diese sind für unterschiedliche Benutzerplattformen schwer zu verwalten. Außerdem ist ein hoher Administrationsaufwand notwendig, um neue Versionen oder Fehlerbehebungen an Benutzer zu verteilen. Mit Applets steht Benutzern automatisch eine Vielzahl von Plattformen zur Verfügung. Zusätzlich entfällt das Problem der Verteilung und der komplexen Administration.

Der Vorteil des Einsatzes von Java ist, daß Programme plattformunabhängig erstellt werden können. Sind Programme als Applets realisiert, dann müssen vorgenommene Änderungen nicht extra verteilt werden, da ohnehin bei jeder Benutzung das Applet neu über das Netz geladen wird. Um die Netzwerktransaktionen gering zu halten, wird man den Großteil der Funktionalität im Server ablegen. Es ist aber auch denkbar, anstelle eines Applets einen herkömmlichen Client zu entwickeln, der als eigenständige Applikation ausführbar ist.

# 3. Die Sprache

Java ist eine objektorientierte Programmiersprache, die sich konzeptionell nicht wesentlich von anderen Sprachen dieser Kategorie unterscheidet. Die Anlehnung an die Sprachen C und C++ erleichtert überdies den Umstieg für Programmierer, die bereits diese Sprachen verwenden. Eine Umschulung beschränkt sich dann in erster Linie auf die verfügbaren Bibliotheken.

Im folgenden diskutieren wir kurz charakteristische Konzepte der Sprache Java, um damit eine Beurteilung der Stärken und Schwächen zu ermöglichen.

In den letzten Jahren wurde in der Fachwelt häufig heftig pro und kontra mehrfacher Vererbung argumentiert. Java hat nur eine einfache Implementierungsvererbung, bietet dafür aber das Konzept der mehrfachen Schnittstellenvererbung (*interfaces*). Damit ist ein wichtiger Teilaspekt der Mehrfachvererbung abgedeckt. Die Voraussetzungen für die systematische Wiederverwendung von Code sind damit jedenfalls gegeben, da Implementierungsvererbung nach wie vor möglich ist.

Java bietet automatische Speicherfreigabe *(garbage collection)*, d.h. der Programmierer muß sich nicht explizit um die Freigabe von Objekten kümmern. Die Speicherfreigabe ermittelt automatisch Objekte, auf die nicht mehr verwiesen wird, und gibt ihren Speicherplatz frei. Die Speicherfreigabe kann explizit gestartet werden; ein Aufruf garantiert aber nicht, daß bestimmte Objekte dabei tatsächlich freigegeben werden [MSS96]. Echtzeitfähige Speicherfreigabe ist denkbar, in der Sprache Java aber nicht definiert und derzeit für Java-Systeme auch nicht verfügbar.

In Java gibt es Referenzen auf Objekte, nicht aber Zeiger bzw. Adressen, die beliebig verändert werden können (wie etwa mit dem Adreßoperator in C und C++). Das ist ein

Vorteil, weil damit eine umfangreiche Fehlerquelle eliminiert wird und diese Konzepte bei sauberer Programmierung nur in wirklich seltenen Fällen notwendig sind.

Applikationen sollten unter (fast) allen Bedingungen weiterlaufen und nicht einfach abstürzen. Die automatische Speicherfreigabe sowie der explizite Ausschluß von Zeigern bzw. Adressen und der *Zeigerarithmetik* tragen viel zur Robustheit von Java-Applikationen bei. Zusätzlich wird mit der Ausnahmebehandlung *(exception handling*) ein Mechanismus geboten, der dem Programmierer die Möglichkeit gibt, auf Ausnahmefälle in definierter Weise zu reagieren [MSS96]. Das schließt sowohl vordefinierte als auch vom Programmierer definierte Fehler- und Ausnahmesituationen ein. Die konsequente Anwendung der Ausnahmebehandlung in den Bibliotheken trägt wesentlich zu frühzeitigen Fehlererkennungen bei.

Java unterstützt leichtgewichtige Prozesse, sogenannte *Threads*. Damit ist ein nebenläufiger Ablauf von Teilbereichen eines Programms möglich, ohne den Verwaltungs- und Kommunikationsaufwand von separaten Prozessen in Kauf nehmen zu müssen. Die parallele Bearbeitung ist unter anderem für Client/Server-Systeme vorteilhaft, da beispielsweise Anfragen über das Netz parallel zu anderen Aufgaben bearbeitet werden können. Ohne diese Parallelität hätte der Programmierer eine manuelle Verwaltung der verschiedenen Teilaufgaben durchzuführen. Solche Trennungen lassen sich auch in anderen Bereichen vorteilhaft einsetzen [MSS96].

Zur besseren Strukturierung von Softwaressystemen gibt es in Java neben Klassen auch Pakete (*packages*). In Paketen können Klassen und Schnittstellen, die zu einem gemeinsamen Aufgabenbereich gehören, zusammengefaßt werden. Die Klassenbibliothek ist in mehrere Pakete unterteilt. Bei großen Systemen und deren arbeitsteiliger Entwicklung ist eine solche Unterteilung vorteilhaft. Pakete definieren einen Namensbereich und können zusätzlich hierarchisch gegliedert werden.

Für systemnahe Programmierung hat Java durch das Bestreben, plattformunabhängig, sicher und robust zu sein, Grenzen, die für die Systemprogrammierung zum Teil hinderlich sind. Da jedoch die Anbindung von Programmen anderer Programmiersprachen möglich ist (derzeit C, das direkt ausgeführt und nicht wie Java interpretiert wird, können bzw. müssen systemnahe oder plattformspezifische Teile ausgelagert werden. Eine solche klare Trennung ist aus softwaretechnischer Sicht ohnehin empfehlenswert. Auch äußerst zeitkritische Bereiche können in anderen Sprachen implementiert und integriert werden. Eine mögliche Unterbrechung, z.B. durch die automatische Speicherfreigabe, kann aber auch dadurch nicht ausgeschlossen werden.

# 4. Klassenbibliotheken

Objektorientierte Programmiersprachen alleine sind für die wirtschaftliche Entwicklung von Softwaresystemen nicht ausreichend. Es bedarf umfangreicher Klassenbibliotheken, um effizient die vielfältigen Anforderungen an heutige Software erfüllen zu können. Die Auswahl von Programmiersprachen wird häufig durch die Verfügbarkeit von Bibliotheken beeinflußt. Die Mächtigkeit von Programmiersprachen wird zunehmend weniger wichtig, da mit Bibliotheken flexibler auf verschiedene Bedürfnisse reagiert werden kann.

Für Java gibt es eine Klassenbibliothek, die in mehrere Pakete unterteilt ist [Fla96]. Man unterscheidet grundsätzlich zwischen den Kernpaketen (*standard* oder *core APIs*) und erweiterten Paketen (*extended APIs*). Diese Pakete haben plattformunabhängige Schnittstellen, aber teilweise plattformabhängige Implementierungen. Wenn ein Java-Programm von einer virtuellen Maschine interpretiert wird, dann werden die jeweiligen Pakete der

entsprechenden Implementierung für diese Plattform dazu verwendet. Neben den vordefinierten Paketen gibt es auch Pakete von anderen Herstellern, die, falls sie nicht ausschließlich in Java implementiert sind, nicht notwendigerweise auf allen Plattformen verfügbar sein müssen.

Zu den Kernpaketen gehören u.a. folgende Pakete:

- ***Ein-/Ausgabe*** (java.io)
  Zur Ein- und Ausgabe stehen eine Vielzahl von Klassen zur Verfügung. Diese bieten plattformunabhängige Schnittstellen (APIs) zu verschiedenen Ein-/Ausgabemedien. Man kann beispielsweise verschiedenste Datenströme auf unterschiedlichen Datenträgern ein-/ausgeben.
- ***Netzzugriffe*** (java.net)
  Zugriffe auf ein Netzwerk sind insbesondere für Applets von Bedeutung. Es gibt Klassen zum Zugriff auf beliebige Internetadressen (z.B. URLs), aber auch zur Implementierung von Verbindungen zwischen Clients und Servern.
- ***"Spracherweiterungen"*** (java.lang)
  In diesem Paket sind Klassen zusammengefaßt, die als Spracherweiterung gelten, z.B die Wurzelklasse Object, diverse Klassen zur Ausnahmebehandlung (*exceptions*), Wrapperklassen für elementare Datentypen (Integer, Boolean, etc.).
- ***Praktische Hilfsmittel*** (java.util)
  Es gibt zusätzliche nützliche Klassen, z.B. für Hashtabellen und Vektoren.
- ***Applets*** (java.applet)
  Zum einfachen Erstellen von Applets gibt es eine Klasse, die deren Grundgerüst darstellt. Der Applet-Programmierer kann sich auf das Wesentliche, die Funktionalität, konzentrieren.
- ***Fenstersystem*** (java.awt)
  Das Fenstersystem von Java wird AWT (*Abstract Windowing Toolkit*) genannt. Es bietet Klassen für Graphiken (z.B. Farben, Zeichensätze, Bilder), für Dialogelemente (z.B. Dialogboxen, Menüs, Knöpfe, Auswahllisten) und für das Layout von Dialogelementen (z.B. vertikale Anordnung von Dialogelementen) an.

Ab Version 1.2 bietet die Klassenbibliothek eine Reihe von Verbesserungen und Erweiterungen in verschiedenen Bereichen [Sun97i, San98, Sun98c]. Viele dieser Erweiterungen werden als *Java Foundation Classes* (JFC) subsumiert, z.B.:

- flexiblere Sicherheitsvorkehrungen
- mächtige Benutzerschnittstellenkomponenten (*swing*)
- geräteunabhängige, zweidimensionale graphische Dokumente (*Java 2D*)
- Applikationsübergreifender Datentransfer (*drag & drop*)
- CORBA-Anbindung (*Java IDL*)
- bessere Performance

Zusätzlich gibt es unterschiedlichste Entwicklungen, die für einen breiten Einsatz von Java interessant sind. Einige dieser Entwicklungen werden im folgenden kurz aufgeführt [Sun97j]. Die Liste ist keineswegs vollständig, sie soll nur einen Eindruck geben über verfügbare und geplante Bibliotheken in verschiedenen Anwendungsbereichen.

- ***Java Commerce***

zur Erstellung von sicheren Applikationen für den Handel, siehe z.B. [JSp98]

- ***Java Management***
  zur Erstellung von Applikationen zum System- und Netzwerkmanagement
- ***PersonalJava, EmbeddedJava***
  zur Verfügbarkeit von Java auf verschiedensten Plattformen, z.B.: Fernsehgeräte, Kopierer, Faxgeräte, Drucker, Handys [Sun97h]. PersonalJava und EmbeddedJava sind jeweils durch den Umfang der APIs definiert.
- ***Java Enterprise***
  Java Enterprise enthält mehrere Pakete zur Verwaltung von Unternehmensdaten, z.B.: *Enterprise JavaBeans* (Erweiterung der JavaBeans-Komponentenarchitektur mit besserer Unterstützung von kommerziellen Anwendungen), *Java Protected Domains* (Erweiterung des Sandbox-Modells mit feinerer Unterscheidung von Zugriffsrechten), *Java Input Method* (zum einfachen Erstellen von Software für den asiatischen Markt, Unterstützung der Sprachen Japanisch, Koreanisch und Chinesisch), *Java IDL* (zur einfachen Kommunikation mit CORBA)
- ***Java Media and Communication***
  Java Media and Communication unterstützt eine breite Palette von Medien und Kommunikationsmechanismen, z.B.: *Java 3D* (zum Erzeugen und Darstellen von geräteunabhängigen, dreidimensionalen graphischen Dokumenten mit Optimierungen für hohe Performance), *Java Advanced Imaging* (zum Bearbeiten von hochauflösenden Bildern), *Java Sound* (zum Bearbeiten, Mischen, etc. von Sound), *Java Animation* (zum Erzeugen und Bearbeiten von interaktiven Objekten), *Java Collaboration* (zur Zusammenarbeit über verschiedenste Netzwerke, Dokumentationsaustausch, etc.), *Java Speech* (zum Erkennen und Erzeugen von Sprache)

Von unterschiedlichen Herstellern werden auch Bibliotheken für verschiedene Aufgabenbereiche zur Verfügung gestellt, z.B. für numerische Berechnungen oder für Animationen [Jav97].

# 5. Standardisierung

Durch die Lizenzierungspolitik von Sun entstand quasi ein de-facto-Standard, da sich alle Lizenznehmer an Vorgaben von Sun halten müssen. Dieser de-facto-Standard ist allerdings an eine einzige Firma gebunden, obwohl Sun versucht, im Einklang mit anderen Firmen Weiterentwicklungen vorzunehmen. Nunmehr zeichnet sich ab, daß für Java auch ein internationaler Standard bei ISO (*International Organization for Standardization*) etabliert werden wird.

Um die Vorgehensweise der Standardisierung zu verstehen, geben wir kurz die drei von ISO gebotenen Möglichkeiten einer Standardisierung an:

1. Es wird ein Kommittee gebildet, welches einen Standard ausarbeitet.
2. Es wird ein bereits bestehender Standard von einem anderen Standardisierungsgremium übernommen, z.B. von ANSI.
3. Eine Firma oder Organisation wird als sogenannter *Publically Available Specification (PAS) Submitter* anerkannt und kann somit Standards einreichen, über die bei ISO, wie auch unter Punkt 1 und 2, abgestimmt wird und die dann als ISO-Standard erklärt werden können.

Im November 1997 wurde Sun als PAS Submitter anerkannt und kann nun verschiedene Teile von Java, d.h. die Sprache, die virtuelle Maschine, verschiende APIs, zur Standardisierung einreichen. Während Sprache und virtuelle Maschine schon weitgehend stabil sind, ist dies bei den APIs allerdings noch nicht der Fall. Wann es nun tatsächlich zu Standardisierungen kommen wird, läßt sich derzeit noch nicht sagen, eine erste wichtige Hürde dorthin ist allerdings durch die Anerkennung Suns als PAS Submitter genommen [Day97, Sun98a].

Um zu gewährleisten, daß mit unterschiedlichen Java-Bibliotheken entwickelte Software kompatibel ist, gibt es von Sun ein "Java Compatible"-Logo [Sun97b]. Dieses wird nur vergeben, wenn eine Applikation ausschließlich mit den Klassen der von Sun vordefinierten Bibliothek entwickelt wurde. Außerdem gibt es sogenannte *compliance tests* für die virtuelle Maschine [Bow97]. Für JDK 1.1 existieren über 5.000 solcher Tests.

In diesem Zusammenhang ist die Strategie von Microsoft interessant. In der von Microsoft angebotenen Bibliothek werden die von Sun geforderten Schnittstellen zur Verfügung gestellt. Darüber hinaus werden aber viele zusätzliche, für die Programmierung von Windows-Programmen nützliche Klassen angeboten (*Windows Foundation Classes, WFC*). Die Verwendung dieser zusätzlichen Klassen verhindert die Portabilität der damit entwikkelten Systeme. Entscheidend wird sein, ob Entwickler Plattformunabhängigkeit oder bestmögliche Unterstützung einer weit verbreiteten Plattform als wichtiger ansehen. Microsoft könnte bei entsprechend großer Beliebtheit der Microsoft-Klassen vom Java-Zug abspringen und (unter anderem Namen, z.B. J++) eigene Entwicklungen verfolgen. Aus eben diesem Grund ist es zwischen Sun und Microsoft zu einer gerichtlichen Auseinandersetzung gekommen[Nic98, Sun98b]. Sun wirft Microsoft vor, sich nicht an das Lizenzabkommen zu halten und damit die Portabilitätsbestrebungen um Java zu untergraben. Im Oktober 1997 wurde eine gerichtliche Klage eingebracht. In einer einstweiligen Verfügung vom 24. März 1998 wurde Microsoft untersagt, für ihre Produkte das "Java Compatible"-Logo zu verwenden.

# 6. Kosten

Für den Einsatz von Java zur Softwareentwicklung ist die Anschaffung einer Programmierumgebung, einer Java-Klassenbibliothek (nicht unbedingt notwendig, aber sinnvoll), sowie einer virtuellen Java-Maschine notwendig. Zur Entwicklung von Applets benötigt man zusätzlich ein System, das die Anzeige solcher Applets erlaubt, z.B. diverse WWW-Browser. Typischerweise umfaßt eine Java-Programmierumgebung Editor, Browser, Compiler und Debugger, sowie Werkzeugunterstützung zur Erstellung von graphischen Benutzerschnittstellen und Klassenbibliotheken. Die Preise liegen zwischen US$100 für einfache Umgebungen und US$3.000 für Umgebungen mit spezieller Unterstützung für Datenbankanbindungen und Client/Server-Lösungen.

Um Java-Applikationen ausführen zu können, braucht man für seine Plattform eine virtuelle Maschine und die Kernpakete der Bibliothek. Diese werden üblicherweise gemeinsam angeboten. Derzeit werden virtuelle Maschinen und die Kernpakete kostenlos von verschiedenen Herstellern abgegeben. Es ist auch zu erwarten, daß diese in zukünftigen Betriebssystemversionen schon integriert sein werden. Als Anbieter von Java-Applikationen bzw. Applets kann man davon ausgehen, daß der Benutzer die notwendige Infrastruktur bereits verfügbar hat. Daß in diesem Zusammenhang in Zukunft Kosten anfallen ist unwahrscheinlich, kann aber nicht ausgeschlossen werden.

# 7. Entwicklungsumgebungen und Werkzeuge

Es gibt bereits Entwicklungsumgebungen auf allen gängigen Hardwareplattformen, die unterschiedlich ausgereift und komfortabel sind. Der Werkzeugmarkt ist sehr dynamisch. Fast ständig gibt es neue Anbieter oder neue Versionen von bestehenden Werkzeugen, so daß es schwierig ist, ein aktuelles Bild von längerer Gültigkeit zu zeichnen. Die angeführten Produkte haben unterschiedliche Funktionalität. Eine entsprechende Evaluierung ist vor dem Einsatz von Java unumgänglich. Einen Vergleich von 15 verschiedenen Produkten findet man beispielsweise in [EK97].

Beim Erstellen von graphischen Benutzerschnittstellen wird von den meisten Herstellern von Entwicklungsumgebungen Unterstützung angeboten. Meist wird dazu graphisch die Benutzerschnittstelle erstellt und dann automatisch Java-Code erzeugt, der bei nachträglichen Änderungen neu generiert und ersetzt werden muß.

# 8. Sicherheit

Die Verteilung von Programmen über das Internet stellt besondere Anforderungen an die Sicherheit solcher Programme. Mögliche Attacken von über das Netz geladenen Programmen betreffen [Sur96]:

- ***Integrität***
  z.B. Löschen oder Verändern von Daten (Dateien oder Hauptspeicher), frühzeitiges Beenden von Prozessen
- ***Verfügbarkeit***
  z.B. Erzeugen von Prozessen mit hoher Priorität, Anfordern von Speicherplatz in großen Mengen
- ***Vertraulichkeit***
  z.B. Senden von Information über den Benutzer und/oder dessen Firma
- ***Belästigung***
  z.B. Darstellung obszöner Bilder, Abspielen von lästigem Lärm

Es gibt unterschiedliche Möglichkeiten, solche Attacken durchzuführen, z.B. mit trojanischen Pferden [Sur96]. Dabei führt ein Applet eine offizielle und eine versteckte Operation durch, z.B. sendet ein Applet unbemerkt Dateien über das Netz, während es eine Animation auf dem Bildschirm anzeigt oder mit dem Benutzer Schach spielt.

Vom Java-Compiler und von der *Java Virtual Machine* werden verschiedene Sicherheitsstufen realisiert, um solche Attacken zu verhindern. Es stehen mehrere Mechanismen zur Verfügung, um fehlerhaften und/oder inkorrekten Code zu erkennen und dessen Ausführung zu verhindern [GM95].

- ***Verifikation des Bytecodes***
  Wenn Javacode über das Internet geladen wird, kann nicht ausgeschlossen werden, daß dieser Code mit einem schadhaften (oder sogar "böswilligen") Java-Compiler übersetzt oder manipuliert wurde. Deshalb wird vor Ausführung von Code tatsächlich sichergestellt, daß keine unsicheren Aktionen ausgeführt werden, wie das "Verbiegen" von internen Zeigern, die Verletzung von Zugriffsrechten, implizite Typumwandlungen, das Aufrufen von Methoden mit falschen Parametern (Typ und Anzahl) und ein möglicher Überlauf des Kellerspeichers (Stacks).

- ***Überprüfungen beim Laden von Klassen***
  Beim Laden von Klassen wird sichergestellt, daß keine Gültigkeitsbereiche oder Zugriffsbeschränkungen verletzt werden. Insbesondere dürfen keine Verwechslungen bei Klassen mit gleichem Namen aber unterschiedlichen Quellen entstehen.
- ***Netzsicherheit***
  Um Sicherheitsangriffe von außen zu verhindern, können lokale Klassen, beispielsweise für Dateizugriffe nicht mit über das Netz geladenen Klassen dynamisch erweitert oder ersetzt werden. Außerdem gibt es Restriktionen für solche Klassen beim Zugriff auf lokale Dateien oder andere Rechner.
- ***Digitale Unterschriften***
  Um Hersteller bzw. Ursprung und Integrität von Javacode ermitteln zu können, werden digitale Unterschriften (Paare von privaten und öffentlichen Schlüsseln) erzeugt, verwendet und überprüft. Dazu können die Algorithmen DSA oder MD5 mit RSA verwendet werden.

Das Sicherheitspaket von Java (das Paket *java.security* und eine Klasse *SecurityManager* des Pakets *java.lang*) definiert Schnittstellen, zu denen verschiedene Implementierungen verwendet werden können, z.B. die von Sun. So hat der Benutzer die Freiheit, in diesem kritischen Bereich nach eigenem Belieben unterschiedliche Algorithmen und/oder Implementierungen (sofern verfügbar) auszuwählen. In einer Java-Implementierung wird das Sicherheitspaket daher separat installiert und kann auch dynamisch verändert werden (jedoch nicht von *remote applets*). Der Begriff *Sandbox Model* spiegelt die Tatsache wider, daß Applets durch diese Maßnahmen in ihren Aktivitäten eingeschränkt sind und praktisch nur in einem geschützten und abgeschlossenem Raum (Sandbox) operieren und außerhalb dieses Raumes keine Aktivitäten durchführen können.

Die oben angeführten Sicherheitsvorkehrungen sind für Anwendungen über ein Netzwerk gedacht. Bei herkömmlichen Softwaresystemen sind diese irrelevant. Mit Java entwikkelte Systeme sind damit nicht weniger sicher als mit anderen Programmiersprachen entwickelte. Durch Javas Speichermodell sind Systeme sogar weniger fehleranfällig und damit robuster als beispielsweise mit C oder C++ entwickelte Systeme.

# 9. Effizienz und Ressourcenverbrauch

Grundsätzlich bewirkt die Verwendung einer virtuellen Maschine mit Interpretation eines plattformunabhängigen Bytecodes längere Ausführungszeiten als die Ausführung von reinem Maschinencode (um einen Faktor bis zu 20). Da in Java aus Sicherheitsgründen viele Überprüfungen durchgeführt werden, die beispielsweise bei C nicht üblich sind, ist Javacode grundsätzlich um einen Faktor bis zu 2 langsamer, d.h. auch bei direkter Übersetzung. *Benchmarks* von verschiedenen Java-Implementierungen gibt es u.a. in [Pen97].

Durch den Einsatz von Übersetzern, die während der Laufzeit eines Java-Programms dessen Bytecode in Maschinencode übersetzen (*Just-In-Time-Compiler*), können Effizienzeinbußen stark verringert werden. Man kommt damit schon in die Liga von guten C++-Übersetzern [Man98]. Mit dem Projekt *Hotspot* will Sun in punkto Performance durch sog. dynamische Übersetzung noch schneller werden. Bei dynamischer Übersetzung werden Laufzeitinformationen zur Optimierung eingesetzt, wodurch Verbesserungen erzielbar sind, die selbst optimierenden, statischen Übersetzern nicht möglich sind [Arm98]. Im Sommer 1998 soll eine erste Version von *HotSpot* verfügbar sein.

Wenn mehrere Applets von einem WWW-Browser angezeigt bzw. ausgeführt werden, dann geschieht dies üblicherweise mit nur einer virtuellen Maschine. Die Applets laufen

in verschiedenen *Threads*. Gelangen mehrere Applikationen gleichzeitig zur Ausführung, dann hat jede davon eine eigene virtuelle Maschine, was einen hohen Ressourcenverbrauch verursachen kann. Durch die Konzepte von Java ist es aber auch möglich, Applikationen in verschiedenen *Threads* mit nur einer virtuellen Maschine auszuführen.

# 10. Anbindung von C-Code

Viele Vorteile von Java kommen nur zur Geltung, wenn ein System ausschließlich in Java implementiert wurde, z.B. Plattformunabhängigkeit. Dieses Ziel wird nicht immer realisierbar sein, da beispielsweise eine Applikation speziell für eine Plattform entwickelt wird und/oder bereits in anderen Sprachen entwickelter Code wiederverwendet werden soll.

Zur Anbindung von nicht in Java implementiertem Code über eine offene Schnittstelle wird das *Java Native Interface* (JNI) vorgegeben [Sun96]. Diese Schnittstelle wurde von Sun in Absprache mit Lizenznehmern erstellt. Der Programmierer kann somit eine einzige Schnittstelle für die Anbindung von Fremdcode auf allen Plattformen verwenden. Bei zeitkritischem und plattformabhängigem Code kann aber grundsätzlich auf plattformabhängige Schnittstellen zurückgegriffen werden.

Mit dem *Java Native Interface* sind Aufrufe in beide Richtungen möglich, d.h. man kann aus Java C-Code aufrufen und auch von C aus Methoden von Java benutzen. Damit die automatische Speicherbereinigung keine Objekte aufräumt, auf die in Nicht-Java-Code noch verwiesen wird, unterscheidet man zwischen lokalen und globalen Referenzen auf solche Objekte. Allerdings liegt es (wie in C und C++) in der Verantwortung des Programmierers, daß auf keine schon aufgeräumten Objekte zugegriffen wird.

# 11. Entwicklungszeiten

Konkrete Aussagen bzw. Vergleiche von Entwicklungszeiten (Einarbeitung, Übersetzungszeiten, etc.) liegen derzeit nicht vor. Wir gehen aber davon aus, daß nach erfolgter Einarbeitung die Produktivität, beispielsweise gegenüber C++, um einiges höher sein wird, da viele Fehlerquellen von vornherein ausgeschlossen sind, und so das Suchen von daraus resultierenden Fehlern wegfällt. Erste eigene Erfahrungen mit Java haben diese Vermutung bestätigt. Produktivitätssteigerungen um einen Faktor 2 bis 4 verbunden mit der Erstellung von Code höherer Qualität scheinen durchaus realistisch zu sein [Orc97].

# 12. Check-Liste für den Einsatz von Java

Die Weiterentwicklung von Java schreitet rasant voran, so daß jedes Dokument darüber in wenigen Wochen veraltete Informationen enthalten kann. Ein ständiges Beobachten der unterschiedlichen Entwicklungen ist deshalb ratsam. Für den Einsatz von Java in der Produktentwicklung ist wichtig, daß eine gewisse Stabilität bei Sprachdefinition, virtueller Maschine, Klassenbibliothek, etc. eintritt, daß vernünftige Werkzeuge zur Softwareentwicklung auf der gewünschten Plattform zur Verfügung stehen, daß eine Weiterentwicklung und Betreuung von Sprache und Werkzeugen gegeben ist und daß die Lizenzkosten gering sind. Einige dieser Punkte sind schon relativ klar, während andere vor einem tatsächlichen Einsatz von Java noch konkret geprüft werden sollten. Im folgenden wollen wir diese Punkte kurz erläutern:

- ***Stabilität von Sprache und Kernbibliothek***
  Die Stabilität von Java wird bereits jetzt für eine Produktentwicklung in kleinerem oder mittlerem Umfang als ausreichend eingeschätzt. Sobald ein internationaler Stan-

dard vorliegt, sind wohl auch die letzten Zweifel ausgeräumt. Auf den Standard muß nicht notwendigerweise gewartet werden, da ohnehin bereits ein de-facto-Standard vorliegt, der sich voraussichtlich nicht wesentlich vom späteren, tatsächlichen Standard unterscheiden wird. Aber man sollte verfolgen, wie die Standardisierung vor sich geht und welche Teile wann als Standard zu erwarten sind.

- ***Werkzeuge***
  Es gibt bereits eine Unmenge von Werkzeugen, und es kommen fast täglich neue hinzu. Dadurch ist es schwierig, ein genaues Bild der aktuellen Werkzeugsituation darzustellen. Umgekehrt läßt diese Situation auch Rückschlüsse auf die Bedeutung von Java sowie deren zukünftige Unterstützung durch verschiedene Firmen zu. Für die Entwicklung von Produkten in Java ist jeweils die aktuelle Situation auf dem Werkzeugmarkt zu eruieren und festzustellen, ob die Unterstützung für die Entwicklung auf einer bestimmten Plattform ausreichend ist. Zum Beispiel sind uns Testwerkzeuge derzeit nicht bekannt. Außerdem sind Java-Werkzeuge, im Unterschied zu Java selbst, nicht immer plattformunabhängig, da sie häufig als Weiterentwicklungen bestehender Werkzeuge mit anderen Sprachen implementiert werden.

- ***Bibliotheken***
  Ob eine wirtschaftliche Softwareentwicklung mit Java möglich ist, wird je nach Anwendungsgebiet nicht zuletzt von der Verfügbarkeit von Klassenbibliotheken auf diesem Gebiet abhängen. Derzeit steht eine Fülle von Entwicklungen an. Es ist festzustellen, ob geeignete Bibliotheken für die geplanten Entwicklungen zur Verfügung stehen werden.

- ***Weiterentwicklung***
  Es ist davon auszugehen, daß Java von Sun weiterentwickelt und gewartet wird. Eine diesbezügliche Absichtserklärung wurde, unter anderem an ISO im Ansuchen als PAS Submitter anerkannt zu werden, abgegeben [Sun97a]. Gleiches gilt es für Werkzeuge und Bibliotheken abzuklären, da diese nicht notwendigerweise von Sun stammen.

- ***Lizenzkosten***
  Um Softwaresysteme in Java zu entwickeln, genügt es, eine der erhältlichen Programmierumgebungen zu erwerben. Diese enthalten in der Regel die wichtigsten Werkzeuge zur Softwareentwicklung wie Compiler, Debugger, eine virtuelle Maschine und eine Klassenbibliothek. Oft können in die Umgebungen auch eigene Werkzeuge integriert werden. Die Lizenzkosten hängen somit primär von der Auswahl der Werkzeuge ab. Höhere Lizenzkosten entstehen nur, wenn beabsichtigt ist, Java-Quellcode von Sun in eigene kommerzielle Entwicklungen zu integrieren. In diesem Fall ist eine Quellcodelizenz erforderlich und Sun muß kontaktiert werden, um nähere Informationen über Bedingungen und Lizenzgebühren zu erhalten.

- ***Abhängigkeiten***
  Obwohl Java von Sun in Zusammenarbeit bzw. in Abstimmung mit vielen anderen Firmen weiterentwickelt wird, ist eine gewisse Abhängigkeit von Sun nicht zu vermeiden. Sobald ein internationaler Standard vorliegen wird, ist diese nicht mehr unmittelbar gegeben. Eine Mitarbeit bei der Standardisierung könnte unter diesem Aspekt durchaus überlegenswert sein.

- ***Komponenten***
  Komponentenbasierte Softwareentwicklung steckt noch in den Kinderschuhen. Hier ist ein Abwarten vor einer breit angelegten Entwicklung von Beans sicherlich sinnvoll, um zu sehen, wie andere Firmen reagieren und wie sich die Technologie weiterentwik-

kelt. Es sei denn, man strebt aus firmenstrategischen Gründen Technologieführerschaft bei komponentenbasierter Softwareentwicklung an.

- ***"Geheimnisprinzip"***
  Üblicherweise werden Applikationen im Objektcode ausgeliefert. Von diesem Objektcode ist nur sehr schwer auf den ursprünglichen Quellcode zu schließen. Damit können Firmen sicher sein, daß sie durch Auslieferung ihrer Softwaresysteme keinen Wettbewerbsvorteil aus der Hand geben. In Javas Bytecode sind beinahe sämtliche Informationen über den Quellcode vorhanden. Man kann beispielsweise Werkzeuge verwenden, die zumindest Quellcode-Bezeichner unkenntlich machen und damit das Verstehen dieses Quellcodes wesentlich erschweren. Manche Programmierumgebungen bieten auch die Möglichkeit, neben Javacode auch Objektcode für eine bestimmte Plattform zu generieren. Dann ist zwar die ausgelieferte Software nicht mehr plattformunabhängig, man kann aber sein System für unterschiedliche Plattformen ausliefern, sofern dies von der Entwicklungsumgebung unterstützt wird. Es gilt abzuklären, ob die Herausgabe des Quellcodes tragbar ist bzw. welche Möglichkeiten man im konkreten Fall hat, dies zu vermeiden. Bei manchen Client/Server-Lösungen beispielsweise kann es genügen, nur den Client nach außen zu geben.

# 13. Das Komponentenmodell JavaBeans

JavaBeans ist das Komponentenmodell von Java und wurde 1997 (mit JDK1.1) eingeführt. Unter einem Bean versteht man eine wiederverwendbare Softwarekomponente mit der wesentlichen Eigenschaft, daß sie visuell (interaktiv) mit einem sogenannten *Builder* manipuliert werden kann. Ein Builder kann beispielsweise ein einfaches Layouting-Werkzeug oder aber auch eine umfassende visuelle, komponentenbasierte Programmierumgebung sein. JavaBeans unterstützen typischerweise folgende Konzepte:

- ***Properties***:
  "Virtuelle Variablen" zur interaktiven Einstellung bzw. zur Kommunikation zwischen Beans.
- ***Events***:
  Ereignisse als Kommunikationsmittel zwischen Beans.
- ***Introspection***:
  Schnittstelleninformation eines Beans wird implizit oder explizit angeboten.
- ***Persistence***:
  Bean-Zustand kann gespeichert und wieder eingelesen werden.
- ***Customization***:
  Einstellung von Bean-Eigenschaften durch universelle od. spezielle Editoren.

Ein Bean kann eine beliebige Java-Klasse sein. Die Schnittstelle eines Beans wird durch eine Menge von Eigenschaften, Methoden und Ereignissen definiert. Ein Bean kann aus vielen Klassen und persistenten Objekten zusammengesetzt sein. Komplexe Beans werden typischerweise in einem komprimierten Archiv (JAR) zusammengefaßt. Die Schnittstelle eines Beans kann aus der Klassenschnittstelle, die in diesem Fall bestimmten Koventionen genügen muß, abgeleitet werden. Es kann aber auch durch eine sogenannte BeanInfo-Klasse eine explizite Beschreibung eines Beans angeboten werden. Eine wichtige Variante zur Bean-Kommunikation sind sogenannte *bound properties*. Das sind Eigenschaften, über deren Änderungen andere Komponenten verständigt werden. Ein Überblick

über eine Fülle von Dokumentation mit weiteren Details zu JavaBeans ist in [Sun97l] zu finden.

# 14. Erfahrungsbericht zu JavaBeans-Komponenten

Im Projekt SimBeans wurde an der Johannes Kepler Universität Linz ein einfacher Prototyp einer Bausteinsammlung für Simulationsanwendungen entwickelt. Damit sollte das Komponentenmodell JavaBeans sowie die dazu benötigten Werkzeuge in einer konkreten Anwendungsdomäne, nämlich für diskrete Simulationen, evaluiert werden. Bei der komponentenbasierten, ereignisgesteuerten Simulation versucht man, Simulationssysteme aus einfachen Grundbausteinen zusammenzubauen. Neben den Simulationsbausteinen im engeren Sinn bedarf es dabei einer Sammlung von Visualisierungs-, Animations- und Auswertungskomponenten. Gesteuert werden die Anwendungen im allgemeinen durch eine Abfolge von Ereignissen, die in einer zentralen Liste verwaltet werden.

Eine komponentenbasierte Entwicklungsumgebung für Simulationsanwendungen sollte den Entwickler in die Lage versetzen, interaktiv Bausteine auszuwählen und in einer Arbeitsfläche anzuordnen. Die Komponenten müssen im nächsten Arbeitsschritt geeignet in Beziehung gesetzt werden, sodaß Signale und Daten zwischen den Bausteinen ausgetauscht werden können. Weiters soll die Voreinstellung von Parametern für das Simulationsmodell bzw. die Visualisierung durch das Werkzeug gut unterstützt werden. Dieses Komponentensystem soll einfach zur Ausführung gebracht und als neue Anwendung bzw. als neuer Baustein gespeichert werden können. Im Falle, daß ein derartiges Komponentensystem wiederum als Baustein in einem anderen Kontext einsetzbar sein soll, bedarf es einer Möglichkeit, die Schnittstelle des Bausteins exakt zu definieren. Dieses Szenario der Anwendungsentwicklung durch Montage von vorgefertigten Komponenten ist eine Idealvorstellung, die nicht ausreichen wird, um komplexe praxistaugliche Simulationsanwendungen zu erstellen. Vielmehr ist es unverzichtbar, durch manuelle Programmierung neue anwendungsspezifische Komponenten zu realisieren bzw. vorhandene Bausteine zu modifizieren und zu erweitern. Die kombinierte Arbeitsweise durch Komponentenmontage einerseits und durch konventionelle Programmierung andererseits scheint uns geradezu typisch für die Entwicklung von Simulationsanwendungen im speziellen sowie für Softwareentwicklung im allgemeinen. Diesem Umstand muß daher in besonderer Weise Rechnung getragen werden, d.h. der vom Werkzeug generierte Code muß gut lesbar und unter Umständen auch manuell modifizierbar sein. Visuelle Programmierwerkzeuge, die umfangreichen, schwer lesbaren und unveränderlichen Code generieren, eignen sich daher schlecht für eine kombinierte Arbeitsweise.

Auch in anderen Projekten wurde offensichtlich, daß schwerwiegende Probleme der visuellen Programmierung zur Zeit noch ungelöst oder prinzipiell unlösbar sind [Sch98]. Schon bei einigen Dutzend visueller Links geht der Überblick verloren und die Korrektheit aller Kopplungen kann nur mehr schwer sichergestellt werden. Ein weiteres ernsthaftes Problem stellt die Übergabe von Daten zwischen Komponenten mit unterschiedlichen Parametertypen dar. Aufgrund der unverzichtbaren Notwendigkeit einer kombinierten Arbeitsweise und den schlechten Erfahrungen mit visuellen Programmierwerkzeugen (vgl. Erfahrungen mit VisualAge), entschieden wir uns für eine rein manuelle Programmierung der Komponentenkopplung.

## 14.1 Erfahrungen mit SNiFF+ und der BeanBox

Nach dem Entwurf einer ersten Systemarchitektur wurden erste Beans entwickelt, um Erfahrungen in der Manipulation und der Montage der Komponenten zu sammeln. Die

Entwicklung der ersten Beans erfolgte mit Suns JDK und der Programmierumgebung SNiFF+ von TakeFive. Die ersten Tests der Beans erfolgten mit der ebenfalls von Sun entwickelten BeanBox. Entwicklung und Montage der Beans wurde somit mit voneinander unabhängigen Werkzeugen durchgeführt.

SNiFF+ ermöglichte ein bequemes Arbeiten in bereits von C++-Projekten bekannter Souveränität. Die BeanBox kann bestenfalls als rudimentärer Ansatz eines Komponentenwerkzeuges betrachtet werden und dient in erster Linie zur Demonstration der Konzepte von JavaBeans. Als Entwicklungswerkzeug ist die BeanBox untauglich.

## 14.2 Erfahrungen mit VisualAge

Als nächstes Werkzeug setzten wir VisualAge von IBM zur Beanmontage ein, da dort besondere Beans-Unterstützung versprochen wird und auch der Einsatz visueller Programmiertechniken möglich ist. VisualAge stellt sich wie eine Smalltalk-Umgebung dar und ist auch in IBM-Smalltalk implementiert. Die Quelltexte der Klassen werden nicht in einzelnen Quelldateien sondern in einem sogenannten *repository* verwaltet. Die diversen Browser sind nach einer gewissen Einarbeitungszeit gut zu bedienen. Sie bieten eine (beinahe schon verwirrende) Vielzahl von Darstellungsmöglichkeiten und gute Suchfunktionen an. Auch die Performance ist anfänglich akzeptabel (auf PPro200 mit 64 MB). Die Installation der Beans war allerdings ein sehr aufwendiger und mühsamer Prozeß, da VisualAge zu jedem Bean eine BeanInfo-Klasse und ein bzw. mehrere Icons benötigt. Die Änderung von Beans und ihre Neuinstallation im Werkzeug ist ein relativ aufwendiger Vorgang, der gerade in der Anfangsphase häufig notwendig und zermürbend war.

Die Probleme von VisualAge liegen im Bereich der Codegenerierung beim Arbeiten mit dem Montagewerkzeug (visual composition tool). Es wird jeweils eine neue Klasse erzeugt, die viele umfangreiche Methoden für jede Komponente und jede Beziehung enthält. Einige der generierten Methoden werden bei jeder Änderung neu generiert. Bei den meisten Methoden kann der Entwickler an mit Kommentaren markierten Stellen eigenen Code einfügen. Sämtliche Ereignisse und Eigenschaftsänderungen werden von generierten Methoden verarbeitet. Durch die pompöse Codegenerierung ist es mühsam und aufwendig, manuell erstellten Code zu integrieren.

Weiters sind die Möglichkeiten zur Beschreibung komplexer Kopplungsstrukturen zwischen Beans auf der visuellen Ebene nicht ausreichend bzw. schwerfällig. So sind etwa unverzichtbare Typumwandlungen zwischen Ereignisdaten und Methodenparametern visuell nicht möglich. Insgesamt wurde das Arbeiten mit VisualAge im Laufe der Zeit immer unbefriedigender. Ab etwa 100 Klassen brach die Systemleistung dann völlig zusammen.

## 14.3 Erfahrungen mit VisualCafe

VisualCafe von Symantec stellt verglichen mit VisualAge eine eher traditionelle, dateibasierte Entwicklungsumgebung mit integriertem GUI-Builder und Debugger dar. Hinsichtlich des Komforts für den Entwickler ist VisualCafe deutlich spartanischer; es bietet nur bescheidene Browsing-Unterstützung an. Der Vorteil von VisualCafe liegt im vergleichsweise geringen Ressourcenverbrauch, in den schnellen Turn-around-Zeiten und in den schnelleren Programmausführungszeiten (durch JIT-Compiler). Beim ersten Versuch, die Beans in VisualCafe zu installieren, mußten wir erfahren, daß die angeboten Möglichkeit, Beans in Archivdateien (JAR files) zu packen, in der Version 2.0 praktisch nicht zu gebrauchen ist. Erst in der Version 2.1 gab es eine vernünftige Unterstützung für die Erzeu-

gung von Archivdateien und die Installation eigener Beans in die sogenannte *Component Library*. Nach weiterem Ärgernis durch ein fehlerhaftes Bean, wobei ein Laufzeitfehler die gesamte Programmierumgebung zum Absturz brachte, war das Arbeiten dann relativ problemfrei. Wir konnten die Beans weiterentwickeln und diverse Verbesserungen an der Gesamtarchitektur vornehmen.

Im Gegensatz zu VisualAge generiert VisualCafe relativ wenig Code. Probleme ergaben sich dadurch, daß der Name eines Beans wie er im Customizer angezeigt bzw. geändert werden kann, nicht im Bean selbst gespeichert wird. Er wird lediglich als Variablenname im generierten Code verwendet. Die von uns geplante Eigenschaft der Beans, sich durch ihren Namen zu visualisieren, konnte dadurch nicht realisiert werden. Ihre Visualisierung beschränkt sich somit auf die Anzeige ihres Klassennamen. Eine weitere Schwachstelle von VisualCafe besteht darin, daß persistente Daten beim Anlegen neuer Beans nicht kopiert werden (entgegen der JavaBeans-Spezifikation). Die Unterstützung bei der Generierung von BeanInfo-Klassen ist ebenfalls ungenügend. Zwar existiert ein sogenannter BeanWizard. Dieser kann aber keine BeanInfo für bereits existierende Klassen erzeugen und ist zudem durch lange Sequenzen von modalen Dialogen schwerfällig zu benutzen. Eine wichtige, offene Frage besteht derzeit noch darin, inwieweit es möglich ist, den Customizer von VisualCafe zu veranlassen, Eigenschaften mit komplexen Klassentypen einzustellen und dann die zusätzlichen, geschachtelten Eigenschaften im Customizer darzustellen.

## 14.4 Erfahrungen mit JavaBeans

Java Beans stellt als leichtgewichtiges, plattformunabhängiges Komponentenmodell eine attraktive Softwarekomponententechnologie dar. Trotz der primär positiven Erfahrung bleiben einige ernsthafte Kritikpunkte, nicht zuletzt an den verfügbaren Werkzeugen, die wir im folgenden kurz diskutieren:

- Die meisten Werkzeuge verwenden Icons zur Darstellung von Beans. Damit dies funktioniert müssen allen Beans entsprechende BeanInfo-Klassen, in denen u. a. die Dateinamen der Icons abgelegt sind, zugeordnet sein. Hier wäre eine einfache Konvention für eine optionale Beanmethode nützlich, damit Beanklassen selbst Icons zur Verfügung stellen können. Das Erstellen von BeanInfo-Klassen ohne geeignete Werkzeugunterstützung ist mühsam, insbesondere wenn noch viele Änderungen an der Schnittstelle erforderlich sind.
- Es ist restriktiv davon auszugehen, daß Beanwerkzeuge ausschließlich visuelle Manipulation unterstützen. Bei textueller und auch bei visueller Montage sind Beannamen eine wichtige Information, ohne die eine Zuordnung von Programmcode zu visueller Repräsentation unmöglich ist. Beannamen sind in der Beansspezifikation kein Thema! Lediglich die Klasse Component, von der visuelle Beans abgeleitet sein sollen (müssen?), definiert einen Namen als Eigenschaft. VisualCafe beispielsweise nutzt diese Eigenschaft nicht, sondern verwaltet intern einen eigenen Namen.
- In der Beansspezifikation sind keinerlei Angaben über hierarchisch strukturierte Beans enthalten. Es gibt derzeit keine definierten Methoden, um enthaltene oder umgebende Beans anzusprechen. Nur die Klasse Container, die visuelle, zusammengesetzte Objekte definiert, besitzt derartige Methoden. Erst in der nächsten JDK-Version (1.2) wird es entsprechende Interfaces geben (*BeanContext* und *BeanContextChild*).
- Die Art und Weise der Codegenerierung der Werkzeuge ist in keiner Weise eingeschränkt. Dies hat zur Folge, daß unterschiedliche Ansätze von verschiedenen Werk-

zeugherstellern verfolgt werden. Für den Entwickler ist es oft schwierig bis unmöglich, auf ein neues Werkzeug umzusteigen, weil dann Klassen mit generierten Codeteilen visuell nicht mehr weiterbearbeitet werden können.

- Die Spezifikation von JavaBeans stellt komponentenspezifische Event- und Listenerklassen als wichtiges Kommunikationsmittel dar. Demnach sollte nahezu jede Komponente mindestens eine zugeordnete Eventklasse und ein passendes Listenerinterface definieren. (Vielfach brauchen Beans auch mehrere Eventklassen und Listenerinterfaces.) Dies führt zu einer explosionsartigen Vermehrung von Datentypen, die kaum Funktionalität besitzen und die Flexibilität der Komponenten nicht wirklich verbessern. Nach unseren Erfahrungen genügt in der Regel ein universeller Ereignistyp (*PropertyChangeEvent*) und ein zugehöriges Listenerinterface (*PropertyChangeListener*). Damit werden zwar nur wenige Daten transportiert, sie reichen aber aus, um die entsprechenden Aktionen auszulösen.

Eines der Hauptprobleme bei der Erstellung von JavaBeans liegt darin, die Schnittstellen der Komponenten sowie deren Interaktion festzulegen. In einer ersten Version von SimBeans gingen wir daran, die Kommunikation über die Verteilung von Ereignissen durchzuführen. Dies stellte sich jedoch später als unbefriedigend heraus, da dabei die Kopplung zu gering ist, um einen komplexeren Datenaustausch durchzuführen. In einer zweiten Version änderten wir den Kommunikationsmechanismus und koppelten die Komponenten über deren Eigenschaften. Als zentrales Problem entpuppte sich dabei die Festlegung der verschiedenen Schnittstellen. Sind diese einmal festgelegt, so können relativ einfach neue Komponenten hinzugefügt werden. Der *InfoBus* sollte in diesem Zusammenhang ebenfalls eine Besserung bringen [Sag98].

Unseres Erachtens müssen für verschiedene Anwendungsdomänen eine Reihe von Schnittstellen vordefiniert, am besten standardisiert werden, um eine hohe Wiederverwendung von Komponenten sowie einen Komponentenaustausch unter verschiedenen Herstellern zu ermöglichen. Zwar können auch voneinander völlig unabhängige Komponenten miteinander gekoppelt werden, für anspruchsvolle Anwendungen reicht das aber oft nicht aus.

# 15. Resümee

In diesem Abschnitt werden wir die wichtigsten Aspekte der Java-Technologie kurz bewerten. Diese Bewertung ist nur zum Teil durch eigene Erfahrungen abgesichert und stützt sich auch auf die Literatur. Durch die starke Dynamik der Entwicklungen um Java handelt es sich naturgemäß nur um einen Schnappschuß.

- ***Die Sprache Java***
  Java als Programmiersprache ist ausgereift und wird in absehbarer Zeit nur mehr geringfügigen Erweiterungen unterworfen sein. Die wichtigsten Sprachelemente sind klar, zweckmäßig, sicher und elegant definiert. Der relativ geringe Sprachumfang und die Anlehnung an C/C++ machen die Sprache zusätzlich attraktiv.
- ***Die virtuelle Maschine***
  Die Leistungsfähigkeit der verbreiteten virtuellen Maschinen läßt noch immer zu wünschen übrig. Erst 1998 werden JIT-Compiler zur Standardtechnologie. Diese sind zum Teil noch mit Qualitätsmängeln behaftet. Inwieweit sich Java-Programme in der Ausführungszeit optimiertem Maschinencode annähern werden können, muß erst die Praxis zeigen.
- ***Die Java-Klassenbibliotheken***

Die aktuelle JDK-Version (1.1.6) ist bezüglich Mächtigkeit und Reifegrad als minimal hinreichend zu bewerten. Man kann damit ohne Probleme kleinere Anwendungen entwickeln. Für große Projekte benötigt man wesentlich mehr Unterstützung mit höherer Zuverlässigkeit und Performance. Mit der nächsten Version (1.2) ist zu erwarten, daß jedenfalls hinsichtlich des Funktionsumfangs die gröbsten Schwachstellen eliminiert sein werden.
Die Windows-basierten Alternativbibliotheken von Microsoft liegen erst in Beta-Version vor und wurden unsererseits noch nicht bewertet.
Die Verfügbarkeit von Spezialbibliotheken ist zur Zeit noch gering. Vielfach sind Bibliothekserweiterungen noch wesentlich dynamischer in ihrer Entwicklung als die Kernbibliothek.

- ***Die Entwicklungswerkzeuge***
  Verfügbarkeit und Reifegrad von Entwicklungsumgebungen kann für kleine bis mittlere Projekte als hinreichend bezeichnet werden (jedenfalls für Windows-Plattformen). Das große Angebot mit häufigen Updates erschwert aber eine Auswahl.
- ***JavaBeans-Komponenten***
  Das plattformunabhängige Komponentenmodell weist viele attraktive Merkmale auf; es ist aber auch noch mit Schwachstellen behaftet. Inwieweit sich JavaBeans als Alternative zum ActiveX-Modell behaupten wird, wird sich noch zeigen. Zur Realisierung von GUI-Widgets sind die Konzepte von Beans weitgehend ausreichend.

# 16. Referenzen


[Arm98] Eric Armstrong. *HotSpot: A new breed of virtual machine*, JavaWorld, http://www.javaworld.com/javaworld/jw-03-1998/jw-03-hotspot.html, March 1998.

[Ber98] Klaus Berg. *Bau und Integration von Softwarekomponenten am Beispiel von JavaBeans*, JavaSpektrum 1/98, Januar/Februar-Ausgabe, SIGS-Publications 1998.

[Bow97] Barry D. Bowen. *Getting There With JDK and JavaBeans*. March 10 1997. http://www.javasoft.com:80/features/1997/feb/jdkbeans.html.

[Bro97] Dieter Brors. *Internet-Office: Office-Pakete zunehmend als Application Server*. c't magazin für computer technik, pages 76 ff., May 1997. http://www.ix.de/ct/art_ab97/9705070.

[Chu97] Eric Chu. *Raising the bar with JDK 1.1: An interview with Eric Chu*. JavaWorld, March 1997. http://www.javaworld.com/javaworld/jw-04-1997/jw-04-jdk.html.

[Day97] Bill Day, *The Impact of Java Standardization*, JavaWorld, http://www.javaworld.com/javaworld/jw-12-1007/jw-12-iso.commentary.html, December 1997.

[Die97] Stephan Diehl. *Java & Co; Die Sprachen des Webs: HTML, VRML, Java, JavaScript*. Addison-Wesley, 1997.

[EK97] Claus Eikemeier, Christian Kirsch. *Nicht die Bohne, fünfzehn Java-Entwicklungsumgebungen*. iX, pp. 48-61, November 1997.

[Fla96] David Flanagan. *Java in a Nutshell*. 2nd Edition. O'Reilly & Associates, Inc., 1997.

[GM95] James Gosling, Henry McGilton. *The Java programming language*. OOPSLA '95 Tutorial Notes, Austin, TX, 1995.

[Jav97] JavaWorld. *Developer Tools Guide*. http://www.javaworld.com/javaworld/common/jw-toolstable.html, 1997.

[Jav98] JavaSoft, *Certified 100% Pure JavaTM Applications*, http://www.javasoft.com:80/100percent/latestlist.html, February 1998.

[JSp98] JavaSpektrum 2/98, März/April-Ausgabe, Schwerpunkt *Electronic Commerce*, SIGS Publications 1998.

[Man98] Carmine Mangione, *Performance tests show Java as fast as C++*, JavaWorld, http://www.javaworld.com/jw-02-1998/jw-02-jperf.html, February 1998.

[MSS96] Stefan Middendorf, Reiner Singer, Stefan Strobel. *Java Programmierhandbuch und Referenz*. dpunkt, 1996.

[Mur97] Kieron Murphy. *The pros and cons of JDK 1.1: JavaSoft, Java developers describe the key benefits and shortcomings of the long-awaited Java software toolkit -- and discuss the pain of moving from 1.02*. JavaWorld, March 1997. http://www.javaworld.com/javaworld/jw-04-1997jw-04-jdk.html.

[Nic98] James Niccolai. *Sun, Microsoft prepare for court battle over Java logo*. News (InfoWorld), http://www.infoworld.com/cgi-bin/displayStory.pl?980223.ehlogo.htm, Feb. 23, 1998.

[Orc97] Dave Orchard. *A look at Java's future*, JavaWorld, Vol. 2, Issue 6, June 1997. http://www.javaworld.com/javaworld/jw-06-1997/jw-06-javafuture.html

[Pen97] Pendragon Software Corporation. *Java Performance Report*, April 1997. http://www.webfayre.com/pendragon/jpr/jpr049702.html

[Sag98] A. Sagar & J. Sigler: *Taking a Ride on the Infobus*, Java Developer's Journal, Vol 3, Issue 2, SysCon Publications 1998.

[San98] John Sands. *JFC: An in-depth Look at Sun's Successor to AWT*. JavaWorld, http://www.javaworld.com/javaworld/jw-01-1998/jw-01-jfc.html, January 1998.

[Sca98] Kane Scarlett. *Get Ready to Swing (1.0)*. JavaWorld, http://www.javaworld.com/javaworld/jw-03-1998/jw-03-swinggui.html, March 1998.

[Sch98] Stefan Schiffer. *Visuelle Programmierung - Grundlagen und Einsatzmöglichkeiten*, Addison-Wesley-Longman, 1998.

[Ste98a] Uwe Steinmüller: *Java Beans in der Praxis, Teil 1: "Konfektioniert"*, iX Magazin fur professionelle Informationstechnik, Ausgabe März 1998, Verlag Heinz Heise GmbH & Co KG, 1998.

[Ste98b] Uwe Steinmüller: *Java Beans in der Praxis, Teil 2: "Diashow"*, iX Magazin fur professionelle Informationstechnik, Ausgabe März 1998, Verlag Heinz Heise GmbH & Co KG, 1998.

[Sun96] Sun Microsystems. Java Native Interface Overview. Part of JDK 1.1 Documentation, December 1996.

[Sun97a] Sun Microsystems. *Application from Sun Microsystems, Inc. for Recognition as a Submitter of Publicly Available Specifications for Sun's Java Technologies*, ftp://dkuug.dk/JTC1/SC22/JSG/docs/j1n4615.html, March 14 1997.

[Sun97b] Sun Microsystems. *Do Business with JavaSoft*. http://www.javasoft.com/nav/business/index.html, 1997.

[Sun97d] Sun Microsystems. *Summary of the Year in Review*. March 10 1997. http://www.javasoft.com:80/features/1996/december/yearnotes.html.

[Sun97g] Sun Microsystems. *Sun Microsystems Announces JavaPC*. Press Release, April 2 1997. http://java.sun.com/pr/1997/april/pr970402-07.html

[Sun97h] Sun Microsystems. *Java Challenges Definition of a Computer*. Press Release, April 2 1997. http://java.sun.com/pr/1997/april/pr970402-01.html

[Sun97i] Sun Microsystems. *Sun announces Java Platform Roadmap and Java Foundation Classes*. Press Release, April 2 1997. http://java.sun.com/pr/1997/april/pr970402-04.html

[Sun97j] Sun Microsystems. *Sun Microsystems Extends the Power of Java with new APIs*. Press Release, April 2 1997. http://java.sun.com/pr/1997/april/pr970402-05.html

[Sun97k] Sun Microsystems. *Performance Measurements*. http://java.sun.com/products/jdk/1.1/performance/index.html, 1997.

[Sun97l] Sun Microsystems. *JavaBeans Documentation*. http://java.sun.com/beans/docs/index.html. 1997.

[Sun98a] Sun Microsystems. *Java Standardization*. http://java.sun.com/aboutJava/standardization/index.html, 1998.

[Sun98b] Sun Microsystems. *Lawsuit-related Information*. http://java.sun.com/aboutJava/info/index.html, 1998.

[Sun98c] Sun Microsystems. *What is JavaPC?* http://java.sun.com/products/javapc/index.html, February 1998.

[Sun98d] Sun Microsystems. *Sun Microsystems releases Java Foundation Classes Software*, Press Release, http://java.sun.com/pr/1998/02/pr980225-01.html, February 25, 1998.

[Sun98e] Sun Microsystems. *U.S. District Court Judge Grants Sun Microsystems Preliminary Injunction in Javatm Technology Case Against Microsoft*, http://java.sun.com/pr/1998/03/pr980324-07.html, March 24, 1998.

[Sur96] Vijay Sureshkumar. *Java Security*. http://csgrad.cs.vt.edu/~vijay/chap16/index.html, 1996.