Workshop on Java in Telecommunication
Deutsche Telekom AG, 12.-13. May 1997
Darmstadt

Erfahrungen in der kooperativen Java-Entwicklung: Ein Praxisbericht

Abdulmotaleb El-Saddik[1], Gunter Weiß[1], Heiko Straulino[1], Utz Roedig[1],
Ralf Steinmetz[1,2]

[1]Darmstadt University of Technology
Department of Electrical Engineering and Information Technology
Industrial Process and System Communications[1]
Merckstr. 25, D-64283 Darmstadt, Germany
{Ralf.Steinmetz,A.El-Saddik,Gunter.Weiss,Utz.Roedig}@KOM.th-darmstadt.de
straule@hrzpub.th-darmstadt.de

[2]GMD-IPSI Institut für integrierte Publikations- und Integrationssysteme
Dolivo Str.15 64293 Darmstadt
Ralf.Steinmetz@darmstadt.gmd.de

Zusammenfassung

An der Technischen Hochschule Darmstadt wurden im Rahmen eines Projektseminars verschiedene Java-Anwendungen für den Einsatz in der interaktiven Lehre konzipiert und entwickelt. Es hat sich herausgestellt, daß Java als objektorientierte Programmiersprache und Entwicklungsumgebung bei einer kooperativen Implementierung hinsichtlich der graphischen Elemente und der Parallelität der Prozesse in den Programmen noch einige Defizite besitzt. Deshalb werden hier die dabei gesammelten Erfahrungen dargestellt. Außerdem werden zu den graphischen Problemen die dazu passenden Lösungen diskutiert.

1. Diese Arbeit wurde teilweise unterstützt von: Volkswagen-Stiftung, D-30519 Hannover, Germany

1. Einleitung

Die Ursprünge der Programmiersprache Java liegen schon einige Jahre zurück. Die Entwicklung begann ca. 1990 bei der Firma Sun in einem Team, das unter der Leitung von James Gosling stand. Ursprünglich trug Java den Namen "Oak" und sollte als allgemeine höhere Programmiersprache für hybride Systeme Verwendung finden. Aus diesem Grund eignen sich Javaprogramme besonders für Anwendungen, die nicht auf eine Plattform beschränkt sind. Solche Anwendungen werden in heterogenen Rechnernetzwerken benötigt, da dort viele verschiedene Rechnersysteme dieselben Programme verwenden. Deshalb hat sich Java als Programmiersprache des Internet durchgesetzt.

Damit Programme von einem Server im Internet auf Clients übertragen werden und dort ablaufen können, müssen bestimmte Forderungen an das eigentliche Programm sowie die Laufzeitumgebung gestellt werden [3]:

- Der Programmcode muß plattformunabhängig sein, damit er auf allen Systemen verfügbar ist.
- Es müssen Sicherheitsanforderungen erfüllt werden, damit das Programm keine unerwünschten (und womöglich destruktiven) Aktionen ausführt.
- Der Programmcode muß robust sein und auch bei Änderungen der Laufzeitumgebung stabil ablaufen.
- Die Programmierumgebung muß genügend Flexibilität bereitstellen, um dennoch die unterschiedlichen Anwendungen realisieren zu können.
- Die Programmiersprache soll objektorientiert sein, weil damit Erstellungs- und Wartungskosten im allgemeinen verringert werden.
- Die Programmiersprache soll hohe Verbreitung (z.B. über das Internet) finden, um eine große Akzeptanz zu erfahren und damit wirtschaftliche Chancen zu haben.

Diese Forderungen werden im allgemeinen von Java erfüllt. Es sind jedoch im jetzigen Entwicklungsstadium von Java im Detail noch Probleme vorhanden. Im nachfolgenden werden unsere wesentlichen Erfahrungen dargestellt. Darunter befinden sich einige Lösungsvorschläge für Probleme, die auftreten, wenn man versucht ein Programm zu schreiben, das "absolut plattformunabhängig" ist, sowie Lösungsvorschläge zu Problemen, die bei der Verwendung von Threads entstehen. Die Eignung von Java zur Entwicklung umfangreicher Programme wird ebenfalls diskutiert. Dies betrifft Aspekte der Wirtschaftlichkeit und der Möglichkeit, ein Projekt auf mehrere Personen aufzuteilen.

2. Eignung von Java zur Entwicklung im Team

In den letzten Jahren wurde die Entwicklungszeit eines Produktes, die sogenannte "Time-To-Market", immer mehr zum wesentlichen Faktor, der über den Erfolg oder Mißerfolg eines Produktes entscheidet. Von den Programmierern werden daher immer kürzere Entwicklungszyklen gefordert, natürlich ohne Qualitäts- und Zuverlässigkeitseinbußen. Die Entwicklungszeit einer Anwendung läßt sich im allgemeinen drastisch verkürzen, indem man eine große Aufgabe in kleine Bereiche unterteilt und diese parallel von mehreren Entwicklern im Team bearbeiten läßt. Wie effizient eine solche verteilte Bearbeitung von Aufgaben ist, wird zu einem nicht unerheblichen Teil von der verwendeten Methodik und damit auch von der Programmiersprache bestimmt.

Sie muß es dem Entwickler unter anderem ermöglichen, klare Trennlinien zu ziehen und Schnittstellen zu schaffen, mit deren Hilfe sich die Teilprojekte zusammenfügen lassen.

An der Technischen Universität Darmstadt[1] setzen wir multimediale Elemente in der Lehre ein. Diese werden in enger Zusammenarbeit mit Lehrstühlen an anderen Universitäten (Braunschweig, Linz, Mannheim und Stuttgart) erstellt. Wegen der gegebenen unterschiedlichen Software und verschiedenen Hardwareplattformen haben wir beschlossen, die notwendigen Programme in Java zu realisieren.

Im praktischen Einsatz von Java zeigten sich in unseren Projekten unter diesem Aspekt einige interessante Eigenschaften. Als äußerst nützlich erwies es sich, daß, bedingt durch die Systemunabhängigkeit von Java, der Programmierer frei in der Wahl der Entwicklungsplattform war. So konnte der eine Teil des Teams auf Macintosh Rechnern arbeiten, der andere auf Windows basierten PC-Systemen und wieder andere an Unix-Workstations, je nach Vorlieben. Dabei wurden die Anwendungen während der Entstehung direkt auf einem Rechner ausgeführt, d.h. ohne Umweg über z.B. einen Cross-Compiler. Dadurch ergab sich ein durchaus beachtlicher Produktivitäts- und damit Zeitgewinn. Leider mußten dennoch die Programme wegen der nachfolgend aufgeführten Probleme auf verschiedenen Systemen regelmäßig auf Lauffähigkeit geprüft werden.

Als weitere Hilfestellung in der verteilten Entwicklung erwiesen sich die strenge Typisierung und die Möglichkeit zur Kapselung mittels *Sichtbarkeits-Modifikatoren* [6]. Dadurch war eine sehr viel klarere Festlegung der Schnittstellen möglich. Es ist daher unbedingt zu empfehlen, bei der Schnittstellendefinition intensiv und überlegt Gebrauch von diesen Sprachkomponenten zu machen. Damit wird es den Entwicklern schwer gemacht, diese Schnittstellen unabsichtlich zu umgehen und damit eventuelle Inkompatibilitäten der einzelnen Programmfragmente zu erzeugen.

Eine weitere Möglichkeit zur effizienten und kooperativen Zusammenarbeit an den Projekten ergab sich durch die Organisation des Java-Bytecodes. Der vom Interpreter erzeugte Code ist sehr kompakt (üblicherweise in der Größenordnung weniger KBytes) und wird für jede Klasse in einer separaten Datei abgelegt. Diese Dateien lassen sich aufgrund ihrer Größe leicht über Netz austauschen, um dann von den anderen Mitgliedern des Teams, ohne das in z.B. C oder C++ nötige Binden (to link), verwendet zu werden.

3. Kritische Anmerkungen zu der Plattformunabhängigkeit der Darstellung

Als erstes gilt es zu beachten, daß die verschiedenen Clients bezüglich der Hard- und Software unterschiedlich ausgestattet sind (z.B. Monitorgröße, Rechenleistung, installierte Schriftarten, Farben). Soll mit dem Programm eine große Zielgruppe erreicht werden, so sind die Ressourcenanforderungen so gering zu halten, daß ein Betrieb der Anwendung auch auf leistungsschwächeren Systemen als dem eigentlichen Entwicklungsystem möglich ist. Regelmäßige Tests auf verschiedenen Clients während der

1. Die Technische Hochschule Darmstadt wird mit Wirkung vom 1.Okt. 1997 Technische Universität Darmstadt genannt.

Entstehungsphase des Projekts sind unerläßlich.
Des weiteren muß man sich darüber im Klaren sein, daß die Möglichkeiten zur Gestaltung der Benutzungsoberfläche erheblich geringer sind, als bei einer spezifisch für einen bestimmten Rechnertyp entwickelten Anwendung. Um die Plattformunabhängigkeit zu gewährleisten, kann Java nur die Basisobjekte implementieren, die auf allen Systemen zur Verfügung stehen. Der Entwickler muß sich also bei der Wahl der Basisobjekte einschränken, da systemspezifische Komponenten nicht berücksichtigt werden können.

Das weitaus größte Problem ergibt sich durch das unterschiedliche Aussehen der Bedienelemente auf den verschiedenen Plattformen. Dies liegt daran, daß das Java AWT (Abstract Windowing Toolkit) immer die von dem System zur Verfügung gestellten Bedienelemente verwendet. Daraus ergeben sich weitreichende Konsequenzen für die Gestaltung des Bildschirmlayouts.

Um diese Problematik vorab zu verdeutlichen, ist nachfolgend ein Beispiel dargestellt. Es zeigt zweimal dieselbe Anwendung, auf einem Unix-Betriebsystem (Linux, Abbildung 1) und auf einem Windows NT Rechner (Abbildung 2). Das dargestellte Programm wurde auf einem Windows NT Rechner entworfen.

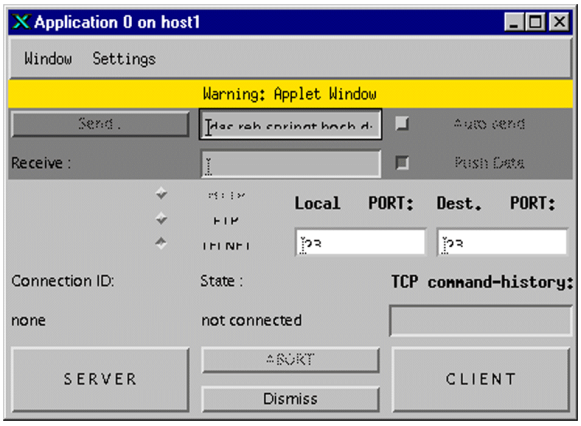

Abbildung1: Fenster einer Anwendung unter Linux ohne Größenveränderung

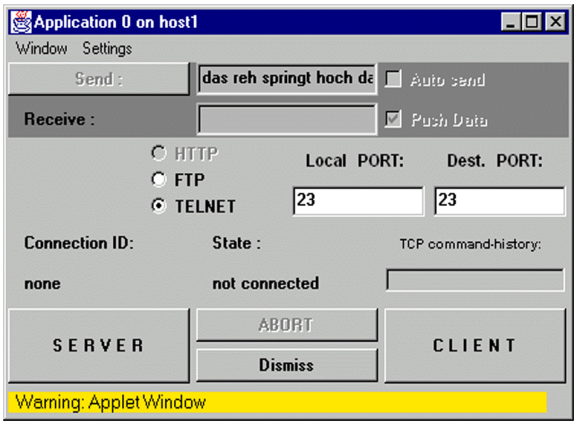

Abbildung 2: Fenster einer Anwendung unter Win-NT ohne Größenveränderung

Grundsätzlich existieren zwei Möglichkeiten, eine graphische Oberfläche zu entwikkeln. Die erste Möglichkeit besteht darin, ein *Werkzeug* (Resource-Editor) einer Entwicklungsumgebung zu benutzen. Einige dieser Werkzeuge sind z.B. das Café Studio von Symantec bzw. von Symantec's Visual Café [7], der Layout-Editor von Jamba [8], der Resource-Editor von Microsoft Visual J++ und Spec-Java [9]. Dadurch wird es dem Programmierer möglich, Bedienelemente visuell am Bildschirm zu plazieren und so das GUI (Graphical User Interface) der Anwendung graphisch zu gestalten. Der entsprechende Java-Code wird dann automatisch erzeugt und in das Projekt eingebunden. Die zweite Möglichkeit besteht darin, den zur Erstellung einer graphischen Oberfläche nötigen Code "von Hand" zu erstellen. Man übernimmt also die Programmierung der Oberfläche selbst und überläßt sie nicht einem Programm. Dazu werden die Bedienelemente mit einem Hilfsmittel der Java-Sprache, den sogenannten *Layoutmanagern*, zusammengestellt.

Von der Verwendung graphischer Hilfswerkzeuge ist, wenn man eine 100%-ige Plattformunabhängigkeit erreichen will, zumindest im jetzigen Stadium der Entwicklung aus unserer Sicht heraus abzuraten. Der Hauptgrund dafür liegt darin, daß der so erzeugte Code mit absoluter Positionierung zwar auf allen Plattformen funktioniert, aber nach unseren Erfahrungen nur auf dem Systemtyp, auf dem er entwickelt wurde, das gewünschte Aussehen liefert. Dies ist nicht zu vermeiden, da alle Bedienelemente, wie zuvor beschrieben, auf unterschiedlichen Systemen verschiedenes Aussehen und verschiedene Größe haben. Im schlimmsten Fall sind die dargestellten Elemente aufgrund von Überschneidungen nicht einmal zu erkennen.

Dieses Problem kann umgangen werden, indem man die graphische Oberfläche unter Benutzung der von Java zur Verfügung gestellten Layoutmanager von Hand programmiert. Durch die Verwendung der Layoutmanager können die Elemente bei einer Größenveränderung des Fensters mitwachsen. Dadurch kann das Fenster auf jedem System in eine Größe gebracht werden, in der jedes Element gut zu erkennen ist. Es ist daher immer darauf zu achten, daß es dem Benutzer ermöglicht wird, die Größe eines Fensters zu verändern, um einen zerstückelten Bildschirmaufbau wieder korrigieren zu können.

Man betrachte nochmals Abbildung 1 und 2:
Es ist zu erkennen, daß z.B. bei einem Eingabefeld unter Linux der Abstand der Schrift zum Rand deutlich größer ist als unter Windows. Auch die Schriftarten sind in Form und Größe sehr verschieden. Aus diesem Grund ist der Text in solch einem Feld dieser Größe unter Linux nicht lesbar. Auch die Auswahlfelder (hier bezeichnet mit HTTP, FTP, TELNET) haben auf beiden Systemen ein unterschiedliches Aussehen und einen anderen Platzbedarf. Die daraus resultierenden Überschneidungen führen zu einer unleserlichen Darstellung. Da bei der Erstellung des Projektes ein Layoutmanager verwendet wurde, kann durch eine Größenveränderung des Fensters eine gute Darstellung erreicht werden. Das Fenster kann auf jedem System in eine Größe gebracht werden, in der jedes Element gut zu erkennen ist (siehe Abbildung 3).

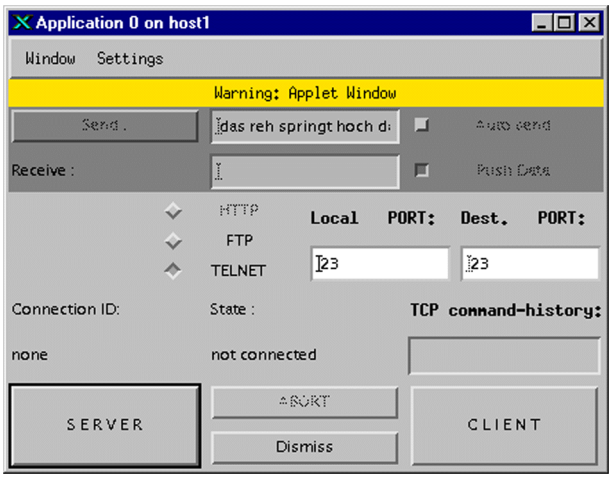

Abbildung 3: Fenster einer Anwendung unter Linux nach Größenveränderung

In Java sind fünf Standard-Layout-Manager integriert (FlowLayout, BorderLayout, CardLayout, GridLayout und GridBagLayout).

Der *FlowLayout* Manager ist der einfachste Layout Manager, er ordnet alle seine Komponenten einfach nebeneinander in einer Reihe an. Ist der das Layout enthaltende Container[1] zu schmal, werden die Komponenten in mehreren Zeilen dargestellt. Das Layout wird dabei der Reihe nach von links nach rechts gefüllt. Bei dem *BorderLayout* Manager können fünf Komponenten an den Positionen North, South, East, West und Center angeordnet werden. Das *CardLayout* unterscheidet sich von den anderen Layout Managern, bei ihm werden die Komponenten wie ein Stapel Spielkarten angeordnet, von denen nur die oberste sichtbar ist. Zum „Weiterblättern“ existieren bestimmte Methoden. Das *GridLayout* ordnet seine Komponenten in einem Gitter an. Die Breite der Zellen richtet sich nach der breitesten Komponente in diesem Layout, entsprechendes gilt für die Höhe der Gitterzellen. Das Layout wird dabei der Reihe nach von links nach rechts und von oben nach unten mit Komponenten aufgefüllt. Das *GridBagLayout* basiert ebenfalls auf einem Gitter, ist allerdings erheblich flexibler als das GridLayout. Einzelne Komponenten können sich über mehrere Gitterzellen erstrecken, außerdem können die einzelnen Zeilen und Spalten unterschiedlich hoch sein. Dadurch ist das GridBagLayout der vielseitigste, aber auch der am aufwendigsten zu programmierende Layoutmanager.

Neben diesen vordefinierten Layoutmanagern besteht noch die Möglichkeit eigene Layoutmanager zu definieren [3]. In den meisten Fällen reichen das BorderLayout und das GridLayout völlig aus, da als Komponenten auch solche eingefügt werden können, die ihrerseits wieder einen Layout-Manager enthalten. Auf diese Weise lassen sich Layout-Manager ineinander verschachteln, und es kann jede gewünschte Anordnung von Komponenten erreicht werden. Damit läßt sich ein Aussehen erreichen, das einer absoluten Positionierung in nichts nachsteht, jedoch die damit verbundenen Probleme nicht aufweist. Gleichzeitig sind die Komponenten in der Größe veränderbar, und jeder Anwender kann die Darstellung an seine Gegebenheiten anpassen. Die oben gezeigte Anwendung wurde zum Beispiel mit mehreren ineinander verschachtelten GridLayout-Managern realisiert.

4. Programmierung paralleler Prozeßabläufe: Threads

Java hebt sich gegenüber vielen Programmiersprachen dadurch ab, daß Threads ein fester Bestandteil der Sprache sind. Damit lassen sich komfortable Anwendungen, die komplexe parallele Abläufe enthalten, einfach erstellen. Die Vorteile des Multithreading verlangen jedoch eine neue Denkweise vom Programmierer[2].

Ein Thread ist eine Abfolge von Anweisungen, die sequentiell abgearbeitet werden [1,2,3,4,5]. Mehrere Threads können parallel ablaufen, was ein gleichzeitiges Abarbeiten der Anweisungen verschiedener Threads zur Folge hat. Jedoch sollte man sich darüber im Klaren sein, daß echte Parallelität nur mit Multiprozessorcomputern möglich ist. Bei Einprozessorsystemen sorgt das Betriebssystem über den Scheduler für eine geeignete Aufteilung der Prozessorzeit.

1. Weitere Informationen zur graphischen Komponenten von Java findet man unter http://java.sun.com
2. Der Begriff Thread wird im weiteren so verwendet wie er im Kontext von Java realisiert ist.

Dieses sogenannte Multithreading ist nicht mit Multiprocessing zu verwechseln [1]. Threads bringen hinsichtlich der Verarbeitungsgeschwindigkeit keinen Gewinn, da der Verwaltungsaufwand für die Threads und die Kontextwechsel Rechenzeit benötigen. Deshalb sollten sie möglichst sparsam eingesetzt werden. Sie bieten dem Programmierer jedoch mehr Möglichkeiten zur Umsetzung der gestellten Aufgaben. Im Laufe unserer Arbeit hat es sich als sehr nützlich erwiesen, mit Hilfe von Threads komplexe, in der Realität parallele Abläufe darzustellen. In zwei unserer Projekte mit den Themen *TCP* [10] und *SlidingWindow* [12] wurden Übertragungssyteme modelliert, wobei die einzelnen Rechner, der Übertragungskanal und die Übertragungsprotokolle jeweils durch eigenständige Threads realisiert wurden. Dadurch ergab sich ein sehr realitätsnahes Verhalten der Anwendungen. Ebenfalls wurden Threads eingesetzt, um das Antwortverhalten der Programme und damit die Benutzerfreundlichkeit zu erhöhen. So liefen die Programmteile für Benutzerinteraktionen, Darstellung und das eigentliche Protokoll in Threads mit unterschiedlichen Proritäten ab, wodurch eine schnelle und zuverlässige Reaktion des Protokolls auf Eingaben des Anwenders ermöglicht wurde. Allerdings kommt es im Zusammenhang mit dem parellelen Ablauf der Threads für einen die sequentielle Programmierung gewohnten Programmierer zu einigen neuen Problemen, die nachfolgend im Kontext von Java verdeutlicht werden sollen.

Durch den Einsatz von Threads ergeben sich zwei neue Arten von Fehlerquellen beim Zugriff auf gemeinsame Daten: Kollisionen und Deadlocks. Threads, die zu einem Deadlock führen, können das komplette Programm zum Stillstand bringen. Treten solche Probleme in einem fortgeschrittenen Entwicklungsstadium auf, ist es äußerst schwierig, die Ursachen zu lokalisieren und zu beseitigen, da das Fehlerbild meist nichtdeterministischer Natur und schwer rekonstruierbar ist (Abbildung 4). Diese Erfahrung haben wir bei der Realisierung unserer Projekte gemacht. Es empfiehlt sich daher, die Resource Thread überlegt einzusetzen.

Bei zu vielen parallel laufenden Java-Threads verliert man außerdem leicht den Überblick. Als Faustregel gilt: Zusätzliche Java-Threads machen nur dann Sinn, wenn noch Aufgaben zu erledigen sind, während sich alle bisherigen im Wartezustand befinden.

Hat man erst einmal die Anzahl und Aufgabenbereiche der Threads festgelegt, muß man sich über deren Zusammenwirken und Abhängigkeiten genaue Gedanken machen. Auch müssen die Stellen im Programmablauf lokalisiert werden, an denen es zu Überschneidungen bezüglich des Zugriffs auf gemeinsame Daten kommen kann. Zur Lösung derartiger Konflikte bietet Java verschiedene Konstrukte zur Threadsynchronisation an. So lassen sich bestimmte Methoden und Zugriffe auf Instanzen von Klassen in sog. *synchronized* Statements einschließen. Dies bewirkt, daß der Zugriff auf diesen geschützten Bereich exklusiv nur von einem Thread erfolgen darf.

Hierbei sollte man folgende Richtlinien beachten [4], die wir in unseren Projekten verifiziert und angepaßt haben.

- In *synchronized*-Methoden sollten nach Möglichkeit keine weiteren aufgerufen werden, die ihrerseits *synchronized* sind. Dadurch kann man sich aus unserer Erfahrung heraus einige Probleme ersparen.
  Sperrt man ein fremdes Objekt aus einem bereits gesperrten Objekt heraus, dann kann es zu einem Deadlock kommen. Das folgende Beispiel soll dies verdeutlichen.

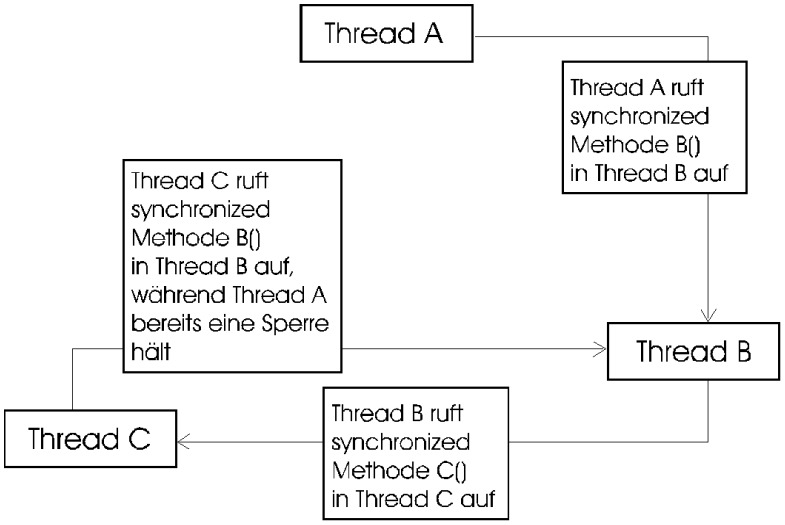

Abbildung 4: Beispiel für das Auftreten eines Deadlock

- Absolut tabu in *synchronized*- Blöcken sind Aufrufe von *suspend(), sleep(), yield(), join(), stop()* und *destroy()*.
  Ruft man die oben genannten Methoden aus einem *synchronized* Block auf, kann es zu einem Deadlock kommen. Ein Beispiel verdeutlicht dies. Verwendet man die Methode *destroy()*, um einen Thread "gewaltsam" zu beenden, so wird dessen Sperre nicht aufgehoben. Ist der Programmcode nun so strukturiert, daß ein anderer Thread erst nach Aufheben dieser Sperre fortfahren kann, kommt es an dieser Stelle zu einem Stop des Programmablaufs. Threads sollten daher immer mit der Methode *stop()* von sich selbst beendet werden.

- Keine Ein- und Ausgaben, Systemaufrufe oder ähnlich zeitaufwendige Dinge in *synchronized*- Blöcken.
  Da sich hinter einem Systemaufruf eine Vielzahl von Befehlen verstecken kann, die viel Prozessorzeit in Anspruch nehmen, kann es ebenfalls zu langen Wartezeiten im Programmablauf kommen. Derselbe Effekt tritt bei Ein- und Ausgabemethoden auf. Die Zeit, die man für Ein- und Ausgaben einrechnen sollte, ist wesentlich größer, als die für die Abarbeitung des Programmcodes.

Weiterhin hat man die Möglichkeit, den Threads aufgrund ihrer Aufgaben Prioritäten zuzuordnen. Dabei sollten die Threads, für welche die schnellste Antwortzeit auf Ereignisse gefordert wird, die höchste Priorität besitzen. Bei der Erstellung unserer Java-Programme haben wir dies berücksichtigt, indem wir den Protokoll-Threads eine höhere Priorität zuteilten als den Darstellung-Threads. Hiermit wurde eine zuverlässige Funktionsweise des Protokolls gewährleistet.
Man muß sich bei der Vergabe von Prioritäten genau überlegen, welche Auswirkungen

dies auf die Funktionsweise des Programmes oder der Anwendung hat. Da ein Thread höherer Priorität bei Bedarf einen Thread niedriger Priorität vom Prozessor verdrängt, können bei der Vergabe der Prioritäten ungewollte Probleme auftreten. Bekommt z.B. ein Berechnungsthread eine höhere Priorität als ein Thread für Benutzerinteraktion, dann nutzt der Berechnungsthread freie Prozessorkapazität für seine Arbeit. Der Thread für Benutzerinteraktion kann aufgrund der niedrigeren Priorität diesen Thread nicht vom Prozessor verdrängen. Er muß warten, bis die Berechnungen beendet sind. Erst jetzt kann er auf die Eingaben des Benutzers reagieren. Dies vermindert die Komfortabilität der Anwendung für den Benutzer.

5. Ausblick

Es ist wahrscheinlich, daß einige der hier beschriebenen Probleme durch das relativ frühe Entwicklungsstadium der Sprache Java bedingt sind und in folgenden Versionen des JDKs korrigiert werden. Die unserer Arbeit zugrundeliegende Version ist die JDK 1.02. Mittlerweile erfolgt die Migration auf die Version 1.1. Man muß bedenken, daß andere etablierte Programmiersprachen wie C und C++ immerhin schon eine "Reifezeit“ von 20 bzw. 10 Jahren hinter sich haben.

Andere Probleme wiederum ergeben sich aus den Kompromissen, die einfach zugunsten einer möglichst hohen Plattformunabhängigkeit gemacht werden müssen. So werden Steuerelemente wie Buttons und Menüs auf verschiedenen Plattformen immer unterschiedliches Aussehen und unterschiedliche Abmessungen haben und dadurch Probleme mit der Positionierung des Bildschirmlayouts verursachen.

Im Laufe der verschiedenen Projekte hat sich jedoch gezeigt, daß die Sprache Java durchaus zum Erstellen anspruchvoller Anwendungen, die deutlich über die üblichen Animationen auf Webseiten hinausgehen, in Betracht gezogen werden kann. Wenn man die zuvor genannten Hinweise berücksichtigt, sollte ein zeit- und kostengünstiger Einsatz der Sprache Java möglich sein und zu ansprechenden Ergebnissen führen.

6. Literaturverzeichnis

[1] Davis, Stephen R. : "Java jetzt!“, Microsoft Press, 1996
[2] Flanagan, David: "JAVA IN A NUTSHELL“, O’Reilly, 1.Auflage, 1996, Deutsche Ausgabe für Java 1.0
[3] Middendorf, Singer, Strobel: "JAVA Programmierhandbuch und Referenz“, dpunkt, 1.Auflage, 1996
[4] Gims, Dr. Klaus: "Multithreading Teil1“, Java Spektrum, Ausgabe 2, März/April 97, Nr.6
[5] Lemay, Perkins: "JAVA in 21 days“, sams net, first edition, 1996
[6] Sun Microsystems Inc.: "The Java Tutorial“, http://Java.sun.com, 1995
[7] Symantec Café und Symantec Visual Café, http://www.symantec.com/index.html
[8] Jamba, http://www.jamba.com
[9] Spec-Java, ftp://ftp.sunlabs.com/pub/tcl/specjava-0.3bin.tar.gz
[10] RFC 793, TCP, http://sunsite.auc.dk/RFC/

[11] David M. Gery, Alan L. McClellan: "Graphic Java - Mastering the AWT", Prentice Hall ISBN 0-13-565847-0
[12] Andrew Tannenbaum, "Computer Networks"' third edition, Printice Hall' 1996 ISBN: 0-13-394248-1
[13] Ralf Steinmetz, Klara Nahrstedt, "Multimedia: Computing, Communications & Applications", Printice Hall, ISBN: 0-13-324435-0

Während der Arbeiten für dieses Projektseminar wurden von uns eine Vielzahl von Büchern zu Rate gezogen, welche sich mit der Programmierung von Java beschäftigen. Es hat sich dabei jedoch herausgestellt, daß nicht jedes Buch bezüglich Umfang und Qualität des Inhalts für eine erfolgreiche Programmierung geeignet ist. Wir möchten deshalb zwei Werke erwähnen, die sich als besonders zweckmäßig für unsere Projekte erwiesen:

- Flanagan, David: "JAVA IN A NUTSHEL", O'Reilly, 1.Auflage, 1996, Deutsche Ausgabe für Java 1.0
- Middendorf, Singer, Strobel: "JAVA Programmierhandbuch und Referenz", dpunkt, 1.Auflage, 1996