Unterstützung der Lehre durch Visualisierung von wissensbasierten Suchalgorithmen mit Java

Jörg Denzinger, Bernd Löchner, and Sebastian Scheffler

Fachbereich Informatik, Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern, Germany
{denzinge,loechner,s_scheff}@informatik.uni-kl.de

Zusammenfassung Dieser Bericht enthält unsere Erfahrungen mit Java bei der Entwicklung und Implementierung eines wissensbasierten Suchsystems. Der Schwerpunkt lag auf der Visualisierung der Vorgänge und Entscheidungen im System, um es Studenten zur Nachbereitung der entsprechenden Vorlesung zur Verfügung zu stellen. Das System wurde im Rahmen eines Praktikums implementiert, und Java hat sich sowohl bei der Entwicklung als auch im Hinblick auf die Visualisierung der Vorgänge als sinnvolle Wahl als Implementierungssprache erwiesen. Überraschend war auch die gute Effizienz des Systems.

1 Motivation

Eines der wesentlichen Ziele bei der Konzeption von Java war die Plattformunabhängigkeit von in Java implementierten Programmsystemen. Zusammen mit guten graphischen Darstellungs- und Interaktionsmöglichkeiten und durch die Integration in Webbrowser läßt dies Java als geeignete Sprache für Lernunterstützungssysteme erscheinen. Durch die Visualisierung dynamischer Aspekte, je nach Benutzerwunsch oder -verständnis auf beliebigen Ebenen eines Vorgangs (*Zooming*), und die Nutzung der implementierten Verfahren mit eigenen Beispielen wird ein tiefergehendes Verständnis und eine weitgehend individuell bestimmte Planung des Lernens ermöglicht. Speziell im universitären Bereich, für Studenten unterschiedlicher Begabung und mit unterschiedlichen (Vor-)Kenntnissen, die unterschiedlichste Computer und Betriebssysteme benutzen, ist die Kombination der angesprochenen Sprach- und Infrastruktureigenschaften eine Voraussetzung für die Erstellung und Akzeptanz jeglicher Systeme zur Lernunterstützung oder -nachbereitung.

In diesem Bericht wollen wir auf unsere Erfahrungen mit Java bei der Entwicklung und Implementierung eines wissensbasierten Suchsystems eingehen, das die praktische Umsetzung der in einer Vorlesung theoretisch vorgestellten Sachverhalte darstellt und Hörer in ihrem Lernprozeß unterstützen soll. Da es sich bei der Vorlesung (Reduktionssysteme, vgl. [1]) um eine Vorlesung der theoretischen Informatik handelt, wird dort sehr großer Wert auf eine Präsentation der Konzepte in einer Form gelegt, die den formalen Nachweis von Eigenschaften wie Korrektheit und Vollständigkeit erlaubt. Zwar stehen effiziente Implementierungen zu den Fragestellungen der Vorlesung zur Verfügung, aber diese zeigen

nicht die Verbindung zwischen Konzept und schließlicher Realisierung auf und fördern nicht unbedingt das Verständnis des Stoffes.

Benötigt wird vielmehr ein System, das sowohl ausreichend effizient ist, um auch etwas schwierigere Aufgaben in einer akzeptablen Zeit zu lösen, als auch in der Lage ist, Brücken zwischen den theoretischen Konzepten und der Implementierungsebene zu schlagen und dabei auf die individuellen Verständnisprobleme eines Benutzers interaktiv einzugehen. Insbesondere muß es möglich sein, die Vorgänge graphisch und/oder animiert darzustellen, um so Eindrücke von der Verhaltensdynamik des Verfahrens zu vermitteln.

Die Erstellung des Systems selbst erfolgte auch in Form einer Lehrveranstaltung, nämlich eines Praktikums. Dadurch war es möglich, zwei konkurrierende Systeme erstellen zu lassen und somit auch einen guten Überblick über die Eigenschaften von Java und diverser Entwicklungstools zu erhalten, der nicht durch Präferenzen eines einzelnen Implementierers eingeschränkt wurde. Insbesondere wurde jedes System von einer Gruppe von Studenten entwickelt, so daß auch zum organisatorischen Teil des Software-Engineering und seine Unterstützung durch Java und die benutzten Tools Aussagen gemacht werden können.

Im folgenden wollen wir zunächst kurz auf die gewünschte Funktionalität des zu entwickelnden Systems eingehen und dann unsere Vision vom Endsystem und dessen Gebrauch vorstellen. Danach werden wir die Erfahrungen während der Entwicklung des Systems (bzw. der Systeme) schildern und dann Endresultat und Vision vergleichen und auf die aufgetretenen Diskrepanzen und ihre Gründe eingehen. Schließlich werden wir im Fazit auf unsere weiteren Pläne bezüglich des Systems eingehen.

2 Suchproblem und Suchverfahren

Das zu implementierende Suchsystem ist typisch für wissensbasierte Suchsysteme, wie sie zur Lösung von Optimierungsproblemen, Planungsproblemen oder Problemen des Symbolischen Rechnens eingesetzt werden. Der Suchzustand wird beschrieben durch eine Menge von Termstrukturen und die Zustandsübergänge durch schematische Transitionsregeln.

Abbildung 1 zeigt den typischen Ablauf der Suche. Auf den aktuellen Suchzustand können unterschiedliche Transitionen angewendet werden, die zu vielen möglichen Folgezuständen führen. Die Kontrollkomponente wählt einen dieser Folgezustände aus, der dann zum neuen aktuellen Suchzustand wird. Dieser neue Zustand hat in der Regel einige der Folgezustände mit seinem Vorgänger gemein, aber manche der früheren Folgezustände sind nun nicht mehr möglich und dafür gibt es neue Folgezustände. Die Suche ist beendet, wenn ein Zustand erreicht ist, der das vorgegebene Endekriterium erfüllt.

In der Regel stellt ein Suchsystem verschiedene Kontrollen zur Verfügung, die alle Wissen über das zu lösende Suchproblem beinhalten. Leider sind diese Kontrollen praktisch nicht kombinierbar, so daß der Benutzer des Systems für eine Probleminstanz das Kontrollverfahren vorgeben muß. Die vielen Möglichkeiten für eine Kontrolle sind auch dafür verantwortlich, daß die Darstellung

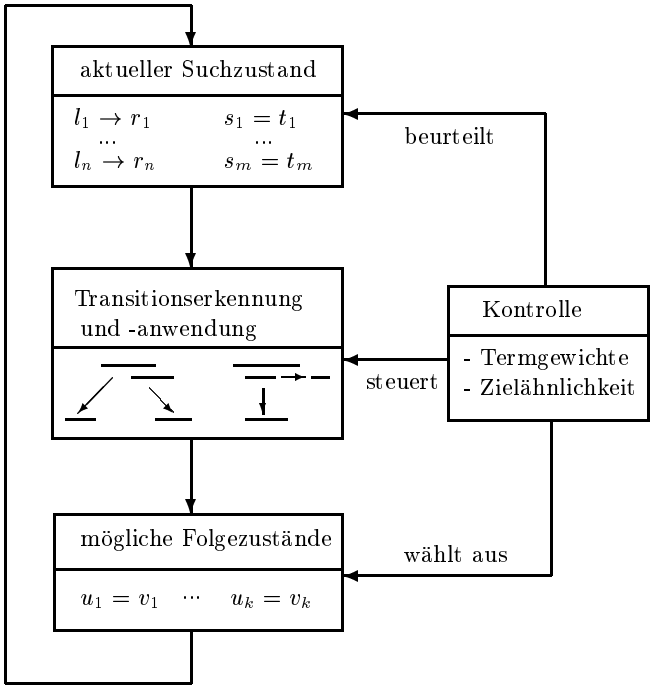

Abbildung1. Ein typischer wissensbasierter Suchzyklus

solcher Verfahren in der Literatur sehr abstrakt ist, um Aussagen, die für viele Kontrollen gültig sind, zu erlauben. Dies bedeutet aber, daß ein Benutzer sowohl die zu Grunde liegende Theorie als auch Implementierungsaspekte kennen muß, um das System sinnvoll nutzen zu können.

Die unteren Teile der Kästen in Abbildung 1 zeigen die Instanziierungen des generellen Konzepts für unsere Anwendung, das symbolische Rechnen mit Reduktionssystemen. Der aktuelle Zustand besteht aus einer Menge von Regeln und Gleichungen. Es gibt zwei Typen von Transitionen, die Kritische-Paar-Bildung (links im Kasten symbolisiert), die eine neue Regel oder Gleichung erzeugt, und die Normalformbildung (rechts im Kasten), die mittels der Regeln und Gleichungen Terme vereinfacht. Üblicherweise wird eine Kritische-Paar-Bildung mit mehreren Normalformbildungen kombiniert, um einen Ausblick auf eine neue Gleichung zu geben. Jeder Folgezustand entsteht durch Aufnahme einer solchen Gleichung in den Suchzustand und anschließende weitere Normalisierungen. Die Kontrolle wählt aus den Folgezuständen die Gleichung aus, die am besten zum aktuellen Zustand paßt, z. B. weil sie sehr klein ist oder Ähnlichkeit mit dem Suchziel hat. Dieses Suchziel ist selbst eine Gleichung, deren Seiten durch Anwendung der Regeln und Gleichungen äquivalent gemacht werden sollen.

Generell gilt, daß Bedingungen für die Anwendbarkeit einer Transition zusammen mit der konkreten zu lösenden Instanz eines Suchproblems die speziellen Eigenschaften der Menge der möglichen Transitionen in einem Zustand bestimmen. Das Erkennen von für die Suche wichtigen Eigenschaften erlaubt es zum einen, Kontrollen zu entwerfen, die diese Eigenschaften nutzen, und zum anderen Verwaltungsstrukturen zu definieren, die das Erfassen dieser Eigenschaften erlauben. Dabei sind diese Eigenschaften sowohl spezifisch für das Suchverfahren als auch für Klassen von Suchinstanzen. Ersteres erfordert das Verständnis der Theorie, letzteres in der Regel Erfahrungen mit dem System.

3 Ziele des Projekts

Da wir Systeme zur Lernunterstützung im Rahmen eines Praktikums entwickelt haben, lassen sich unsere Ziele zunächst in solche, die das Praktikum betreffen, und solche, die die spätere Verwendung der Systeme (bzw. eines davon) betreffen, untergliedern. Allerdings gibt es bei den didaktischen Zielen eine große Übereinstimmung.

Die am Praktikum beteiligten Studenten sollten

- ihr Verständnis des Vorlesungsstoffs vertiefen und praktisch anwenden,
- Erfahrungen im Entwurf und der Implementierung größerer Softwaresysteme sammeln und
- Erfahrungen mit der Arbeit in einem Team gewinnen.

Da dies die üblichen Ziele eines Praktikums in der Informatik sind, wollen wir auf sie nicht weiter eingehen. Das Endprodukt sollte folgendes ermöglichen:

- Das System soll es Hörern der Vorlesung erlauben, ihr Verständnis des Vorlesungsstoffes zu testen und zu vertiefen und außerdem eine praktische Umsetzung der Konzepte demonstrieren.
- Das System soll die dynamischen Vorgänge auf allen Ebenen des Suchsystems visualisieren.
- Das System soll sich individuell auf den Kenntnisstand eines Benutzers konfigurieren lassen und interaktiv diesen Kenntnisstand verbessern helfen.
- Das System soll den Benutzern über das World-Wide-Web zugänglich sein.

Konkret sollte man sich die Benutzung des Systems so vorstellen, daß ein Student oder sonstiger Nutzer sich von seinem Rechner zu der Startseite des Systems im WWW begibt und dort Darstellungen der theoretischen Konzepte findet, eine Sammlung von Suchinstanzen, die bestimmte Konzepte und Effekte beleuchten und die Möglichkeit eigene Suchinstanzen einzugeben. Beim Lösen einer Suchinstanz ist es möglich, unter verschiedenen Stufen der Ausführlichkeit an begleitenden Informationen zu wählen und Interaktionspunkte zu setzen, wobei die Informationen meistens in graphischer Form erfolgen.

Zum Beispiel sollten die Voraussetzungen für Transitionsgruppen hervorgehoben und das Berechnen des Folgezustandes animiert werden. Speziell bei der

Kontrolle der Suche sollte es auch möglich sein, den Benutzer aus den möglichen Transitionen auswählen zu lassen, wobei es in unserer Anwendung möglich war, die durch eine Transitionsgruppe erfolgenden Änderungen am Suchzustand schon a priori darzustellen (als Liste kritischer Paare), um dies zu erleichtern. Man beachte, daß diese Möglichkeit nicht nur für Studenten interessant ist, sondern auch für viele Forscher.

Selbstverständlich sollte es möglich sein, Transitionsgruppen in die Einzeltransitionen aufzulösen und die entsprechenden Zustandsänderungen nach und nach durchzuspielen. Bei einer einzelnen Transition sollte das theoretische Konzept jederzeit dargestellt werden können und die Anwendung, die zum konkreten Schritt führte, vergleichend präsentiert werden. Bei unserer Anwendung involviert der Anwendbarkeitstest mitunter zusätzliche Algorithmen (Unifikation, Matching von Termen und Vergleichen von Termen bezüglich komplexer Ordnungen), die ebenfalls visualisiert und schrittweise durchgespielt werden sollten.

Damit werden recht hohe Anforderungen an die Visualisierung gestellt, die über eine Standardanwendung hinausgehen. Für die verschiedenen Ebenen sind Darstellungen mit unterschiedlichem Detaillierungsgrad zu entwickeln. Beispielsweise sollen Terme einerseits kompakt in textueller Form, andererseits in einer der Theorie näheren Form als Bäume repräsentiert werden. Zweite Form gestattet auch einfacherer Interaktionen, z. B. die Selektion von Teiltermen. Weiterhin sind Konzepte zur Darstellung von Vorgehensweisen und Abläufen zu entwickeln, die den jeweiligen Algorithmen angemessen sind.

4 Die Systementwicklung

Auf Grund der später intendierten Verwendung sollten die Studenten ihre Systeme auf Sun-Workstations unter Solaris 2.5 entwickeln. Während des Praktikums wurde Java 1.1 verfügbar, und wir haben uns zusammen mit den Studenten dafür entschieden, von Java 1.0 auf Java 1.1 umzusteigen. Dies hatte sowohl für die Systementwicklung als auch für unsere Systemziele positive und negative Auswirkungen, auf die wir später noch eingehen werden.

Über das JDK 1.1.1 hinaus wurden von uns keine Java-spezifischen Werkzeuge zur Verfügung gestellt, was dazu führte, daß viele der Studenten auf andere Rechner/Betriebssystem-Kombinationen auswichen, um dort ihre Systemteile zu entwickeln. So wurden als Betriebssysteme Linux, Windows 95, Windows NT, OS/2, Solaris und AIX verwendet, und bei der Entwicklung kamen als Werkzeuge neben dem JDK der *Java WorkShop 1.0*, *Visual Café 1.0* und *VisualAge for Java 1.0* zum Einsatz. Auf Wunsch aller Studenten war auch die Verwendung der *Java Generic Library* (JGL, siehe [5]) zulässig.

Die Studenten bildeten zwei Gruppen zu je 7 Personen, von denen jede ein eigenes System entwickelte (allerdings wurde von uns durch gemeinsame Besprechungen der Erfahrungsaustausch zwischen den Gruppen gefördert, so daß insbesondere die Basissysteme (ohne Visualisierung) der Gruppen sehr ähnlich waren). Die Praktikumsteilnehmer verfügten über keine oder allenfalls geringe Java-Kenntnisse, und die Fertigkeiten in objektorientierter Analyse, objektori-

entiertem Design, objektorientierter Programmierung und in teamorientierter Entwicklung eines größeren Softwaresystems waren sehr unterschiedlich.

Im folgenden werden wir zunächst auf die Java-spezifischen Aspekte im Entwurf der Systeme eingehen und danach auf die Erfahrungen bei der Implementierung. Die Erfahrungen bezüglich der Visualisierung der Suche werden wir anschließend gesondert behandeln. Schließlich werden wir auch kurz auf die Erfahrungen der Studenten mit den von ihnen gewählten Werkzeugen eingehen.

4.1 Der objektorientierte Entwurfsprozeß

Die Studenten entwarfen nach dem OMT-Ansatz gemäß J. Rumbaugh et. al. [3] iterativ ein objektorientiertes Modell der Aufgabenstellung. In der Analyse-Phase ging es zunächst darum, ein gemeinsames Verständnis der Anwendungsdomäne und der Aufgabenstellung herzustellen und eine konzeptionelle Übersicht über die zu entwickelnde Anwendung zu erhalten. Dazu wurden die erforderlichen Objektklassen, erste Attribute und Operationen und die Beziehungen der Klassen zueinander identifiziert und die anzuwendenden funktionalen Transformationen der Daten in Form eines ersten statischen Objektmodells samt Data Dictionary und eines funktionalen Modells graphisch beschrieben. Im Rahmen des Objektentwurfs wurden beide Modelle nach und nach durch Implementierungsdetails verfeinert, die Spezifikationen der Klassen und Assoziationen vervollständigt, die Schnittstellen und die wichtigsten Algorithmen der Operationen festgelegt und interne Objektklassen zur Speicherung der zu bearbeitenden Datenmengen dem Objektmodell hinzugefügt. Schließlich wurde das derart modellierte Problem in Java umgesetzt.

In der Phase des Objektentwurfs stellten sich schnell die Java-eigenen *Collection*-Klassen als zu unflexibel heraus, woraufhin die Studenten nach einer zusätzlichen Bibliothek suchten und sich nach kurzem Test für JGL entschieden. Die Umsetzung des Modells in Java gestaltete sich insofern einfacher als in C++, als in Java mit Ausnahme der primitiven Typen alles ein Objekt ist und selbst primitive Typen sich bei Bedarf in Objekte einbetten lassen: Es gibt keine globalen Funktionen, keine globalen Variablen und Konstanten, keine Aufzählungstypen, ja selbst Exceptions und Events sind als Objekte modelliert. Da zudem Java im Falle nicht-statischer Methoden grundsätzlich die dynamische Bindung verwendet, wurde das polymorphe Verhalten von Objekten klarer und dadurch weniger fehleranfällig.

Mit Hilfe des objektorientierten Konzeptes und durch den Einsatz von Java war es den Studenten möglich, während des gesamten Entwicklungszykluses statt in Begriffen der technischen Lösung in Begriffen der Anwendungsdomäne zu denken und dadurch ohne Paradigmenwechsel zwischen den Phasen abstrakte Konzepte klar auszudrücken und sich darüber zu verständigen. Außerdem konnten die Schnittstellen zwischen den einzelnen Zuständigkeitsbereichen in überschaubarem Ausmaß gehalten werden, wodurch der Abspracheaufwand während der Implementierung erfreulich gering ausfiel.

4.2 Die Implementierung

Die Studenten beider Gruppen zerlegten das Objektmodell in fünf inhaltlich zusammenhängende Subsysteme und teilten deren Implementierung untereinander auf; sie haben größtenteils getrennt voneinander je nach persönlicher Vorliebe auf einer der genannten Plattformen entwickelt und trafen sich periodisch, um sich abzusprechen, das Objektmodell anzupassen und auf dem Referenzsystem die fortschreitenden Systemteile zu integrieren. Sie stießen bei der Integration der verteilt und unter Einsatz verschiedenster Werkzeuge entwickelten Komponenten auf überraschend wenig Probleme. Die folgenden Aspekte von Java haben dazu entscheidend beigetragen:

- strenge Typisierung,
- keine Zeigerarithmetik,
- *garbage collection*,
- strenger Java-Compiler,
- voll integriertes Exception-Konzept.

Die strenge Typisierung von Java erlaubt es sowohl dem Compiler als auch der Laufzeitumgebung, strenge Überprüfungen durchzuführen. Die Studenten waren dadurch dazu gezwungen, sehr sauber und überlegt ihre Systemteile zu implementieren.

Das Java-Exception-Konzept erlaubt es, den eigentlichen Anwendungsquellcode von der Fehlerbehandlung sauber zu trennen und Ausnahmebedingungen nicht unbedingt an der Stelle des Auftretens behandeln zu müssen, sondern an den Teil der Anwendung hochreichen zu können, der qualifiziert ist, darüber zu entscheiden, wie weiter zu verfahren ist.

Da Java keine Destruktoren kennt und somit allein der *garbage collector* über die Lebenszeit eines Objekts entscheidet, waren ungültige Objektreferenzen und Speicherlecks sehr unwahrscheinlich, wenn auch nicht ganz unmöglich. Der *garbage collector* hat sich im Praktikum auch dadurch als sehr hilfreich erwiesen, daß sich die Teilnehmer über ihre Subsysteme hinweg nicht aufwendig abstimmen mußten, wer Objekte erzeugen darf und wer sie wieder zerstören muß. Damit und durch Verzicht auf Zeiger im Sinne von C und C++ entfallen die Hauptfehlerquellen aus C- und C++-Programmen.

Probleme bereiteten unseren Studenten

- zu unflexible Datenstrukturen und Iteratoren,
- Objektübergabe nur als Referenz und
- keine parametrisierten Typen.

Die wissensbasierte Suche benötigt zur Verwaltung des Suchzustandes und der möglichen Transitionen Sprachkonstrukte, die eine variable Anzahl von Objekten repräsentieren und bearbeiten. So müssen nach jedem Zustandsübergang die neumöglichen Transitionen integriert und nun unmögliche Transitionen eliminiert werden. Oder es müssen alle möglichen Transitionen neu bewertet werden, da der Benutzer eine alternative Kontrolle übernehmen lassen will. Das Array-Konstrukt ist dafür nicht geeignet, und die wenigen *Collection*-Klassen in Java

erwiesen sich als zu starr und restriktiv. In Java fehlt bisher hinsichtlich Funktionalität und Flexibilität ein Analogon zur *Standard Template Library* von C++. Die *Java Generic Library* konnte diese Lücke aber teilweise füllen.

Nachdem Speicherfreigabe und Zeigerfehler als größte Fehlerquellen eliminiert wurden, stellte sich bei unseren Studenten die *call-by-reference*-Übergabe anfänglich als Hauptfehlerquelle beim Umgang mit Java- und JGL-Objekten heraus. Versehentliche Änderungen per *call-by-reference* übergebener Objekte waren oft erst sehr viel später, im schlimmsten Fall nur in Form eines falschen Endergebnisses feststellbar. Ursache und Wirkung liegen hier weit auseinander, wodurch sich Fehler dieser Art nur schwer lokalisieren lassen.

Java läßt es mangels parametrisierter Typen zu, in *Collections* Objekte von gänzlich verschiedenen Typen gleichzeitig zu halten. Dabei geht bei der Aufnahme eines Objekts in eine *Collection* scheinbar die Typinformation zunächst verloren, die bei der Entnahme über einen expliziten *type down cast* umständlich wiederhergestellt werden muß. Java weiß dabei diese Typwandlung aber sehr wohl zu überprüfen und weist gegebenenfalls zur Laufzeit auf eine unzulässige Typwandlung hin. Die expliziten Typwandlungen machen den Quelltext umfangreicher und schlechter zu lesen, immerhin aber macht Java im Gegensatz zu C++ auf unzulässige Typwandlungen aufmerksam, auch wenn hier wieder Ursache und Wirkung weit voneinander entfernt sein können.

Alles in allem hat sich aber gezeigt, daß Java zu kürzeren Entwicklungszeiten führt und für Entwicklergruppen, die heterogene Entwicklungsplattformen verwenden, sehr gut geeignet ist. Die reine Java-Syntax ist zwar schnell zu erlernen und zu überblicken, es ist aber recht schwierig und dauert recht lange, ein Gefühl für die Sprache und die der Sprache eigene Philosophie zu bekommen (also dafür, wie man klar, strukturiert, robust und effizient innerhalb der umfangreichen und komplexen Java-Umgebung größere ernsthafte Anwendungen entwickelt).

4.3 Erfahrungen mit der Visualisierung

Natürlich ist die Visualisierung der Ebenen des Suchvorganges Teil der Implementierung, so daß die folgenden Bemerkungen eigentlich zu 4.2 gehören. Allerdings waren die damit verbundenen Ziele ja maßgeblich für unsere Wahl von Java als Implementierungssprache, so daß wir uns für einen eigenen Abschnitt entschieden haben.

Der zentrale Aspekt von Java für die Programmierung graphischer Oberflächen und somit für unsere Visualisierung von Suchprozessen ist das *Abstract Window Toolkit* (AWT). Das Ziel des AWT besteht eigentlich darin, graphikorientierte Anwendungen entwickeln zu können, die auf allen Plattformen gleich gut aussehen. Unsere Studenten bestehen auf Grund ihrer Erfahrungen allerdings darauf, daß das AWT Anwendungen produziert, die auf allen Plattformen gleich *schlecht* aussehen. Das AWT ist relativ komplex, führt zu einem hohen Quellcodeaufwand, stellt nur sehr eingeschränkte graphische Konstrukte zur Verfügung und sieht keine Möglichkeit vor, die fortgeschritteneren graphischen Elemente des jeweils zugrunde liegenden Betriebssystems anzusprechen,

weshalb Java-Anwendungen stets um einiges schlechter aussehen als native Anwendungen.

Die Layout-Manager des AWT stellen eine wichtige Technik zur portablen Programmierung graphischer Oberflächen dar. Sie zeichnen dafür verantwortlich, daß alle graphischen Komponenten in den jeweils plattformspezifischen Dimensionen richtig dargestellt werden, ohne absolute Größenangaben fest in die Anwendung aufnehmen zu müssen. Die derzeit in der Java- Bibliothek angebotenen Layout-Manager sind für unsere Zwecke entweder zu restriktiv und unflexibel oder wiederum zu komplex und zu universell (wie das `GridBagLayout`). An diversen Stellen mußten wiederholt aufwendig Panels ineinander verschachtelt werden, um ein Dialogfenster annähernd so zu gestalten, wie man es von anderen graphischen Benutzungsoberflächen her gewohnt ist.

Unsere über Standardanwendungen hinausgehenden Anforderungen an die Visualisierung führten zu nicht unerheblichen Problemen. Anfängliche Versuche, eine graphisch dargestellte Baumstruktur per *drag and drop* animiert über eine andere Baumstruktur zu ziehen, gaben die Studenten schnell wieder auf, weil aus unerklärlichen Gründen Maus-Events verloren gingen und ab einer schon geringen Baumgröße die Auffrischung der graphischen Darstellung schlicht zu lange dauerte.

Die Visualisierung war der Hauptgrund für die Umstellung von Java 1.0 nach Java 1.1 während des Praktikums. Zwar war diese Umstellung für die Studenten lern- und, wegen mangelnder Dokumentation und Literatur, experimentieraufwendig, aber sie hat sich gelohnt: Das neue Event-Modell fügt sich konsequenter in das objektorientierte Konzept von Java ein und ermöglicht eine klarere, übersichtlichere und strukturiertere Eventbehandlung, auch wenn die geschachtelte Deklaration von Klassen zunächst dem Wiederverwendungsgrundsatz zu widersprechen schien und der Einsatz von Interfaces gewöhnungsbedürftig war.

Alles in allem gestaltete sich der direkte Einsatz des AWT ohne visuelle Entwicklungswerkzeuge als sehr umfangreich, aufwendig und fehlerintensiv.

4.4 Erfahrungen mit Entwicklungswerkzeugen

Die folgenden Berichte und Wertungen beruhen auf den Ausarbeitungen der Studenten, die im Rahmen des Praktikums angefertigt wurden. Sie spiegeln den Stand der Werkzeuge im Zeitraum April bis Juli 1997 wieder und sind natürlich unter den Aspekt der gestellten Aufgabe und individuellen Präferenzen zu sehen.

Java Develpment Kit Wir haben uns bei unseren Studenten nicht unbedingt dadurch beliebt gemacht, daß wir nur das JDK als Entwicklungsumgebung zur Verfügung gestellt haben. In der Ausarbeitung einer Gruppe las sich das wie folgt:

> Diese Entwicklungsumgebung ist rein kommandozeilenorientiert und nur sehr spartanisch ausgestattet. Insbesondere der Debugger `jdb` verbreitet das Flair der 70er-Jahre und ist bei ernsthaften Anwendungen mit mehreren Threads fast nicht zu gebrauchen; es lebe das `println`-Debugging! Der Java-Compiler `javac` "quält" sich regelrecht durch den Quellcode.

Allein das `javadoc`-Werkzeug war zusammen mit der *java documentation comment syntax* bei den Praktikumsteilnehmern geachtet, da sich mit `javadoc` aus besonders formatierten und direkt in den Quelltext eingefügten Kommentaren eine detaillierte und strukturierte Klassendokumentation auf HTML-Basis generieren läßt. Diese Dokumentation war für die Studenten eine wichtige Grundlage für die Koordination und Kooperation während der Implementierung.

Java WorkShop 1.0 Der *Java WorkShop 1.0* von Sun bietet einen visuellen GUI-Builder, einen schnellen Compiler und einen komfortablen Debugger. In der Version 1.0 war der *Java WorkShop* zumindest unter Windows 95 viel zu langsam, um mit ihm vernünftig arbeiten zu können, und steht leider nur für Solaris und Windows95/NT zur Verfügung. Der GUI-Builder hält sich zwar streng an die AWT-Vorgaben zur portablen Programmierung, ist aber recht umständlich zu handhaben. Die mittlerweile ausgelieferte Version 2.0 läßt sich um einiges zügiger und angenehmer bedienen.

Visual Café 1.0 *Visual Café 1.0* von Symantec enthält einen schnellen Compiler und einen komfortablen Debugger, einen Klassen-Browser und Hierarchie-Editor und außerdem ein Tool, um Java 1.0-Anwendungen nach Java 1.1 zu konvertieren.

Alles in allem ließ sich mit *Visual Café 1.0* recht angenehm arbeiten, einige üble Programmfehler trübten jedoch das Bild immer wieder. Zudem verleitet der visuelle GUI-Builder (FormDesigner) dazu, plattformspezifische, auf Windowssysteme zugeschnittene graphische Anwendungen zu entwickeln, gleichwohl besteht versteckt dennoch die Möglichkeit, durch Einsatz von Layout-Managern portablen AWT-konformen Quelltext zu erzeugen. Leider gibt es *Visual Café* nur für Windows95/NT und für Macintosh-Rechner.

VisualAge for Java 1.0 *VisualAge for Java 1.0* von IBM bietet einen komfortablen Debugger, einen inkrementellen Compiler, einen visuellen GUI-Builder zur Entwicklung portabler Anwendungen und ein integriertes Versionskontrollsystem. Die Studenten fanden *VisualAge for Java* sehr leistungsfähig und komfortabel. Der Entwickler wird von Routinetätigkeiten entlastet und aufmerksam unterstützt.

So angenehm sich allerdings das *"VisualAge for Java works hard so you don't have to"* auch ausnimmt, verlangt VisualAge doch nach einer recht üppigen Hardwareausstattung insbesondere an Hauptspeicher, um volle Leistung zeigen zu können. Das Workbench-Konzept (keine Dateien, sondern Projekte, Packages, Klassen, Methoden) ist sehr komfortabel, so lange alle Entwickler VisualAge durchgängig einsetzen. Weil VisualAge aber gerade für die im Praktikum eingesetzte Referenzplattform unter Solaris nicht verfügbar war, mußten alle Projektdateien immer wieder aufwendig in die VisualAge-Ablage importiert und aus der Ablage exportiert werden.

5 Das Endsystem

Im folgenden wollen wir die Erfüllung der Ziele aus Abschnitt 3 durch die Praktikumssysteme beschreiben und außerdem auf die Effizienz der Systeme eingehen.

Beide im Rahmen des Praktikums entstandenen Systeme erfüllen die Ziele bezüglich der Visualisierung und der Interaktion mit dem Benutzer sehr gut, wobei aber für die unterschiedlichen Darstellungsebenen die Testbenutzer unterschiedliche Systeme bevorzugen. Bei den Gründen, die für die Wahl von Java sprachen, hat sich die Plattformunabhängigkeit ja schon bei der Entwicklung eindrucksvoll bestätigt und dies war auch beim Endprodukt der Fall. Auf allen von uns getesteten Hardware/Betriebssystem-Kombinationen lief der für die Java Virtual Machine erzeugte Code in den entsprechenden Java-Umgebungen problemlos ab und auch die Visualisierungselemente zeigten sich gleich (bei Verwendung des im JDK enthaltenen *appletviewer*).

Die Verwirklichung unserer Ziele bezüglich des WWW-Zugriffs auf die Systeme kann man im Gegensatz zu den bisherigen Bemerkungen nur als Desaster bezeichnen. Zur Zeit des Praktikums war HotJava der einzige verfügbare WWW-Browser, der Java 1.1 und insbesondere das darin enthaltene neue Event-Modell unterstützte. Auch zum jetzigen Zeitpunkt hat sich die Situation nicht viel verbessert. Zwar gibt es mittlerweile ein Patch für den Netscape Communicator 4.04, das die Darstellung von Java 1.1 ermöglicht, aber dafür sind unsere Testbenutzer mit anderen Teilen von Netscape sehr unzufrieden. Die Problematik Java–Microsoft ist auch hinlänglich bekannt. Sollten sich die Browser mit Einführung von Java 1.2 ähnlich (schleichend) entwickeln, stellt sich für uns die Frage, ob wir nicht zurück zu Java 1.0 gehen sollen, um unser Ziel verwirklichen zu können.

Obwohl wir bei diesem Praktikum den Schwerpunkt auf andere Ziele gelegt haben, haben wir trotzdem auch die Effizienz der entwickelten Systeme getestet. Frühere Praktika, bei denen wir Studenten algorithmisch gleiche Aufgaben in Pascal, C, C++, Common Lisp, Prolog und Caml lösen liesen, hatten bisher gezeigt, daß interpretierte Programme compilierten deutlich unterlegen waren. Die Vergleiche von C-Programmen mit C++-Programmen, die wirklich objektorientiert geschrieben waren, endeten mit einer großen Überlegenheit der C-Programme bezüglich der Effizienz. Deshalb, und weil für Optimierungen der Praktikumssysteme im Hinblick auf Effizienz keine Zeit vorhanden war, erwarteten wir von den Java-Systemen eigentlich keine Effizienz beim Lösen etwas schwierigerer Beispiele.

Da sich aber die ersten Tests erfolgreich gestalteten, haben wir die Systeme an den Beispielen getestet, die beim CASC-14 (siehe [6]), dem jährlichen internationalen Theorembeweiserwettbewerb im Rahmen der CADE, benutzt wurden (ohne Verwendung des Visualisierungsmodus). Dabei konnte das beste der Java-Systeme 30 der 50 Probleme in der vorgegebenen Zeit lösen (zum Vergleich: das Gewinnersystem, Waldmeister ([2]), löste 49, das schlechteste System im Wettbewerb 37). Diese Zahlen lassen das Java-System nicht unbedingt als effizient erscheinen, aber in Anbetracht der Ziele des Praktikums ist dies eine

überraschend gute Leistung, die zu einem nicht kleinen Teil Java und der Java-Laufzeitumgebung zuzuschreiben ist.

Alle Systeme, die an CASC-14 teilnahmen, hatten auch an CASC-13 (siehe [4]) teilgenommen. Dort löste das beste System 43 der 50 (anderen, aber vergleichbar schwierigen) Probleme, das schlechteste 24. 3 Systeme konnten nicht mehr als 30 Probleme lösen. Dies zeigt, welche Verbesserungen in einem Jahr möglich sind (bei Waldmeister wurde zum Beispiel die wissensbasierte Kontrolle und die initialen Aktionen weiterentwickelt) und unsere Erfahrungen mit Waldmeister lassen uns glauben, daß entsprechende konzeptionelle Verbesserungen auch an den Java-Systemen möglich sind.

6 Fazit

Die Verwendung von Java hat sich positiv sowohl auf das Praktikum als auch auf die Erfüllung der Ziele für das zu erstellende System ausgewirkt. Der praktische Totalausfall der Browser im Anwendungsszenario ist nur indirekt Java, über die sehr großen Änderungen in Java 1.1, zuzuordnen und wird hoffentlich bald der Vergangenheit angehören. Natürlich wird dadurch das intendierte Fernstudiumspotential des Systems drastisch eingeschränkt, aber alle anderen Ziele wurden zu unserer Zufriedenheit, und der der Testbenutzer, erfüllt. Dies wäre bei Verwendung einer anderen Implementierungssprache mit ziemlicher Sicherheit in der zur Verfügung stehenden Zeit so nicht möglich gewesen. Auch die Effizienz der entstandenen Systeme war mehr als zufriedenstellend. Herauszuheben ist die Bedeutung der Plattformunabhängigkeit auch bei der Entwicklung.

Unser nächstes Ziel ist ein System, das die jeweils besten Ideen zur Visualisierung der unterschiedlichen Ebenen und Vorgänge vereint und außerdem eine retrospektive Sicht auf die Zielerreichung zu jedem Zeitpunkt gibt. Sobald von einer größeren Verbreitung von entsprechend fähigen Browsern auszugehen ist, wollen wir dieses System in WWW-Seiten, die die Definitionen aller Vorgänge darstellen, zur beispielhaften Visualisierung dieser Definitionen nutzen (gemäß unserer ursprünglichen Vision) und außerdem das System um Übungsaufgaben und einen entsprechenden Dialog mit dem Übenden zu Verständnisproblemen erweitern.

Literatur

1. Avenhaus, J.: *Reduktionssysteme*, Springer, 1995.
2. Hillenbrand, T.; Buch, A.; Vogt, R.; Löchner, B.: WALDMEISTER. High Performance Equational Deduction, *Journal of Automated Reasoning* **18**(2), 1997, pp. 265–270.
3. Rumbaugh, J.; Blaha, M.; Premerlani, W.; Eddy, F.; Lorensen, W.: *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
4. Sutcliffe, G.; Suttner, C.: The Results of the CADE-13 ATP System Competition, *Journal of Automated Reasoning* **18**(2), 1997, pp. 271–286.
5. `http://www.ObjectSpace.com`
6. `http://www.cs.jcu.edu.au/~tptp/CASC-14/`