Visual J++, nicht visuell

Silke Seehusen, Fachhochschule Lübeck

Visual J++ soll der einfachen Entwicklung und teilweisen Generierung von Anwendungen mit grafischen Benutzungsoberflächen dienen, die in Java realisiert und auf allen Systemen mit einer virtuellen Java-Maschine lauffähig sind. Es wird untersucht, wie leicht und visuell intuitiv es wirklich ist, eine solche Anwendung zu erstellen, wie gut der erzeugte Quellcode weiterverwendet werden kann und wie maschinenunabhängig der Code ist.

Visual J++ und Visual C++

Visual J++ ist ein Name, der sofort an Visual C++ erinnern soll. Deshalb erwarten wir eine ähnlich komfortable und mächtige Programmentwicklungsumgebung, wie Visual C++ es ist ([1]). Da das **J** (wahrscheinlich) für Java stehen soll, muß J++ viel mehr als Java sein. C++ ist, im Gegensatz zu C, objektorientiert. Java ist schon strikt objektorientiert, und was ist nun J++ und was VJ++?

J++ steht für Java (und einen guten Werbegag). Die Programmiersprache Java hat auch Microsoft noch nicht erweitert. Diese Aufgabe übernimmt zur Zeit Sun Microsystems mit ausgewählten Partnern.

Visual steht dafür, daß die Bedienoberfläche an die von Visual C++ angelehnt ist. Visual J++ ist eine Komponente des Microsoft Developer Studios und hat hinsichtlich Fensterrahmen, Knopfleisten, Menüs und Tastatureingaben (*Short Cuts*) die gleiche Bedienoberfläche wie die anderen Komponenten, wie z.B. Visual Basic oder Visual C++. Es soll deshalb im folgenden nicht im Detail darum gehen, wie durch welchen Mausklick welche Aktion hervorgerufen wird, sondern darum, in welchem Umfang Code erzeugt wird, wie einfach es ist, im erzeugten Code die Stellen mit Leben zu füllen, damit es die gewünschte Anwendung wird, wie änderungsfreundlich und wie plattformunabhängig der erzeugte Code ist.

Plattformunabhängigkeit

Java ist hauptsächlich deshalb so bekannt geworden, weil es sehr leicht ist, HTML-Seiten mit Animationen zu schreiben. Des weiteren können Anwendungen wie z.B. die Klienten-Software eines Bestellwesens in HTML-Seiten im Internet zur Verfügung gestellt werden.

Folgende Aspekte von Java, die später auch noch einmal in Verbindung mit Visual J++ untersucht werden, sind dafür notwendig und wichtig:

- Java-Programme sind plattformneutral, d.h., der ausführbare Code (Bytecode) ist auf allen Plattformen lauffähig, auf denen eine virtuelle Java-Maschine verfügbar ist, und die Programme liefern auf jeder Plattform die gleichen Berechnungsresultate. Die Oberflächengestaltung kann geringfügig voneinander abweichen, die Logik einer grafischen Bedienoberfläche muß überall gleich sein.
- Zu Java gehört eine sehr umfangreiche Bibliothek, die sogenannten APIs (*Application Programming Interface*), die u.a. alle Elemente enthalten, die zur Programmierung von grafischen Bedienoberflächen benötigt werden.
- In Java können elegant Inter- und Intranet-Anwendungen geschrieben werden, da *Remote Method Invocation*, Datenbankzugriff über JDBC und Internetprotokolle wie *http* und *ftp* über URLs aus der Bibliothek direkt verfügbar sind.

Der alte Traum von Rechnerunabhängigkeit soll vielleicht wahr werden. Hilft uns VJ++ dabei?

Der Aspekt, daß Visual J++ eine Entwicklungsumgebung zur Entwicklung allgemeiner Java-Programme ist, wird hier nicht näher beschrieben. Details können in der Begleitdokumentation von Visual J++ und Beispiele z.B. in [2] nachgeschlagen werden.

Ein leeres Applet

Wir beginnen mit einem einfachen Beispiel und entwickeln unter maximaler Hilfestellung von Visual J++ ein leeres Applet, d.h., ein Applet, das ausführbar ist, aber eigentlich nichts tut (also auch nichts Böses).

Dazu definieren wir mit Visual J++ ein neues Projekt namens *LeerApp*, das vom sogenannten Applet Wizard erzeugt werden soll, wobei wir mit den Voreinstellungen einverstanden sind, daß es nur ein Applet sein soll, daß automatisch eine passende HTML-Seite erstellt wird, daß aber auf eine automatisch erstellte Animation verzichtet werden soll.

Die Aufgaben des Applet Wizards bestehen allgemein darin,

- ein Projekt für das Applet zu erzeugen,
- die Klassen und *.java*-Dateien anzulegen, die für die gebräuchlichsten Programmieroptionen erforderlich sind,
- das Grundgerüst einer Reihe von Methoden für diese Optionen zu erstellen,
- Kommentare hinzuzufügen, die die Funktion der einzelnen Methoden beschreiben und der Programmiererin einen Hinweis geben, wo der anwendungsspezifische Code eingefügt werden muß, und
- eine (einfache) HTML-Seite zum Ausführen des Applets zu erstellen.

In unserem einfachen Beispiel werden folgende Dateien für das Applet erstellt:

LeerApp.java	Applet-Klasse, Java-Quellcode.
LeerApp.html	HTML-Datei zum Aufruf des Applets.

Für das Projekt werden entsprechend den Konventionen des Microsoft Developer Studios die Projektdatei *LeerApp.mdp*, eine *nmake*-Datei *LeerApp.mak* und eine Datei mit den persönlichen Einstellungen *LeerApp.ncb* erstellt. Diese drei Dateien werden vom Microsoft Developer Studio verwaltet, deshalb betrachten wir sie nicht weiter.

Wenn wir das Applet jetzt per Menüauswahl übersetzen und ausführen lassen, wird der voreingestellte HTML-Browser, der Microsoft Internet Explorer, aufgerufen und präsentiert sich wie in Abbildung 1 auf dem Bildschirm. Das Applet belegt den Bereich zwischen den beiden horizontalen Linien und schreibt die erste Zeile in das Fenster. Unten steht ein Hyperlink auf die Quelldatei *LeerApp.java*.

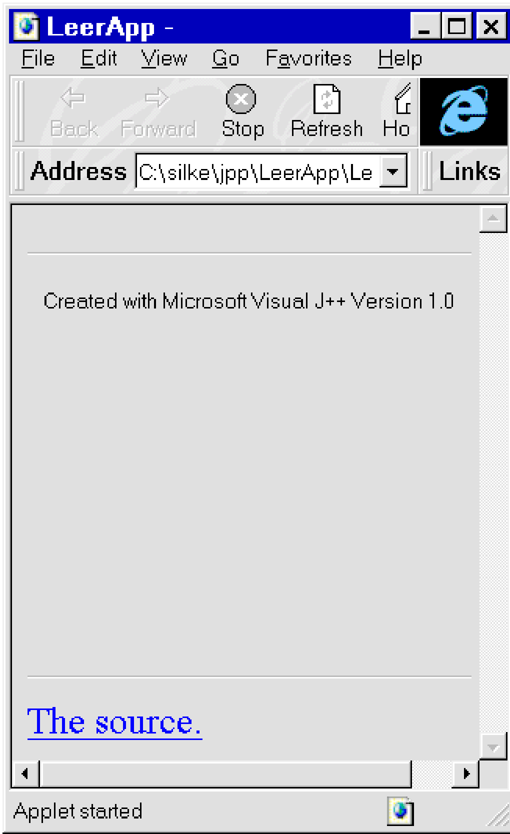

Abbildung 1: Applet LeerApp

Die Klasse LeerApp

Einen Überblick über den erzeugten Code der Klasse *LeerApp* bietet die Klassenübersicht, die das Microsoft Developer Studio anzeigt wie in Abbildung 2.

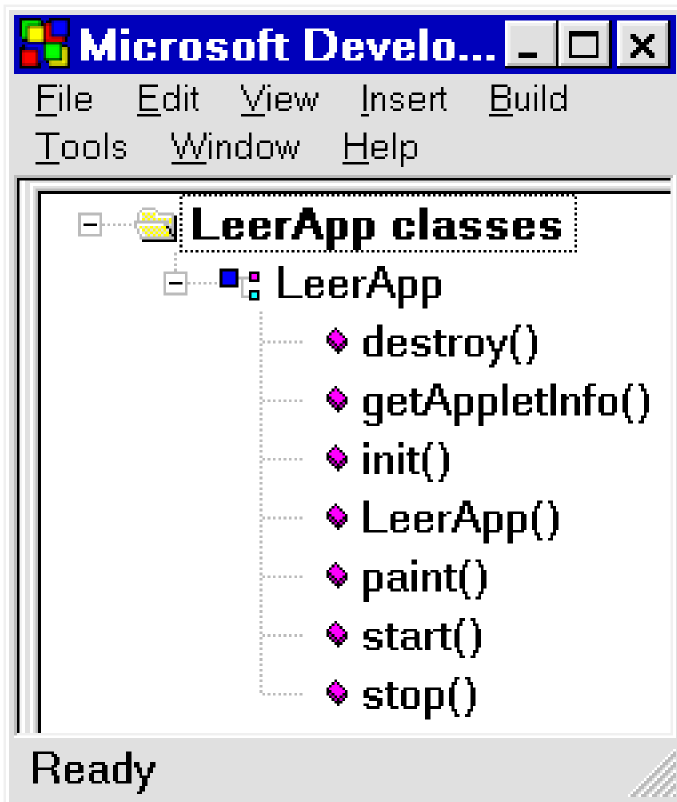

Abbildung 2: Erzeugte Klasse LeerApp eines "leeren" Applets

Die Klasse wird deklariert als

```
public class LeerApp
       extends Applet
```

Es werden neben dem leeren Konstruktor die Methoden für das Applet erzeugt, die vom Browser aufgerufen werden und die von einem Applet immer überschrieben werden sollten. Es sind dies *init()*, *start()*, *stop()* und *destroy()*.

Der erzeugte Code für eine Methode besteht aus dem Methodenkopf, teilweise sinnvollem und teilweise eigenwilligem Code und entsprechendem Kommentar, der der Programmiererin einen Hinweis darauf gibt, was noch manuell einzutragen ist.

Die Methode *init()* z.B. besteht aus viel Kommentar und einer Codezeile:

```
// The init() method ...
// reloaded. Override ...
// ...
//---------------------...
public void init()
{
  // If you use a Reso...
  // ...
  //-------------------...
  resize(320, 240);

  // TODO: Place addi...
  // initialization c...
}
```

Die drei Punkte stehen jeweils für weiteren Kommentar. Der erste Kommentar beschreibt noch einmal, wofür die Methode eigentlich da ist. Das steht auch in jedem Java-Buch. Die Generierung des Kommentars kann per Option verhindert werden.

Der zweite Kommentar bietet einen Hinweis darauf, daß die einzige generierte Code-Zeile in der Regel gelöscht werden sollte. Der dritte Kommentar, der mit *TODO* beginnt, markiert die Stelle, wo Code einzufügen ist.

Diese Art Kommentar findet sich in allen Methoden. Leider gibt es keinerlei Kommentar, weder vor den Methoden noch vor den Klassen, aus dem das Werkzeug *javadoc*, das zum Java Development Kit (JDK) gehört, eine sinnvolle Dokumentation im HTML-Format generieren könnte. Visual J++ kennt diese Art Kommentar offensichtlich nicht.

Die Methoden *start()*, *stop()* und *destroy()* besitzen alle einen leeren Rumpf. Die Methode *paint()* schreibt den in Abbildung 1 sichtbaren Text in das Fenster:

```
.ps 8
// LeerApp Paint Handler
//——————————————————————————————
public void paint(Graphics g)
{
  g.drawString("C...",10,20);
}
```

Neben diesen Methoden wird die öffentliche Methode *getAppletInfo()* generiert, die immerhin den Namen des Applets und den Namen der Erstellerin zurückliefert.

In dieser einfachsten Benutzung des Applet Wizards liefert er eigentlich nicht mehr, als eine Java-Programmiererin sowieso in einer Art Rahmendatei für ein Applet ablegt und immer wieder benutzt.

Mit Threads

Wenn wir bei der Generierung des Applets Threads wünschen, hat die generierte Applet-Klasse, wir nennen sie *LeerAppT*, zusätzlich das Attribut *m_LeerAppT*:

```
.ps 8
// THREAD SUPPORT:
// m_LeerAppT is the Thread
//  object for the applet
//——————————————————————————
Thread m_LeerAppT = null;
```

Eine Übersicht über die Klasse zeigt Abbildung 3.

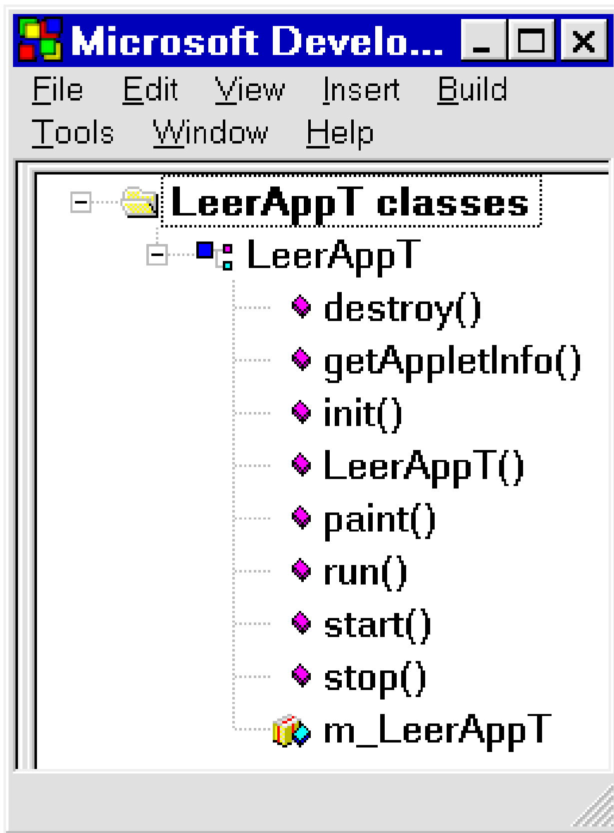

Abbildung 3: Erzeugte Klasse LeerAppT mit Threads

Da das Applet *LeerAppT* das Interface *Runnable* implementiert, muß es die Methode *run()* implementieren. Generiert wird Code, der in einer Endlosschleife *repaint()* aufruft und zwischendurch 50 ms schläft, was nicht besonders sinnvoll ist. Auf Ausnahmen reagiert die Methode mit dem Aufruf von *stop()*.

Die Methode *paint()* zeigt einen Text, der jeweils eine andere Zufallszahl darstellt, wie in Abbildung 4. Die Betrachterin sieht eine sich ständig ändernde Zahl.

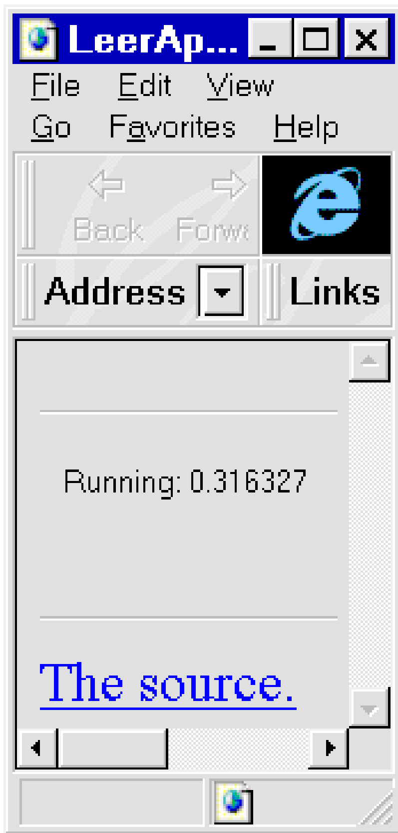

Abbildung 4: Applet LeerAppT mit Threads

Der Thread *LeerAppT* wird in der Methode *start()* erzeugt und gestartet:

```
.ps 8
public void start()
{
 if (m_LeerApp == null)
 {
  m_LeerAppT=new Thread(this);
  m_LeerAppT.start();
 }
 // TODO: Place additional
 // applet start code here
}
```

In der Methode *stop()* wird der Thread gestoppt und auf *null* gesetzt. Die Methode *destroy* wird mit einem leeren Rumpf generiert.

Applet mit allem

Lassen wir vom Applet Wizard alles erzeugen, was er kann, erhalten wir eine Klasse *Leer* für ein Applet und eine Klasse *LeerFrame* für eine eigenständige Applikation, die das Applet in einem Frame ausführt. Eine Übersicht der Klassen ist in Abbildung 5 gegeben.

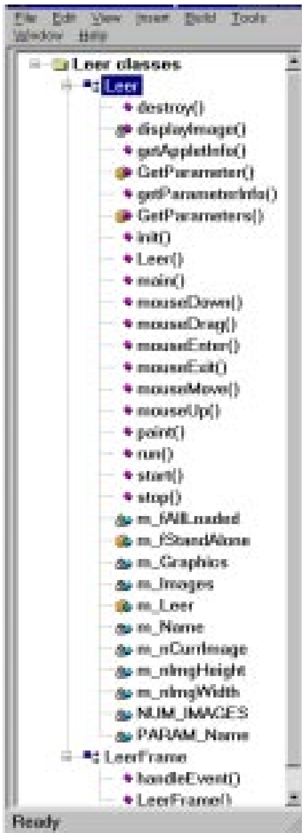

Abbildung 5: Erzeugte Klasse Leer “mit allem”

Die Ausführung der Applikation liefert eine rotierende Erde wie in Abbildung 6.

Abbildung 6: Applikation Leer

Dasselbe gibt es auch als Applet nach dem obigen Schema. Die rotierende Erde ist eine Folge von Gif-Bildern. Dieser sogenannte Animation-Support steckt in den Methoden *display-Image()* zum Zeichnen eines Bildes und *run()* zur entsprechenden Verwaltung eines Objektes der Klasse *MediaTracker*.

Zum Einlesen von Applet-Parametern wird die Methode *GetParameters()* generiert.

Applikation

Für die Applikation wird ein Hauptprogramm *main()* generiert, das einen *Frame* erzeugt und darin das Applet zeigt.

Das Attribut *m_fStandAlone* zeigt an, ob es als Applikation oder als Applet ausgeführt wird.

Zur Ereignisbehandlung wird die Methode *handleEvent* generiert, die erst einmal nur auf das Ereignis *WINDOW_DESTROY* mit *System.exit(0)* reagiert.

Mausereignisse

Für die Mausereignisse wie Mausdrücken werden für das Applet leere Methoden wie *mouseDown()*, *mouseDrag()* etc. generiert, die allerdings schon mal *true* zurückliefern, obwohl sie das Ereignis gar nicht behandeln. Die generierten Mausbehandlungsroutinen sind in Abbildung 5 aufgelistet.

Resource Wizard

Die generierten Klassen mögen vielleicht für echte Java-Anfänger ganz nett sein, um so weniger nützlich sind sie jedoch für richtige Anwendungen. Aber vielleicht liegt die Stärke der Zauberei in dem Resource Wizard.

Der Resource Wizard soll einen Teil der Routineaufgaben übernehmen, die beim Erstellen komplizierterer Fenster anfallen. Die Programmiererin erstellt das Layout mit dem sogenannten Resource Editor. Der Resource Wizard erzeugt daraus entsprechende Java-Klassen. Zur Zeit wird nur die Erstellung von Dialogen und Menüs unterstützt. Stringtabellen z.B. können mit dem Resource-Editor erstellt werden, aber der Resource Wizard kann damit nichts anfangen.

Um den Resource Wizard auszuprobieren, nehmen wir uns vor, eine Anwendung zu schreiben, mit der Studierende sich in einem Campus-Netz beim Systemadministrator anmelden können, in der Hoffnung, daß sie wenig später ein Login eingerichtet bekommen.

Anmeldung

Eine Anwendung der Applikation wird in Abbildung 7 gezeigt. Dort sind nur Elemente verwendet, die der Resource Wizard kennt und die mit *Drag&Drop* mithilfe des Resource Wizards generiert werden können, als da sind Knöpfe (Button), Marken (Label), Textfelder (TextField), Checkboxen (Checkbox) und Auswahlen (Choice).

Abbildung 7: Applikation Anmelde

Diese Elemente sind mit dem wirklich visuellen Resource Wizard erstellt und deren ungefähre Position im Fenster bestimmt worden. Generiert werden daraus zwei Klassen, die Klasse *AnmeldeDialog.java* für die Zusammenstellung der Elemente und die Klasse *DialogLayout.java*, in der ein spezieller Layoutmanager implementiert ist. Dieser Layout-manager implementiert die Schnittstelle *LayoutManager*. Es ist kein Standard-Layoutmanager aus der API, sondern speziell für Visual J++ geschrieben. Er benutzt nur absolute Positionen im Fenster, wobei die Schriftgröße in die Berechnungen miteinbezogen wird.

Das Resultat ist ein Layout, an dem noch viel Zeit mit Versuch-und-Irrtum investiert werden muß, um ein halbwegs vernünftiges Layout wie in Abbildung 7 zu erreichen. Gleiche Größe von Textfeldern etc. wird eigentlich nur durch Änderung der entsprechenden Zahlen im generierten Code erreicht. Die relative Ausrichtung von Elementen erfolgt auch fast ausschließlich über Codeänderungen.

Ein Relikt von Versuchen, es ausschließlich mit dem Resource Wizard zu erreichen, ist bei der Ausrichtung der Labels Name, Vorname und Matrikelnummer zu sehen. Sie sollten alle rechtsbündig übereinanderstehen.

Es stellte sich bei weiteren Versuchen heraus, daß der erzeugte Layoutmanager immer gleich ist (identischer Code), was an und für sich gut wäre, wenn er nicht so unhandlich wäre. Mit den Layoutmanagern der Java-API wie *FlowLayout* oder *GridBagLayout* wäre ich schneller und zu einem besseren Ergebnis gekommen.

Die Einbettung der generierten Klasse in das zunächst als leer generierte Applet kann nach dem Schema erfolgen, das in der Online-Dokumentation beschrieben ist. Etwas Code ist dabei zu schreiben. Den ClassWizard aus Visual C++ gibt es nicht.

Die erzeugten Klassen haben nach der Erzeugung keinerlei Verbindung mehr zur Ressource. Eine Änderung der Ressource hat somit keinen Effekt auf die Anwendung. Nur durch Neugenerierung aus der Ressource wird die Änderung wirksam. Dabei werden, ohne jegliche Rückfrage, schon existierende Java-Dateien überschrieben.

Mit dem Resource Wizard können noch Elemente wie Ikonen erstellt werden, aber die Einbindung in das Java-Programm wird nicht unterstützt. Der Resource Wizard unterstützt z.B. auch keine Textresourcen wie der Resource Wizard von C++.

Der Resource Wizard von Visual J++ trägt den gleichen Namen wie der Resource Wizard von Visual C++. Während aber mit dem Resource Wizard von Visual C++ wirklich visuell vernünftige Bedienoberflächen erzeugt werden können, ist der Resource Wizard von Visual J++ weit davon entfernt. Die jetzige Version ist meiner Meinung nach sogar kontraproduktiv.

Online-Dokumentation

Über die Online-Hilfe erreichbar sind die Übersetzungen der Bücher über die Sprachspezifikation und die API-Referenz, die im Addison-Wesley-Verlag erschienen sind. Sie sind mit entsprechenden Hyperlinks versehen worden, die auch in der HTML-Version zu finden sind (siehe [3]).

Systemabhängige Erweiterungen

Visual J++ bietet, wie bei Microsoft üblich, systemabhängige Erweiterungen, die nur auf Systemen von Microsoft lauffähig sind und damit den Zielen von Java widersprechen.

Zu diesen Erweiterungen gehört der Durchgriff auf ActiveX und COM-Objekte. Der Java-Type-Library Wizard generiert sogenannte Java-Wrapper-Klassen für COM-Objekte.

Die Java-Programme, die diese Erweiterungen benutzen, sind sehr eingeschränkt einsetzbar, da sie plattformabhängig sind. Für den Anwender von solchen Java-Programmen bedeutet es darüber hinaus, daß das Sicherheitskonzept von Java damit durchbrochen wird. Ein Schutz des lokalen Systems ist nicht mehr gewährleistet, da z.B. auf die lokale Platte zugegriffen werden kann, die vollständig gelöscht werden könnte.

Um dieses Problem zu entschärfen, gehört zu Visual J++ ein CAB&Sign-Toolkit. Cabinet-Dateien (CAB files), eine Eigenentwicklung von Microsoft, entsprechen ungefähr von der Intension und der Anwendung her den Java-Archiven (JAR files) in Java 1.1. Die digitalen Signaturen, die mit dem CAB&Sign-Toolkit erzeugt und benutzt werden können, sind auch microsoftspezifisch. In Java 1.1 gibt es dazu die Java Security API, mit der Schlüssel und digitale Signaturen erzeugt und verwendet werden können.

Übrigens: Das schöne Beispiel Exploder-Control zeigt, daß man einen Virus mit z.B. ActiveX-Erweiterungen erstellt, dann einfach bei einem Anbieter von Zertifikaten wie z.B. Microsoft verifizieren und mit einer digitalen Signatur versehen läßt und dann im Internet zur Verfügung stellt. Das Beispiel führte zum Glück nur zu einem Shutdown des Rechners (aus Demonstrationszwecken), auf dem das Applet ausgeführt wurde, und ließ die lokale Platte unversehrt.

Zusammenfassung

Visual J++ ist keine große Hilfe bei der Entwicklung von richtigen Java-Anwendungen, wie es Visual C++ bei der Entwicklung von C++-Anwendungen ist.

- Der Applet Wizard erzeugt ein Programmgerüst, das auch aus einem normalen Java-Buch abgeschrieben werden kann. Das Programmgerüst hilft im wesentlichen Java-Anfängern.
- Der Resource Wizard erzeugt Code, der per Hand mit Versuch-und-Irrtum nachgebessert werden muß, um ein akzeptables Layout zu bekommen.
- Der erzeugte Layoutmanager kennt eigentlich nur absolute Positionen. Die Layoutmanager der API sind besser benutzbar.
- Eine nachträgliche Änderung einer Ressource mit dem Resource Editor hat keinen Effekt auf schon überarbeiteten Code.
- Es gibt keinen Class Wizard (wie in Visual C++). Die erzeugten Klassen müssen per Hand eingefügt werden.
- Es wird keinerlei Kommentar für eine Dokumentation erzeugt, aus der *javadoc* eine Programmdokumentation im HTML-Format erzeugt.
- Der Durchgriff auf ActiveX und COM-Objekte wird leicht gemacht. Dadurch verliert ein Java-Programm jedoch alle wesentlichen Vorteile gegenüber einem C++-Programm wie Plattformunabhängigkeit und Sicherheitsüberprüfung.

Visual J++ bietet einem Benutzer von Visual C++ zunächst einen leichten Übergang zur Erstellung eines ersten Java-Programms, da die Entwicklungsumgebung hinsichtlich Menüs, Tastenbelegungen und der Struktur der Bedienoberfläche wie die von Visual C++ aussieht. Aber eine weitergehende Unterstützung für eine professionelle Java-Entwicklung bietet Visual J++ nicht.

Es ist anzunehmen, daß es neben der angekündigten Erweiterung auf Java 1.1 weitere Zusätze wie einen Class Wizard oder einen Multidokument-Rahmen geben wird.

Nach dem, wie Microsoft bisher agiert hat, wird versucht werden, mehr plattformabhängige Elemente aufzunehmen. Das zerstört die Ideen von Java. Plattformneutralität soll wieder ein Fremdwort werden.

Literatur

[1] S. Seehusen “Visuelle Zauberei oder Visual C++” in: Die Blauen Blätter, Nr. 1/1997.
[2] Stephen R. Davis “Java jetzt!” Microsoft Press, Unterschleißheim, 1996.
[3] Sun Microsystems “The Java Language Specification” 1996.