08.04

Javamagazin

Deutschland € 5,50 Österreich € 6,50 Schweiz SFr 10,70

S&S

Javamagazin™

Internet & Enterprise Technology

XML extra included

Tomcat & Apache

Die coolsten Tricks für Tomcat & Apache Webserver

Hibernate

Übersicht, Design Patterns, Performance-Tipps

www.javamagazin.de

Logging API
Crashkurs für Anwendungsprogrammierer

Jetzt neu! Der Java-Knigge
Ist Java wirklich langsam?

eBay mit JCA
Eigene Resource Adapter entwickeln

Meta-Framework Keel
Frameworks integrieren

Identity Management
Authentifizierungslösungen für das WWW

mit CD!

D 45867

4 194586 705506 08

Konzepte einer performanten Prozessvisualisierung für Web Clients

von Edgar Dehm

Im Bilderrausch

Vierzig Kilometer Transporttechnik, dargestellt auf einer Videowand in einem riesigen Übersichtsbild. Details der Anlage, die aberwitzig viele Gepäckstücke verteilt, umfassen knapp 200 Bilder mit dynamischen Eigenschaften, die in einem Applet angezeigt werden. Um dieser Herausforderung gerecht zu werden, sind konzeptuelle Planung, automatisierbare Testverfahren sowie Kenntnisse über das Java 2-D Graphics API unabdingbar.

Einführung

Sicherlich ist Beratern auf dem Gebiet der Java-Technologie folgende Situation vertraut: In einem Softwareprojekt wird über die mäßige Ausführungsgeschwindigkeit eines Applets geklagt, das die Anwendungsschicht einer Web-basierten Lösung bereitstellt. Ohne Umschweife wird die Ursache in der Java-Plattform vermutet. Die Begründung erfahrener Veteranen aus dem industriellen Umfeld vor Ort, die die Legacy-Anbindung zur Hardware pflegen, klingt zunächst einleuchtend: „Vor Java ging's doch auch!"

Es folgt eine Analyse der historisch gewachsenen Lösung. Bei Projektbeginn gab es keine Alternative zu den Zeichenroutinen des Abstract Windowing Toolkit, aber auch nachträgliche Erweiterungen lassen die Möglichkeiten einer modernen, optimierten Grafik-Bibliothek [1] ungenutzt. Schon bald stellt sich heraus, dass überschaubare Refactoring-Maßnahmen [2] nicht ausreichen, um die Ausführungsgeschwindigkeit spürbar zu steigern.

Diese auf den ersten Blick verfahrene Ausgangssituation bietet die Chance für ein erfolgreiches Redesign. Die Anforderungen liegen mit dem Code vor und die als zeitraubend identifizierten Faktoren können von Anfang an ausgemerzt werden. Während die Altanwendung zum Prototypen degeneriert, kann sie weiterhin zu Planspielen mit Testszenarien herangezogen werden. Die folgende Auswahl von konzeptuellen Fehlern entstammt der Analysephase:

- Missachtung von OOA/D-Paradigmen
- Einsatz veralteter Methoden und Packages
- Lange synchronisierte Bereiche
- Hintereinanderschaltung von String-Operationen
- Wiederholte Erzeugung anstelle Wiederverwendung
- Unnötige Ausführung von Zeichenfunktionen

Im Unterschied zu serverseitigem Java bleibt anzumerken, dass die letztgenannten Punkte eine besondere Bedeutung für die Akzeptanz eines Frontend erlangen. Ein Server kann auch nach einem Rollout vergleichsweise einfach mit zusätzlichem Hauptspeicher zum Vorhalten von Objekten oder einer höheren Taktfrequenz nachgerüstet werden, wenn die Virtual Machine mehr Feuer unter dem Kessel braucht. Vor allem jedoch setzt eine Middleware zur Bereitstellung ihrer Dienste keine Benutzeroberfläche voraus, in der jedes kleinste Flackern, selbst unter Maximallast, als störend wahrgenommen wird. Bemerkenswert ist, dass es sich bei den Schwachstellen um allgemein gültige Problemklassen handelt, die allerdings technologiespezifisch (C, .NET, Java) angegangen werden müssen.

Bevor die Maßnahmen erläutert werden, mit denen die Performance des Web Client gesteigert wurde, soll zunächst der mehrstufige Visualisierungsprozess anhand der bestehenden Infrastruktur veranschaulicht werden.

Die Hardware besteht aus Förderelementen, die Statusänderungen (z.B.: Motor an/aus) über speicherprogrammierbare Steuerungen zum Backend senden. Das Backend lässt sich als Sammlung systemnah programmierter Serverdienste umschreiben. Ein Dienst filtert die für den Visualisierungsserver relevanten Nachrichten. Der Server, eine in C++ implementierte Legacy-Applikation, bündelt zusammengehörige Benachrichtigungen und erzeugt daraus Telegramme, die der Client als Zeicheninformationen interpretieren kann. Client und Visualisierungsserver kommunizieren mithilfe eines Plugins, das für die Dauer einer Sitzung in einem Application Server gestartet wird. Bei Letzterem handelt es sich nicht um einen EJB Application Server, sondern um eine in Java realisierte Middleware, die neben der Visualisierung zahlreiche Prozesse innerhalb eines Frameworks für logistische Prozesse koordiniert.

Der Browser wird auf den Clients nicht nur als Laufzeitumgebung für das Applet benötigt, sondern stellt auch die anlagenspezifische Dokumentation in Form von HTML-Seiten bereit. Ansonsten könnte die auf Swing basierende Applikation aus tech-

nischer Sicht ebenso gut mit Web Start verteilt werden.

Konzepte einer performanten Architektur

Die Kommunikation wird über mehrere Stationen abgewickelt, bevor der Web Client Zustandsänderungen einzelner Förderelemente anzeigen kann (Abb. 1). Ausschlaggebend für die Reaktionsgeschwindigkeit des Client scheint dabei in erster Linie seine Fähigkeit zu sein, eingehende Telegramme effizient in Bildinformationen umzuwandeln. Das ist aber noch nicht alles. In der Anlagenvisualisierung ist es üblich, Störfälle mit dem Blinken von Streckenabschnitten zu signalisieren. Der optische Reiz soll die Aufmerksamkeit in der Leitwarte sicherstellen. Um die Kommunikationswege nicht mit zyklisch ausgelösten „Blink"-Telegrammen aus der Middleware zu belasten, wird die Aufgabe, bestimmte Blinkfrequenzen bereitzustellen, an den Client abgeschoben. Wenn im schlimmsten Fall alle Baugruppen eines Detailbilds zu blinken beginnen und der Visualisierungsserver fortlaufend Statusänderungen sendet, wachsen die Synchronisationsprobleme dramatisch an. Ein Anwender wird die Belastung der CPU sogar noch weiter erhöhen, während er mit dem Mauszeiger ein Objekt innerhalb des Bildes ansteuert, um auf die Störung mit einem Klick zu reagieren.

Den Kern der Lösung liefert das MVC-Paradigma. View (Abb. 2) zeichnet das Model, also den aktuellen Zustand der Anlage. Gleichzeitig werden Benutzeraktionen entgegengenommen und an den Controller weitergeleitet. Der Controller empfängt außerdem die Telegramme vom Visualisierungsserver und erzeugt daraus Kommandos im Sinne des Command Pattern [3]. Blinksignale sind ebenfalls als Kommandos modelliert und werden nebenläufig in einem separaten Thread erzeugt. Um die Synchronisationsprobleme in den Griff zu bekommen, genügt es, alle Kommandos in einer thread-safe implementierten Queue einzureihen, die von einem Prozessor nach einem Fließbandprinzip, jeweils ohne Unterbrechung, abgearbeitet werden. Zusammenfassend lässt sich das robuste und flexible Design der Visualisierungskomponente mit dem MVC-Konzept, einer Datenstruktur aus dem Lehrbuch und dem Command Pattern vollständig beschreiben.

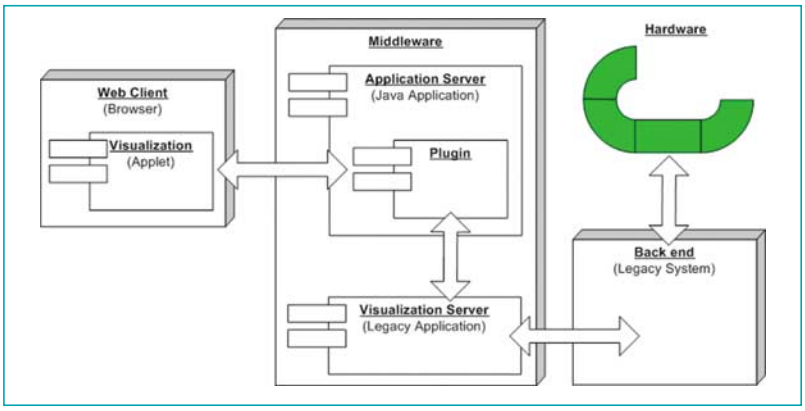

Abb. 1: Kommunikation innerhalb der Schichten des Gesamtsystems

Grob- und Feintuning

Nachdem ein solides Fundament skizziert wurde, geht es im Folgenden darum, die zentralen Bestandteile der Komponente unter dem Aspekt hoher Ausführungsgeschwindigkeit zu entwerfen. Das Model, mit dem die Bildinformationen verwaltet werden, spielt die Hauptrolle und muss auf maximale Geschwindigkeit ausgelegt werden. Die Objekte für ein Bild entstehen beim Parsen von ASCII-Dateien, die mit einem proprietären Zeichenprogramm bearbeitet werden können. Im Laufe einer Sitzung lädt der Client die vom Anwender gewählten Abbildungen von der Middleware.

Angesichts guter Literatur, in der die Ursachen für zeitkritische Vorgänge innerhalb der Java-Plattform [4] erläutert werden, möchte ich mich im Folgenden darauf beschränken, konkrete Vorschläge zu liefern, die auch zur Lösung vergleichbarer Aufgabenstellungen herangezogen werden können.

Die Visualisierung verfügt über einen Parser, der für die sequenzielle Analyse von Zeichenketten optimiert wurde, damit auf String-Manipulationen gänzlich verzichtet werden kann. Er erzeugt das Model, während die Bilddateien geparst werden, und ist außerdem für die Interpretation eingehender Telegramme verantwortlich. In Abhängigkeit von der Größe eines Bildes müssen mitunter sehr viele Objekte erzeugt werden. Um diesen zeitintensiven

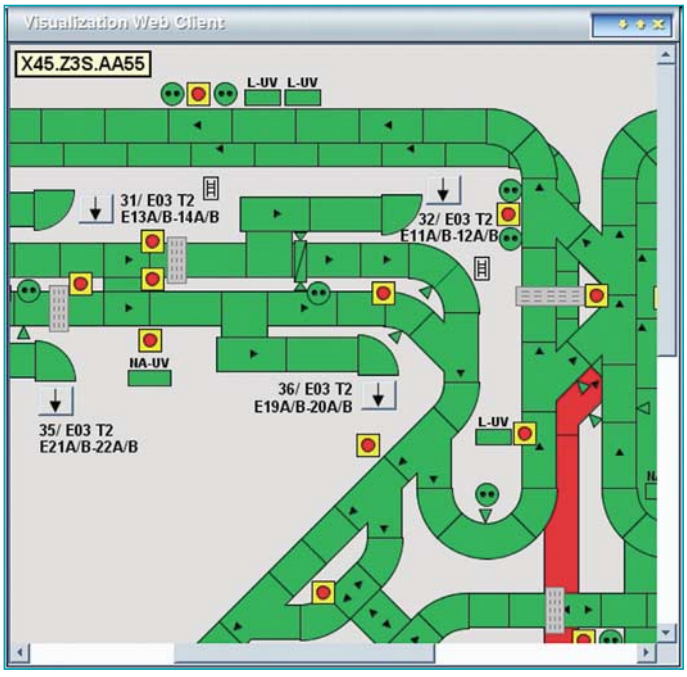

Abb. 2: Ausschnitt aus einem Detailbild in Originalgröße mit Tool-Tip oben links

Vorgang nicht bei jedem Bildwechsel wiederholt ausführen zu müssen, wird ein Model nach dem Anlegen in einem primären Zwischenspeicher vorgehalten. Natürlich reicht der Speicher des Client nicht für alle 200 Bilder aus. Deshalb verbleiben lediglich diejenigen Bilder, die vom Benutzer häufig ausgewählt werden. Elemente, die in verschiedenen Bildern dargestellt sind, z.B. Images auf Schaltflächen oder wiederkehrende Hintergrundbilder, werden in einem separaten Cache mithilfe von *WeakHashMap* verwaltet. Sobald kein Bild mehr im primären Zwischenspeicher vorhanden ist, das einen bestimmten Eintrag in der Hash-Tabelle referenziert, kann der Garbage Collector automatisch den hierfür reservierten Speicher freigeben.

Ein speicherschonender Programmierstil muss den Methodennamen verwendeter APIs Aufmerksamkeit schenken. Die meisten grafischen Objekte, die durch die Visualisierung gezeichnet werden, setzen sich aus Shape-Objekten der Java 2-D API zusammen. Vor dem Rendering werden alle Transformationen, die für die Darstellung an der richtigen Position und in der gewünschten Größe notwendig sind, auf diese Shape-Objekte angewendet. Das API stellt hierfür Methoden bereit, die mit ihren Namen *createTransformedShape* bereits auf Objekterzeugung hindeuten. Wenn alle Objekte eines großen Bildes mit einer Frequenz von einer Sekunde blinken, dann generieren diese Methoden während des zyklischen Rendering derartig viele Objektinstanzen, dass der Garbage Collector seiner Aufgabe innerhalb kürzester Zeit nicht mehr nachkommen kann, weil die CPU hundertprozentig ausgelastet ist. Der Speicher-GAU kann verhindert werden, indem alle transformierten Objekte so lange wiederverwendet werden, bis eine Animation die erneute Anwendung der Transformation erfordert.

In diesem Zusammenhang drängt sich die Frage auf, ob es effizienter ist, nach jeder Änderung im Modell das ganze Bild oder nur die beschädigten Bereiche zu zeichnen. Hier liefert eine Worst-Case-Analyse, also das Blinken aller Bildelemente, die Antwort. Wenn ein blinkendes Element andere Objekte überdeckt, dann muss es eine Möglichkeit geben, alle Objekte zu bestimmen, die in dem Bereich liegen, der erneut gezeichnet werden muss. Unabhängig davon, wie effizient diese Methode implementiert werden kann, sobald sämtliche Elemente blinken, ist es in jedem Fall schneller, das Modell komplett zu zeichnen, ohne vorher alle beschädigten Bereiche zu ermitteln.

Damit kommt *JComponent* als performante Basisklasse für Malobjekte nicht in Betracht. Bei einem Test mit gefüllten Kreisbögen wurde in einer Spezialisierung die *paint*-Methode überschrieben. Mehrere dieser Objekte werden überlappend auf einem Panel gezeichnet und eines anschließend mit *setVisible(false)* unsichtbar gesetzt. Die Grafik wird durch Swing fehlerfrei gezeichnet, allerdings läuft der Rendering-Prozess im Schneckentempo ab und kann auf dem Bildschirm mitverfolgt werden.

Ein weiterer Punkt während der Implementierung einer Rendering Engine ist die Klärung der Frage, warum es beim Double Buffering eines umfangreichen Bildes zu Flackereffekten kommen kann. Die View ist von der Klasse *JComponent* abgeleitet und implementiert eine Methode, die das Modell offscreen rendert (Listing 1). Die *paint*-Methode wird überschrieben, damit Swing das Bild im Offscreen-Puffer auf die Oberfläche der Komponente zeichnen kann, wenn die View aktualisiert werden muss. Der nebenläufige Swing Thread kann *paint* jedoch aufrufen, bevor die arbeitsintensive Methode *renderOffscreen* vollständig abgearbeitet ist. Es lässt sich vermeiden, dass ein unvollständiger Puffer-Inhalt, der letztendlich zum Flackern führt, gezeichnet wird, wenn der Paint-Mechanismus von Swing mit dem Zeichnen des Modells synchronisiert wird.

Neben dem Einsatz optimierter Zeichenroutinen muss zudem die Anzahl der Zeichenvorgänge minimiert werden, indem beispielsweise zeitnahe Statusänderungen unterschiedlicher Objekte in einem Aufruf zusammengefasst werden. Eine Folge von Statusänderungen kann sich in dem Legacy-System über mehrere Telegramme erstrecken. Der Client kann nicht selbstständig entscheiden, wann eine Update-Sequenz zu Ende ist. In diesem Fall ist er auf eine explizite Kennung vom Visualisierungsserver angewiesen, der damit das Zeichnen eines Models auslöst, das vollständig aktualisiert wurde.

Die Darstellungsqualität beeinflusst die Ausführungszeit der Zeichenroutinen. Anti-Aliasing verhindert Treppeneffekte, wie sie beim Zeichnen schräger Linien und Kreise auftreten, und lässt Kanten an den Rändern weich erscheinen. Der Prozessor und die Grafikkarte werden dafür jedoch überdurchschnittlich beansprucht. Sobald der Weichzeichner eingeschaltet wird, steigen die CPU-Belastung beim Zeichnen eines Bildes um den Faktor drei und die Dauer der Zeichenoperation sogar auf das Fünffache an. Die Option kann für jeden Client individuell ein- oder ausgeschaltet werden.

Auch ein zunächst harmlos wirkender Tool-Tip-Mechanismus lässt sich optimieren. Der Tool-Tip-Manager aus dem Swing Package führte zu einer nicht lokalisierbaren Race Condition, wenn der Mauszeiger schnell innerhalb eines Bildes bewegt wurde. Die Applikation blieb sang- und klanglos stehen. Eine alternative Lösung, basierend auf einem Label, das in einem Fenster dargestellt wird, beseitigte dieses Problem. Erst beim Integrationstest auf den vergleichsweise langsamen Produktivrechnern machte sich ein Flacker-Effekt in Kombination mit dem Tool-Tip-Fenster bemerkbar. Es hat den Anschein, als ob das AWT nicht schnell genug für die Aktualisierung der Oberfläche zum Zuge kommt und der Tooltip kurzzeitig eine Schmierspur hinterlässt. Um diesen Schmutzeffekt zu verhindern, kann der Tooltip in der linken oberen Ecke der View angepinnt werden. Da-

Listing 1

Synchronisation zwischen Swing und der Rendering Engine

```
// Paint the model offscreen.
public void renderOffscreen( Model model ){
 setOffscreenReady( false );
 // 1. Clear _offscreenBuffer
 // 2. Render model to _offscreenBuffer
 setOffscreenReady( true );
 repaint();
}

// Overrides JComponent's paint() method.
public void paint( Graphics g ){
 if ( isOffscreenReady() )
  g.drawImage( _offscreenBuffer, 0, 0, this );
}
```

mit verschwindet selbst bei starken Zeigerbewegungen und blinkenden Strecken jegliches Flackern aus den Bildern.

Extreme Testing

Das Test-Framework JUnit eignet sich wegen seiner einfachen Handhabung ausgezeichnet für automatisiertes Testen. Ohne individuelle Anpassungen ist das Framework jedoch nicht für das Testen eigenhändig gerenderter Bilder geeignet. Im Folgenden wird erörtert, inwieweit JUnit eingesetzt wird und warum es für spezielle Aspekte nötig scheint, einen individuellen Weg einzuschlagen.

Der Parser wird mit JUnit getestet. Eine Testklasse verifiziert den Analysealgorithmus und die Fehlererkennung. Wann immer eine weitere Parse-Methode oder zusätzliche Tokens hinzukommen, kann das gesamt Funktionsspektrum auf Knopfdruck überprüft werden.

Sämtliche Zeichenroutinen der Altanwendung mussten auf das Java 2-D Graphics API portiert werden, um sowohl die optimierten Routinen als auch anspruchsvolle Features, wie z.B. die Skalierung, verwenden zu können. Wie lässt sich nach Möglichkeit automatisiert sicherstellen, ob die neu entstehenden Klassen die gleichen Bilder zeichnen wie die alten? Im Grunde ist das einfach: Es wird eine Assert-Methode implementiert, die zwei Bilder miteinander vergleicht. Nur wenn beide Bilder in jedem Bit übereinstimmen, gilt der Test als bestanden. Die Originalbilder werden mithilfe einer Erweiterung der Altanwendung als gepackte Bitmaps gespeichert und stehen damit als Zielvorgaben zur Verfügung. Für das Erzeugen einer Bitmap aus dem Inhalt der gezeichneten Oberfläche kommt das Java Advanced Imaging API zum Einsatz. Damit steht ein augenschonender Soll-Ist-Abgleich bei fortschreitender Entwicklung zur Verfügung.

Ab dem Zeitpunkt, als verschiedene Threads damit begannen, dynamische Zustandsänderungen an visualisierbaren Objekten durchzuführen, stieß das Testkonzept mit statischen Testfällen an seine Grenzen. Von nun an kam eine Klasse zum Einsatz, die einen Testroboter simuliert. Ihre Aufgabe besteht darin, einzelne Testszenen abzuspielen, die sowohl aus Benutzeraktionen als auch aus Telegrammen des Visualisierungsservers bestehen. Damit die Testszenarien realitätsnah gestaltet werden können, wird die Kommunikation zwischen dem Client und dem Visualisierungsserver mit dem Plugin des Application Server aufgezeichnet.

Ein Szenario spielt alle realisierten Features durch, z.B. Blinken, Fortschritt einer Füllstandsanzeige, Animation und Drehung von Objekten. Eine Testszene kann in beliebig vielen Fenstern gleichzeitig dargestellt werden. Ein Fenster zeigt das Detailbild in Originalgröße und stellt, wenn der sichtbare Bereich nicht ausreicht, Schieberegler zur Verfügung. Die anderen Fenster zeigen das vollständige Bild, das mithilfe der Skalierung in den Fensterrahmen eingepasst wird (Abb. 3). Ein erfolgreicher Verlauf dieser online gerenderten Animation kann leider nicht mehr mit vertretbarem Aufwand automatisiert, sondern nur durch „manuelle“ Kontrolle überprüft werden.

Während eines Stresstests werden 400 Bildwechsel mit einer Frequenz von unter 500 ms simuliert und in hoher Darstellungsqualität auf zwei Fenstern angezeigt. Gefordert war ein Bildaufbau von unter einer Sekunde für lediglich ein Fenster in Originalgröße und ohne Anti-Aliasing. Der Test stellt hohe Anforderungen an die CPU, die Grafikkarte und den Garbage Collector, der in dem mehrstufigen Cache-Mechanismus immer wieder Platz schaffen muss. Erst als der Lasttest stabil verlief, konnte davon ausgegangen werden, dass selbst ein „schwachbrüstiger“ Web Client die geforderte Reaktionszeit einhalten würde. Mehrere Integrationstests auf unterschiedlichen Rechnerkonfigurationen bestätigten später diese Annahme.

Bis zum Bestehen des Stresstests waren regelmäßige Untersuchungen mit einem Analysewerkzeug unablässig. Die Er-

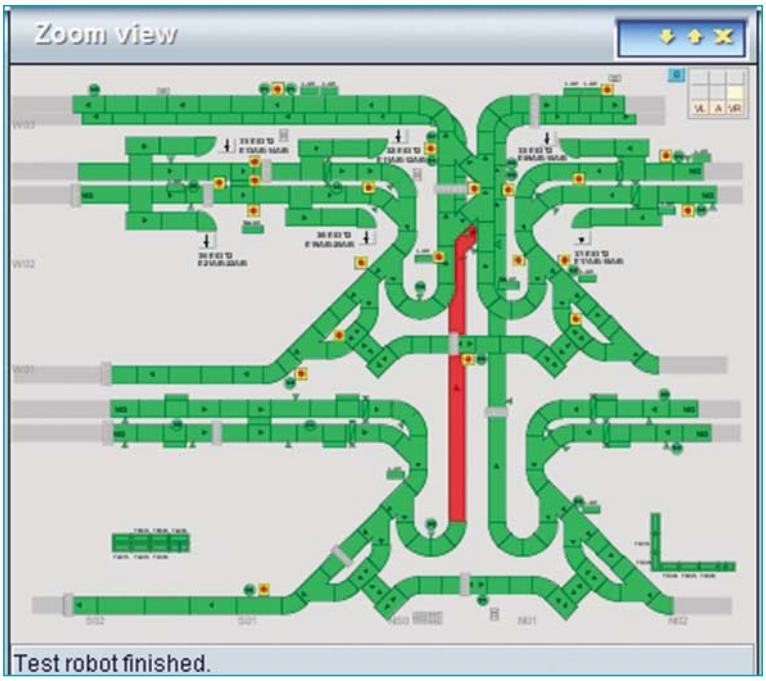

Abb. 3: Der Testroboter zeigt eine stark verkleinerte Ansicht des Detailbilds

kenntnisse daraus hatten immer wieder Einfluss auf Details des Programmentwurfs, ohne jedoch die objektorientierte Architektur aufzuweichen.

Kurioses aus dem Testlabor

Zu Beginn des Redesigns wurde noch das JDK 1.3.1 verwendet. Darin führt ein Bug im Java 2-D Graphics API dazu, dass Flächen mit schrägen Kanten nicht immer korrekt gefüllt werden. Der Umstieg auf das JDK 1.4.1 trieb zunächst den Teufel mit dem Beelzebub aus: Die Flächen werden zwar vollständig gefüllt, dafür kann es zu Kompatibilitätsproblemen mit gängigen Grafikkarten auf Windows-PCs kommen. Sollten ungewöhnliche Effekte in der Oberflächendarstellung einer Java-Anwendung auffallen, dann kann versuchsweise die Hardwarebeschleunigung der Grafikkarte deaktiviert oder mit den Laufzeitparametern für Direct Draw (*Dsun.java2d.noddraw, ddoffscreen, ddscale*) experimentiert werden. Wahrscheinlich sind die Darstellungsfehler anschließend verschwunden. Vielleicht aber nicht alle. Wiederholtes Zeichnen eines Textes mit dem Font-Attribut „fett“ an unveränderter Position durch eine AWT-Funktion kann dazu führen, dass der Text perforiert dargestellt wird. Wenn Text gezeichnet werden muss, dann sollten unabhängig von der JDK-Version, die Funktionen des Java 2-D Graphics API verwendet werden, die immer eine korrekte Darstellung liefern.

Manchmal kann es bei der Analyse optischer Phänomene hilfreich sein, die Kommentare im Code der JDK Packages zu studieren. Einmal erschien ein Tool-Tip an einer Position, an der kein Objekt gezeichnet war. Die Recherche in der zur Positionsbestimmung verwendeten Methode förderte einen „Merkzettel“ zu Tage, der besagte, dass das Verhalten für einen bestimmten Wertebereich noch überprüft werden müsse … Ungeachtet dieser Widrigkeiten sind die Möglichkeiten, die Java in den Disziplinen Grafik und Bildbearbeitung bereitstellt, weit gereift und selbstverständlich frei verfügbar.

Die wichtigsten Tuning-Maßnahmen im Überblick

- Verzicht auf String-Operationen
- Implementierung von Cache-Strategien
- Vermeidung von Objekterzeugung
- Zusammenfassen zeitnaher Ereignisse
- Einsatz von Datenstrukturen mit kurzen synchronisierten Abschnitten
- Verzicht auf Swing-Klassen, wenn deren komplexe Funktionalität nicht benötigt wird
- Regelmäßige Kontrolle mit automatisierten Stress-Tests
- Optimierung zeitkritischer Passagen mit einem Profiling-Werkzeug

Fazit

Die Leser des *Java Magazins* wissen, dass die Java-Plattform schnell ist. Moderne Virtual Machines mit ausgefeilten Mechanismen, neue, aber auch altehrwürdige APIs, die mit steigenden Versionsnummern an Geschwindigkeit zulegen, bieten kaum mehr Anlass zur Sorge. Im Umkehrschluss bedeutet es aber auch, dass die Java-Welt nicht schon von Anfang an eine heile war. Eine Ursache liegt meiner Meinung nach in der Länge der Lernkurve, die widerspiegelt, wie viel Zeit verstreicht, bis eine komplexe Technologie gemeistert werden kann [5]. Dieses Argument trifft gleichermaßen auf die Anwender einer kontinuierlich weiterentwickelten Plattform zu [6]. Es wäre daher kurzsichtig anzunehmen, es reiche aus, die Syntax einer Sprache, ihre wichtigsten Klassen und ein paar OO-Grundlagen zu erlernen, um auf Anhieb beeindruckende Lösungen zaubern zu können. Bei der Portierung eines Altsystems in eine neue Welt, sei es nun J2EE oder .NET, sollte dieser Umstand ausreichend berücksichtigt werden, um nicht bei Problemen ausschließlich den Reifegrad der verwendeten Technologie verantwortlich zu sehen.

Seit etwa einem Jahr wird die neue Visualisierung auf Web Clients in unterschiedlichen industriellen Anlagen eingesetzt. Wegen der guten Erfahrungen wird heute darüber nachgedacht, auch den einstmals im Visual Studio geschmiedeten Visualisierungsserver durch eine plattformunabhängige Java-Komponente abzulösen. ■

Edgar Dehm ist Projektleiter im Geschäftsbereich Business Applications der develop group in Erlangen.

Links & Literatur

[1] Vincent J. Hardy: Java 2D API Graphics, Prentice Hall, 2000

[2] Martin Fowler: Refactoring, Addison-Wesley, 1999

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Addison-Wesley, 1999

[4] Jack Shirazi: Java Performance Tuning, O'Reilly & Associates, 2002

[5] Pete Mc Breen: Software Craftsmannship, Addison-Wesley, 2002

[6] Alistair Cockburn: Surviving Object-Oriented Projects, Addison-Wesley, 1998