

Durch dick und dünn

# Probleme bei Thin-Clients

Bruno Schäffer

Seit einigen Jahren gibt es eine starke Tendenz von Fat-Clients zu HTML-basierten Thin-Clients. Allerdings bietet HTML nicht nur Vorteile. Die Kombination der positiven Eigenschaften von Fat- und HTML-basierten Thin-Clients käme Benutzern, Entwicklern und Betreibern zugute. Dieser Artikel diskutiert aktuelle Ansätze und zeigt auf, welche Probleme und mögliche Lösungen es bei Java-basierten Thin-Clients gibt.

Die Landschaft der Client-Server-Architekturen hat sich in den letzten 15 Jahren mehrmals drastisch verändert. Ausgehend von Terminal-basierten Lösungen über Fat-Clients sind nun HTML-basierte Anwendungen die bevorzugte Wahl. Neben neuen Technologien wie z. B. dem Internet waren vor allem Kostenüberlegungen treibende Kraft hinter diesem Wandel. Zu einem guten Teil haben sich diese Kostenüberlegungen allerdings auf betriebliche Aspekte konzentriert, während die Entwicklungsseite häufig vernachlässigt wurde.

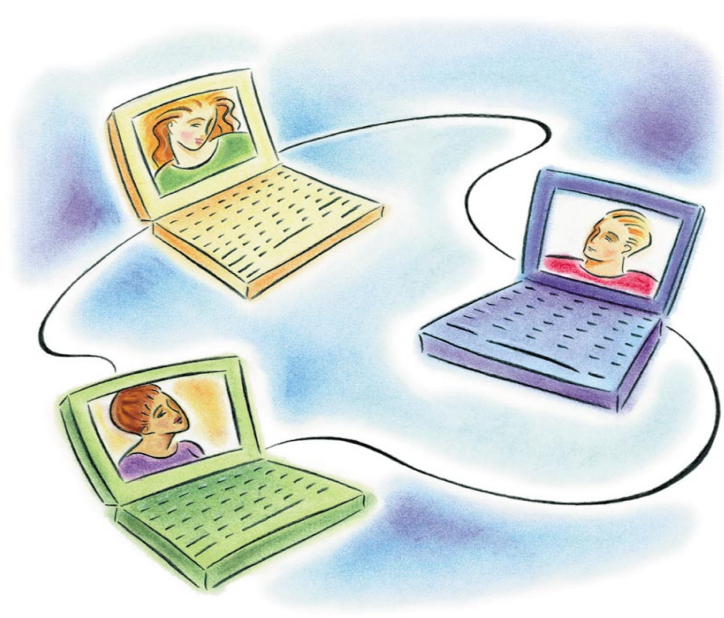
Die heutigen Client-Server-Anwendungen sind charakterisiert durch eine starke Server-Zentrierung mit Thin-Clients. HTML-basierte Anwendungen als klassische Vertreter dieser neuen Generation betonen die leichte Ausbreitung zu Lasten der Entwickler- und Benutzerfreundlichkeit.

Dieser Artikel vergleicht aktuelle Ansätze und zeigt auf, welche Probleme und mögliche Lösungen es bei (Java-basierten) Thin-Clients gibt. Neben betrieblichen Vorteilen sind diese Thin-Clients auch einfach zu entwickeln und weisen eine zeitgemäße Benutzerschnittstelle auf. Die geschilderten Probleme, Lösungen und Schlussfolgerungen basieren auf den Erfahrungen bei der Entwicklung von komplexen HTML-Applikationen inkl. Infrastruktur und der Entwicklung einer Infrastruktur für Java-basierte Thin-Clients.

## Fat-Clients

Client-Server-Architekturen beschäftigen sich mit der Frage, wie die Aufteilung einer Anwendung auf zwei oder mehrere logische Schichten (Engl.: Tiers) erfolgen kann, welche Eigenschaften eine Schicht aufweist und wie die Kommunikation zwischen den Schichten erfolgt. Grundsätzlich lassen sich folgende grobe applikatorischen Komponenten identifizieren: Präsentation, Anwendungslogik (Business-Objekte und -Prozesse) und Datenhaltung.

Mit dem Aufkommen leistungsfähigerer Hard- und Software auf der Benutzerseite entstand das Bedürfnis, diese Leistung für den Anwender auch einzusetzen. Die ersten Client-Server-Anwendungen wiesen eine 2-Schichten-Architektur auf: die (Fat-)Client-Schicht enthielt Präsentations- und Anwendungslogik während die Server-Schicht Datenbankzugriffe/Transaktionen zur Verfügung stellte.



Der Nachteil dieses Modells ist vor allem die schlechte Skalierbarkeit aufgrund der direkten Anbindung an die Datenbank bzw. Transaktionen. Außerdem verringern die beschränkten Wiederverwendungsmöglichkeiten von Anwendungslogik die Entwicklungseffizienz und Wartbarkeit.

Die serviceorientierte Architektur versucht die Wiederverwendung von Anwendungslogik durch Einführung von Services zu erhöhen. Dazu wird eine weitere Schicht – der Middle-Tier – eingeführt. Damit können wesentliche Teile der Anwendungslogik zentral entwickelt und unterschiedlichen Anwendungen zur Verfügung gestellt werden. Beispiele für unterstützende Infrastrukturen sind Corba oder J2EE. 3-Schichten-Architekturen erhöhen die Wiederverwendbarkeit und Skalierbarkeit ohne die nach wie vor umfassenden Möglichkeiten für die Benutzerschnittstelle einzuschränken.

Klassische 3-Schichten-Architekturen haben jedoch zwei gravierende Nachteile. Der anwendungsspezifische Fat-Client bedingt bei jeder Version eine kostenträchtige Neuinstallation. Dazu kommt der hohe Konfigurationsaufwand für unterschiedliche Fat-Clients auf dem Anwenderrechner. Unterschätzt wird auch der notwendige Entwicklungsaufwand. Dieser resultiert vor allem aus der Aufteilung der Anwendungslogik auf Client- und Middle-Tier. Hier muss sich der Entwickler mit Problemen von verteilten Objekten beschäftigen, wie z. B. Kommunikation, Caching, Synchronisation zur Übersetzungs- und Laufzeit etc.

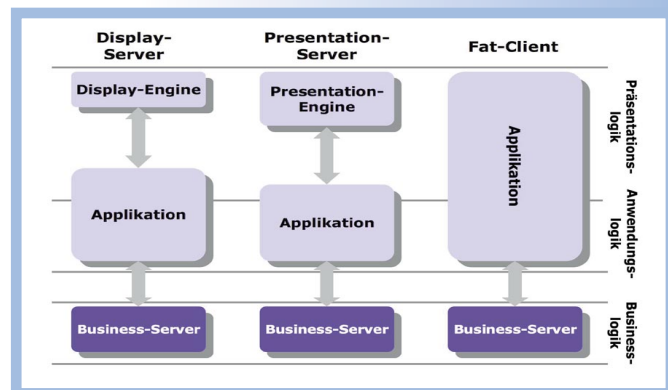


Abb. 1: Display-Server vs. Presentation-Server vs. Fat-Client

Die meisten Lösungen wurden primär aus der Softwareverteilungsperspektive entwickelt. Der *Distribution-Server*-Ansatz reduziert den Installationsaufwand durch eine transparente Verteilung (z. B. beim Starten der Anwendung). Applets sind ein klassischer Vertreter dieses Ansatzes. *Display-Server* und *Presentation-Server* verwenden einen anwendungsunabhängigen Thin-Client, womit Software-Updates normalerweise nur auf dem Server erfolgen müssen. Beide Ansätze unterscheiden sich im Wesentlichen durch die Funktionalität des Client bzw. Granularität der Kommunikation. X-11 ist ein Beispiel für den Display-Server-Ansatz und HTML ist der bekannteste Vertreter des Präsentation-Server-Ansatzes. Abbildung 1 zeigt die drei Ansätze im Vergleich.

## HTML-basierte Thin-Clients

### Motivation

Thin-Clients verwenden eine anwendungsunabhängige Client-Komponente, die für die Darstellung und Ereignisbehandlung der Benutzerschnittstelle zuständig ist. Die Grundlage für HTML-basierte Thin-Clients bilden Internet-Technologien wie HTML und HTTP. Ein Web-Browser kann heutzutage als Bestandteil praktisch jeder Client-Konfiguration vorausgesetzt werden. Die Erweiterung der Seitenbeschreibungssprache HTML um grundlegende Interaktionselemente bietet die Möglichkeit, Client-Server-Anwendungen für einen generischen Thin-Client zu realisieren. Internet-Technologien wie z. B. HTTP(S) erlauben den Zugriff auf diese Anwendungen ohne weitere Software-Installation beim Anwender und ohne Vernachlässigung von Sicherheitsaspekten (z. B. Firewall, Verschlüsselung etc).

Dies hat eine massive Migration von Fat-Clients zu HTML-basierten Thin-Clients herbeigeführt, unabhängig davon, ob eine Anwendung im Internet oder nur im Intranet eingesetzt wird.

### Realisierung

Auf der Server-Seite kommen im Java-Umfeld je nach Gewichtung des statischen Anteils der Benutzerschnittstelle Servlets oder JSP zum Einsatz. Die Entwicklung von Benutzerschnittstellen wird damit aber nur unzureichend unterstützt. Als Ergänzung werden Bibliotheken wie z. B. Struts oder Java Server Faces angeboten, um auf der Entwicklungsseite wieder in die Nähe des Abstraktionsniveaus von Fat-Client-Bibliotheken zu kommen. Auf der Client-Seite wird als Ergänzung zu HTML oftmals JavaScript eingesetzt.

### Vorteile

Die Vorteile von HTML-basierten Thin-Clients liegen vor allem in den Bereichen Ausbreitung und Konfiguration. Der Web-Browser als generischer Client kann vorausgesetzt werden. Damit muss eine Anwendung nur auf dem Server installiert werden. Zusätzlich ist die Konfigurationsschnittstelle relativ schlank. Viele heutige Umgebungen bieten eine Unterstützung für diese Art von Anwendungen. Einfache formularorientierte Anwendungen lassen sich effizient mit HTML-basierten Thin-Clients realisieren.

### Nachteile

Die offensichtlichsten Nachteile eines HTML-basierten Thin-Clients liegen in der limitierten Benutzerschnittstelle. HTML war und ist ausgerichtet auf statischen Hypertext.

Alles, was über eine einfache formularbasierte Anwendung hinausgeht, lässt sich oft nur schwer oder gar nicht realisieren. Das größte Hindernis in der Entwicklung ist die Seitenorientierung. Interaktive Benutzerschnittstellen erfordern eine Interaktion mit der Präsentationslogik auf dem Server auf der Basis einzelner Interaktionselemente. Die seitenweise Interaktion ist aber nicht nur aus Benutzersicht mühsam, sie benötigt auch unverhältnismäßig viel Bandbreite.

Durch den Einsatz von Erweiterungen wie CSS oder JavaScript können manche Einschränkungen umgangen werden. Dieser Technologiemix geht jedoch klar zu Lasten des Entwicklers mit steigenden Entwicklungskosten als Folge davon. Die Seitenorientierung von HTML hat auch erhebliche negative Auswirkungen auf das Programmiermodell. Bibliotheken und Frameworks wie z. B. Struts können das Abstraktionsniveau für den Entwickler anheben, kommen aber letztendlich nicht um die Limitierungen des Web-Browsers herum.

HTTP als Protokoll ist unidirektional und mit einer Ursache für bestimmte Limitierungen in HTML-basierten Benutzerschnittstellen. Applikatorisch gesteuerte Updates in der Benutzerschnittstelle sind schwer realisierbar (Paradebeispiel: Stock Ticker) und seitenweise Updates als Alternative skalieren schlecht.

Der Einsatz von JavaScript ist kritisch im Bezug auf die Architektur. Es besteht die Gefahr, anwendungsspezifische Funktionalität (z. B. semantische Plausibilitätsprüfungen) auf die Client-Seite zu bringen, womit der Thin-Client langsam wieder zum Fat-Client mutiert. Auch die Integration von Fat-Client-Applets ist keine Lösung. Die Vermischung von Server-Orientierung (HTML) mit Client-Orientierung (Applet) ist aufwändig und problematisch. Außerdem tendieren Applets architekturell schnell in Richtung Fat-Client.

Die Benutzerschnittstellen von HTML-Anwendungen sind nicht nur limitiert, sondern auch extrem heterogen. Fehlende Interaktionselemente werden je nach Anwendung unterschiedlich „nachgebaut“. Im Vergleich zu Fat-Clients fehlt vielfach die anwendungsübergreifende Konsistenz.

Die Verwendung von Web-Browsern als Client ist ebenfalls mit einigen Nachteilen verbunden. Aufgrund der (Versions-)Vielfalt von Web-Browsern ist das Entwickeln und Testen von HTML-Anwendungen mit erheblichem Aufwand verbunden, insbesondere bei komplexeren Benutzerschnittstellen. Vor einigen Jahren gab es die Vorstellung von einem „Business-Desktop“, eine nahtlose Integration von unterschiedlichen Anwendungen über eine Desktop-Metapher. Mit HTML lassen sich Anwendungen kaum und über die Browser-Grenze hinweg praktisch gar nicht integrieren. Portale sind lediglich eine schwache Imitation des Desktops.

## Java-basierter Thin-Client

Anspruchsvolle Anwendungen benötigen interaktive Benutzerschnittstellen. Trotzdem ist man bestrebt, die Nachteile des Fat-Client zu vermeiden. Java bietet gute Voraussetzungen für die Entwicklung einer Thin-Client-Architektur, die die Vorteile von HTML mit denen eines Fat-Client verbinden kann.

Nachfolgend wird auf die Anforderungen und Probleme eingegangen, die für eine Java-basierte Thin-Client-Architektur zu lösen sind. Dabei liegt der Schwerpunkt auf der Realisierung der Benutzerschnittstelle. Wie beim HTML-basierten Thin-Client ist dabei die Präsentationslogik auf dem Middle-Tier. Die Präsentationslogik wird

hierbei durch den Code definiert, der das anwendungsspezifische Erscheinungsbild und Verhalten der Benutzerschnittstelle definiert.

**Entwicklungsmodell**

Entwicklungskosten lassen sich unter anderem mit einem möglichst homogenen und einfachen Entwicklungsmodell senken. Möglichkeiten hierfür sind:

- ▼ *Reduzierung des Technologiemix:* Als verwendete Technologie ist Java ausreichend.
- ▼ *Transparenz der Verteilung:* Eine Thin-Client-Anwendung ist wie eine Fat-Client-Anwendung zu entwickeln. Daher sind die Verteilungs- und Kommunikationsaspekte zu abstrahieren.
- ▼ *Anlehnung an bestehende Bibliotheken:* Um den Umstieg von Fat-Client zu Thin-Client möglichst einfach zu gestalten, orientiert sich die Infrastruktur an vorhandenen Klassenbibliotheken (z. B. AWT/Swing).
- ▼ *Nebenläufigkeit:* Die Präsentationslogik läuft sessionspezifisch auf dem Server. Daher ist der Entwickler soweit wie möglich von Aspekten der Nebenläufigkeit abzusichern.

**GUI-Bibliothek**

Gängige Fat-Client GUI-Bibliotheken wie z. B. Swing sind auf Client-seitige Entwicklung und Betrieb ausgerichtet. Da sich bei einer Thin-Client-Anwendung die Präsentationslogik auf dem Server befindet, müssen die Interaktionselemente ebenfalls auf dem Server (u. U. ohne Bildschirm) lauffähig sein. Beim Thin-Client wird das bekannte MVC-Modell (s. Abb. 2) beibehalten. Sowohl auf der Server- wie auch der Client-Seite gibt es je eine MVC-Triade, die transparent synchronisiert sind.

Auf der Server-Seite bekommen Anforderungen bezüglich Skalierbarkeit und Effizienz ein wesentlich stärkeres Gewicht. Es ist daher notwendig, die serverseitigen Interaktionselemente möglichst leichtgewichtig zu gestalten. Bestimmte Server-Umgebungen, wie z. B. EJB-Container, setzen zudem voraus, dass ihre Objekte passiviert und wieder aktiviert werden können. Um diesen Aufwand zu reduzieren, ist auf einen minimalen Zustand der Interaktionsobjekte zu achten. Unter diesen Voraussetzungen ist eine Erweiterung einer bestehenden GUI-Bibliothek wie z. B. Swing über Vererbung nicht möglich.

Client- und Server-Seite sind sowohl statisch als auch dynamisch zu entkoppeln. Die statische Entkopplung fördert die Austauschbarkeit der Client-seitigen GUI-Bibliothek. Die dynamische Entkopplung vermindert die Blockierung der Benutzerschnittstelle, die sich durch synchrone Kommunikation mit limitierter Bandbreite bzw. hoher Latenzzeit ergibt. Das Aufdatieren der Benutzerschnittstelle kann im Normalfall asynchron erfolgen. Benutzeraktionen, die den Zustand auf der Server-Seite verändern, müssen dagegen synchron ausgeführt werden.

**Ausbreitung**

Der Client ist anwendungsunabhängig, er muss aber trotzdem (zumindest einmal) ausgebreitet werden. Für Java-Anwendungen gibt es grundsätzlich drei Möglichkeiten: Installation als Anwendung, Starten als Applet oder über Java WebStart.

Ein optimiertes Protokoll zwischen Client und Server bedeutet eine starke Kopplung bezüglich der Versionierung. Die Ausbreitung aufgrund einer neuen Ver-

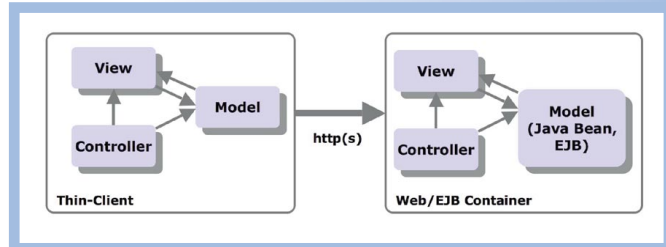


Abb. 2 Thin-Client Model – View – Controller

sion der Anwendung kann durch die Thin-Client-Infrastruktur abgefangen werden. Die neue Version der Anwendung muss nur auf dem Server installiert werden. Versionsänderungen in der Thin-Client-Infrastruktur bedingen teilweise einen Update des Client. Hier empfiehlt es sich, Softwareverteilungsinfrastrukturen wie Applet oder Java WebStart zu benutzen. Die Entkopplung zwischen Anwendungs- und Infrastrukturversionen kann jedoch den Ausbreitungsdruck auf dem Client entscheidend reduzieren.

**Betrieb und Sicherheit**

Die Serverseite einer Thin-Client-Infrastruktur muss kompatibel mit heutigen Applikationsservern sein. Die spezifische Eigenentwicklung eines Applikationsservers kommt aus Kosten- und Marktüberlegungen wohl kaum in Frage.

Daraus folgt, dass sich der Serverteil einer Thin-Client-Anwendung – obwohl entwickelt wie eine Fat-Client-Anwendung – einfach in den Servlet- oder EJB-Container installieren lassen muss. Aus Entwicklungssicht ist die Frage interessant, inwieweit eine Abstraktion des Containers vorgenommen wird. Dies ermöglicht die Festlegung des Containertyps (Servlet oder EJB) erst spät im Entwicklungsprozess. Andererseits nimmt es dem Entwickler die Möglichkeit, Container-spezifische Eigenschaften auszunutzen.

Sicherheit ist immer noch ein unterschätzter Faktor im Thin-Client-Umfeld. Probleme mit der Kommunikationssicherheit (siehe unten) sind dabei noch am leichtesten zu lösen. Schwierig wird es, sobald von einer Thin-Client-Anwendung auf sicherheitsrelevante Ressourcen auf dem Client zugegriffen werden kann. Beispiele hierfür sind Zugriff auf Dateien, Starten von Anwendungen etc., Funktionalität, die aber unter Umständen entscheidend für die Akzeptanz einer Anwendung sind. Der Client ist grundsätzlich in der Java Security Sandbox zu starten. Diese Umgebung muss jedoch anwendungsspezifisch konfiguriert werden können.

**Kommunikation**

Wird die Präsentationsdarstellung und -behandlung von der Präsentationslogik getrennt, wie das beim Thin-Client der Fall ist, so wird für anspruchsvolle Benutzerschnittstellen eine bidirektionaler Kommunikation vorausgesetzt.

Ausgehend von der Forderung nach Kompatibilität mit heutigen Applikationsservern tritt ein weiteres Hindernis auf. Deren Kommunikationsmodell beruht praktisch ausnahmslos auf dem Request-Response-Modell und ist folglich unidirektional (vom Client zum Server). Eine Thin-Client-Lösung ist daher gezwungen, auf diesem unidirektionalen Kanal einen bidirektionalen Kanal zu simulieren. Polling kann hier eine Möglichkeit sein, falls die dafür benötigte Bandbreite ausreicht. Das Polling-Intervall muss sowohl von außen als auch programmatisch konfigurierbar sein.

Für Thin-Clients ist die benötigte Bandbreite zu minimieren. Das gilt sowohl für die Aufstartzeit als auch für die Bedienung insgesamt. Die Zielvorstellung ist, eine An-

wendung auch über eine Modemverbindung (>= 33 Kbps) ohne merkliche Einschränkungen benutzen zu können. Folgende Lösungsmöglichkeiten können hier zur Anwendung kommen:

- ▼ **Lazy-Loading:** Alle Daten für die Benutzerschnittstelle und die Anwendung werden nur dann übertragen, wenn sie für die Darstellung benötigt werden. Einmal übertragene Daten werden dann allerdings auf dem Client zwischengespeichert. Ein Paradebeispiel für Lazy-Loading sind große Tabellen.
- ▼ **Lokale Behandlung von Ereignissen:** Anfragen werden nur dann zum Server geschickt, wenn zur Behandlung eines Ereignisses Code auf dem Server ausgeführt werden muss. Andere Ereignisse wie z. B. Formatierung oder syntaktische Konsistenzprüfungen können dagegen lokal auf dem Client behandelt werden.

Aus Sicht eines Entwicklers muss das verwendete Transportprotokoll transparent sein und das verwendete Protokoll erst beim Verbindungsaufbau vom Client zum Server bestimmt werden. Die möglichen Transportprotokolle hängen natürlich vom Einsatzgebiet ab. Im Intranet wird die Wahl vor allem durch den Applikationsserver bestimmt. Sicherheitsaspekte können diese Wahl noch einschränken. Im Internet und Extranet steht als Protokoll praktisch ausschließlich HTTP(S) zur Verfügung, da nur dieses von Firewalls durchgelassen wird.

### **Multichannel**

In einigen Fällen lassen sich Anwendungen nicht ausschließlich als „für HTML geeignet“ oder „für einen Java-basierten Thin-Client geeignet“ klassifizieren. Anwendungen besitzen oft größere Anteile, die sich sehr wohl mit HTML realisieren lassen, und daneben Teilbereiche, die kaum oder gar nicht für HTML tauglich sind.

Aus architektureller Sicht ergänzen sich die beiden Ansätze:

- ▼ Bei beiden Lösungen wird die Präsentationslogik im Applikationsserver ausgeführt – eine ideale Voraussetzung für die Integration. Sowohl Entwicklungs- als auch Laufzeitmodell gehen von einer server-basierten Anwendung aus.
- ▼ Der Java-basierte Thin-Client kann als Applet in die HTML-Anwendung eingebettet werden.

- ▼ Die Synchronisation zwischen dem Applet- und HTML-Teil kann entweder im Browser über JavaScript oder auf dem Server über eine gemeinsame Session erfolgen. Die erste Variante ist eleganter und bietet eine engere Integration, ist aber auch schwieriger und z. T. Browser-abhängig.

### **Zusammenfassung**

In den letzten Jahren wurde massiv von Fat-Clients nach HTML migriert. Teilweise ist aufgrund der Limitierungen von HTML wieder eine Kehrtwende im Gange. Dabei sollte man aber die Vorteile von HTML nicht einfach über Bord werfen. Die Kombination der positiven Eigenschaften von Fat- und Thin-Client kommt Benutzern, Entwicklern und Betreibern zugute. Die dafür notwendige Infrastruktur zu entwickeln ist allerdings aufwändig und nur realistisch, wenn sie für mehrere Anwendungen eingesetzt werden kann.



**Bruno Schäffer** ist Mitgründer und CTO der Softwarefirma Canoo in Basel. Er beschäftigt sich seit Jahren mit Java und Objektorientierung, in der letzten Zeit vor allem mit der Entwicklung von Infrastruktur für Java-basierte Thin-Clients.  
E-Mail: [bruno.schaeffer@canoo.com](mailto:bruno.schaeffer@canoo.com).

▼ **Weiterführende Informationsquellen**

<http://www.canoo.com/ulc>

## **Stellensuche oder Stellenanzeige**

– hier in JavaSPEKTRUM sprechen Sie garantiert die richtige Zielgruppe an.

Bitte wenden Sie sich mit Ihren Anzeigenwünschen an:  
Brigitta & Karl Reinhart  
Tel.: +49 / 89 / 46 47 29 · Fax: +49 / 89 / 46 38 15  
E-Mail: [Brigitta.Reinhart@sigs-datacom.de](mailto:Brigitta.Reinhart@sigs-datacom.de)  
[Karl.Reinhart@sigs-datacom.de](mailto:Karl.Reinhart@sigs-datacom.de)