Java in der Hosentasche

# Java Virtual Machines für PDAs

**Daniel Mettler und Sönke Gold**

Seit Ende Mai 2000 liegt die Spezifikation der *Connected Limited Device Configuration (CLDC)* vor, welche Teil der *Java 2 Micro Edition Platform (J2ME)* von Sun Microsystems ist. Dieser Artikel gibt zunächst einen Überblick über den gegenwärtigen Stand der J2ME-Spezifikation und eine kurze, prägnante Einführung in diese Plattform, welche für mobile Geräte wie PDAs gedacht ist. Im Hauptteil werden die Ergebnisse mehrerer Benchmarktests von *Java Virtual Machines (JVMs)* für die CLDC-Spezifikation auf gängigen PDA-Plattformen präsentiert.

Es zeigt sich, dass die J2ME-Spezifikation selbst für anspruchsvolle Java-Programmierung auf PDAs geeignet ist. Der Wahl der „richtigen" JVM fällt dabei je nach Zielplattform eine unterschiedliche Bedeutung zu.

Sun Microsystems Java 2-Plattform verfügt mit der *Java 2 Enterprise Edition (J2EE)*, der *Standard Edition (J2SE)* und der *Micro Edition (J2ME)* über Java-Spezifikationen für die Server-, Desktop- und Mobil-/Embedded-Bereiche. Im Rahmen einer *Advanced Technologies Study* der *Credit Suisse Financial Services* wurde die Java 2 Micro Edition-Spezifikation genauer untersucht.

In der Studie ging es in einer ersten Phase darum, praxisorientierte Leistungs- bzw. Geschwindigkeitsmessungen bei aktuellen CLDC Java Virtual Machines für Personal Digital Assistants (Abk. PDAs, auch portable devices oder Mobile Endbenutzergeräte) vorzunehmen. Dadurch sollte eruiert werden, inwiefern zurzeit eine plattformübergreifende Software-Entwicklung für mobile Endbenutzergeräte mittels Java möglich ist. Es sollte anhand von Benchmarks verglichen werden, wie die Java Virtual Machines (JVMs) leistungsmäßig relativ zu einander und im Vergleich zu einer herkömmlichen J2SE-Virtual Machine abschneiden.

Der vorliegende Artikel dokumentiert diesen Evaluierungsprozess in geraffter Form. Dazu wird zuerst ein kurzer Überblick über die J2ME-Spezifikation und im Speziellen über die für PDAs relevanten CLDC- und MIDP-Spezifikationen vermittelt. Danach wird kurz auf die getesteten JVMs und ihre verwendeten Verfahren eingegangen. Die Eigenschaften des für die Messungen entwickelten Benchmarks werden erklärt und die Testanordnung dokumentiert. Schließlich werden die Resultate diskutiert und resümiert.

## Die Java 2 Micro Edition im Überblick

Zwei Prinzipien bei der ursprünglichen Spezifikation von Java waren „compile once, run everywhere" und „one size fits all": Ist ein Java-Programm erst einmal in den *Java-Bytecode* kompiliert, sollte es dank virtueller Maschinen unverändert auf beliebigen Hardware-Plattformen lauffähig sein.

Im Hinblick auf die Veröffentlichung der Java 2-Plattformspezifikation mussten diese Absichten jedoch revidiert werden. Die Verschiedenartigkeit der erhältlichen Zielgeräte (vom großen Enterprise-Server bis zur kleinen Chipkarte) bezüglich ihrer Eingabe-, Ausgabe- und Vernetzungsmöglichkeiten sowie Leistungsfähigkeiten hatte sich als zu großes Hindernis für dieses Konzept erwiesen. Stattdessen garantiert die Java 2-Plattform und im Speziellen die J2ME-Spezifikation von vornherein nur eine begrenzte Plattformunabhängigkeit. Java-Applikationen, die auf der J2ME-Plattform laufen sollen, sind also nicht komplett plattformunabhängig.

J2ME definiert als architektonisches Grundkonzept so genannte Konfigurationen und Profile. Eine *Konfiguration* spezifiziert die minimalen Anforderungen, die eine bestimmte „Mobilgerätekategorie" (ein bestimmtes *horizontales* Marktsegment) bezüglich Java erfüllen muss. Sie beschreibt somit die Eigenschaften und den Funktionsumfang der Zielgeräte und der JVM. Außerdem sind darin die unterstützten Klassenbibliotheken spezifiziert.

Die Ausarbeitung von allen Java-Spezifikationen erfolgt nach den Regeln des *Java Community Process (JCP)*. Der Stand der jeweiligen Spezifikation wird in den *Java Specification Requests* [JSR] festgehalten.

Zurzeit sind zwei Konfigurationen verabschiedet: Die *Connected Device Configuration* (JSR-36), die sich an festinstallierte, vernetzte Geräte wie Set-Top-Boxen oder Bildtelefone richtet, und die *Connected Limited Device Configuration* (JSR-30) [CLDC], die für persönliche, vernetzte, mobile Informationsgeräte wie PDAs oder Mobiltelefone gedacht ist. Die CLDC-Spezifikation entspricht dabei gewissermaßen einer Teilmenge der CDC-Spezifikation. Beide Konfigurationen verfügen mit der *CVM* resp. *KVM* über Referenzimplementierungen ihrer JVM-Spezifikationen [KVM].

*Profile* setzen entweder direkt auf einer Konfiguration oder einem anderen Profil auf und definieren zusätzliche Klassenbibliotheken, die von den jeweiligen, profilkonformen Geräten einer „Mobilgerätefamilie" zwingend zu unterstützen sind. Eine Mobilgerätefamilie kann dabei als ein bestimmtes *vertikales* Marktsegment angesehen werden. In einer ersten Version verabschiedete Profile sind das *Foundation Profile* (JSR-46) für die CDC,die auf dem Foundation Profile aufsetzenden Profile *RMI Profile* (JSR-66) und *Personal Basis Profile* (JSR-129) sowie das *Mobile Information Device Profile* (JSR-37) [MIDP] für die CLDC.

Eine noch nicht verabschiedete CDC-Profilspezifikationen (Stand: 12.07.2002) ist das *Personal Profile* (JSR-62) das als Nachfolger der *PersonalJava*-Spezifikation gedacht ist. Das oben erwähnte Personal Basis Profile wiederum ist eine abgespeckte Variante des Personal Profile und verzichtet auf die Rückwärtskompatibilität zu PersonalJava.

Ebenfalls noch in der Ausarbeitungsphase befinden sich das *Mobile Information Device Profile Next Generation* (MIDP_NG; JSR-118) sowie das *PDA Profile* (PDAP, JSR-75), die beide auf der CLDC aufsetzen und stark an die MIDP-Spezifikation angelehnt sind. Abbildung 1 zeigt einen Überblick der Architektur der Java 2 Micro Edition.

Für den vorliegenden Artikel wurden Leistungstests auf gängigen PDA-Plattformen durchgeführt. Dafür eignen sich die CDLC und das MIDP am besten, da einerseits das PDA-Profil weitgehend auf der MIDP-Spezifikation basiert und andererseits die CLDC PDAs als angepeilte Zielplattform spezifiziert. Von der MIDP-Spezifikation sind zudem bereits mehrere Implementierungen für verschiedene PDA-Geräte vorhanden, was Leistungsmessungen und -vergleiche überhaupt erst ermöglicht. So gibt es von Sun z. B. eine komplette Referenzimplementierung des CLDC und MIDP für Palm-Geräte. Der nächste Abschnitt stellt die wichtigsten Punkte der CLDC- und MIDP-Spezifikationen übersichtsmäßig dar.

## Connected Limited Device Configuration (CLDC)

Welche hardware- und softwaremäßigen Voraussetzungen müssen Zielgeräte nun erfüllen, um überhaupt für eine CLDC-Kompatibilität in Frage zu kommen? Die CLDC-Spezifikation nennt folgende Anforderungen an ein Zielgerät:

- 160 bis 512 KB verfügbares, totales Speicherbudget für die Java-Plattform (mindestens 128 KB nichtvolatiler Speicher für die JVM, die CLDC- sowie die Profil-Bibliotheken; mindestens 32 KB volatilen Speicher für die Java-Laufzeitumgebung und als Objektspeicher für Bibliotheken und den Applikationscode),
- Vorhandensein eines 16- oder 32-Bit-Prozessors,
- geringer Energieverbrauch; häufig batterie- resp. akkubetrieben,
- Vorhandensein eines Netzwerkzugangs; oftmals über drahtlose, zeitweilig unterbrochene Verbindungen mit einer limitierten Bandbreite, oft mit einer Durchsatzrate von 9600 bps oder weniger,
- Vorhandensein eines minimalen Host-Betriebssystems oder -Kernels für den Hardware-Zugriff und zur Ausführung der Java Virtual Machine.

Aufgrund dieser einschränkenden Rahmenbedingungen kann das CLDC nicht den gesamten Umfang der *Java Language Specification* [JLS] und *Java Virtual Machine Specification* [JVMS] unterstützen. So müssen in einigen Bereichen Abstriche gemacht werden, in anderen sind zusätzliche Maßnahmen nötig, damit Java-Applikationen auf den erwähnten Geräten in sinnvoller Weise funktionieren können. Bezüglich der Unterstützung von Java formuliert die CLDC-Spezifikation dementsprechend folgende Vorgaben:

- *Weitgehende Kompatibilität mit der JLS. Ausnahmen:*
  - keine Unterstützung für Fliesskommazahlen (Datentypen `float` und `double`),
  - keine *Finalization* von Klasseninstanzen,
  - auf bestimmte Fehlerklassen limitierte Fehlerbehandlung.
- *Weitgehende Kompatibilität mit der JVMS. Einzige Unterschiede:*
  - keine Unterstützung für Fliesskommazahlen,
  - keine Unterstützung für das *Java Native Interface (JNI)*. Das JNI dient in Java der nativen, hardwarenahen Programmierung,
  - keine benutzerdefinierten Java-Klassenlader,
  - kein *Reflection API*. Dieses dient bei Java 2 der (Selbst-)Inspektion und Serialisierung von Objekten,
  - kein Support für *Thread-Groups* und *Daemon-Threads*,
  - keine *Finalization* von Klasseninstanzen,
  - keine schwachen Referenzen,
  - Fehlerbehandlung nur in limitiertem Umfang,
  - Bytecode-Verifikation zur Kompilier- und Laufzeit: Bei der so genannten *Pre-Verification* werden Klassendateien zur Kompilierzeit (in der Regel auf einem Server- oder Desktop-Computer) auf Korrektheit überprüft und ein in der CLDC neu definiertes „stack map" Attribut in den Standard-Java-Bytecode der entsprechenden Datei eingefügt. Die Dateigröße der vorüberprüften Klassen nimmt dadurch um etwa 5% zu, die Klassendateien bleiben jedoch durch eine J2SE-kompatible JVM unverändert ausführbar, da dieses neue Attribut nur durch eine CLDC-JVM ausgewertet wird.
  - vorgeschriebene Verwendung regulärer Dateiformate: Dazu zählen die Class- und JAR-Dateiformate. Für der Öffentlichkeit zugänglich gemachte Java-Applikationen und -Ressourcen ist die Verwendung des JAR-Dateiformats vorgeschrieben. Klassendateien müssen stets die bei der Pre-Verification eingefügten „stack map" Attribute enthalten. Zukünftige Versionen der CLDC-Spezifikation können weitere, zulässige Dateiformate definieren.
  - Vorhandensein einer unveränderlichen, implementierungsabhängigen Klassendatei-Suchreihenfolge: Systemklassen müssen vor *Overriding* geschützt werden.
  - optionales, transparentes *Pre-Loading/Pre-Linking* von Klassen: In Zusammenhang mit PDAs wird das auch *ROMizing* genannt.
- *Low-Level-Sicherheit auf Stufe der JVM* (Classfile Pre-Verification auf der Entwicklungsmaschine und Classfile-Verification zur Laufzeit auf dem Gerät selbst)
- *High-Level-Sicherheit auf Stufe der Applikation* (Einsatz eines im Vergleich zu J2SE vereinfachten Sandbox-Modells, Schutz der Systemklassen, optionale Unterstützung mehrerer, simultan ausgeführter Java-Applikationen). Ein Profil kann darüber hinaus weitere Sicherheitsfunktionen definieren.
- *Vorgeschriebene Unterstützung aller CLDC-Bibliotheken*

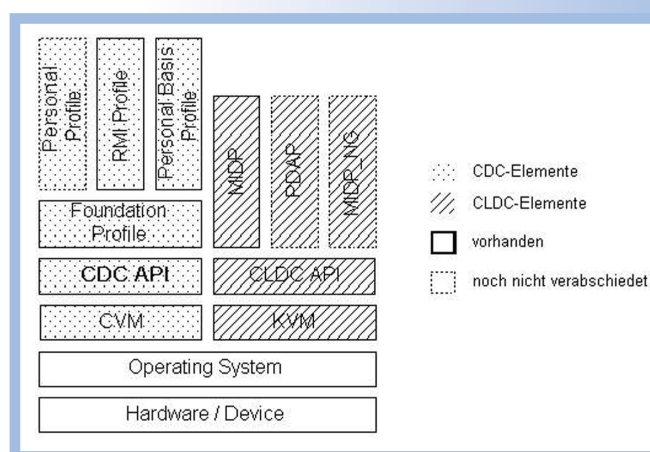


**Abb. 1: Die J2ME-Architektur.**

Grundsätzlich sieht die CLDC keine *optionalen* Merkmale vor. Implementierungen von CLDC-kompatiblen Virtual Machines ist es aber erlaubt, zusätzliche, nicht-spezifizierte Funktionalitäten anzubieten, sofern diese gegenüber den darunter und darüber liegenden Schichten transparent implementiert sind. Die Unterstützung eines Just-in-Time-Compilers oder des Pre-Linking von Klassen sind unter dieser Bedingung also erlaubt.

Die Limitiertheit der CLDC spiegelt sich auch im Umfang der unterstützten Java-Bibliotheken wider. Folgende Bibliotheken und Klassen werden durch die CLDC spezifiziert:

- Java-Basisbibliotheken (System-, Datentypen- und Fehlerbehandlungsklassen in `java.lang.*`; Collection-, Kalender-, Zeit- und zusätzliche Hilfsklassen in `java.util.*`),
- Eingabe-/Ausgabeklassen (`java.io.*`),
- Netzwerkzugriffsklassen im Paket `javax.microedition`. Dieses Framework stellt eine Generalisierung der Netzwerkzugriffsklassen der J2SE dar. Im Paket `javax.microedition.io` befinden sich Interfaces, durch welche die Profile entsprechende Netzwerkprotokollimplementierungen zur Verfügung stellen können. Das Binden eines bestimmten Protokolls an diese Interfaces geschieht zur Laufzeit.
- leicht eingeschränkte Internationalisierungs-Unterstützung, keine Unterstützung für Lokalisierung,
- limitierte Unterstützung für die Verwaltung von *Properties* (Speichern/Lesen von Systemeigenschaften).

Die CLDC-Bibliotheken und -Klassen sind mit Ausnahme der Klassen des `javax.microedition`-Pakets *Teilmengen* der J2SE-Klassen oder sogar *identisch* mit ihnen. Die Aufwärtskompatibilität zur J2SE ist gewährleistet, da die Klassen des `javax.microedition`-Pakets in J2SE abgebildet werden können. Die CLDC-Bibliotheken bilden außerdem eine Teilmenge der CDC-Bibliotheken.

Durch die CLDC hingegen explizit *nicht* spezifiziert werden die Verwaltung des Applikationslebenszyklus (Installation, Starten, Löschen einer Applikation), die Funktionalität der grafischen Benutzeroberfläche, die Ereignisbehandlung, das Anwendungsmodell auf abstrakter Stufe (die Interaktion zwischen dem Benutzer und der Applikation) und die Persistenz von Daten und Programmen.

Dadurch können Java-Applikationen, welche nur die in der CLDC-Spezifikation definierten Bibliotheken benutzen, auf beliebigen CLDC-Geräten unverändert ausgeführt werden. Sobald hingegen eine Applikation auch Bibliotheken eines bestimmten Profils verwendet, schränkt sich deren Portabilität auf Geräte des jeweiligen Profils ein. Definitionsgemäß wird die weniger enge CLDC-Spezifikation durch eine größere Anzahl an Geräten unterstützt als die Spezifikation eines spezialisierten Profils.

Durch die Auslagerung der Spezifikation der meist stark geräteabhängigen Merkmale in die Profilspezifikationen wird also eine bessere Wiederverwendbarkeit und Portabilität von Teilen einer Java-Applikation für Mobilgeräte erreicht.

## Das Mobile Information Device Profile (MIDP)

Die MIDP-Spezifikation deckt nun noch diejenigen Bereiche ab, die nicht bereits durch die CLDC spezifiziert werden. So muss ein *Mobile Information Device* mindestens über ein 96 Pixel breites und 54 Pixel hohes Schwarzweißdisplay verfügen und entweder eine einhändige oder zweihändige Bedienung ermöglichen. Eine kombinierte Bedienung mit Stift, Touchpad und virtueller Tastatur, wie man sie bei PDA-Geräten oft findet, ist ebenfalls zulässig.

Eine MIDP-Applikation wird in Form einer *MIDlet Suite* verwaltet. Eine MIDP-Applikation wird auch *MIDlet* genannt, weil sie stets von der Klasse `javax.microedition.midlet.MIDlet` erben muss, welche eine Schnittstelle zwischen der *Application Management Software (AMS)* und dem eigentlichen Applikationscode darstellt.

Die Midlet Suite kann aus mehreren, gebündelten MIDlets bestehen und wird als JAR-Datei z. B. auf einem Server zum Herunterladen zur Verfügung gestellt. Die Beschreibung dieser JAR-Datei (welche MIDlets in der Suite enthalten sind, deren Versionsinformationen, Urheberrecht usw.) geschieht einerseits in einer separaten JAD-Datei, andererseits in der Manifestdatei innerhalb der JAR-Datei selbst. „JAD" steht demnach für *Java Archive Descriptor*.

### Abkürzungsverzeichnis

| | |
|---|---|
| AMS | Application Management Software |
| AOT | Ahead Of Time |
| API | Application Programming Interface |
| CDC | Connected Device Configuration |
| CLDC | Connected Limited Device Configuration |
| FastBCC | Fast Bytecode Compiler |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transfer Protocol |
| IDEA | International Data Encryption Algorithm |
| J2EE | Java 2 Enterprise Edition |
| J2ME | Java 2 Micro Edition |
| J2SE | Java 2 Standard Edition |
| JAD | Java Archive Descriptor |
| JAR | Java Archive |
| JCP | Java Community Process |
| JIT | Just In Time |
| JNI | Java Native Interface |
| JSR | Java Specification Request |
| JVM | Java Virtual Machine |
| MD5 | MD5 Message Digest Algorithm |
| MIDP | Mobile Information Device Profile |
| MIDP_NG | Next Generation Mobile Information Device Profile |
| PDA | Personal Digital Assistant |
| PDAP | PDA Profile |
| RMI | Remote Method Invocation |
| RMS | Record Management System |
| RPC | Remote Procedure Call |
| SHA1 | SHA-1 Secure Hash Algorithm |
| SOAP | Simple Object Access Protocol |
| WAT | Way Ahead Of Time |
| XML | Extensible Markup Language |

| | KVM 1.0 | Jbed 2.0.5 | J9 1.4 |
|---|---|---|---|
| PocketPC / StrongARM | | | x |
| PocketPC / SH3 | | | x |
| PalmOS / 68k | x | x | x |

Tabelle 1: Die Matrix der getesteten JVMs und Geräte

Auf jedem MIDP-kompatiblen Gerät muss zur Verwaltung des Lebenszyklus einer MIDlet Suite (Installation, Upgraden, De-Installation) sowie der drei möglichen Laufzeitzustände eines MIDlets (*active*, *paused*, *destroyed*) eine Application Management Software (AMS) vorhanden sein.

Ein MIDlet kann sowohl ein High-Level- als auch ein Low-Level-GUI benützen. Beide sind in Form von Klassenframeworks im Paket `javax.microedition.lcdui` vorhanden.

Beim High-Level-GUI ist das zentrale Abstraktionselement der so genannte *Screen*. Ein Screen ist vergleichbar mit einem Fenster bei einer fensterbasierten Oberfläche. Es können mehrere Screens als Anzeigebehälter definiert werden, jedoch kann nur ein einziger Screen zu einem bestimmten Zeitpunkt angezeigt werden und ausschließlich dieser kann Benutzereingaben empfangen. Als Screens stehen vier verschiedene, selbsterklärende Varianten/Klassen zur Verfügung: `List`, `Alert`, `TextBox` sowie `Form`.

Der „Look and Feel" der Komponenten eines auf Screens basierenden MIDlets kann dabei durch den Programmierer nur minimal beeinflusst werden. So sind z. B. keinerlei Farb-, Form- oder Schriftänderungen, kein benutzerdefiniertes *Scrolling* und kein Abfangen einzelner Tastatureingaben möglich.

Das Low-Level-GUI mit der `Canvas`-Komponente bietet demgegenüber die mächtigeren Funktionen, jedoch zum Preis einer schlechteren Code-Portierbarkeit – die bei der Verwendung des low-level GUI Framework explizit *nicht* mehr garantiert wird – zwischen verschiedenen Mobile Information Devices. Dafür erhält der Entwickler die volle Kontrolle über das Erscheinungsbild und Verhalten der Applikation, kann einzelne

Tastatureingaben via low-level Event API abfangen und trotzdem aber auch high-level Events behandeln.

Als Display Manager fungiert in beiden Fällen die Klasse `javax.microedition.lcdui.Display`, von der pro aktivem MIDlet stets nur eine Instanz vorhanden ist.

Analog zum GUI gibt es auch ein high- und low-level API zur Ereignisbehandlung (mit sinngemäßen Konsequenzen, was die Mächtigkeit der Funktionen und die Portabilität des jeweiligen Programmcodes betrifft). Die Ereignisbehandlung des high-level Event API beruht dabei auf dem Konzept der *Listeners*, währenddessen das low-level API via Instanzmethoden der `Canvas`-Klasse auf Ereignisse reagiert. Alle GUI-Callbacks werden stets in serieller Form abgearbeitet, allfällige Timer-Callbacks parallel dazu.

Der Kommunikation dient das Paket `javax.microedition.io`. Als einziges Kommunikationsprotokoll der Applikationsebene wird standardmäßig auf jedem MIDP-Gerät das weitverbreitete Hypertext Transfer Protocol (HTTP) unterstützt. Via *Remote Procedure Calls* (z. B. mittels *XLM-RPC/SOAP*) und *HTTP-Tunnelling* steht MIDlets damit ein potentiell breites Spektrum an Kommunikationsmöglichkeiten zur Verfügung. Optional können durch ein MID auch weitere Protokolle direkt unterstützt werden, jedoch ist bei deren Verwendung die Portabilität des Quell- und Bytecodes zwischen verschiedenen MIDP-Geräten nicht mehr gewährleistet.

Zur persistenten Speicherung von Daten dient das *Record Management System (RMS)* im Paket `javax.microedition.rms` mit dem `RecordStore` als konzeptuell zentrale Klasse. Im RMS gespeicherte *Records* bleiben nur während der Lebensdauer einer MIDlet Suite erhalten. Innerhalb der eigenen MIDlet Suite kann ein MIDlet stets auf alle Records, also auch auf diejenigen eines anderen MIDlets zugreifen. Der Record Store wird für die MIDlet Suite auf transparente Weise in einer Art „Sandbox" gespeichert – ein direkter Zugriff eines MIDlets auf eine native Datenbank eines Gerätes ist somit bewusst ausgeschlossen (resp. nur über optionale, zusätzliche Bibliotheken realisierbar). Im Gegenzug sind dafür die bestehenden, „nativen" Daten auf dem Gerät vor Veränderungen durch MIDlets geschützt, was ein mögliches Schadenspotential eines MIDlets verringert.

## Getestete Java Virtual Machines

Getestet wurden drei verschiedene Java Virtual Machine Implementierungen für die CLDC in den jeweils aktuellen Versionen, nämlich

- die Referenzimplementierung von Sun Microsystems, die *KVM 1.0* [KVM],
- die *Jbed Micro Edition CLDC 2.0.5* von Esmertec [JBED] sowie
- die *J9 1.4* Virtual Machine von IBM.

IBM J9 ist seit kurzem nicht mehr separat erhältlich, sondern nur noch als Gesamtpaket mit der Entwicklungsumgebung IBM *WebSphere Studio Device Developer* [J9].

Alle drei JVMs wurden auf der weitverbreiteten PalmOS-Plattform getestet und zwar unter PalmOS 4.0 auf einem *Palm m500*-Gerät. IBMs J9 stand zudem auch für die PocketPC-Plattform zur Verfügung; getestet wurde sie auf Microsoft Windows CE 3.0 basierten Geräten von HP (Jornada 545) und Compaq (iPAQ H3660). In Tabelle 1 ist die Testmatrix dargestellt.

Die getesteten PDA-Plattformen weisen eine unterschiedliche Hardware- und Softwareausstattung auf:

- *PocketPC / StrongARM*: *Compaq iPAQ H3660* mit einer 206 MHz *Intel StrongARM* CPU, 64 MB RAM, 16 MB ROM. Betriebssystem: Microsoft Windows CE 3.0
- *PocketPC / SH3*: *HP Jornada 545* mit einer 133 MHz *Hitachi SH3* CPU, 16 MB RAM, 16 MB ROM. Betriebssystem: Microsoft Windows CE 3.0
- *PalmOS / 68k*: *Palm m500* mit einer 33 MHz *Motorola DragonBall VZ* CPU, 8 MB RAM, 4 MB ROM. Betriebssystem: PalmOS 4.0

Zum Vergleich der Testresultate wurden die Tests auch auf einer aktuellen Desktop-Workstation (*Intel Pentium III* 800 MHz CPU, 256 MB SDRAM, IDE-Harddisk) unter *Microsoft Windows NT 4* als Betriebssystem und *Sun Java 2 SDK 1.3.1* durchgeführt. Die getesteten JVMs differieren maßgeblich in der Art und Weise, wie Java-Bytecode ausgeführt wird. Man kann dabei grob vier Verfahren unterscheiden:

- Die *Interpretation* von Java-Bytecode ist das ursprüngliche Verfahren, wie eine JVM eine Java-Applikation handhabt. Dabei werden die Bytecode-Anweisungen Schritt für Schritt interpretiert, d. h. analysiert und ohne einen Kompilierungszwischenschritt ausgeführt. Die Interpretation von Bytecode ist ein zeitaufwändiger und wiederholt in gleicher Form auftretender Prozess, was die Performance von Java-Applikationen im Vergleich zu nativen Applikationen um (große) Faktoren geringer ausfallen lässt. Die KVM ist eine Vertreterin dieser JVM-Klasse. Java-Bytecode wird durch die KVM ausschließlich interpretiert.
- Um die Geschwindigkeitsnachteile der Interpretation von Code zu umgehen, wird bei der *Just-In-Time Compilation* Bytecode zu nativem Code (Maschinencode) kompiliert und dieser anstelle des Bytecodes ausgeführt. Der Kompilierungszwischenschritt geschieht dabei zur Laufzeit und Methode für Methode, un-

abhängig vom jeweiligen Kontext. Die dadurch dazu gewonnene Ausführgeschwindigkeit einer Methode wird jedoch durch einen hohen Zeitaufwand für die Kompilierung erkauft, sodass sich die JIT-Kompilierung nicht immer lohnt. Bei einer moderneren Variante der JIT Compilation, der *Dynamic/Adaptive JIT Compilation*, werden deshalb nur noch diejenigen Methoden kompiliert, welche häufig aufgerufen werden. Auf welche Java-Methoden dies zutrifft, ermittelt ein *Profiler* zur Laufzeit. Die *Hot Spot* JVM des Java 2 SDK 1.3.1 bedient sich einer dynamischen JIT-Kompilierung.

- Bei der *Ahead-Of-Time (AOT) Compilation* (auch *Offline Compilation* genannt) wird der Bytecode zu nativem Code kompiliert *bevor* die Applikation auf dem Zielgerät installiert wird [FAB]. Dies kann entweder gleich für alle Klassen geschehen oder selektiv nur für diejenigen Klassen, welche performanzkritisch sind, wobei die restlichen Klassen normal interpretiert werden. Die IBM J9 verwendet die AOT-Kompilierung, jedoch ergänzt durch die JIT-Kompilierung.
- Die *Way-Ahead-Of-Time (WAT) Compilation* wird je nach Quelle etwas unterschiedlich definiert. Gemeinsam ist den Definitionen jedoch, dass bei der WAT-Kompilierung im Gegensatz zur JIT- und AOT-Kompilierung stets der *gesamte* Bytecode einer Java-Applikation *vor der Laufzeit* in nativen Code für die Zielplattform umgewandelt wird. Dadurch kann die zeitaufwändige Interpretation von Code vollständig umgangen werden. Zur Beschleunigung der geforderten Java-Laufzeitumgebung auf dem Zielgerät selbst (benötigt z. B. zum dynamischen Laden von Klassen) werden bei WAT-Systemen JIT- oder vorzugsweise ebenfalls WAT-Compiler eingesetzt. Der Fast Bytecode Compiler (FastBCC) von Jbed fällt in letztere Kategorie. Jbed benutzt ausschließlich die WAT-Kompilierung, sie verzichtet also vollständig auf eine Interpretation von Bytecode.

## Der Benchmark

Zum Zeitpunkt der Evaluation waren auf dem Markt keine geeigneten Benchmark-Frameworks zum Testen von J2ME Virtual Machines verfügbar. Es gab zwar den MIDP low-level GUI Benchmark *Amark* [AM] und den *TaylorBench* [TB]. Beide Benchmarks sind aber ausschließlich für MIDP-Geräte mit vollständiger Unterstützung des `javax.microedition.lcdui`-Pakets geeignet. Da die IBM J9 JVM dieses Paket zur Zeit der Evaluation noch nicht unterstützte, wurde beschlossen, selbst einen Benchmark ohne Einbezug von GUI-Tests zu implementieren.

Der Benchmark sollte dabei eine gute Vergleichbarkeit der Leistungsfähigkeit der verschiedenen JVM-Implementierungen ermöglichen und dazu zu den Java-Spezifikationen CLDC/MIDP, PersonalJava und J2SE kompatibel sein. Beim Entwurf der Architektur des Benchmarks wurde deshalb auf eine gute Modularisierung und Erweiterbarkeit geachtet.

Aufgrund dieser Vorgaben und der teilweise noch unvollständigen und fehlerhaften Unterstützung der MIDP-Bibliotheken durch die zu testenden JVMs musste jedoch auf den Einbezug von GUI-Tests in den Benchmark verzichtet werden. Stattdessen beschränkt sich der Benchmark auf Tests von Kryptografie-Funktionen (Generieren von MD5-, SHA1-Hashes und IDEA-Chiffren) sowie einiger Basisfunktionen (Instanzieren eines Objekts, Aufrufen einer Methode, Vergleichen von Objektreferenzen, Initialisieren eines Arrays).

Die Implementierung der Kryptografie-Tests erfolgte gestützt auf die *Bouncy Castle Crypto APIs*, welche als Opensource-Software frei zur Verfügung stehen [BC].

Das API des Benchmarks wurde in drei Pakete aufgeteilt, wobei eines davon die Klassen der erwähnten, spezifikationsübergreifenden Sub-Benchmarks enthält (Tabelle 2), eines PersonalJava- resp. J2SE-spezifische Klassen umfasst (Tabelle 3) und eines schließlich Klassen beinhaltet, die ein MIDlet formen, somit auf MIDP-Geräten lauffähig sind (Tabelle 4).

Das Bouncy Castle Crypto API ist lose an das Benchmark API gekoppelt: Einzig die Klassen `IDEABench`, `MD5DigestBench` und `SHA1DigestBench` sind davon abhängig. Die Abstraktion der Datenausgabe via `GenericPrinter`-Klasse wurde zur Sicherstellung der Spezifikationsunabhängigkeit des `com.csg.bench.generic`-Paketes eingeführt.

| Klasse | Beschreibung |
|---|---|
| *ArrayBench* | Initialisiert einen Array mit 10 Elementen. |
| *FunctionCallBench* | Ruft eine kleine Methode (mit einem Rückgabewert) auf. |
| *GenericBench* | Eine generische Benchmarkklasse, von welcher alle Sub-Benchmarks erben. Sie enthält Funktionen zur exakten Messung der Zeitdauer und des Speicherverbrauchs. |
| *GenericPrinter* | Das generische Interface zur Ausgabe der Messdaten, welches von *GenericBench* verwendet wird. Die spezialisierten Printerklassen in den *com.csg.cs.bench.j2me*- und *com.csg.cs.bench.j2se*-Paketen implementieren dieses Interface. |
| *IDEABench* | Berechnet eine IDEA-verschlüsselte Chiffre eines 256 Bit großen, konstanten Eingabewertes. |
| *IdentityBench* | Vergleicht zwei Referenzen auf ein Testobjekt auf Identität. |
| *MD5DigestBench* | Berechnet einen MD5-Hashwert eines zufälligen, neunstelligen Ganzzahlwertes. |
| *ObjectCreateBench* | Initialisiert und dereferenziert ein Testobjekt. |
| *SHA1DigestBench* | Berechnet einen SHA1-Hashwert eines zufälligen, neunstelligen Ganzzahlwertes. |
| *TestObject* | Ein kleines Testobjekt, welches von *IdentityBench* und *ObjectCreateBench* verwendet wird. |

Tabelle 2: Das Paket *com.csg.cs.bench.generic*

Die Implementierung der teilweise verwendeten Zufallsgeneratoren wurde so gestaltet, dass der Prozess der Zufallsgenerierung die Zeitmessung nicht beeinflusst.

## Testanordnung

Die Testanordnung war so ausgelegt, dass jeweils hundert Durchläufe aller Sub-Benchmarks pro Benchmarkdurchlauf stattfanden. Die hohe Zahl an Wiederholungen wurde gewählt, weil typische JVM-Beschleunigungstechniken wie die JIT-Kompilierung erst unter diesen Bedingungen überhaupt eine Wirkung entfalten können.

Pro Testplattform/-konfiguration wiederum wurden zur Erhöhung der Messgenauigkeit drei solche Benchmarkdurchläufe seriell auf dem gleichen Gerät durchgeführt, wobei das Gerät vor jedem Benchmarkdurchlauf per Hardware-Reset in den Auslieferungszustand zurückgesetzt wurde.

Die getesteten Geräte waren während der Tests stets ans Stromnetz angeschlossen, um einen möglichen Einfluss der Akkus auf die Messungen auszuschließen. Der Benchmark wurde jeweils mit den in den Dokumentationen der getesteten JVMs angegebenen Verfahren kompiliert und ausgeführt. Dementsprechend wurden nur jene Optimierungen vorgenommen, die ausdrücklich in der Dokumentation beschrieben und empfohlen wurden. Die voreingestellten Heap-Größen der JVMs wurden nicht verändert.

Auf eine Ausgabe der berechneten Hash- und Chiffrewerte wurde verzichtet, sie kann jedoch bei Bedarf eingeschaltet werden. Die korrekte Funktionsweise der JVMs wird durch den Benchmark jeweils vor Ausführen der kryptografischen Sub-Benchmarks anhand von Testvektoren überprüft.

Als Vergleichsplattform diente eine Intel Pentium III basierte Workstation mit einem Sun JDK 1.3.1, wie oben beschrieben unter „Getestete Java Virtual Machines".

## Benchmarkresultate

Von den drei Benchmarkdurchläufen je Gerät und JVM wurden das arithmetische Mittel sowie das zugehörige Konfidenzintervall (bei einer statistischen Sicherheit von 95%) auf Stichprobenbasis berechnet.

Im Folgenden sind die Ergebnisse nun grafisch aufbereitet dargestellt. Auf eine Auflistung der detaillierten Messergebnisse wird aus Platzgründen verzichtet, sie sind auf Anfrage bei den Autoren dieses Artikels erhältlich. Aufgeführt sind die standardisierten Ergebnisse der JVMs der Testmatrix. Die Standardisierung erfolgte für jeden Sub-Benchmark separat, wobei die Benchmarkzeitdauer der KVM 1.0 auf der Palm-Plattform jeweils einem Wert von 100% entspricht. Die KVM wurde als Standardisierungsbasis herangezogen, da diese Suns Referenzimplementierung einer CLDC-JVM ist und somit ein gutes Vergleichsmaß darstellt. Auf der X-Achse sind somit Prozentangaben aufgetragen, die in entsprechende Benchmarkdurchlaufzeiten, gemessen in Millisekunden (ms), umgerechnet werden können.

Dargestellt sind pro Sub-Benchmark die arithmetischen Mittelwerte aus drei Durchläufen. Tiefere Werte entsprechen kürzeren Benchmarkdurchlaufzeiten bzw. besseren Leistungsdaten und umgekehrt. Die zugehörigen 95%-Vertrauensintervalle sind als dünne, horizontale Linien um die Mittelwerte herum eingezeichnet (sehr kleine Vertrauensintervalle sind aufgrund der geringen Auflösung der Grafiken jedoch nicht mehr als solche erkennbar). Je schmaler ein Vertrauensintervall ist, desto geringer ist die Streuung der zugrundeliegenden Messergebnisse. Ein Messunterschied ausgedrückt durch eine Differenz der arithmetischen Mittel kann als signifikant (bezüglich einem Sicherheitsniveau von 95%) betrachtet werden, wenn sich die zugehörigen Vertrauensintervalle nicht überschneiden. Wenn im Folgenden von Signifikanz die Rede ist, dann ist implizit stets eine Signifikanz mit 95%-iger Wahrscheinlichkeit gemeint.

Man beachte, dass den aufgeführten JVMs teilweise unterschiedliche Hardware-Plattformen (mit entsprechend unterschiedlicher Leistungsfähigkeit) zugrunde liegen.

| Klasse | Beschreibung |
|---|---|
| *Bench* | Die Startklasse für PersonalJava-/J2SE-kompatible JVMs. |
| *BenchJ9* | Eine angepasste Startklasse zum Testen der IBM J9 (Bug-Workaround). |
| *Printer* | Eine spezialisierte Printerklasse, die Messdaten auf der Systemkonsole ausgibt. |

Tabelle 3: Das Paket *com.csg.cs.bench.j2se*

| Klasse | Beschreibung |
|---|---|
| *Bench* | Ein MIDlet, welches als Startklasse für MIDP-kompatible JVMs dient. |
| *BenchSuite* | Eine MIDlet Suite, welche aus Kompatibilitätsgründen für die Tests der Esmertec Jbed Virtual Machine benötigt wird. |
| *Printer* | Eine spezialisierte Printerklasse, die Messdaten auf einem MIDlet-Formular ausgibt. |

Tabelle 4: Das Paket *com.csg.cs.bench.j2me*

Abbildung 2 zeigt die Ergebnisse des SHA1DigestBench. Bei einem Durchlauf des SHA1DigestBench werden aus einhundert zufälligen, neunstelligen Ganzzahlwerten einhundert SHA1-Hashwerte berechnet (wobei die Zeit, die zur Generierung der Zufallszahlen nötig ist, nicht in die Messergebnisse einfließt).

Aus den Messergebnissen ist ersichtlich, dass sich alle Mittelwerte signifikant voneinander unterscheiden. Auffallend sind einerseits die stark unterschiedlichen Messresultate der drei JVMs der Palm-Plattform, andererseits der große Performance-Unterschied zwischen PalmOS-basierten JVMs und JVMs für die beiden PocketPC-Plattformen iPAQ und Jornada. Letzteres ist vermutlich auf die generell großen Leistungsunterschiede der darunter liegenden Hardwareplattformen zurückzuführen. So ist der StrongARM-Prozessor des iPAQ mit 206 MHz getaktet, die SH3-CPU mit 133 MHz, während der Palm m500 nur mit einer 33 MHz DragonBall CPU bestückt ist. Den Geräten steht auch unterschiedlich viel Speicher (RAM/ROM) zur Verfügung.

Die Messwertunterschiede innerhalb der gleichen Plattform können jedoch nur durch die unterschiedliche Leistungsfähigkeit der jeweiligen JVMs erklärt werden. Bezogen auf die Palm-Plattform ist die J9 JVM bei diesem Vergleich fast doppelt so schnell wie die Referenz Sun KVM, Jbed aber sogar über zehnmal schneller als die KVM.

Die Ergebnisse des MD5DigestBench (ohne Abbildung) und des IDEABench (Abbildung 3) sind ebenfalls alle signifikant und vergleichbar mit den Ergebnissen des SHA1DigestBench. Analog zum SHA1DigestBench werden beim MD5DigestBench pro Benchmarkdurchlauf einhundert MD5-Hashes aus zufälligen, neunstelligen Ganzzahlen berechnet. Wiederum fließt die Zeit, die zur Erzeugung der Zufallswerte nötig ist, nicht in die Messergebnisse ein.

Bei einem Durchlauf des IDEABench werden einhundert mit dem IDEA-Algorithmus verschlüsselte Chiffren aus ebenso vielen, 256 Bit langen konstanten Eingabewerten erzeugt. In Zahlen ausgedrückt benötigen hier die J9 VM auf dem Palm m500 etwa 70% und Jbed rund 10% der Zeit der KVM, um die gleiche Aufgabe zu bewältigen. Die gleichen Benchmarks werden auf den PocketPC-Plattformen sogar in nur 0.6 (iPAQ J9) bis 1.67 Prozent (Jornada J9) dieser Zeit absolviert. Diese beeindruckende Performance der PocketPC-Plattformen entspricht etwa einem Zwanzigstel bis einem Siebentel der Leistung eines aktuellen Desktop-PCs (vgl. Resultat des Java 2 SDK 1.3.1 auf der Workstation).

Des Weiteren wurden mit folgenden Sub-Benchmarks Messungen vorgenommen:

- Der ObjectCreateBench misst, wie lange es dauert, 100 kleine Testobjekte vom Typ `TestObject` zu initialisieren und zu dereferenzieren.
- Der FunctionCallBench ruft pro Durchlauf 100 kleine Methoden mit Rückgabewert auf.
- Beim IdentityBench werden hundertmal zwei Referenzen des gleichen Testobjekts auf Identität verglichen.
- Der ArrayBench schließlich initialisiert hundert Arrays mit je zehn Feldern.

Die bei diesen vier Benches gemessenen Durchlaufzeiten haben ein ähnliches Bild der relativen JVM-Leistungen wie bei den ersten drei Sub-Benchmarks gezeigt. Es sind also keine domänenspezifischen Leistungsschwächen der JVMs aufgefallen. Aus Platzgründen wird an dieser Stelle deshalb auf eine detaillierte Darstellung der Resultate dieser Sub-Benchmarks verzichtet.

## Schlussbetrachtung

Wie sind diese Resultate nun zu interpretieren? Insgesamt kann aufgrund der gemessenen, absoluten Leistungen festgehalten werden, dass Java-Programmierung auf PDAs mittels J2ME aus Performance-Sicht eine gute Alternative zu nativer Programmierung (z. B. mit C/C++) darstellt. Insbesondere eine JVM mit WAT-Kompilierung (wie sie Esmertec Jbed verwendet) ermöglicht eine sehr hohe Leistungsfähigkeit, wobei trotzdem die Vorteile von Java wie gute Code-Portabilität (dank J2ME), hohe Sicherheit durch ein Laufzeitsystem und moderater Entwicklungsaufwand er-

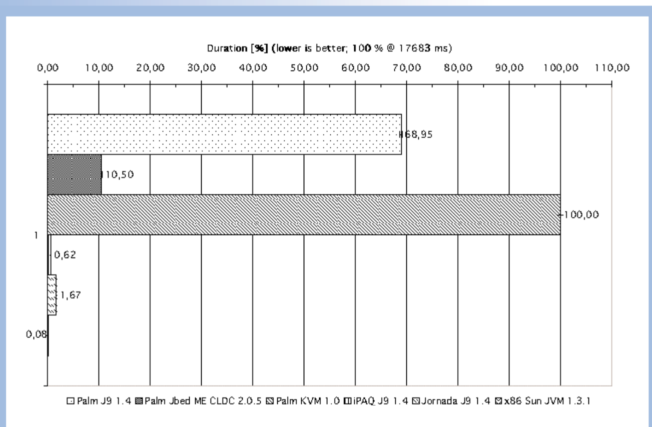


Abb. 2: SHA1DigestBench (100% entsprechen 17683 ms)

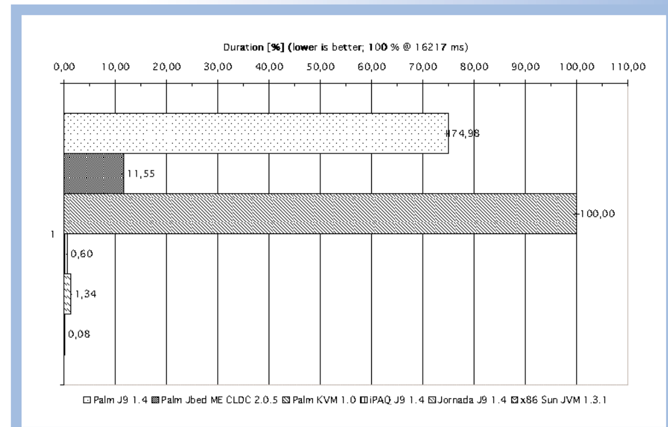


Abb. 3: IDEABench (100% entsprechen 16217 ms)

halten bleiben. Dadurch sind selbst rechenintensive Aufgaben wie starke Verschlüsselung in Java auf PDAs realisierbar.

Aus den Messergebnissen ersichtlich ist der deutliche Leistungsvorteil der beiden PocketPC/JVM-Kombinationen gegenüber denjenigen der Palm-Plattform. Erst die nächste Generation von Palm-Geräten (basierend auf PalmOS 5 und ARM-Prozessoren) wird hinsichtlich Hardwareleistungsfähigkeit voraussichtlich mit den PocketPC-basierten Geräten gleichziehen können. Während auf der PocketPC-Plattform der Wahl der verwendeten JVM vorderhand somit eine weniger bedeutsame Rolle zufällt, ist die Wahl der richtigen JVM auf der Palm-Plattform für performanzkritische Anwendungen entscheidend. Esmertec Jbed Micro Edition CLDC 2.0.5 hat sich auf der Palm-Plattform als klare Gewinnerin präsentiert, welche die getestete Konkurrenz (Sun KVM 1.0 und IBM J9 1.4) leistungsmäßig bis um den Faktor 10 zu distanzieren vermag.

Man kann somit festhalten, dass Sun mit der J2ME-Spezifikation den richtigen Weg eingeschlagen hat und die Performanz der JVMs zufrieden stellend ist. Einziger Wermutstropfen ist die teilweise noch etwas fehlerbehaftete und unvollständige Implementierung der JVMs.

Es wird spannend zu verfolgen sein, wie sich sowohl die Leistungen der JVMs als auch die Funktionalitäten der bestehenden und noch kommenden CLDC-Profile im dynamischen Marktumfeld des Mobile Computing weiter entwickeln werden.

## Quellen

Der Quellcode des Benchmarks kann auf Anfrage via E-Mail bei den Autoren bezogen werden.

Alle Rechte an in diesem Artikel verwendeten Markennamen gehören den jeweiligen Besitzern.

## Links

**[AM]** Amark, Fabio Ciucci, http://www.anfymobile.com

**[BC]** Bouncy Castle Crypto APIs, Legion of the Bouncy Castle, http://www.bouncycastle.org/

**[CLDC]** Connected, Limited Device Configuration, Specification Version 1.0a, Sun Microsystems, Inc., 2000, http://jcp.org/aboutJava/communityprocess/final/jsr030/

**[FAB]** Java Compilation for Embedded Systems: Current & Next Generation, Christian Fabre, 2000, http://www.nacse.org/HPjava/fabre/fabre.pdf

**[J9]** Websphere Studio Device Developer, International Business Machines Corporation, http://www.embedded.oti.com/wdd/

**[JBED]** Jbed Micro Edition CLDC, Esmertec, Inc., http://www.esmertec.com/p_jbed_cldc_long.html

**[JLS]** B. Joy, G. Steele, J. Gosling, G. Bracha, The Java Language Specification, Second Edition", Addison-Wesley, 2000, ISBN 0-201-31008-2, http://java.sun.com/docs/books/jls/

**[JSR]** Übersicht aller Java Specification Requests (JSRs), http://www.jcp.org/jsr/all/

**[JVMS]** T. Lindholm, F. Yellin, The Java Virtual Machine Specification (Java Series), Second Edition, Addison-Wesley, 1999, ISBN 0-201-43294-3, http://java.sun.com/docs/books/vmspec/

**[KVM]** The K Virtual Machine (KVM), A White Paper, Sun Microsystems, Inc., 2000, http://java.sun.com/products/cldc/wp/

**[MIDP]** Mobile Information Device Profile (JSR-37), JCP Specification, Java 2 Platform, Micro Edition, 1.0a, Sun Microsystems, Inc., 2000, http://jcp.org/aboutJava/communityprocess/final/jsr037/

**[TB]** TaylorBench, Richard Taylor, http://www.poqit.com/midp/bench/

**Daniel Mettler** studiert im 8. Semester Wirtschaftsinformatik an der Universität Zürich und arbeitet neben dem Studium am Departement IT Architecture and Standards der Credit Suisse Financial Services.
E-Mail: Daniel.Mettler@csfs.com oder mettlerd@icu.unizh.ch.

**Dr. Sönke Gold** arbeitet seit 1999 am Departement IT Architecture and Standards der Credit Suisse Financial Services. Seine Arbeitsschwerpunkte sind neue Endgeräte, PKI und Smartcards. E-Mail: Soenke.Gold@csfs.com.