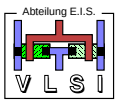
JControl – ein speichereffizienter JAVA-Ansatz

Tagungsbeitrag *Embedded Intelligence 2002* vom 14. Dezember 2001, Helge Böhme[1], Gerrit Telkamp[2]



[1]Abteilung E.I.S., TU BRAUNSCHWEIG
Mühlenpfordtstraße 23
38106 Braunschweig
ϟ h.boehme@tu-bs.de
☎ 0531/391-3108, Fax 0531/391-5840

[2]Domologic Home-Automation GmbH
Rebenring 33
38106 Braunschweig
ϟ g.telkamp@domologic.de
☎ 0531/3804-340, Fax 0531/3804-342

Zusammenfassung

Gezeigt werden einige Möglichkeiten, eingebettete JAVA-Anwendungen speichereffizient zu realisieren. Zum Einsatz kommen dabei Optimierungen auf der Anwendungsebene (z. B. Design-Rules), der API-Ebene (Strukturierung) und der Laufzeitumgebungsebene (spezielle Mechanismen der JAVAVM).

Einleitung

Die Begriffe „JAVA“ und „Speichereffizienz“ werden in der Praxis nur selten in einem Zusammenhang genannt. Der Grund mag darin liegen, dass die meisten JAVA-fähigen Systeme heute im High-End-Bereich zu finden sind. Dabei gibt es ein enormes Anwendungspotenzial im Bereich der eingebetteten Low-End-Systeme, die konventionell in Assembler oder „C“ programmiert werden. Hier kann nur selten die Hardware an erhöhte System-Anforderungen einer JAVA-Laufzeitumgebung angepasst werden. Die einzige Möglichkeit, JAVA als Programmiersprache für Systeme dieser Kategorie einzusetzen besteht darin, die Laufzeitumgebung an die geringe Hardware-Leistung anzupassen.

Für JAVA als Programmiersprache im Embedded Bereich sprechen vielerlei Gründe: Das objektorientierte Konzept, die leistungsfähigen Entwicklungswerkzeuge, Multi-Threading, Plattformunabhängigkeit, verbesserte Software-Wiederverwendung durch strukturierte Bibliotheken (APIs) usw. Der Preis für diesen Komfort ist bekannt: der Resourcenverbrauch. Beispielsweise werden beim Start einer Anwendung eine Reihe von Klassen dynamisch nachgeladen, initialisiert und dann nur teilweise genutzt. Dadurch wird nicht nur verhältnismäßig viel Speicher belegt, sondern es steigt auch die Startzeit der Anwendung. Gerade eingebettete Low-End-Systeme mit wenig System-Resourcen können auf diese Weise schnell an ihre Leistungsgrenze geführt werden.

JControl ist eine JAVA-Plattform, die auf die besonderen Anforderungen eingebetteter Low-End-Systeme optimiert wurde [1]. Eine Reihe spezieller Mechanismen ermöglichen es, *JControl* sogar auf 8-Bit-Mikrocontrollern mit 64 KB Adressraum einzusetzen. Auf diese Weise werden JAVA-programmierbare Single-Chip-Systeme realisierbar. *JControl* übernimmt dabei die Rolle des Betriebssystems.

Als Beispiel wurde *JControl* in einer konkreten Anwendung für ein Single-Chip-System (8-Bit) mit lediglich 3KB SRAM realisiert. Abbildung 1 zeigt die qualitativen Unterschiede beim Speicheraufbau dieses JAVA-Systems im Vergleich zu einem Desktop-System. Bei einem Desktop-System werden die JAVAVM und die JAVA-Klassen vom Filesystem in den Arbeitsspeicher geladen, die Klassen liegen dabei möglicherweise archiviert vor, werden also ggf. entpackt. In der Regel ist dabei der Arbeitsspeicher (heutzutage typisch 128 MB und mehr) größer als die für die Ausführung einer JAVA-Anwendung nötigen Klassen (falls nicht, verfügen moderne Systeme über eine virtuelle Speicherverwaltung).

Bei der hier vorgestellten *JControl*-Realisierung ist dieses Verhältnis umgekehrt, d.h. der Speicherplatzbedarf der Klassen übersteigt das verfügbare RAM um ein Vielfaches. Daher verbleiben die Klassen im

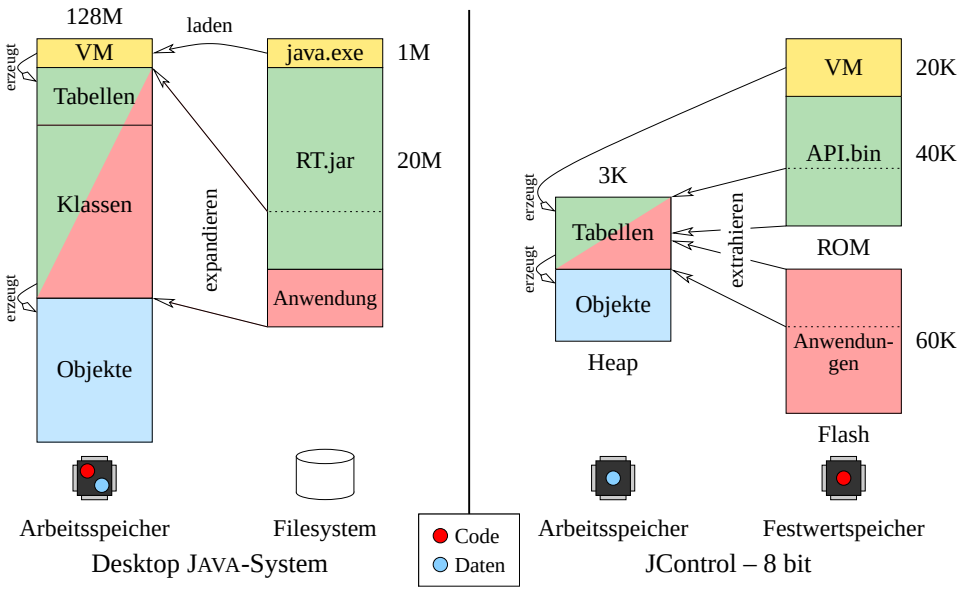


Abbildung 1: Vergleich der Speichernutzung

Festwertspeicher und werden direkt von dort ausgelesen und ausgeführt. Im RAM befinden sich lediglich dynamisch erzeugte Informationen der Klassen (Methodenreferenzen, statische Variablen, Zugriffstabellen auf den Konstantenpool; genaueres hierzu in Abschnitt 3) und natürlich die Objekte. In Zahlen können auf dem verwendeten Mikrocontroller Class-Files mit einer Gesamtgröße von 100KB abgelegt werden (verteilt auf zwei Speicherbänke), wobei einer ablaufenden JAVA-Anwendung ein Arbeitsspeicher von lediglich 3KB zur Verfügung steht. Je nach Komplexität der Anwendung müssen hier noch weitere Speichereinspar-Mechanismen eingesetzt werden, die nach unserer Erfahrung durchaus auch für Systeme mit deutlich großzügigerer Ausstattung sinnvoll sind (z.B. 32-Bit-System mit 4MB DRAM).

Dieses Papier zeigt einige Mechanismen, wie JAVA-Anwendungen resourcenschonend realisiert werden können. Drei Aspekte für die Einsparung des Speicherverbrauchs werden hier dargestellt:

optimierte Anwendungen ein Programmierer kann unter Einhaltung einiger Regeln den Speicherplatzbedarf der Anwendung verringern,

optimiertes API ein standard-mäßiges JAVA-API ist für eingebettete Systeme ungeeignet, da es mit der großen Hierarchie aus Vererbungen nur wenig speichereffizient benutzt werden kann,

optimierte Laufzeitumgebung die Architektur der zur Ausführung verwendeten JAVAVM kann einen entscheidenden Beitrag dazu leisten, Objekte und Klassen effizient zu kodieren.

1 Speicherplatzsparende JAVA-Anwendungen

Bei der JAVA-Programmierung im Allgemeinen gibt es einige Fallen, die unnötig den Speicherbedarf einer Anwendung erhöhen. Im Folgenden werden einige dieser Fallen dargestellt. Teilweise sind diese eventuell bereits als (vielfach unbeachtete) JAVA-Binsenweisheiten bekannt, teilweise wurden sie im praktischen Einsatz mit *JControl* erkannt. Wichtig ist es hierbei, zu wissen, dass nicht nur Objekte Heap-Speicherplatz belegen, sondern vor allem die dynamisch erzeugten Informationen der Klassen, Ziel soll es also sein, die Anzahl der residenten Klassen zu verringern.

1.1 Falle 1: Garbage or not Garbage

Die Aussage „Wieso, ich habe doch den Garbage-Collector“ wird des öfteren von JAVA-Programmierern gemacht, wenn auf eine resourcenintesive Anwendung hingewiesen wird. Ja, der Garbage-Collector entfernt nicht mehr benötigte Objekte (in machen Fällen auch Klassen)[1] aus dem Heap. Es muss ihm aber auch die Möglichkeit gegeben werden, diesen Speichermüll zu erkennen. Folgende Quellcode-Sequenz verdeutlicht dies:

```
1  import jcontrol.comm.*;
2  import jcontrol.io.*;
3  import java.io.IOException;
4  public class LocalVariableReferenceTest {
5    public static void main(String[] args) throws IOException {
6      DisplayConsole con=new DisplayConsole(new Display());
7      ROMresource res=new ROMresource("Guide.txt");
8      RS232 rs=new RS232();
9      rs.println("Watch the Display");
10     rs.close();
11     try {
12       for (;;) {
13         con.println(res.readLine());
14       }
15     } catch (IOException e) {
16       con.println("==EOF==");
17     }
18   }
19 }
```

Dieses (sinnfreie) Beispiel eines eingebetteten JAVA-Programms demonstriert einige I/O-Streams. Zunächst werden die benötigten Klassen instanziert und eine Meldung über eine serielle Schnittstelle verschickt, der Anwender möge das Display betrachten, dann wird eine Textdatei aus dem Festwertspeicher des eingebetteten Systems auf dem Display dargestellt. Wo ist hier die Falle?

Innerhalb der `for`-Schleife werden fünf Klassen benötigt (plus Anwendungsklasse(n)). Offensichtlich sind dies `ROMresource` und `DisplayConsole` aber auch `Display` (wird von `DisplayConsole` benutzt), `String` (ist Parameter von `println`) und `RS232`. Die Klasse `RS232` wird an dieser Stelle offensichtlich nicht mehr benötigt, der Stream wurde in Zeile 10 geschlossen und die Variable `rs` wird unten nicht mehr referenziert. Aber auch wenn die Anwendung nicht mehr auf `rs` zugreift, der Inhalt der Variablen bleibt erhalten und somit auch das `RS232`-Objekt und die dazugehörige Klasse. Ein explizites `rs=null;` löst dieses Problem, da nun keine Referenz mehr vorhanden ist und die Klasse vom Garbage-Collector freigegeben werden kann.

Manche JAVA-Compiler erkennen nicht mehr benötigte lokale Variablen und benutzen sie später für andere Zwecke. So kann es ausreichend sein, die Instanzen von `con` und `res` erst zu erzeugen, nachdem `rs` geschlossen wurde. Um sicher zu gehen, kann die Benutzung der Variable `rs` auch eingeklammert werden, so dass der Compiler einen genau definierten Bereich für die Gültigkeit von `rs` zugewiesen bekommt[2]:

[1] Viele JAVAVMs sind nicht in der Lage, einmal geladene Klassen wieder aus dem Speicher zu entfernen.

[2] Bei *Suns* `javac` ist diese Vorgehensweise sogar erforderlich.

```
[...]
  public static void main(String[] args) throws IOException {
    {
      RS232 rs=new RS232();
      rs.println("Watch the Display");
      rs.close();
    }
    DisplayConsole con=new DisplayConsole(new Display());
    ROMresource res=new ROMresource("Guide.txt");
[...]
```

Dieses Beispiel erzeugt dabei nicht mehr Code als das obige, wie es bei einem explizitem `null`-setzen der Variable der Fall ist.

1.2 Falle 2: versteckte Klassen

Aber nicht nur das Entfernen von Klassen kann der Programmierer steuern, er hat auch einen direkten Einfluss auf die Zahl der für eine Anwendung nötigen Klassen. Dies ist allerdings nicht immer einfach zu erkennen, z. B. :

```
String world="world";
rs.println("hello " + world);
```

Hier werden explizit nur zwei Klassen verwendet: `String` und eine `OutputStream`-Klasse in `rs`, das kleine „+“ verbirgt jedoch eine Menge Code, die der Compiler automatisch erzeugt, in etwa so:

```
String world="world";
rs.println(new StringBuffer().append("hello ")
           .append(world).toString());
```

Dieser Zusammenhang wird in der Dokumentation des *JDK* [3] erläutert, jeder JAVA-Programmierer sollte ihn kennen. Der Ausweg

```
String world="world";
rs.println("hello ".concat(world));
```

sieht zwar im Quelltext weniger übersichtlich aus, erzeugt aber im Bezug auf Speicherplatz optimaleren Code. Der Performance-Nachteil dieser Methode zählt in diesem Fall nicht, da das Initialisieren einer weiteren Klasse auch entsprechende Rechenzeit benötigt. Ferner handelt es sich hierbei auch bei Desktop-Systemen um einen Grenzfall, da der Geschwindigkeitsvorteil des `StringBuffers` eher bei umfangreicheren bzw. zahlreicheren Zeichenkettenoperationen greift (hier ist es ja nur eine).

1.3 Falle 3: zu viele Abhängigkeiten bei Aufrufen

Nicht nur Instanzen von Klassen erzeugen Referenzen, die die Klassen-Abbilder resident halten, sondern auch der Aufruf von Methoden, auch statischer (im Invoke-Stapel des Threads). Ein Anwendungsbeispiel mit Benutzerinteraktion soll dies verdeutlichen. Dem Benutzer soll ein Auswahlmenü präsentiert werden, ob dazu auf einem Display grafische Elemente dargestellt werden und die Auswahl per Tastendruck erfolgt oder eine Kommunikation über einen Stream erfolgt, soll an dieser Stelle irrelevant sein. Entscheidend dabei ist, dass das Menü eine gewisse Komplexität beherbergt und auch eine bestimmte Menge Klassen benötigt, die für dessen Darstellung sorgen. Erfolgt eine Auswahl, so soll eine entsprechende weitere Anwendung gestartet werden. Abbildung 2 veranschaulicht dies links.

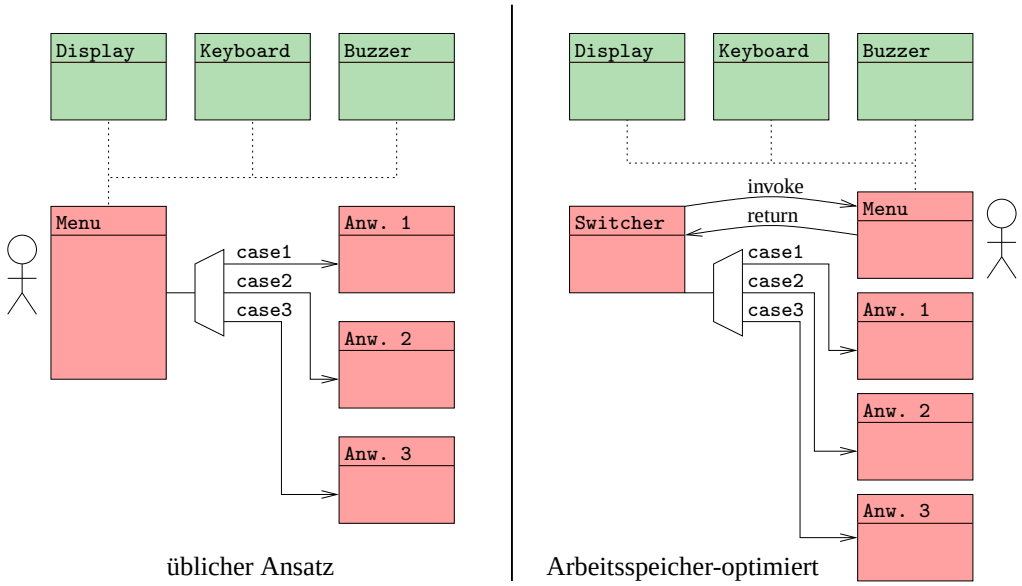


Abbildung 2: Abhängikeiten beim Methodenaufruf

Die übliche Vorgehensweise ist, in einer Klasse, die das Menü darstellt, die Benutzereingabe entgegenzunehmen und schließlich entsprechend zu verzweigen. Zur Darstellung des Menüs müssen ggf. weitere Klassen resident gehalten werden. Die gewählten Aktionen erfordern dann weitere Klassen (Teilanwendungen), während diese ausgeführt werden, bleibt die Menü-Klasse samt ihrer abhängigen Klassen resident. Um dies zu verhindern ist es sinnvoll, die Verzweigung von der Darstellung des Menüs zu trennen. Lediglich eine kleine Klasse muss permanent resident bleiben, die genauso, wie sie die übrigen Anwendungen startet, das Menü aufruft und die Entscheidung des Benutzers entgegennimmt. Die Gesamtzahl der benötigten Klassen nimmt dann zwar zu, aber die Anzahl der gleichzeitig im Arbeitsspeicher residenten Klassen wird reduziert (siehe Abbildung 2 rechts).

Der Trick mit den Threads

Ist diese Trennung nicht leicht möglich, etwa wenn es kein zentrales Menü gibt, sondern an unterschiedlichen Stellen des Programmablaufs Entscheidungen gefällt werden, die den Ablauf beeinflussen und ggf. weitere Klassen erfordern, so gibt es noch eine weitere Möglichkeit. Um zu verhindern, dass die aufrufenden Klassen auf dem Invoke-Stapel liegenbleiben, ist einfach der Invoke-Stapel selbst zu entfernen. Sind also Funktionalitäten weiterer Klassen nötig, die keinen direkten Rückgabewert und eine Weiterverarbeitung erfordern, so können diese in einem neuen Thread gestartet werden und der aufrufende Thread wird beendet.

Diese Optimierungen können bei allen JAVA-Anwendungen auf allen Systemen angewendet werden.

2 Speichereffizientes API-Konzept

In den meisten Anwendungsfällen wird bei eingebetteten Systemen nur ein Bruchteil der Funktionälität des Original-*JDK* benötigt. Daher kann der Speicherbedarf deutlich reduziert werden, wenn der Umfang und die Komplexität des APIs gegenüber dem *JDK* reduziert wird. Das größte Optimierungspotenzial besteht aber dann, wenn das API komplett neu strukturiert wird. Anstatt eine große vertikale Vererbungshierarchie aufzubauen, kann auf das zweite bei JAVA verfügbare Konzept zur Vererbung gesetzt werden, die horizontal arbeitenden *Interfaces*. Diese sind gegenüber gewöhnlichen Klassen eingeschränkt, sie enthalten nur Konstanten und abstrakte Methodendeklarationen jedoch keinen Code und keine Daten. Die Kenntnis dieser Einschränkungen kann eine JAVAVM ausnutzen und muss die Interfaces bei einer Referenz nicht in den Speicher laden. Und da ein JAVA-Compiler in der Regel die primitiven Konstanten in die implementierenden Klassen direkt übernimmt und die implementierenden Klassen die Interfaces vollständig enthalten kann auf die Interfaces im Festwertspeicher des Systems verzichtet werden (siehe Abbildung 3).

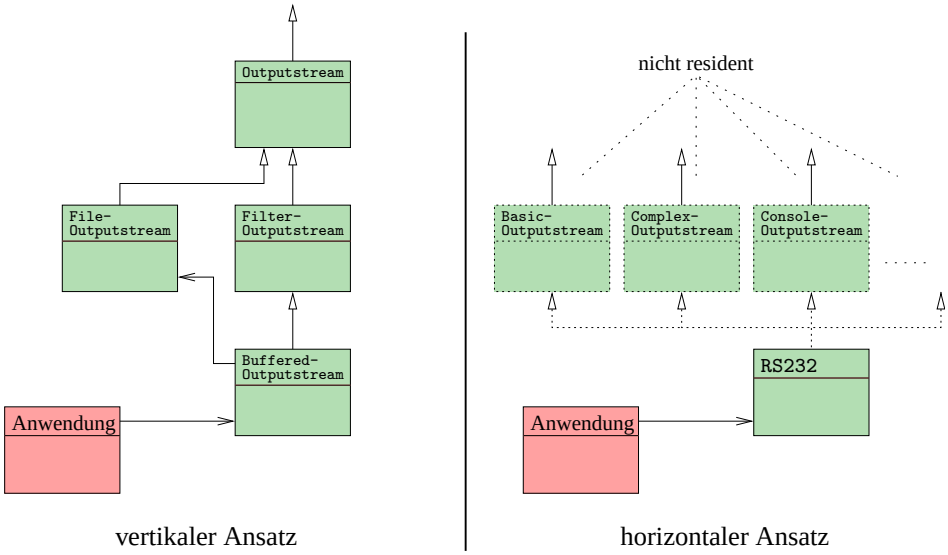


Abbildung 3: Vertikale vs. horizontale Vererbung

2.1 Beispiel I/O-Streams

Hier ist ein Ansatz für I/O-Streams, die vollständig als Interfaces definiert sind:

```
BasicInputStream    BasicOutputStream
ComplexInputStream  ComplexOutputStream
ConsoleInputStream  ConsoleOutputStream
```

Dabei existiert eine klare Trennung zwischen lesenden und schreibenden Streams und eine Gliederung bzgl. der Komplexität der Kommunikation. `BasicI/OStream` verfügt dabei lediglich um einfachste Fähigkeiten zum Übertragen von Einzelzeichen und Byte-Arrays, `ComplexI/OStream` ermöglicht die Übertragung von JAVA-Datentypen (z. B. `Strings`) und `ConsoleI/OStream` schließlich die formatierte Übertragung mittels `println()` bzw. `readLine()`. Die I/O-Klassen des APIs implementieren nun nur die benötigten Interfaces entsprechend der eigenen Fähigkeiten, z. B. :

```
public class CAN implements
  BasicInputStream, BasicOutputStream,
  InputMessage, Runnable;
public class DisplayConsole implements
  ConsoleOutputStream;
public class ROMresource implements
  BasicInputStream, ConsoleInputStream, File;
public class RS232 implements
  BasicInputStream, BasicOutputStream,
  ComplexInputStream, ComplexOutputStream,
  ConsoleInputStream, ConsoleOutputStream;
```

Es fällt dabei auf, dass die Klassen `ROMresource` und `DisplayConsole` auf Nur-lese- bzw. Nur-schreib-Geräte zugreifen und daher auch nur die Hälfte der Interfaces implementieren müssen. Die Klassen `ROMresource` und `CAN` implementieren beide `BasicInputStream`, dass dahinter eine unterschiedliche Komplexität steht, bleibt genauso gekapselt wie bei abgeleiteten Stream-Klassen. Beim ersten muss nur ein Speicherbereich umkopiert werden, beim zweiten findet dagegen eine protokollierte Kommunikation zwischen den beiden Partnern statt, da der CAN-Bus paketorientiert arbeitet und nicht direkt für einen Datenstrom vorgesehen ist.

Hierbei wird auch ein Nachteil der Interface-Lösung deutlich. Da Interfaces keinen Code enthalten, muss jede Klasse den Code selbst zur Verfügung stellen und dies erhöht scheinbar den Speicherplatzbedarf. Allerdings betrifft dies nur den Festwertspeicher, da die Anzahl von Objekten und Referenzen gleich bleibt, die Zahl der residenten Klassen verringert sich jedoch, da Interfaces nicht zwingend geladen werden müssen. Und auch im Festwertspeicher sind Optimierungen möglich, da ein Teil des APIs mit *Native*-Code arbeitet und hiermit Querverbindungen zwischen den Klassen aufgebaut werden können, die mit Interfaces alleine nicht möglich wären.

3 Speicherplatzsparende JAVA-Laufzeitumgebung

Die Hauptlast eines speicherplatzsparenden Umgangs mit den JAVA-APIs und -Anwendungen liegt natürlich bei der JAVA-Laufzeitumgebung selbst. Die obigen Techniken haben allesamt zum Ziel, die Anzahl den verwendeten Klassen zu reduzieren, der JAVAVM obliegt es hingegen, den flüchtigen Speicherplatz pro verwendeter Klasse zu verringern. Zunächst skalieren Optimierungen wie z. B. die Anpassung der Wortbreite (gewöhnlich 32 bit) an vorhandene 16 bit-Adressbereiche den Speicherbedarf insgesamt.[3] Um den Speicherplatz im flüchtigen Speicher wie im Festwertspeicher weiter zu verringern, sind jedoch aufwändigere Techniken nötig. Wie schon in der Einleitung erwähnt, liegen bei diesen eingebetteten Systemen die Klassen im Festwertspeicher, der Zugriff darauf erfolgt direkt, somit scheidet dort die übliche Entropiekodierung aus, denn für einen Zugriff müsste eine entpackte Kopie erzeugt werden. Stattdessen wird das Dateiformat der Klassen modifiziert, lediglich der Bytecode selbst bleibt unverändert. Um zu verstehen, warum dies auch und vor allem den Verbrauch flüchtigen Speichers reduziert, ist die Kenntnis über den Umgang einer JAVAVM mit den Klassen zur Laufzeit nötig.

[3] Über diese Tatsache ist der JAVA-Programmierer zu informieren, da sich dann einige primitive Datentypen ändern, `int` ist dann beispielsweise auch nur 16 bit breit und verhält sich genauso wie `short`.

3.1 Laufzeitrepräsentationen

Beim Zugriff auf eine Klasse legt eine JAVAVM Zugriffstabellen auf deren Elemente (Variablen, Methoden, Konstanten) an [5, 6]. Die Variablen sind dabei nicht nur die Datenfelder, die den Inhalt (den Wert) enthalten, sondern auch Tabellen auf die symbolische Information, damit diese über ihren Namen aufgefunden werden können. Für Methoden gilt dasselbe, zusätzlich ist bei ihnen eine Referenztabelle nötig, die nicht nur die in einer Klasse deklarierten Klassen enthält, sondern auch die von der Oberklasse ererbten. Alle Referenzen erfolgen bei JAVA durch Einträge im sog. Konstantenpool einer Klasse und zwar durch die Nummer eines Eintrags, daher muss es hierfür auch eine Tabelle geben, die *CPR* (constant pool representation). Sie enthält Zeiger in die jeweilige Klasse (ggf. werden hier aber auch schon Werte, z. B. Konstanten, direkt vermerkt). Da diese Tabellen erst zur Laufzeit erzeugt werden (dynamisches Linken), müssen sie im flüchtigen Speicher abgelegt werden. Aus diesem Umstand ergibt es sich, dass die residenten Klassen ein Vielfaches flüchtigen Speichers belegen, verglichen mit ihren Instanzen, die in der Regel nur wenige Attribute speichern müssen. Wenigen dutzend Bytes pro Instanz stehen dabei mehrere hundert Bytes für eine Klassenrepräsenation gegenüber.

Der Ausweg aus diesem Dilemma ist, diese Tabellen nicht von der JAVAVM zur Laufzeit zu erzeugen, sondern vorab und im Festwertspeicher abzulegen, die Klassen werden *vorverlinkt*. Redundante und irrelevante Informationen in den Klassen selbst können dann entfernt werden, das wiegt dann den Platzbedarf der Laufzeittabellen auf. Dieses Verfahren wurde bereits bei der JAVACard [4] umgesetzt, dabei wurde allerdings auch komplett auf die symbolische Information verzichtet. Bei eingebetteten Systemen, die später um weitere Klassen ergänzt werden sollen, müssen hingegen einige Symbole erhalten bleiben, damit die Referenzen von nicht vorverlinkten Klassen aufgelöst werden können. Daher existiert auch bei vorverlinkten Klassen ein kleiner Repräsentant einer jeden residenten Klasse im flüchtigen Speicher, der auf die vorverlinkten Tabellen verweist. Diese enthalten auch noch den Teil der symbolischen Informationen, die für externe Klassen relevant sind. Stösst das System auf eine Klasse ohne vorverlinkte Tabellen, wird ein Repräsentant erzeugt, der nicht auf die Tabellen verweist, sondern diese enthält (und ist dann natürlich entsprechend größer). Bei einem Zugriff der JAVAVM auf eine Klasse verhält sich dieser Unterschied allerdings transparent.

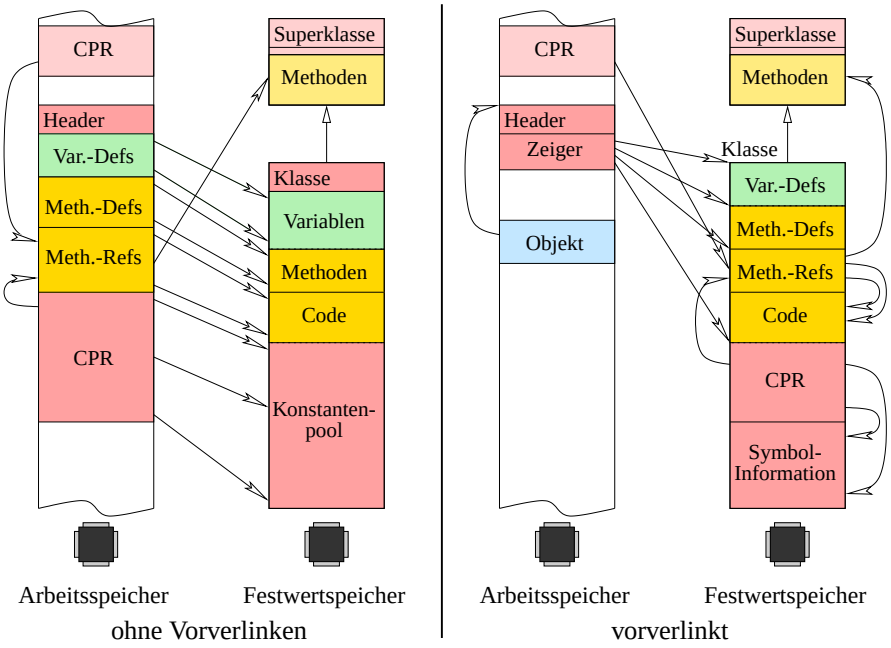


Abbildung 4: Die Repräsentation von Klassen im Heap

Abbildung 4 stellt die beiden Möglichkeiten gegenüber. Zu beachten ist, dass auch klassenübergreifende

Verweise existieren, die erst zur Laufzeit ermittelt werden können. Bei der vorverlinken Variante auf der rechten Seite ist die Klasse ihres Dateiformats beraubt, lediglich ein Teil des Konstantenpools, der die symbolische Information enthält und der Code der Methoden blieben übrig. Zu sehen ist hier auch ein Querverweis von einer anderen – nicht vorverlinkten – Klasse, deren CPR zur Laufzeit der JAVAVM ermittelt wurde, dies mittels des Rests symbolischer Information.

3.2 Automatisiertes Vorverlinken bei *JControl*

Das Vorverlinken läuft auf einem Host-Rechner ab, gewöhnlich auf demselben, der als Entwicklungsplattform für die eingebetteten JAVA-Anwendungen dient. Ein in JAVA realisiertes Werkzeug (siehe Abbildung 5) fasst die Klassen einer Anwendung zusammmen, verlinkt diese vor und kümmert sich auch um die Programmierung der Anwendung in ein *JControl*-Modul. Strukturiert werden die Anwendungen dabei in Archiven (vergleichbar mit JAR-Archiven, die beim *JDK* zum Einsatz kommen), von denen mehrere im wiederprogammierbaren Festwertspeicher verwaltet und ausgewählt werden können. Vom Vorverlinken bemerkt der Anwender dabei nichts.

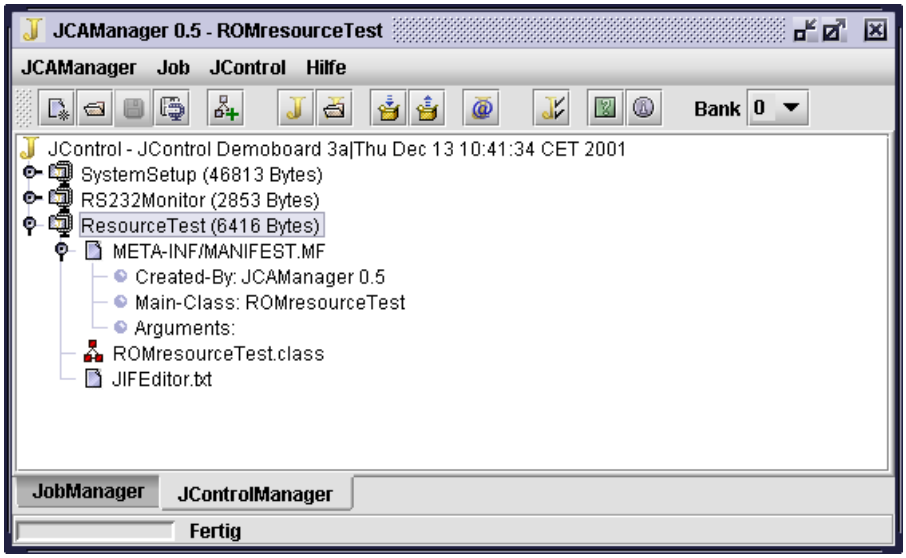


Abbildung 5: Der *JControl*-Archive-Manager bei der Arbeit

4 Fazit

Soll JAVA als Programmiersprache für eingebettete Low-End-Systeme genutzt werden, so gewinnen Techniken für die Einsparung von Speicherplatz eine besondere Bedeutung. Bereits bei der Anwendungsentwicklung lässt sich durch geschickte Programmiertechniken eine Menge Speicherplatz einsparen, ohne dass dabei an der Anwendung selbst gespart werden muss. Aber auch bei der Implementierung der JAVAVM selbst kann durch kompakte Laufzeitrepräsentationen zusätzlicher Speicherplatz eingespart werden. Ein speichereffizienter API-Ansatz setzt diese Techniken fort und erschöpft weiteres Optimierungspotenzial durch horizontale Vererbungen. Dass die Änderung der APIs dabei zu Inkompatibilitäten mit den bekannten JAVA-Implementationen führt, spielt für viele Anwendungen im Embedded-Bereich eine untergeordnete Rolle – viel wichtiger ist, dass die system-spezifischen Funktionen, wie z. B. Ein- und Ausgabeports, Schnittstellen etc., über APIs programmiert werden können.

Alle hier vorgestellten JAVAVM-Techniken wie auch die speichereffiziente API-Bibliothek sind Bestandteil des *JControl*-Systems, einer speziellen JAVA-Realisierung für eingebettete Low-End-Systeme. Bei

JControl können zur Anwendungsentwicklung die gewöhnlichen JAVA-Entwicklungswerkzeuge genutzt werden (IDEs, Compiler, etc.). Spezielle Tools unterstützen dann die Nachbearbeitung der Klassen (z.B. Archivierung und Vorverlinkung). Auf diese Weise lassen sich die Vorteile der Programmiersprache JAVA auch für minimalistische Systeme nutzen.

Literatur

[1] *Böhme, Helge; Telkamp, Gerrit*; Embedded Control mit JAVA; *Embedded Intelligence 2001*

[2] JControl-Homepage:
http://www.jcontrol.org

[3] *Sun Microsystems*; JAVA™ 2 SDK Documentation
http://java.sun.com/j2se/1.4/docs/

[4] *Sun Microsystems*; JAVACard™ Technology
http://java.sun.com/products/javacard/

[5] *Lindholm, Tim; Yellin, Frank*; JAVA™ Die Spezifikation der virtuellen Maschine, Die offizielle Dokumentation von JAVASOFT; *Addison-Wesley*, 1997; ISBN 3-8273-1045-8
http://java.sun.com/docs/books/vmspec

[6] *Venners, Bill*; Inside the JAVA2 Virtual Machine; 2nd edition, *McGraw-Hill*, 1999; ISBN 0-07-135093-4
http://www.artima.com/insidejvm/blurb.html