Inhaltsverzeichnis

1 Unterschiede zu J2SE

Das Ziel einer JVM nach CLDC−Spezifikation ist es, trotz gegebenen Einschränkungen, soweit wie möglich kompatibel zur "Java Language Specification" und "Java Virtual Machine Specification" zu sein. Die JVM von der J2SE−Plattform unterstützt die JLS vollständig. Es folgen nun die Unterschiede zwischen einer JVM nach CLDC−Spezifikation und der J2SE−JVM

1.1 Keine Fliesskommaunterstützung

Der grösste Unterschied der CLDC−Spezifikation zur Standardspezifikation ist das Fehlen der Fliesskommaunterstützung. Dies wurde weggelassen, da die meisten Zielgeräte selber keine Hardware Unterstützung dafür implementiert haben. Das ganze müsste dann per Software implementiert werden, was aber den eng gesteckten Rahmen des Speicherbedarfs der VM sprengen würde.

Daher sind keine Dezimalzahlen erlaubt, es gibt keine Fliesskommatypen und Variablen (kein "float" oder "double") und es gibt keine Fliesskommaoperationen.

1.2 Keine finalize()−Methode

In den CLDC−Bibliotheken existiert Object.finalize() nicht. Diese Methode wird normalerweise, falls vorhanden, aufgerufen, bevor das Objekt aus dem Speicher entfernt wird, z.B. bei der Garbage−Collection.

1.3 Einschränkungen bei der Fehlerbehandlung

Eine CLDC−JVM unterstützt generell die Ausnahmebehandlung (exception), wie in der Java Language Specification (Kap. 11) beschrieben. Hingegen ist die Fehlerbehandlung stark eingeschränkt, da nur wenige Fehlerklassen vorhanden sind. Bei der Ausnahmebehandlung geht man davon aus, dass so ein Ausnahmefehler abgefangen wird und behandelt werden kann.

Hingegen nimmt man bei einem Fehler an, dass das Problem nicht trivial zu lösen ist. Die Lösbarkeit ist zudem stark Geräteabhängig, und bei kleinen, mobilen oder embedded Geräten ist anzunehmen, dass bei einem Fehler einfach ein Softreset ausgeführt wird, anstatt sich genauer um den Fehler zu kümmern. Wollte man sich trotzdem um alle Fehlerfälle kümmern, ergäbe dies wiederum einen enormen Overhead, was sich in mehr Speicherbedarf niederschlägt.

1.4 Fehlende Features der JVM

Einige Eigenschaften wurden von der CLDC−JVM entfernt, da einerseits die CLDC−Javabibliotheken stark eingeschränkt sind, und andererseits würden einige Eigenschaften ein Sicherheitsproblem ergeben, da das Sicherheitsmodell der Standard Java Plattform fehlt.

1.4.1 Kein Java Native Interface (JNI)

Das JNI wurde hauptsächlich wegen 2 Gründen nicht implementiert: 1. würde JNI wieder einmal zu speicherintensiv ausfallen, und 2. diktiert das "limited security model" von CLDC, dass die Menge der native Funktionen nicht erweitert werden darf, wegen dem Sandkastenprinzip.

1.4.2 Keine benutzerdefinierte Klassenlader (class loaders)

Auch diese Art, eine Klasse zu laden, wird vom limited security model verboten, um das Sandkastenprinzip zu wahren. Es muss strikt der vorgegebene Klassenlader verwendet werden. Sowohl die Klassenladerimplementierung als auch die Fehlerbehandlung während dem Laden einer Klasse sind stark geräteabhängig.

1.4.3 Keine Reflection

Eine JVM nach CLDC unterstützt keine reflection Eigenschaften. Reflection wird dazu gebraucht, dass ein Javaprogramm Laufzeiteigenschaften innerhalb der JVM herausfinden kann, wie z.B. Anzahl und Eigenschaften von Klassen, Objekten, Methoden, usw. Das heisst dann aber auch, dass Funktionen, welche abhängig von Reflection sind, nicht vorhanden sind. Das sind z.B. RMI, Serialisierung von Objekten, JVMDI (Debugging Interface), JVMPI (Profiler Interface).

1.4.4 Keine Thread Gruppen

Multithreading ist zwar vorhanden, aber Threads können nur individuell gestartet und angehalten werden. Möchte der Programmierer trotzdem eine Gruppe von Threads gleichzeitig starten, muss er ein Collection Objekt zur Speicherung der Thread−objekte benutzen.

1.5 Überprüfung der Klassendateien

Wie auch bei der Standard JVM, muss auch die CLDC−JVM in der Lage sein, ungültige Klassendateien zu erkennen und zu verwerfen. Diese Dateiüberprüfung würde aber wieder zu viel Speicher beanspruchen (bei J2SE sind das ca. 50 kB Bytecode, während der Überprüfung werden zusätzlich 30−100kB benötigt). Daher wurde für dieses Problem ein anderes Verfahren entwickelt.

1.5.1 Vorüberprüfung und Laufzeitüberprüfung

Die Vorüberprüfung (preverification) findet nicht auf dem Gerät statt, sondern ausserhalb, z.B. vom Server, von wo das Programm auf das Gerät geladen wird, oder sogar schon während der Entwicklung des Programmes. Bei dieser Vorüberprüfung werden in diese Klassendateien zusätzliche Attribute hinzugefügt, welche dadurch um ca. 5 % grösser werden. Durch diese Vorarbeit kann der Haupt−Überprüfer in der CLDC−JVM viel geringer ausfallen. Bei der KVM ist der Bytecode nur noch 10 kB gross, und zur Überprüfung werden weniger als 100 Bytes benötigt. Der Überprüfer muss nun nicht mehr iterative Algorithmen über den Bytecode ausführen, sonder es ist nur noch ein linearer Scan nötig.

Soll das Programm auf der standard JVM ausgeführt werden, werden vom standard Klassendateiüberprüfer diese zusätzlichen Attribute einfach ignoriert, und somit bleibt die Aufwärtskompitibilität erhalten.

1.6 Format und Laden der Klassendateien

Eine wichtige Anforderung an CLDC ist die Fähigkeit, Klassen und Inhalte dynamisch herunterladen zu können.

1.6.1 Unterstützte Dateiformate

Die Standard Klassendateien, erweitert mit den Attributen des Vorüberprüfers, müssen von der CLDC−JVM gelesen werden können. Zusätzlich zur Standard−JVM muss eine CLDC−Implementation auch komprimierte Java Archive Files (JAR) verarbeiten können. Bei der Datenübertragung über langsame Netzwerke ergibt sich natürlich ein Geschwindigkeitsvorteil, wenn komprimierte JARs verwendet werden können (30−50%). Sind die Programme öffentlich zugänglich (z.B. via Internet), dann muss das Programm zwangsweise als JAR veröffentlicht werden.

1.6.2 Suchreihenfolge von Klassendateien

Die 2 Spezifikationen Java Language Specification und JVM Specification definieren nicht, in welcher Reihenfolge nach Klassendateien, welche neu geladen werden müssen, gesucht werden soll. Meistens wird die Suchreihenfolge der Umgebungsvariable "classpath" entnommen, das ist aber von der jeweiligen Implementierung der JVM abhängig.

Die CLDC−Spezifikation geht davon aus, dass die Definition der Suchreihenfolge auf der Implementierungsebene der JVM stattfindet, sie ist teil des "application management" des Gerätes. Die Umgebungsvariable "classpath" kann, muss aber unterstützt werden.

Für die Suchreihenfolge gibt es 2 Einschränkungen: die JVM nach CLDC muss sicherstellen, dass der Anwedungsprogrammierer die Systemklassen nicht umgehen kann, und dass er keine Möglichkeit haben darf, die Suchreihenfolge zu verändern. Beide Einschränkungen sind wegen dem vereinfachten Sicherheitsmodel notwendig.

1.6.3 Optimierung bei der JVM−Implementierung

Die CLDC−Spezifikation erfordert, dass Java−Anwendungen, welche via Netz öffentlich zugänglich sind, im JAR−Format bereit gestellt werden müssen.

In geschlossenen, nicht öffentlichen Netzen, können andere Formate benutzt werden, z.B. stärker komprimierte Formate, welche über schmalbandige Netze schneller übertragen werden können, oder im Gerät mit geringerem Speicherbedarf gespeichert werden können.

1.6.4 Preloading / Prelinking

Eine CLDC−JVM darf Klassen "preloaden / prelinken", typischerweise wird eine JVM für Kleinstgeräte Systemklassen (d.h. alle Klassen der Konfiguration / des Profiles) laden, und nur die Klassen der Anwendungsprogramme während der Laufzeit dynamisch hinzuladen. Der Lademechanismus ist wieder stark Implementierungsabhängig. Auf jeden Fall muss aber die Ausführung des Programmes genau gleich aussehen, als wenn kein Preloading stattgefunden hätte, der einzige erkennbare Unterschied ist somit ein schnellerer Programmstart, da die Standardklassen bereits geladen sind.

1.6.5 Zukünftige Klassendateiformate

Die uns bekannten Klassendateien sind nicht für die Übertragung via schmalbandige Netze

ausgelegt, denn jede Klassendatei ist eine unabhängige Einheit, welche ihre eigene Symboltabelle, Methoden, Felder, Ausnahmetabellen, Bytecode und andere Informationen hat. Dadurch kann erreicht werden, dass eine Applikation aus verschiedenen Einheiten bestehen kann, welche an verschiedenen Orten verteilt sein können, Programme können so während der Laufzeit dynamisch erweitert werden. Diese Flexibilität hat aber seinen Preis: wüsste man, dass eine Applikation eine Einheit wäre, könnte man einiges an Redundanzen einsparen.

Des weiteren wäre es wünschenswert, wenn die wenn die Programme "in place" ausgeführt werden könnten, gerade bei solchen eingeschränkten Geräten wäre es von Vorteil, wenn die Klassendateien nicht zuerst geladen werden müssten. Damit würde der Programmstart beschleunigt, der Speicherbedarf gesenkt, und nicht zuletzt würde auch weniger Strom verbraucht.

2 CLDC Bibliotheken (Libraries)

J2SE und J2EE haben sehr reichhaltige Bibliotheken, zur Entwicklung von Anwendungen für Desktop Computer und Server. Leider benötigen diese Bibliotheken mehrere Megabytes an Speicher zur Ausführung, daher sind diese für kleine, ressourcenbeschränkte Geräte ungeeignet.

Das Hauptziel beim Design der CLDC−Bibliotheken war es, einen kleinstmöglichen, Satz an Bibliotheken zu definieren, welche sinnvoll sind für die Entwicklung von Applikationen und Profilen für möglichst viele unterschiedliche Geräte. Dabei wurden vor allem Bibliotheken integriert, welche Funktionen für die Netzwerkkonnektivität zur Verfügung stellen.

Der grosse Teil der Bibliotheken sind eine Untermenge von den Bibliotheken der grösseren Java Versionen J2SE / J2EE, um die aufwärtskompatibilität und portabilität der Anwendungen zu wahren. Das Problem aber ist, dass man nicht einfach nur eine Untermenge von Bibliotheken übernehmen kann, da viele stark von einander abhängig sind und daher können nicht einfach gewisse Klassen weggelassen werden. Die Schwierigkeiten findet man vor allem in den folgenden Gebieten: security, in/output, Interface Definitionen, Netzwerk und Speicher. Desshalb wurden diese, allen voran die Netzwerk− und IO Bibliotheken, neu implementiert.

Es gibt nun 2 Arten von CLDC−Bibliotheken: solche, welche eine Untermenge der J2SE Bibliotheken sind, und solche, welche J2SE−spezifisch sind. Die J2SE−kompatiblen findet man in den Packages java.lang.*, java.util.* und java.io.*, die CLDC−spezifischen Klassen sind unter javax.microedition.* zu finden (welche aber auch unter J2SE funktionieren).

2.1 CLDC−spezifische Klassen

Ich möchte nun das Generic Connection Framework im Detail vorstellen.

2.1.1 Hintergrund und Motivation

Die J2SE/J2EE bieten vielfältige Bibliotheken für Input/Output Methoden für den Zugriff auf Speicher und Netzwerke. In J2SE gibt es in den Packages java.io.* etwa 60 Klassen + Interfaces, plus 15 Exception Klassen, in den Packages java.net.* hat es weitere 20 Klassen plus 10 Exception Klassen. Zusammen ergibt das eine statische Grösse der Klassendateien von ca. 200 kB, was für die CLDC−Spezifikation eingeutig zu viel ist. Des weiteren sind die Netzwerk−funktionen heute nicht direkt in den mobilen Geräten 1:1 brauchbar, da verschiedenartige Verbindungen neben TCP/IP benötigt werden, wie etwa IR oder Bluetooth.

Die Anforderungen an Netzwerk− und Speicherbibliotheken sind also stark Geräteabhängig, Gerätehersteller, welche mit paketorientierten Netzwerken arbeiten, benötigen datagram−basierte Kommunikationsmechanismen, während für verbindungsorientierte Netze meist stream−basierte Verbindungen benötigen.

Des weiteren besitzen einige Geräte ein gewöhnliches Dateisystem, während andere Devices nur sehr spezifische Mechanismen bieten. Auch das Vorhandensein mehrerer Netzwerk−mechanismen und Protokollen kann für den Applikationsprogrammierer ziemlich verwirrend sein.

2.1.2 The Generic Connection Framework

Die Anforderung an ein speicherfreundliches J2ME System hat zu einer Generalisierung der IO− und Netzwerkklassen geführt. Ziel war es, eine Untermenge der J2SE Klassen zu entwickeln, welche einfach für den Hardwarehersteller implementierbar ist, und genug flexibel ist, um mit verschiedensten Geräten und Kommunikationsprotokollen zusammenzuspielen. Die Idee ist es, verschiedene Abstraktionsebenen für verschiedene Formen der Kommunikation zu bieten.

Die allgemeine Form: alle Verbindungen werden mit der Systemklasse "Connector" erstellt. Es existiert eine ganze Hierarchie von Connectoren, deren Wurzel eben dieser Connector ist. Die Methode akzeptiert ein Argument, die URL, in der Form:
Connector.open("<protocol>:<address>;<parameters>");

Für HTTP sieht das wie gewohnt aus: http://www.mycompany.com, weiter gibt es noch Sockets (socket://192.168.100.111:9000), Kommunikationsschnittstellen (comm:0;baudrate=9600), Datagramme (datagram://123.234.345.456) und Dateien (file:/readme.txt).

Der grosse Vorteil ist, dass der Code praktisch gleich bleiben kann, egal über welchen Kanal etwas ein− oder ausgegeben wird.

Trotzdem braucht es verschiedene Verbindungstypen, da z.B. eine Datei umbenannt oder gelöscht werden kann, was man aber mit einem TCP/IP Socket nicht machen kann. Es gibt daher 6 verschiedene Verbindungstypen:

1. serielles input device,
2. serielles output device,
3. Datagramm−basiertes kommunikationsdevice (z.B. UDP)
4. verbindungsorientiertes device (z.B. TCP)
5. Benachrichtigungsmechanismus (um einen Server über eine gewünschte Client/Server Verbindung zu informieren)
6. Webserververbindung

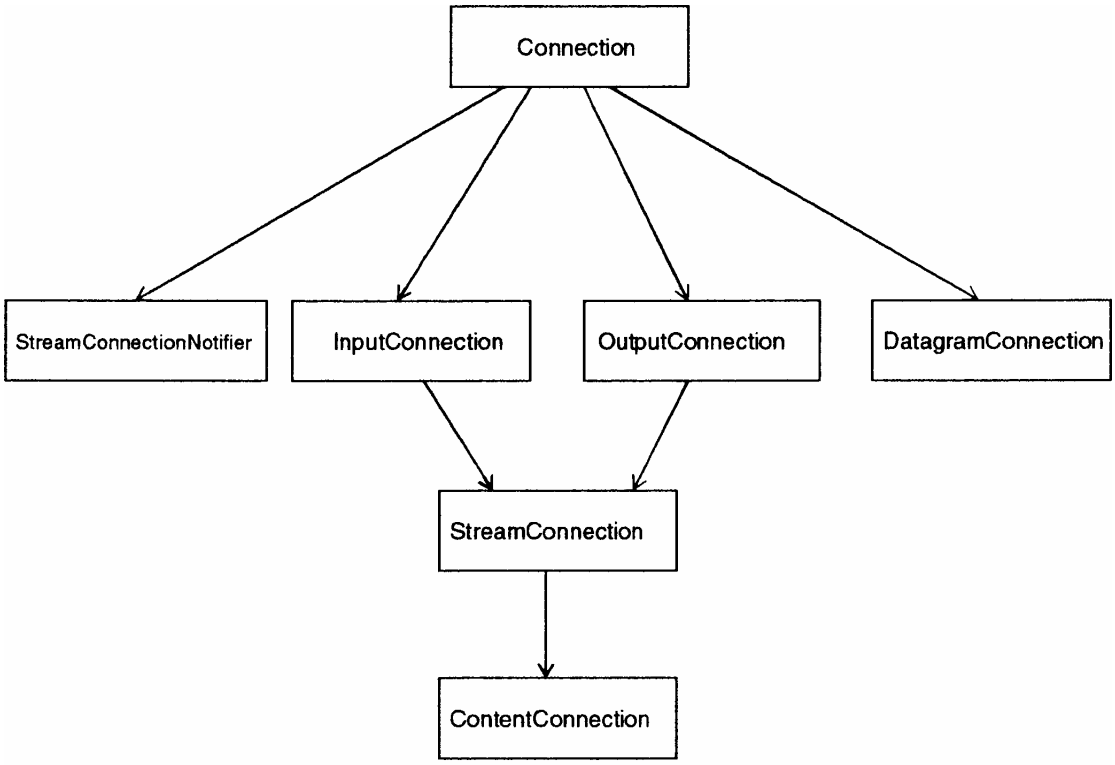

FIGURE 6-1 `Connection` interface hierarchy

2.1.2.1 Interface Connection

Dies ist der einfachste Verbindungstyp, die open Methode ist nicht public, da immer die Klassenmethode open() benutzt wird.

2.1.2.2 Interface InputConnection

Dieser Verbindungstyp repräsentiert ein Gerät, wovon Daten gelesen werden können. Die openInputStream() Methode liefert also einen InputStream zurück.

2.1.2.3 Interface OutputStream

Dieser Verbindungstyp stellt ein Gerät dar, an welches Daten gesendet werden können. die Methode openOutputStream() liefert einen OutputStream zurück.

2.1.2.4 Interface StreamConnection

Dieses Interface verbindet einen Input− und Outputstream, und bildet somit das Fundament für Klassenm welche ein Kommunikationsinterface implementieren wollen.

2.1.2.5 Interface ContentConnection

Dies ist ein Subinterface von StreamConnection, welches einen Streamverbindung definiert, worüber Daten transportiert werden können. 3 Methoden liefern Metainformationen über die zu transportierenden Daten: getType() gibt den Typ des zu übertragenden Inhaltes an, bei http liefert es den "content−type", getLenght() gibt die Grösse der Daten an, und getEncoding() liefert die Codierung.

2.1.2.6 Interface HttpConnection

Dieses Interface erweitert das ContenConnection Interface um viele httpspezifische Methoden.

2.1.2.7 Interface StreamConnectionNotifyer

Dieses Interface wartet auf einen Client, welcher eine StreamConnection aufbauen will. Eine StreamConnection zum Client wird dann mit der acceptAndOpen() Methode erstellt.

2.1.2.8 Interface DatagramConnection

Dieses Interface repräsentiert ein Datagramm−Endgerät. Hier wird von keiner Methode eine URL akzeptiert, da die Zieladressen in den zu sendenden Datagrammpaket selber enthalten sind.

3 Demo

Die folgende Javaapplication, ConnTest.java, funktioniert so, wie sie ist, direkt mit der cvm, der Virtual Machine für CDC mit dem Foundation Profil.

Zum Kompilieren wird J2SE 1.3 wie auch J2ME/CDC benötigt. Es wird vorausgesetzt, dass J2SE standardmässig installiert ist und $JAVAHOME/bin im Pfad vorhanden ist ($JAVAHOME ist das Hauptverzeichnis von J2SE, $CDCHOME das von CDC, $CT ist das Verzeichnis von ConnTest.java). Kompiliert wird mit folgendem Befehl (jedenfalls unter Linux):

```
cd $CT

javac -bootclasspath
$CDCHOME/src/cdcfoundation/build/linux/btclasses.zip:$CDCHOME/src/cdcfoundation/
build/linux/lib/foundation.jar ConnTest.java
```

Zum starten:

```
cd $CDCHOME/bin

./cvm -Djava.class.path=$CT ConnTest
```

es wird 2 Mal die HTML−Datei, welche in der main Methode in der URL angegeben ist, heruntergeladen und ausgegeben. Beim 1. Mal wird mit StreamConnection gearbeitet, beim 2. Mal wird HttpConnection verwendet, und die Verwendung von ein paar Httpmethoden gezeigt.

−−−−−−−ConnTest.java−−−−−−−

```
import javax.microedition.io.*;
import java.io.*;

class ConnTest {

    public static void getViaStreamConnection(String url)
        throws IOException {
        StreamConnection c = null;
        InputStream s = null;
        StringBuffer b = new StringBuffer();
        try {
            c = (StreamConnection)Connector.open(url);
            s = c.openInputStream();
            int ch;
            while((ch = s.read()) != -1) {
                b.append((char) ch);
            }
            System.out.println(b.toString());
        } finally {
            if(s != null) {
                s.close();
            }
            if(c != null) {
                c.close();
            }
        }
    }

    public static void getViaHttpConnection(String url) throws IOException {
        StringBuffer b = new StringBuffer();
        InputStream is = null;
        HttpConnection c = null;
        try {
            long len = 0 ;
            int ch = 0;
            c = (HttpConnection)Connector.open(url);

            c.setRequestProperty("User-Agent",
                                 "Profile/Foundation Configuration/CDC-1.0");

            System.out.println("\nNow let's check out some http header fields:");
            System.out.print("The Content-Type is: ");
            System.out.println(c.getHeaderField("Content-Type"));
            System.out.print("The Date is: ");
            System.out.println(c.getHeaderField("Date"));
            System.out.println("\nNow let's get the html-data:\n");
            is = c.openInputStream();
            len =c.getLength();
            if( len != -1) {
                // Read exactly Content-Length bytes
                for(int i =0 ; i < len ; i++ )
                    if((ch = is.read()) != -1) {
                        b.append((char) ch);
                    }

            } else {
                //Read until the connection is closed.
                while ((ch = is.read()) != -1) {
                    len = is.available() ;
                    b.append((char)ch);
                }
            }
            System.out.println(b.toString());
        } finally {
            is.close();
            c.close();
        }

    }

    public static void main (String args[]) throws Exception {

        String url = "http://tp/index.html";
        //String url = "http://www.ifi.unizh.ch/index.html";
        getViaStreamConnection(url);
        getViaHttpConnection(url);
    }
}
```

Referenzen:
http://java.sun.com/j2me/