Integrating Smart Devices into Java Applications

Marc Jansen
University of Duisburg-Essen
Faculty of Engineering
Institute of Computer Science and Interactive Systems
jansen@collide.info

Contents

1 Introduction

This paper should provide an introduction to a framework for the integration of smart devices into Java applications. The main goal of this framework is to ease the integration task for smart devices of several different kinds. Those devices may be PDA's (Personal Digital Assistants), digital cameras, programmable micro controllers but also devices like telescopes or seismic sensors. To ease the integration task this framework provides several classes and interfaces which divide smart devices basically in two categories: devices we can program and devices we only can exchange data with. This categorization differs from the categorization done by Sun. Sun provides a categorization that divides devices by the amount of memory and computational power they have. This makes sense since Sun is interested in bringing Java to smart devices like cell-phones or PDA's and therefore Sun has to worry about minimal Java subsets that can be run on these devices. Since we have a much brighter variety of devices an approach like this would make no sense for us. For an integration task it is much more natural to categorize the devices by how to access them and the produced data. Additionally this framework provides a factory that allows to build classes for data exchange devices from XML files. This paper will include examples on how to use the framework and how to integrate new devices in Java applications, additionally it will provide an outlook to future extensions of the framework. Furthermore a documentation of the classes and interfaces implemented so far will be presented in this paper and a Java typical documentation based on HTML will be shipped together with this paper.

2 Categorization of Smart Devices

Within this chapter i will discuss two different approaches for the categorization of smart devices. One of the approaches was presented by Sun and is practically implemented in the J2ME (Java 2 Micro Edition)[1]. The second approach is implemented in the framework presented in this paper.

2.1 Categorization by the Amount of Memory and Computational Power

Within the implementation of J2ME Sun had to define the smart devices they want to support with their Java subset for those devices. For their definition they divided smart devices into different groups of devices. This categorization was done by the amount of memory and computational power the device provides. Due to the fact that Sun only wanted to port the Java programming language to smart devices this categorization is reasonable because they had to fiddle the problem of limited capabilities for the user interface, available memory and computational power. Since they do not want to integrate smart devices in an infrastructure where not only smart devices do exist they do not have to worry about the data exchange capabilities of those devices. Beside this, there exist standards like SyncML[2] for the data exchange between different kind of devices.
Sun categorizes smart devices roughly in two different kind of devices, one category really has a short amount of memory and computational power and the other category is a bit richer to wards this properties. They implemented two different, so called configurations, for these categorize: the CDC (Connected Device Configuration) for the richer smart devices and the CLDC (Connected Limited Device Configuration) for the less rich smart devices. Based on these configurations Sun provides different profiles for the devices. The relation between the virtual machine, the configuration and the profile is shown in Figure 1. It is a typical layer architecture where each layer can only communicate with its two neighbors.

2.1.1 The CLDC

The CLDC itself is practically divided into two different implementations:

- The K Virtual Machine (KVM)
- The CLDC Hotspot Implementation

The KVM implementation is for the smallest group of devices that are supported by Sun. To deploy this implementation the device needs at least 40-80KB of RAM depending on the target platform and compilation options. No further restriction is done for the clients.
The CLDC Hotspot Implementation is for devices that have a bit more

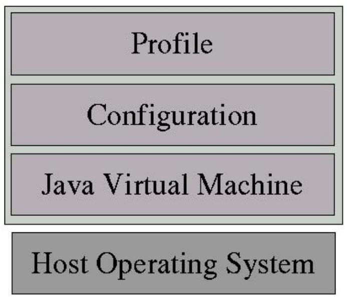

Figure 1: The layer structure of J2ME

memory and computational power but do not fulfill the needs for the CDC configuration. A device must at least have 300KB of RAM, 1MB of ROM and 12MHz of CPU speed.
There exists only one profile that could be deployed with both CLDC implementations: the Mobile Information Device Profile (MIDP). Additional packages that are available for both CLDC implementations are:

- The Mobile Media API
- The Wireless Messaging API
- Bluetooth API

2.1.2 The CDC

Currently the only CDC implementation available is the CDC Hotspot Implementation provided by Sun. This implementation is meant to be deployed on personal mobile devices with a higher amount of memory and computational power than needed for the CLDC implementations. The minimal configuration for a CDC compatible device is:

- 512KB ROM
- 256KB RAM
- The device must be connected to some kind of network.

Based on the CDC there exist three different profiles which differ mostly in the support for Graphical User Interfaces:

- The Foundation Profile
  - most basic profile

  - provides network and I/O support
  - does not provide an GUI or graphics support

- Personal Basis Profile
  - includes the whole Foundation Profile
  - provides Xlet application support
  - provides lightweight GUI support
- Personal Profile
  - includes the whole Personal Basis Profile
  - provides full AWT support
  - provides full Applet support
  - provides limited bean support

At the top of this profiles Sun provides two different additional packages, one for Remote Method Invocation (RMI) and one for Java Database Connectivity (JDBC).

2.2 Categorization by How to Access the Device and the Produced Data

Another way to categorize smart devices is to have a closer look on the possibilities to access the device and the produced data. This leads to two different kind of smart devices: On the one hand we have smart devices we can directly control by programming them, and on the other hand we have smart devices we can only exchange data with. The category of smart devices which we can only exchange data with might itself be split up in two different sections again: the ones we can just get produced data from, and devices we can control and receive the produced data from. Examples of those devices might be:

- Programmable devices
  - Programmable Micro controller - RCX, ...
  - PDA's - Compaq iPaq, ...
  - J2ME Devices - Palm, cell phones, ...
- Data exchange devices we can only receive data from
  - Digital cameras
  - Seismographs
  - Pens with scanners build in

- Data exchange devices we can control and receive data from
  - Telescopes
  - Lego sensors

It is important to mention that the category in which special devices belong might be a subject of change, e.g. are digital cameras today usually devices we only exchange data with, but in the near future this might change since there already exist hybrid devices that combine programmable cell phones with digital cameras.

In the case of data exchange devices we will not divide those devices any more since from an implementation point of view both type of smart devices could be equally dealt with, we just need input, and if we can control them also output, streams to exchange data with those devices.
In the case of data exchange devices we will several different kind of devices still on an abstract level. There might be devices we can exchange data with via:

- Databases
- Network connections
- Java objects
- Processes
- The file system

Most of those devices can easily be used with simple input/output streams, only in case of a device we exchange data with over a database we need a possibility for querying against a database which in our case will most likely be an sql database.
While looking at the devices we can actually program at this point in time we have just two different kind of devices on the next level of abstraction:

- J2ME capable devices
  - CDC capable
  - CLDC capable
- Devices we can program in any other way
  - RCX
  - A Linux equipped iPaq

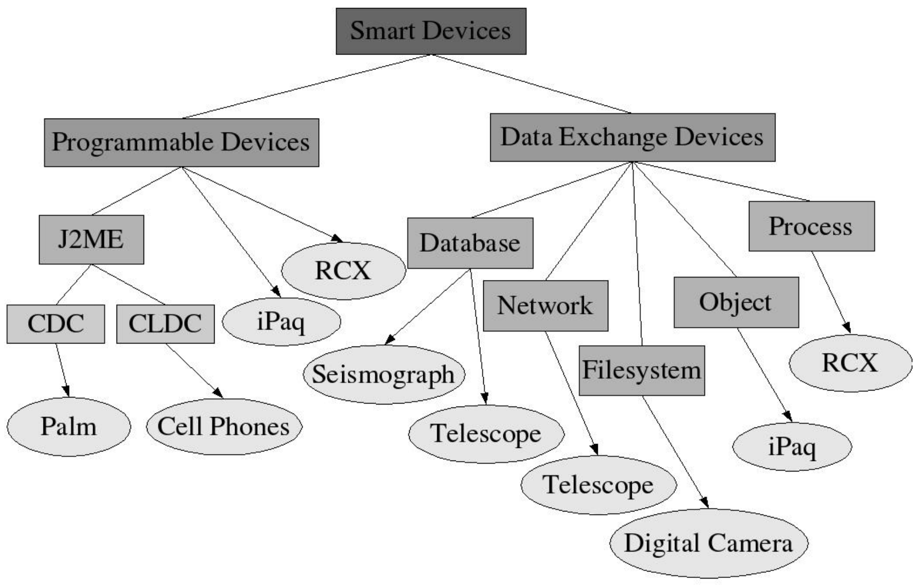

Figure 2: The categorization overview

In both cases of the J2ME capable devices additional packages should be for seen since they would, in the case of use, increase the possibilities of those devices. For devices that we are able to program in another way we might have additional informations like the source code of the program that we want to run on the device, or information about available compilers. The overall situation is shown in Figure 2.

2.3 Discussion of Both Categorizations

Overall the CDC configuration builds a sufficient implementation to deploy Java applications on smart devices. Also the variety of smart devices which fulfill the needs for this configuration is quite rich. Nevertheless both configurations presented by Sun do not solve the problem of integration of devices that do not support Java in any way.
The CLDC implementations provided by Sun builds a good trade off between the low possibilities of those relatively poor devices which are supported and the capability of programming them in Java. For Java developers who want to program applications for these kind of devices it's really helpful that they don't need to learn a new programming language but can stick to patterns they are already familiar with.
What we want to achieve with our framework is to allow an easy integration of smart devices that may not be capable of Java. Therefore the approach of categorization by the way how to access the device and the produced data

is much more natural than the one proposed by Sun.
Nevertheless since on the first view this two approaches for the categorization of smart devices seem to be quite contrary, it is very easy to combine them. This could easily be achieved by implementing a class that represents J2ME capable devices and that encapsulates informations about the configuration, the used profile and additional packages.

3 Java Versions for Smart Devices

This section will provide an overview over different Java implementations that can be deployed on smart devices. Hereby we do not want to provide a complete overview about all existing implementations but discuss those implementations we consider interesting basically when it comes to Java running on an iPaq or an RCX.

3.1 The Java 2 Micro Edition (J2ME)

The J2ME is from an architectural point of view already explained in section 2.1. Within this section we want provide an overview about existing J2ME versions basically for the iPaq, but this will be discussed the section 3.2.1. Other J2ME implementations that are already on the market are one version provided by Sun that is to be deployed on Palm devices running the PalmOS operating system with a version at least 3.5. This one is running with the MIDP profile since it belongs to the CLDC configuration. Additionally most of the currently up to date mobile phones do ship directly with a configured J2ME implementation that is also based on MIDP and therefore on the CLDC configuration. Furthermore no other J2ME implementations are discussed here, beside those that are to be deployed on the iPaq.

3.2 Java for the iPaq

Regarding Java on the iPaq first the question about the operating system running on the iPaq has to be discussed since the existing Java implementations for the different operating systems available for the iPaq do differ a lot. Currently we are aware of three different operating systems for the iPaq:

- PocketPC
- Linux
- SavaJE

The most common one is for sure PocketPC since this is the operating system that is already installed when the iPaq is shipped to the customer. As an alternative we figured out that there exists a Linux distribution that can be deployed on the iPaq. This distribution is named Familiar [1] but is quite tricky to install. The last operating system that is mentioned here is SavaJE [2] which, since it is totally Java based, has currently the best Java support.

[1]http://familiar.handhelds.org
[2]http://www.saveje.org

3.2.1 Java on PocketPC

Using PocketPC as the operating system for the iPaq the developer finds at least two different J2ME implementations which are both based on the CDC configuration. There exists one Personal Profile implementation provided by Sun and one implementation called J9 produced at IBM. Both implementations perform quite well we just had to encounter some incompatibilities when we tried to get RMI (Remote Method Invocation) working with the J9 implementation. Our RMI based applications could not be run on the iPaq using the J9 implementation together with the RMI package for the CDC profiles.

3.2.2 SavaJE

SavaJE is, as already mentioned, a fully Java based operating systems. Therefore it currently provides the best Java support for the iPaq. It supports in its newest version the J2SDK1.4.1 which is almost optimal for a handheld. Our tests with a previous version that only supports the J2SDK1.2.0 were almost sufficient regarding the performance. Since it really supports the full variety of the newest Java implementations also Swing was not a problem when it comes to graphical user interfaces. One problem with the usage of SavaJE is that it has currently no support for the infrared interface of the iPaq and since in a lot of scenarios this infrared port is heavily used this is really a drawback for SavaJE.

3.2.3 Java in Connection with Linux

A good alternative to the usage of SavaJE is to use Linux as the operating system of the iPaq. While using Linux the developer can install a J2SDK1.3.1 implementation provided by Blackdown [3]. Using this implementation we encountered some performance drawbacks while using swing for graphical user interfaces but those could almost be solved by switching to AWT, which is as a matter of fact not the usual way we want to go. Seeing how fast the performance also of handheld devices like the iPaq increases we regard this a minor problem. One big advantage while using Linux on the iPaq is the good support of the infrared interface. It is not only possible to use it with the IRDA protocol but there do also driver exist that allow the usage with proprietary protocols.

3.2.4 Discussion of the Above Mentioned Solutions

The above mentioned possibilities to deploy a Java environment on the iPaq are not meant to provide a complete overview about all possibilities but

[3]http://www.blackdown.org

they do provide an overview about techniques that are almost straight forward and senseful to use. All three of them provide certain advantages and disadvantages so that there is no "real right way", e.g. using PocketPC the developer has the problem that only CDC implementations are available for this operating system and that the usage of the infrared interface is not straight forward. Using SavaJE the usage of the infrared interfaces is almost impossible but on the other hand the developer can use the most up to date Java version. The less drawbacks we encountered using Linux as the operating system since it provides an almost up to date Java implementation and the usage of the infrared port is possible also with proprietary protocols. As a drawback we encountered that the installation of the Linux distribution is really not straight forward and also the usage of the infrared port with proprietary protocols is quite challenging. Nevertheless all the discussed operating systems have a very rich support for wireless LAN cards that are also important in quite a big number of scenarios with smart devices.

3.3 Java on a Programmable Lego Brick

Another smart device that plays an important role in a lot of our scenarios is the programmable Lego brick, the RCX. This device usually ships with a special programming language developed at the MIT Media Lab. While using the RCX in connection with other smart devices that are capable of being controlled from within Java applications it is very useful to program the RCX also in Java. In [5] a lot of ways to control the RCX with Java are provided. Additionally this book provides a first overview about LeJOS [4](Lego Java Operating System) a Java based operating system for the RCX. Regarding the memory opportunities of the RCX (32KB of Ram) it is quite impressive what is possible while using LeJOS. The operating system itself just uses 15KB of Ram so that there are still 17KB free to run Java programs. Of course LeJOS does not provide a full Java support but e.g. the VM (Virtual Machine) does support multi threading.

[4]http://lejos.sourceforge.net

4 The Framework

Within this section we will give a detailed view on the developed framework. The main idea about the categorization of smart devices was already implemented in [3], an older version of the framework. One of the advantages of the new version is a parallel development of classes and interfaces on the level of the categorization. Additionally this version of the framework provides a number of already implemented data exchange devices together with a factory to create the related data handles from an XML description.

4.1 The package hierarchy

The framework currently splits up in four different packages:

- info.coldex.udui.sdevices.core
- info.coldex.udui.sdevices.devices
- info.coldex.udui.sdevices.io
- info.coldex.udui.sdevices.xml

The relationship between the different packages is shown in Figure 3. The

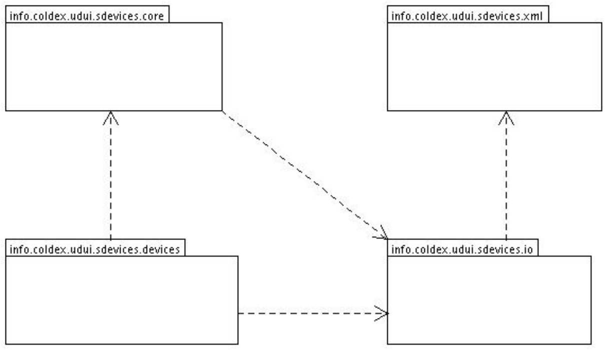
info.coldex.udui.sdevices.core
info.coldex.udui.sdevices.xml
info.coldex.udui.sdevices.devices
info.coldex.udui.sdevices.io

Figure 3: The relationship of the packages in the framework

package core, which basically encapsulates the classes for the parallel development of the interfaces and corresponding classes that reflect the categorization of smart devices, is used by the devices package which capsules the implementation of data exchange devices and programmable devices. The core package itself uses, just like the device package, classes from the io package which basically provides a factory to create data handles from given xml files. The io package itself just uses the xml package which provides interfaces that supports developers in dealing with xml files.

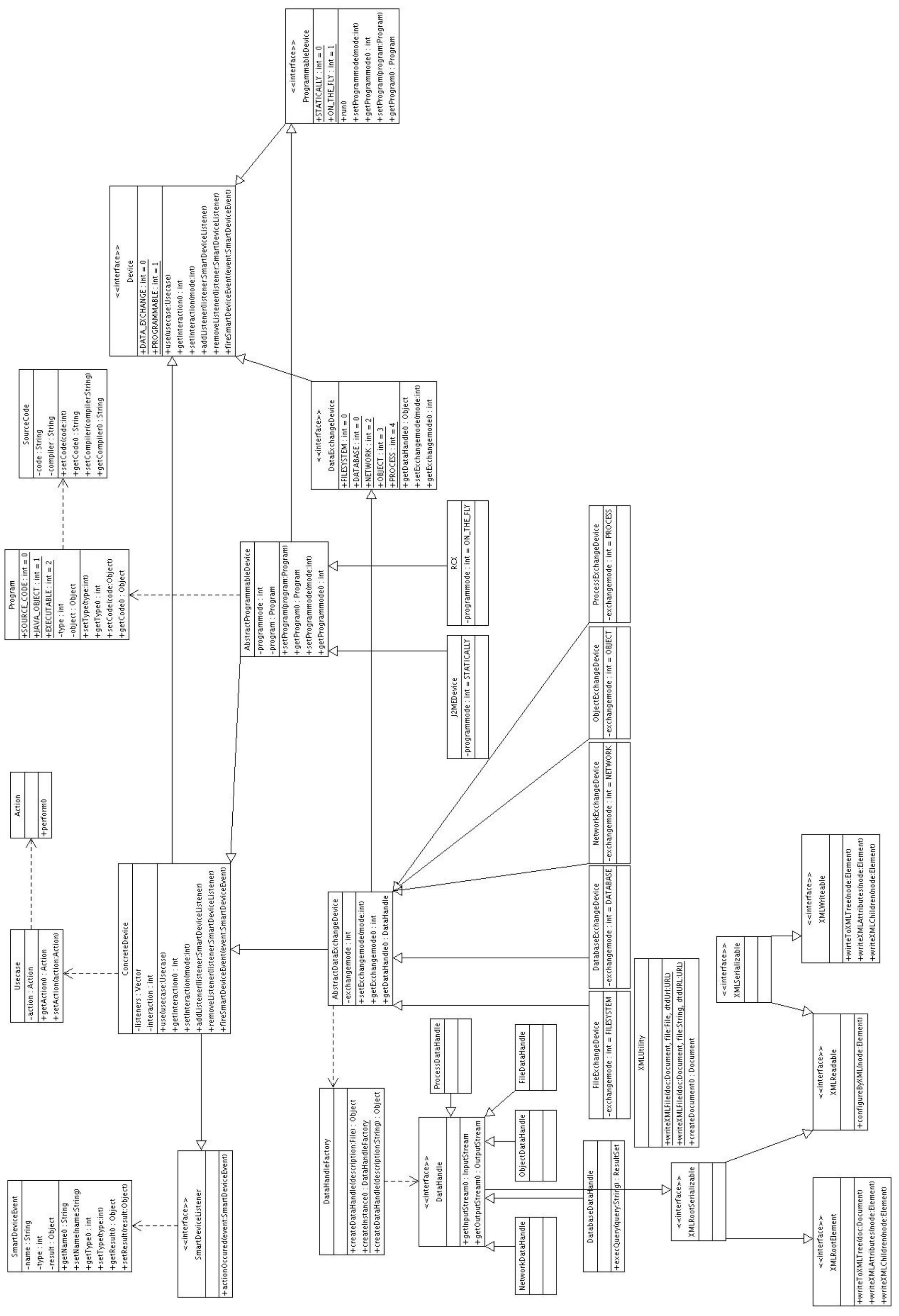

Figure 4: The complete framework as an UML diagram

4.2 The UML description of the framework

Figure 4 provides you with a quick overview about the whole framework, we will present the whole framework here as a UML diagram. In the following subsections we will provide an overview about different sub diagrams.
Within the complete UML diagram the relations between the already implemented classes become obvious, e.g. the parallel development of interfaces and classes that implements these interfaces for different abstraction levels of devices are clearly shown. Further more the relationship between the interfaces for the devices, the implementing classes and the listener for smart devices is explained. The explanation of each sub diagram is done sorted by the encapsulating package.

4.2.1 The core package

This package encapsulates basically the classes needed for the categorization of smart devices and a few additional classes that are already needed on a very high level of abstraction. The UML diagram of this package is shown in Figure 5. Within this figure we can directly see the parallel implementation

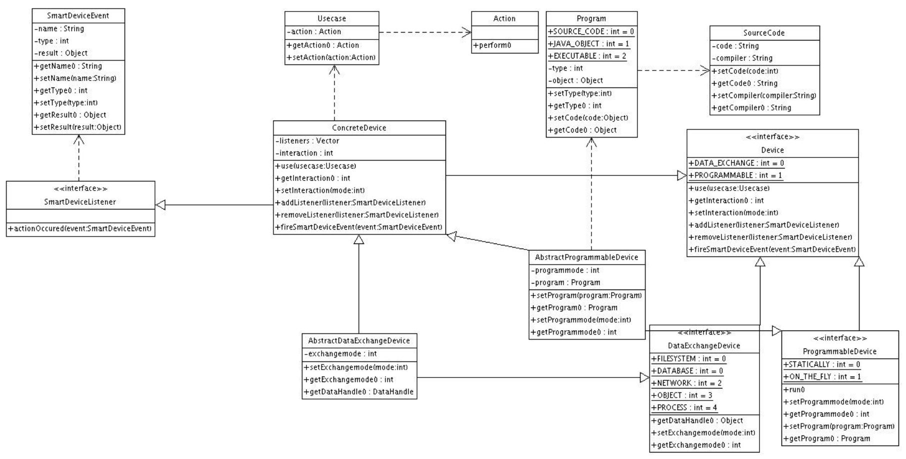

Figure 5: The UML diagram of the core package

of interfaces and the implementing classes. Actually we have three different interfaces with three different corresponding classes.

- interface Device
- interface DataExchangeDevice
- interface ProgrammableDevice

The *ConcreteDevice* class, the one that implements the *Device* interface, is the only class in this context that is implemented concrete and not as an abstract class. The reason that the *AbstractProgrammableDevice* and *AbstractDataExchangeDevice* are abstract classes is that it doesn't make any sense to instantiate them directly, but the developer should be forced to use the corresponding interface or to inherit behavior from the abstract classes. Additionally this package encapsulates the *SmartDeviceListener* and *SmartDeviceEvent* class. Both classes are implemented to allow sending of events if anything in a smart device changes. Using *SmartDeviceListener* interface allows to register listeners for smart device events according to the Java event model. More classes that are implement in this package are the *Usecase* and the *Action* class which are both for the encapsulation of a certain task which should be solved by the device or in which the device should be used. Furthermore is the *Program* and the *SourceCode* class implemented within the core package, both are used to allow the organization of programs, which might be given as source code, of a programmable device.

4.2.2 The devices package

Within this package concrete devices are encapsulated. Those devices extend the *ProgrammableDevice* or the *DataExchangeDevice* class. In Figure 6 the classes of the package are described in UML, both classes from the core package are also in the diagram but specially marked. The reason why they are also in this diagram is just to show the origin of the classes in the devices package. As shown in the UML diagram this package encapsulates seven new

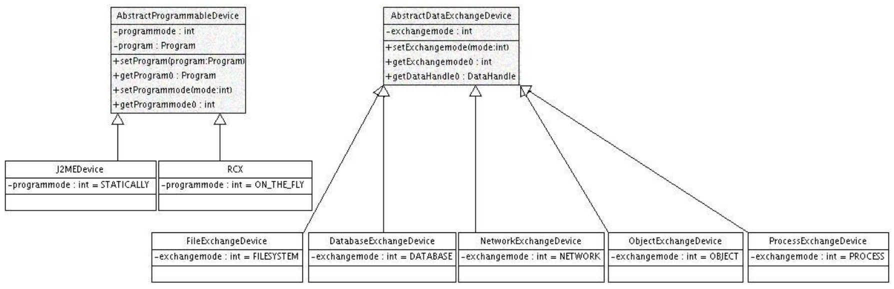
AbstractProgrammableDevice
-programmode : int
-program : Program
+setProgram(program:Program)
+getProgram() : Program
+setProgrammode(mode:int)
+getProgrammode() : int
AbstractDataExchangeDevice
-exchangemode : int
+setExchangemode(mode:int)
+getExchangemode() : int
+getDataHandle() : DataHandle
J2MEDevice
-programmode : int = STATICALLY
RCX
-programmode : int = ON_THE_FLY
FileExchangeDevice
-exchangemode : int = FILESYSTEM
DatabaseExchangeDevice
-exchangemode : int = DATABASE
NetworkExchangeDevice
-exchangemode : int = NETWORK
ObjectExchangeDevice
-exchangemode : int = OBJECT
ProcessExchangeDevice
-exchangemode : int = PROCESS

Figure 6: The UML diagram of the devices package

classes, five that are extended from the *AbstractDataExchangeDevice* class:

- *DatabaseExchangeDevice*
- *NetworkExchangeDevice*
- *FileExchangeDevice*
- *ObjectExchangeDevice*
- *ProcessExchangeDevice*

and currently two classes that extend the *AbstractProgrammableDevice* class:

- *J2MEDevice*
- *RCX*

The devices which extend the *AbstractDataExchangeDevice* class build an almost complete set of classes to ease the task of exchanging data with smart devices. We assume that almost all data exchange could be implemented with the help of these five classes. To make this classes even be easier to use, it is possible with the help of the io package to describe the data exchange capabilities in XML and to create the data handles needed for the data exchange with the help of a factory.
The set of concrete implementations which are extended from the *AbstractProgrammableDevice* class is not at all complete but is meant to create a first impression of how to use these classes. Furthermore those two classes need some improvement since they actually have almost no behavior at all.

4.2.3 The io package

This package encapsulates all needed classes to allow an XML serialization of data handles necessary to describe the io capabilities of data exchange devices. Therefore in this package a factory is developed which allows to read in XML files and which can create the data handle from this description. The used classes in this package are shown in Figure 7. Again in this figure the *AbstractDataExchangeDevice* class is still shown to explain the connections to other packages but marked to make obvious that it belongs to another package. This package defines a new interface *DataHandle* which is implemented by concrete classes that reflect implement the behavior of data handles for the five different kind of data exchange devices that are already implemented in the package devices. Those concrete data handles are:

- *DatabaseDataHandle*
- *NetworkDataHandle*
- *ObjectDataHandle*
- *ProcessDataHandle*
- *FileDataHandle*

All those concrete data handles have to implement to methods which are defined in the interface *DataHandle*:

- *getInputStream*

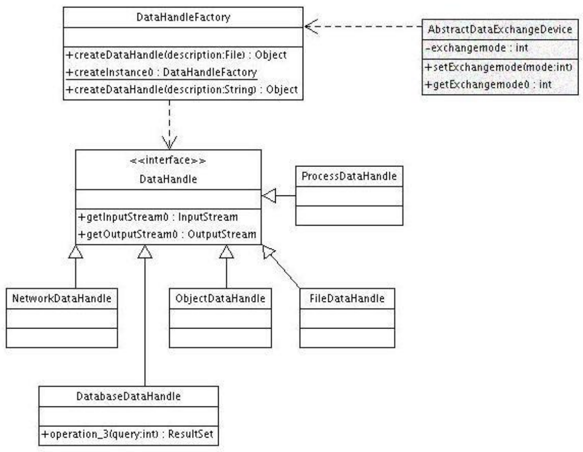
DataHandleFactory
+createDataHandle(description:File) : Object
+createInstance() : DataHandleFactory
+createDataHandle(description:String) : Object
AbstractDataExchangeDevice
-exchangemode : int
+setExchangemode(mode:int)
+getExchangemode() : int
<<interface>>
DataHandle
+getInputStream() : InputStream
+getOutputStream() : OutputStream
ProcessDataHandle
NetworkDataHandle
ObjectDataHandle
FileDataHandle
DatabaseDataHandle
+operation_3(query:int) : ResultSet

Figure 7: The UML diagram of the io package

- *getOutputStream*

Only the *DatabaseDataHandle* class does not use these two methods to provide access to data sets but implements additionally the *execQuery* method that returns a *ResultSet*. Nevertheless this class also implements the *DataHandle* interface but just for the reason of compatibility. This is necessary because the data exchange with databases is in Java done by the usage of the Java internal sql package that does provide an additional abstraction and therefore does not use simple input and output streams.
Additionally this package also provides a factory, regarding the factory design pattern [4], for the automatic instantiation of data handles that is capable of parsing XML files with validated against a given dtd. The dtd which provides the grammar of the XML files readable by the *DataHandleFactory* class is defined by:

```
<!ELEMENT DataHandle EMPTY>
<!ATTLIST DataHandle update CDATA #IMPLIED>

<!ELEMENT FileDataHandle EMPTY>
<!ATTLIST FileDataHandle update      CDATA        #IMPLIED
                         file        CDATA        #REQUIRED
                         isDirectory (true|false) "false">

<!ELEMENT NetworkDataHandle EMPTY>
<!ATTLIST NetworkDataHandle update CDATA #IMPLIED
                            host   CDATA #REQUIRED
                            port   CDATA #REQUIRED>
```

```
<!ELEMENT ObjectDataHandle EMPTY>
<!ATTLIST ObjectDataHandle update CDATA #IMPLIED
                           class  CDATA #REQUIRED>

<!ELEMENT ProcessDataHandle EMPTY>
<!ATTLIST ProcessDataHandle update      CDATA #IMPLIED
                            process     CDATA #REQUIRED
                            environment CDATA #REQUIRED>

<!ELEMENT DatabaseDataHandle EMPTY>
<!ATTLIST DatabaseDataHandle update   CDATA #IMPLIED
                             host     CDATA #REQUIRED
                             user     CDATA #REQUIRED
                             pass     CDATA #REQUIRED
                             driver   CDATA #REQUIRED
                             protocol CDATA #REQUIRED
                             database CDATA #REQUIRED>
```

Overall this package allows easy access to data exchange devices with the help of XML configuration files. The XML support for the framework is basically implemented in a few interface that are encapsulated in the xml package explained in the next section.

4.2.4 The xml package

This package encapsulates a number of interfaces that are needed to provide XML support for the framework presented here. Since Java already has a very good support for the serialization and storage in XML, this package does not provide much functionality but helps the programmer to easily create classes that can them self deal with XML descriptions.
As already mentioned in the previous section, a dtd is defined which can be used to describe the data exchange capabilities of smart devices. This dtd and therefore the data exchange devices are only a first example of how to use the xml package of this framework. The interfaces encapsulated in this package are shown in Figure 8. Again in this figure there is one interface that does not belong to this package, in this case it is the *DataHandle* interface but it is again also put in this figure to explain the context the xml package appears in. Additionally this package provides five interfaces:

- *XMLRootSerializable*
- *XMLSerializable*
- *XMLRootElement*

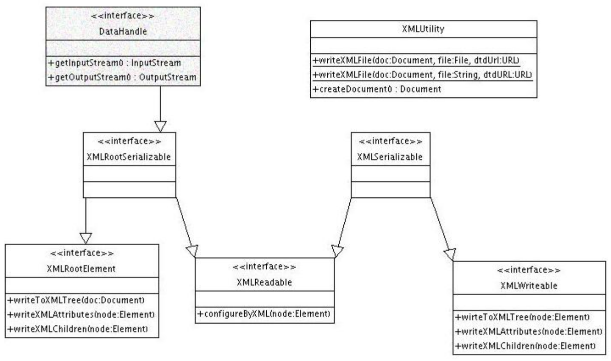
<<interface>>
DataHandle
+getInputStream0 : InputStream
+getOutputStream0 : OutputStream
XMLUtility
+writeXMLFile(doc:Document, file:File, dtdUrl:URL)
+writeXMLFile(doc:Document, file:String, dtdURL:URL)
+createDocument0 : Document
<<interface>>
XMLRootSerializable
<<interface>>
XMLSerializable
<<interface>>
XMLRootElement
+writeToXMLTree(doc:Document)
+writeXMLAttributes(node:Element)
+writeXMLChildren(node:Element)
<<interface>>
XMLReadable
+configureByXML(node:Element)
<<interface>>
XMLWriteable
+wirteToXMLTree(node:Element)
+writeXMLAttributes(node:Element)
+writeXMLChildren(node:Element)

Figure 8: The UML diagram of the xml package

- *XMLReadable*
- *XMLWritable*

Hereby the the interface *XMLRootSerializable* is extends two other interfaces, the *XMLRootElement* and the *XMLReadable* interface. The special need for the *XMLRootSerializable* and the *XMLRootElement* interface is given because a root element of an XML document has to be handled differently than a normal element in the XML tree when writing them out. For writing normal XML elements we have to other interfaces, the *XMLWritable* and the *XMLSerializable* interface. Here also the *XMLSerializable* extends both, the *XMLWritable* and the *XMLReadable* interface.
As an additional class the xml package provides the *XMLUtility* class that provides support in creating new XML documents and in writing XML documents to files.

5 Examples

This section will provide a small example of how to use the framework. Within this example we will collect data from an RCX that is equipped with a light sensor. Hereby the RCX is only a data exchange device, since we assume that the program that sends the light data over the infrared port is already deployed on the RCX. Within this example, shown in Figure 9, we will implement two classes one that is capable of reading data from the RCX and one that enables the database connection. Additionally we need

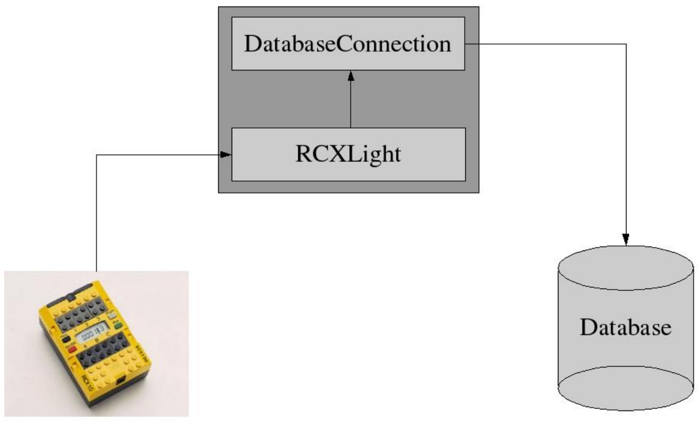

Figure 9: The architecture of our example

two configuration files that can be parsed by the factory that creates the data handles for the data exchange devices. Actually both implemented classes will reflect another data exchange device, the one device should exchange data with the RCX and the other should write the data in a database.

5.1 Example Configuration Files

The configuration files are in XML and can be validated against the dtd explained in section 4.2.3. In this example we need two different configuration files.

5.1.1 The Configuration File for the Device that Exchanges Data with a Database

One configuration file is necessary for the device that should later-on exchange data with the database. Within this file there has to certain informations:

- The host the database is running on.

- A user that has access rights for this database.
- A password for the user.
- The name of the database itself.
- The used protocol.
- A driver to use for the connection.

For this example we used the following XML file which provides all the necessary informations:

```
<?xml version="1.0"?>
<!DOCTYPE DatabaseDataHandle SYSTEM "io.dtd">

<DatabaseDataHandle host="localhost"
                    user="user"
                    pass="password"
                    protocol="jdbc:mysql"
                    database="light_values"
                    driver="org.gjt.mm.mysql.Driver" />
```

At the very beginning we define that the root element of this XML document should be of the *DatabaseDataHandle* type and that this file should be validated against the dtd that is described in the *io.dtd*.
For the data handle assume that the database is running on the *localhost*, that the user name is *user* and the password is *password*. Additionally we selected a database with the name *light_values* and used the standard protocol for a MySQL connection from Java, the *jdbc:mysql* protocol. Last but not least we define that the *org.gjt.mm.mysql.Driver* should be used as the database driver.

5.1.2 The Configuration File for the Devices that Reads Data from the RCX

The second configuration file that we need provides a description for the device that will read the light values from the RCX. In the case of the RCX as a data exchange device we can easily exchange data using an *ObjectExchangeDevice* by using the *RCXPort* class provided by the LeJOS environment. This class actually behaves very much similar to a socket for network connections, therefore it provides an input and an output stream which allows the data exchange with an RCX over its infrared port. Since the only think we have to define for an instance of the *ObjectExchangeDevice* is the class of which we want an object to be created, the XML file for the description of that kind of data handle is quite small:

```
<?xml version="1.0"?>
<!DOCTYPE ObjectDataHandle SYSTEM "io.dtd">

<ObjectDataHandle update="30" class="RCXPort" />
```

Again at the very beginning of this description we define the type of the root element of this XML file, which in this case is an *ObjectDataHandle* and again this file should be validated against the *io.dtd*. Later-on we describe that this data handle object should instantiate an object of the *RCXPort* class. Additionally here we define that this data handle will send new data each 30 seconds. Actually this is just an information that is additionally encapsulated within the data handle object and might be used by the class that uses the data handle instance. It might also be possible to implement the *DataHandle* class that it only sends data after each period of time, but feeling that the actual way is much more flexible it is implemented like this.

5.2 Example Classes

Since most of the work is already done in the XML configuration files explained in section 5.1, the actual work to implement the needed classes is quite small. For this example we at most need to implement two classes.

5.2.1 The Class that Exchanges Data With the RCX

This class actually is very simple, it just extends the *ObjectExchangeDevice* class and passes the name of the XML description, which was already mentioned in section 5.1.2, to the constructor of its super class.

```
    import info.coldex.udui.sdevices.devices.*;

    public class RCXLight extends ObjectExchangeDevice {

        public RCXLight(String desc) {
            super(desc);
        }

    }
```

Actually this class could be skipped easily since it does not implement any behavior and therefore in this case we can just use the *ObjectExchangeDevice* class itself. But for illustrating purposes the class was implemented in this example.

5.2.2 The Class that Exchange Data with the Database

The second class that needs to be implemented simulates a device that is capable of exchanging data with a defined database and therefore we extended this class from the *DatabaseExchangeDevice* class. The XML description is already explained in section 5.1.1. This class actually only needs a constructor and one method for collecting and storing the data in the database. The constructor for this class is fairly simple:

```
public DatabaseConnection() {
    super("db_conf.xml");
    rcx = new RCXLight("ir_conf.xml");
    in = rcx.getDataHandle().getInputStream();
    collectData();
}
```

The first thing we have to do is to call the constructor of the super class with the given XML description for the data handle. After wards we instantiate the *RCXLight* class explained in section 5.2.1 and store the input stream we get via the data handle from this object in a variable of type *InputStream*. The last thing to do in the constructor is to call the method that actually collects and stores the data:

```
private void collectData() {
    DatabaseDataHandle dbDataHandle;
    String query;
    int i;

    dbDataHandle = (DatabaseDataHandle)getDataHandle();
    try {
        while ((i = in.read()) != -1) {
            query = "INSERT INTO entries VALUES
                    ('LIGHT', '" + i + "');");
            dbDataHandle.execQuery(query);
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Within this method the first thing to do is to get a hand on the data handle object provided by this class. Since this class is extended from the *Database-ExchangeDevice* class, we will receive a *DatabaseDataHandle* here. After wards we do a loop that only stops if the RCX does no longer send any data and within this loop we create our SQL (Structured Query Language) query that inserts the data into the entries table of the before defined database. After wards we just have to submit the query to the database with the help of the *execQuery* method of our database data handle. The whole loop has to be put in a try and catch block since the reading from the input stream may throw an input and output exception.

6 Implementation Patterns for Smart Devices

This section will provide an overview over patterns for the usage of smart devices. Since we want to integrate smart devices in Java applications we assume that smart devices have no end in itself but are used in connection with other, probably also smart, devices. Looking at this assumption we found that the usage of smart devices usually follows certain implementation patterns.

6.1 The Direct Communication Pattern

The direct communication pattern is the most simple pattern that could be imagined. It is used to directly communicate with a smart device. This pattern could be seen in our example in section 5.2.1 in the *RCXLight* class. This class is only used to collect the data and is not used any further to distribute the data to any other class.
Usual examples where to use this pattern is e.g. a device is used for data mining and this data is immediately processed on the device itself.

6.2 The Proxy Pattern

The second pattern we want to present here is a bit more complicated as the direct communication pattern from section 6.1. In this case one of the smart devices only works as a proxy to enable communication between two other devices, this might make sense e.g. if these two devices do not have a compatible interface. An example scenario of this is maybe the RCX, the infrared port of the RCX is not IRDA compliant, but it uses some proprietary protocol called cIR. If now some device that only supports IRDA wants to communicate with the RCX it is necessary that some kind of intermediate device, here called a proxy, that supports both protocols is used for the communication.
An example like this is shown in Figure 10 where a Linux equipped iPaq acts as a proxy between the RCX and a PC. Another example for this kind of pattern would be a typical field trip. In this Example the students can collect data from sensors with the help of PDA's and later-on in the classroom they can transfer the collected data to a PC and work with the data their in, e.g. modeling tools. Hereby the PDA just acts as a data collector on the field trip and is, just like in the first example, not used at all to process this data.

6.3 The Advanced-Proxy Pattern

This pattern is quite similar to the proxy pattern from section 6.2 with slight changes in where the data is processed. In the case of a PDA the device itself is on the one hand programmable itself and on the other hand does it provide computational power to some extend. Therefore it is possible to do at least

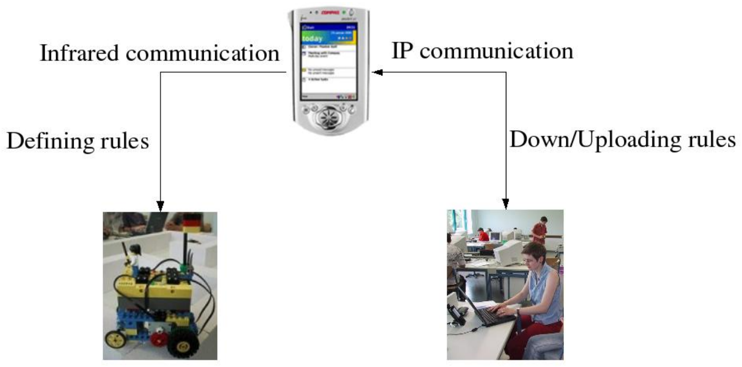

Figure 10: The iPaq as a Proxy for the communication between an RCX and a PC

some easy calculations on the collected data. After wards if any further data processing is necessary that can not be done on the PDA since it does not provide enough computational power the data can be transferred to a usual PC where the necessary tasks could be performed. Another example where this pattern might be used is again a field trip scenario. Here already right on the field trip can some data processing be done and bigger exercises can later-on be done in a laboratory. This allows for example to measure data right on the field trip, process this data and probably directly play this data back to an RCX which might use this already processed data to collect some other information.

7 Outlook and Future Work

This section will provide an overview about the planned work to elaborate the framework presented in this paper. This plant work is basically divided into the different subsections:

- Implementation things to do
- The theoretical background
- The evaluation of the framework

The most of the work is currently done in the first subsection which is the implementation. Nevertheless there is still a lot of work to do and some of the ideas presented in this chapter might be a subject of change. As for the second subsection of this section which deals with the development of the theoretical background of the framework, there is actually not very much already done, only the categorization of smart devices and the comparison with the approach that Sun presents belong to this subsection. The last subsection is almost not touched at all at this point in time, the only thing that can be said is that already at this very early state of the framework, the code that has to be generated by the developer decreases a lot.

This list of future work should not be seen as a complete list of next steps, but should be considered a list of permanent change and additional tasks.

7.1 Implementations

The next step on the implementations side will be a network layer that allows the transparent handling of different network protocols within the framework. At least protocols like:

- HTTP
- RMI
- Sockets
- MatchMaker [6]

should be handled by this layer in a most transparent way. Furthermore the question of so far unhandled protocols should be discussed.
Another thing is to develop a dtd for the description of smart devices in XML. So far, the framework does only support the description of data handles for data exchange devices in XML, but this is only one part of the description of a whole smart device. During the development of this dtd, current standards like SyncML [2] should be regarded as possibilities and

also SOAP (Simple Object Access Protocol) [7] is a standard to mention in this context since it might be a way to describe use cases for smart devices. By having a rich XML description of a smart device, a factory can be implemented that automatically creates instances of the classes that represents those devices. Additionally this factory could register those instantiated representations at a central server, so that the devices are accessible over a network. Hereby Jini [8] should be considered an interesting standard to look at. Furthermore should it be possible to describe a device fully in XML without the need to program a class in Java. This would definitely lead to a much poorer representation compared to a complete Java implementation of this class but it could be one point of research to see how rich those XML descriptions might become.
Last but not least a Cool Modes [9] should be implemented to allow the visual arrangement of different smart devices.

7.2 The Theoretical Background

For the theoretical background of this work the first thing to do will be to discuss the device centered approach in comparison to any other approaches. Since the framework is currently implemented to wards the device centered approach it is an open question whether another approach like a use case centered approach might be reasonable. Additionally the question of alternative approaches to the two already mentioned ones should be discussed.
Currently the framework does only support two major kind of smart devices, on the one hand the programmable devices and on the other hand devices we can only exchange data with. This raises the question whether there exist other type of devices or if those two classes of devices are enough to cover all smart devices on the market.
Additionally the question of patterns occurring during the work with smart devices has to be discussed any further and other patterns have to recognized.
Last but not least a formal description of the framework must be formalized. Therefore a mathematical description of the framework should be done and discussed.

7.3 The Evaluation of the Framework

The evaluation of this framework is still the part that is very badly elaborated so far. Only some ideas of what could be interesting to wards a good software evaluation of the framework are present up to now. Planned are some reference implementations with and without the framework to compare three different things:

- length of code

- quality of code
- time needed for the implementation

There different user groups with explicit different implementations skills should implement the reference implementations with or without the usage of the framework.
Furthermore some example scenarios should be implemented using this framework, both to show the usage of the framework and to give examples of the patterns for the usage of smart devices.

References

[1] http://java.sun.com/j2me/docs/j2me-ds.pdf

[2] http://www.openmobilealliance.org/syncml/download/whitepaper.pdf

[3] Marc Jansen, Kunal Sachdeva, Andreas Harrer: *About a Framework for Integrating Smart Devices in Java Applications*, In: A. Harrer, K. Gassner (eds.), Proceedings of the Workshop on Expressive Media and Intelligent Tools for Learning. German Conference on Artificial Intelligence KI-2003, Hamburg, 2003.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (2000): Design Patterns. ISBN: 0-201-63361-2

[5] Giulio Ferrari, Andy Gombos, Sorem Hilmer, Juergen Stuber, Mick Porter, Jamie Waldinger, Dario Laverde (2002): ”Programming Lego Mindstorms with Java”, Syngress, ISBN: 1-928994-55-5

[6] Jansen, M (2003), MatchMaker - A Framework to Support Collaborative Java Applications, In U. Hoppe, F. Verdejo, J. Kay (eds.): Shaping the Future of Learning through Intelligent Technologies. Proceedings of the 11th Conference on Artificial Intelligence in Education. Amsterdam, IOS Press.

[7] http://www.w3.org/2000/xp/Group/

[8] http://java.sun.com/products/jini/

[9] Pinkwart, N. (2003). A Plug-In Architecture for Graph Based Collaborative Modeling Systems. In U. Hoppe, F. Verdejo, J. Kay (eds.): Shaping the Future of Learning through Intelligent Technologies. Proceedings of the 11th Conference on Artificial Intelligence in Education. Amsterdam, IOS Press.