Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo

Master's Thesis MA-2005-16
Summer Term 2005

Author: Nicole Hatt

Tutor: Matthias Bossardt
Co-Tutors: Vincent Lenders, Dr. Martin May

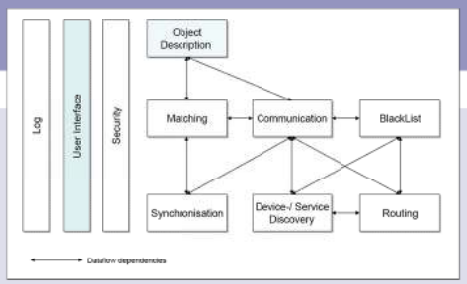
Supervisor:
Prof. Dr. Bernhard Plattner

01.10.2005

BlueFramework

Application Framework for Bluetooth Enabled Mobile Phones

Bluetooth



Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Contents

Figures

Abstract

Semester theses developped lately within the scope of the BlueStar project [1] at the ETH dealt with the development of applications for Java enabled mobile phones that communicate over a Bluetooth connection. An analysis pointed out that those applications share a lot of functionality. The overall goal of this thesis is the design and implementation of a framework which provides common functional blocks in a generic library and development tools to reduce development time and costs of future mobile applications. The main modules include functions for device discovery, user interface generation, object description, matching, communication and logging.
To validate BlueFramework, an example application, BluePinboard, has been successfully implemented and tested on mobile phones. The developed framework has proven to be efficient and time-saving.

1

Introduction

The mobile phone industry has grown explosively over the last few years. Nowadays almost everyone owns a cellular phone. The rapid evolution in mobile computing made it possible that today's embedded devices provide the functionality that was limited to large personal computers years ago. We are currently moving towards a "many computers per user" era where technology will have a large impact on our daily lives. The release of Java 2 Micro Edition brings the advantages of Java to resource constrained devices, such as mobile phones, personal assistants or pagers. The fact that J2ME enables applications to run on different operating systems and on different kind of hardware caused that many mobile phone vendors now support J2ME programs. Because of its portability J2ME has high potential to become the technology for future mobile applications.

The number of wireless Internet devices is increasing and soon Bluetooth will be embedded in every mobile device. Although Bluetooth provides only a fraction of the transmission power of Wireless LAN, it has its advantages in certain applications. Because Bluetooth communication is within short range, it is an interesting technology for location related applications. In additon, it consumes considerable less power and causes lower costs.

Two semester theses developped lately within the scope of the BlueStar [1] project at the ETH dealt with the development of J2ME applications for Bluetooth enabled mobile phones: Bluetella, a file sharing application, and BlueDating, a dating application. Even though the applications are intended for a completely different use, they share a lot of functionality.

The goal of this thesis is to design and implement a framework which comes up with a generic API and development tools to simplify J2ME application program-

ming. To show the framework in use and to validate its functionality, the thesis also includes an example application developped by means of BlueFramework. BluePinboard, a pin board application for Bluetooth enabled mobile phones, has been successfully developed and tested on mobile phones.

The following chapter gives a brief overview over the technologies used for the development of BlueFramework. Chapter 3 takes a closer look at the applications BlueDating and Bluetella and describes the functional blocks they have in common. This analysis forms the basis for defining the required functionalities of BlueFramework. The focus of Chapter 4 of this documentation is on the modular design of BlueFramework. It determines the functionality blocks BlueFramework consists of and describes each of them in detail. The sixth chapter discusses the implementation of the modules introduced in Chapter 5. Chapter 7 describes BluePinboard, the example application developped with the API and the development tools provided by BlueFramework. An evaluation of the the example application is contained in Chapter 8. Closing remarks and a summary of the thesis are contained in the last chapter.

2

Technologies

2.1 Java 2 Micro Edition

The micro edition of the Java 2 platform is an application development environment for devices with limited resources. J2ME includes two Java virtual machines with different configurations. Profiles and optional packages for certain classes of devices extend configurations with additional functionalites..

Configurations

Configurations are a minimal set of class libraries. Currently, there are two J2ME configurations:

- The **Connected Limited Device Configuration(CLDC)** [6] is aimed at devices having limited memory, slow processors and operating on batteries. CLDC defined on top of a kilobyte virtual machine (KVM) which is especially designed for devices like as mobile phones, pagers or personal organizers.
- The **Connected Device Configuration(CDC)** is intended for devices with more memory and faster processors. In contrary to CLDC, this configuration is defined on top of virtual machine that is similar to the Java virtual machine included in J2SE.

As the framework developed in this thesis is intended for mobile devices with limited resources, the CLDC is used in this context.

Profiles

Although configurations offer a J2ME application runtime environment, their functionality is usually not sufficient for a certain kind of device. To provide a

complete solution, the configuration has to be extended with a J2ME profile, which is a collection of additional APIs. The most import profile in the context of application programming for mobile phones is the Mobile Device Information Profile (MIDP) [5], which is built on top of CLDC. MIDP provides a complete Java runtime environment with APIs for:

- user interface components
- networking
- persistant storage

Optional Packages

In addition to profiles and configurations, optional packages provide additional APIs for devices equipped with specific technologies. There are for example optional packages for Bluetooth communication [8], wireless messaging [7] or file system access [9].

2.2 Bluetooth

Bluetooth is a communication protocol which enables ad hoc wireless networks between electronic devices. Bluetooth technology is an open specification developed by the Bluetooth Special Interest Group (SIG), a group initially formed by Ericsson, IBM, Nokia and Toshiba. Interest in Bluetooth increased, and the SIG currently counts more than 2000 member companies. Bluetooth has the following characteristics:

- Bluetooth is a radio technology which operates at 2.4 GHz anywhere in the world.
- It is a short-range technology which reaches a maximum distance of about 10 meters. The transmission speed is about 1Mb/s.
- Because radio waves within short distances consume low power, it is a suitable protocol for small, battery supplied devices.

The Bluetooth API for Java has been specified in the *Java Specification Request 82 (JSR-82)* [8]. It provides functionality to discover Bluetooth enabled devices, to establish connections and to exchange data.

2.3 Wireless Messaging API

The Wireless Messaging API(WMA) [7] is an optional package for sending and receiving messages using the short message service (SMS) or the Cell BroadCast Service (CBS) on GSM networks. The WMA specification has been developed under the *Java Specification Request 120 (JSR-120)* and is implemented by most of the phone vendors.

3

Requirement Analysis

This chapter identifies common functional blocks of BlueDating [2] and Bluetella [3] and describes how these functionalities should be adapted and extended in order to provide a generic library of functions and useful development tools in BlueFramework.

3.1 Analysis of BlueDating and Bluetella

3.1.1 User Interface

Both applications, Bluetella and BlueDating, include a user interface to guide a user through the application. The user interface of both applications is built up in the same way, it consists of a hierarchy of screens, each of them composed of elements and commands defined by the `javax.microedition.lcdui` package. Programming a user interface is not complicated, once this hierarchy is defined and the functions behind each command are known. But adapting and changing a user interface can be quite time consuming. A change on one screen can cause adaption of other screens of the user interface. For this reason, an application developer should be appropriately supported in user interface programming.

BlueFramework shall provide a tool which helps to create and adapt a user interface fast and easily. Instead of programming the entire class, the application developer simply has to determine the screens, the order they are linked and the elements and commands they have appended. Provided with this information, BlueFramework is supposed to generate the Java class which defines the user interface for the mobile application.

3.1.2 Object Description

Bluetella provides methods for describing the files available on a mobile phone as well as for describing a file search. A search request holds information about the file as for example the size, the type, the name or the genre.
BlueDating includes the description functionality as well. In BlueDating however, persons are described instead of objects. A BlueDater, looking for the partner of his dreams, describes himself and the character he expects from his future partner in a profile.

BlueFramework needs to offer a module to manage any kind of data description. The module is supposed to include classes with methods to generate profiles describing persons or objects. Features and natures of objects shall be expressed by different attributes. In addition to profiles and attributes, there must be a structure which defines the scope of all possible attributes allowed for the description of objects of an application.

3.1.3 Matching

Bluetella as well as BlueDating exchange only data matching certain criteria. In BlueDating for instance, the detailed information about an encountered person will only be sent if the base and the core description of the BlueDaters match. Also in Bluetella, a search request and the file information are compared prior to the file exchange.

In order to exchange data selectively, BlueFramework is supposed to offer a structure to match two object description profiles. Two objects are similar or identical if the profile which describes them match. The following matching algorithms have to be provided:

- **Exact Match**
  An exact match compares two profiles accurately. To accomplish a match, all attributes of the profiles must match.
- **Partial Match**
  Compared to an exact match, a partial match verifies if the profiles match to a certain degree. The basic idea of the partial match is to calculate a matching score. If this score reaches a certain threshold, a match is accomplished although there might be small deviations between attributes.

3.1.4 Device Discovery

BlueDating and Bluetella are intended for Bluetooth enabled mobile phones. In BlueDating for example, each time a partner search is initiated, BlueDating scans the environment for other mobile phones running the same application. To retain discovered devices, BlueDating defines a *known devices* list, storing the

Bluetooth address together with the last date of contact.
Bluetella works in a similar way. The device discovery is implemented with a timer, searching in regular time intervals for other Bluetella devices in proximity. Bluetella also maintains a list of already discovered mobile phones.

BlueFramework should not only offer the functionality to discover Bluetooth devices in the area, it has to provide a structure to remember already encountered mobile phones as well. This structure has to include methods to access and manage the information gathered while discovering. The discovery procedure is supposed to run in the background, an application using the BlueFramework for Bluetooth discovery will be notified when another Bluetooth device is in its proximity.

3.1.5 Data Exchange

Once another mobile phone has been located, Bluetella and BlueDating establish a connection in order to exchange data.
An objective of BlueFramework is to provide methods for sending and receiving data. Each mobile device should be able to listen to incoming data as well as sending data. BlueFramework has to take on the task of serialisation and deserialisation of objects sent and received. An application developer shall not have to care about the serialisation of a profile or an attribute, it has to be all included in a module for communication. In additon to communication over Bluetooth, BlueFramework needs to include communication over SMS as well. The API for sending and receiving data has to be the same, regardless of the underlying technology.

3.2 Additional Requirements

The analysis of BlueDating and Bluetella builds a base of requirements BlueFramework should meet. In order to provide a complete solution for the development of mobile applications, BlueFramework has to meet the following requirements as well:

3.2.1 Log

Logging output for J2ME applications in not trivial. The *System.out* stream is defined but not usable on many mobile devices because there is no console the output can be directed to. BlueFramework shall provide this missing functionality. The following features are expected:

- **Statement classification**
  The module has to allow a classification of the log statements in different levels. An error statement for instance should be distinguishable from an information statement.

- **Different log targets**
  Log statements are to be written in different persistant data repositories as well as to screens of mobile devices.
- **Selective reading**
  The module should allow to read logs selectively, for example all log statements written in a certain time period, all log statements written in a certain class or all log statements of a specified level.

3.2.2 Routing

Data exchange between two devices raises the question of routing. BlueFramework needs to provide a module which provides algorithms and data structures to route data from a sender device to a receiver device. If data cannot directly be transmitted to the receiver, the routing module should allow mechanisms to find out alternative paths along which data can be sent to reach the receiver. The module needs to maintain a routing table storing the best routes to various devices.
Bluetooth currently only supports point-to-multipoint connections for mobile phones. Building up a network structure is not possible. Bluetella [3] provides a solution to work around this incapability. It spreads data using a "store and forward" algorithm, which profits from the user's mobility. The objective of BlueFramework is to generalize this routing approach in order to make it available to future mobile applications.

3.2.3 BlackList

When interacting with other devices, it might happen that a device does not send the data expected. To avoid reception of corrupt or false data again, the device should be put on a blacklist and not be contacted any more. In addition to the security mechanisms, BlueFramework needs to provide a list structure holding information about corrupt devices. Using this list in an application will avoid establishing a connection to a device which has recently sent corrupt data. This blacklist module should not only include methods to put devices to a blacklist, there should also be the possibilty to remove them again. Supplementary to the blacklist structure, the module shall also supply a whitelist. The whitelist keeps track of known and trusted devices.

3.2.4 Synchronisation

In combination with the module for communication, BlueFramework is supposed to provide methods for data synchronisation between devices. The functions offered by the synchronisation module ensure that devices are automatically updated with the latest version of the data they want to share.

A simple form of synchronisation just copies non-existing data from one

device to another without remembering versions. If both devices involved are already in possession of the same data, it is not possible to judge which device has the more recent version. To allow proper synchronisation, all data need to have a unique ID or date and time of the last update. There are two alternative method to keep track of the synchronisation status:

- **ID of the data**
  A device must remember which data has been sent to which device already. This information has to be stored in a structured form. When the next synchronisation takes place, it can be determined which data has yet to be transferred.
- **Date and time of last synchronisation**
  Instead of keeping the IDs of data transferred, each device remembers the date and time the last synchronisation with another device took place.

A more sophisticated synchronisation mechanism keeps track of different versions as well and can synchronize the most recent information to the other device. In this case all data has to include a unique ID and the date and the version (e.g. the date and time of the last update).

3.2.5 Security

J2ME applications run on resource constrained devices. Because security mechanisms usually need a lot of CPU, the implementation of security aspects have so fare been neglected. Nevertheless, a MIDlet is exposed to different kind of threats, especially when sensitive data or money is involved. Currently, there exists no general security solution for applications intended for mobile devices. Configurations, profiles and optional packages address a limited number of security issues. The objective of BlueFramework is to assembly the existing security concepts and to extend them with the missing functionality. In order to provide a secure mobile environment, BlueFramework has to include the following security concepts:

Authentication

Authentication is the procedure to determine whether someone or something is who or what it pretends to be. With the existing security features, authentication can be done trough PIN codes and digital certificates. JSR-82 involves PIN for authentication. When Bluetooth devices communicate for the first time they have to agree on a PIN. JSR-82 allows only authentication of a device, there are no means to verify a user's identity.

MIDP 2.0 brings in the new concept of trusted MIDlets [12]. A trusted MIDlet is digitally signed based on X.509 Public Key Infrastructure. Each mobile device holds a set of root certificates to verify a MIDlet signature. This new concept enables authentication of the MIDlet signer.

In addition to the concepts mentioned, BlueFramework is supposed to implement a generally usable mechanism for the authentication of devices as well as the persons using them.

Confidentiality

A mobile device may contain sensitive data. It the data is transmitted to other mobile devices, the security of the data depends on the security of the network. To protect it from eavesdroppers intercepting the transmission, the data has to be encrypted. MIDP 1.0 lacks of a secure network protocol as well as encryption algorithms. In MIDP 2.0, end-to-end security is provided through the HTTPS standard, cryptographic algorithms however are still not implemented.

Applications interacting over a Bluetooth link could also use JSR-82 which allows encryption of data transmission. For this purpose symmetrical encryption is used. Encryption of information is completely transparent, it is all done by the underlying JSR-82 implementation.

As encryption is limited to Bluetooth connections, BlueFramework needs to provide classes for data encryption and decryption.

Integrity

When data is transmitted, the receiver of the data has to be provided with methods for verifying if the data has been changed or replaced during transmission. The concept of trusted MIDlets mentioned earlier in this chapter allows verification of a MIDlet's integrity as well. When a MIDlet is downloaded to a device, the device has the possibilty to verifiy its integrity with a set of root certificates. For the development of signed and trusted MIDlets, a certificate conform the X.509 Public-Key Infrastructure is necessary.

As integrity is not only important for MIDlet download but also for data transmission, the security module of BlueFramework must include mechanisms to verify integrity.

4

BlueFramework Architecture

As a result of the requirement analysis in Chapter 3, BlueFramework is composed of the modules shown in Figure 4-1. It consists of development tools and Java class libraries to simplify the development of future applications for mobile devices. The shaded boxes represent the development tools, the white ones represent libraries. The object description module is both a development tool and

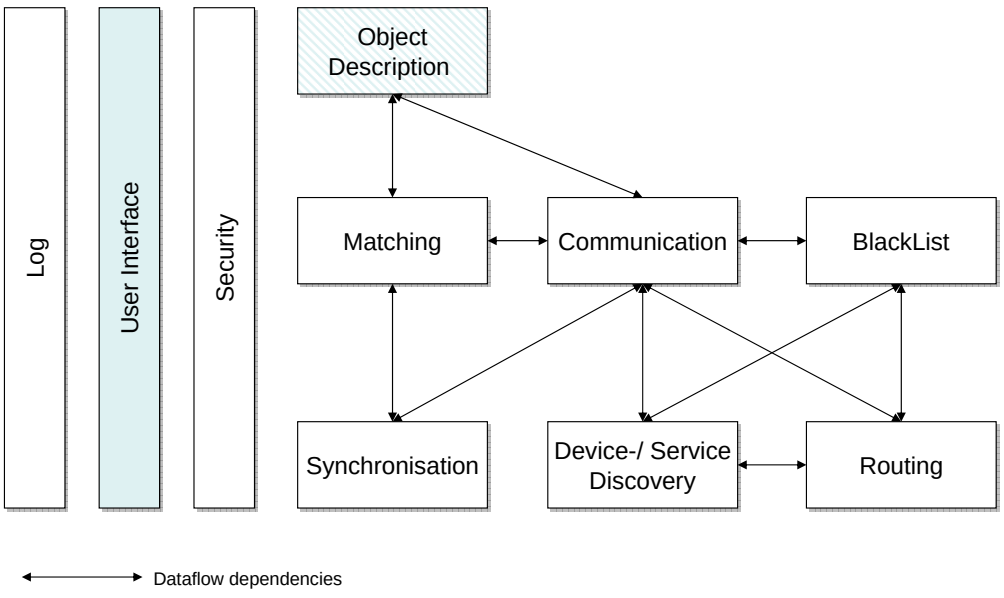


Figure 4-1
Survey BlueFramework

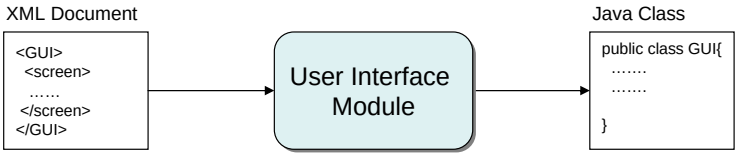


Figure 4-2
User interface module

a library. Eventual data flow dependencies are indicated by arrows between the modules. The three upright blocks on the left of the figure interact with all of the modules shown on the right. The following sections focus on the design goals of each module in more detail.

4.1 User Interface Module

The user interface module is a development tool to generate and adapt a user interface fast and easily. A user interface can be seen as a collection of screens, each of them composed of elements and commands to interact with the user. Building up the entire screen hierarchy is a quite time consuming and not very challenging programming task. A change on one screen can cause adaption of other screens of the user interface. For this reason, an application developer should be appropriately supported in user interface programming. The module illustrated in Figure 4-2 generates a user interface class directly from an XML document. The XML document holds all necessary information to define screens and the appended elements and commands.

4.1.1 User Interface API

The package `javax.microedition.lcdui` includes a set of features for implementation of user interfaces for MIDP applications. The API is divided in *high-level* and *low-level* APIs. With the *low-level* API, the programmer has exact control over position and appearence of the elements on the screen. A MIDlet[1] programmed with the *low-level* API might not be portable to other devices. Because BlueFramework aimes at supporting a wide range of devices, the user interface module is programmed with the *high-level* API. Contrary to the *low-level* APIs, *high-level* APIs only define the content of a screen, whereas its appearence is set by the device running the application. Figure 4-3 gives an overview over the

[1] A MIDP application is called a MIDlet.

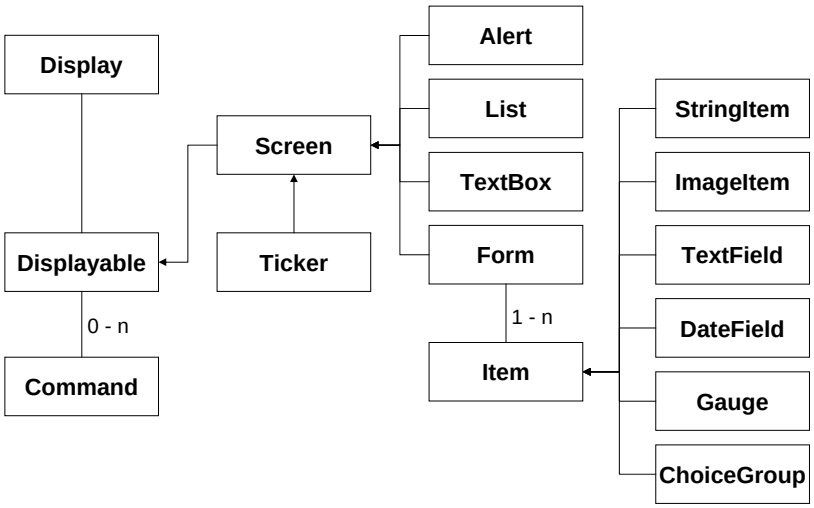


Figure 4-3
MIDP user interface class hierarchy

high-level class hierarchy. A MIDlet consists of exactly one *Display* object which allows to place different *Displayable* elements on the screen, but only one at a time. As shown in Figure 4-3, there are four different subclasses of *Screen*: *Form*, *List*, *TextBox* and *Alert*. Their significance is discussed in the following section.

4.1.2 XML Document

The XML schema *GUISchema.xsd* provided in C.2 specifies the elements and attributes that can appear in an XML document in detail. Figure 4-4 gives an overview about the structure of the XML documents used for user interface definition. The oval boxes represent XML elements, square boxes stand for XML attributes. Dashed lines signal that the occurrence of the attribute or element is optional. The structure of the *Form* element shaded in grey is shown in Figure 4-5. Every document has a main element *GUI* and a number of subelements *Screen* which represent one of the four *Screen* objects. The parameters necessary to define one of those elements are defined as XML attributes. One XML attribute they all have in common is *objectname*. It refers to the name of the object defining the screen. All *Screen*s may have appended *Command*s.

- **List**
  A *List* consists of list items which include a *StringPart* and an optional *ImagePart*. A *List* has a *title* and a *type*. The *type* defines if only one list item can be selected (`Choice.EXCLUSIVE`) or if multiple selection is possible

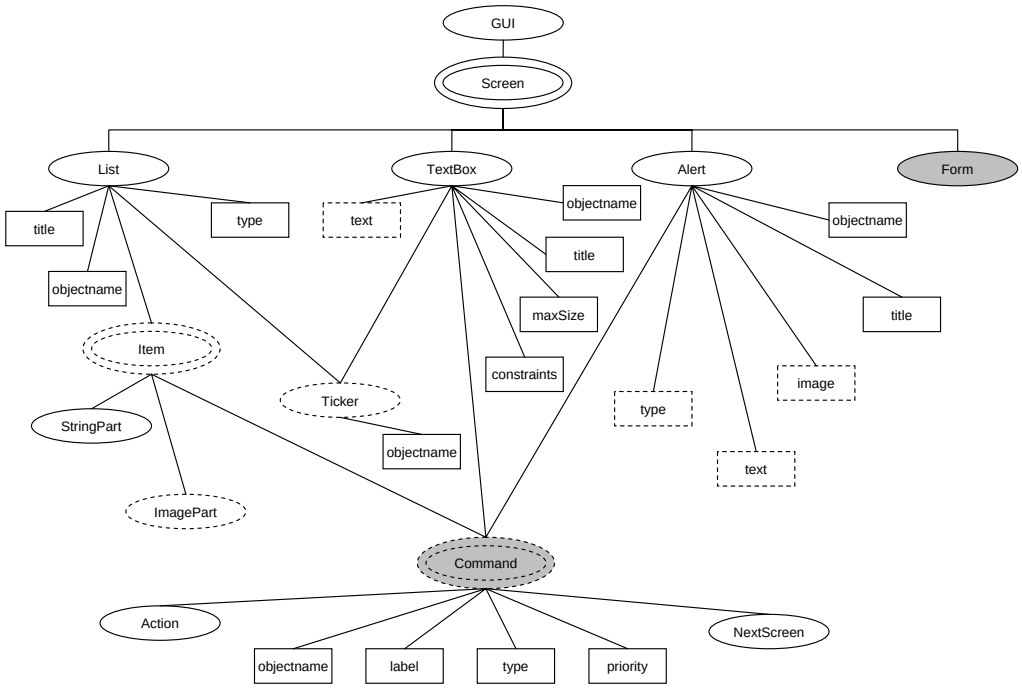


Figure 4-4
Structure of an XML document to generate an user interface

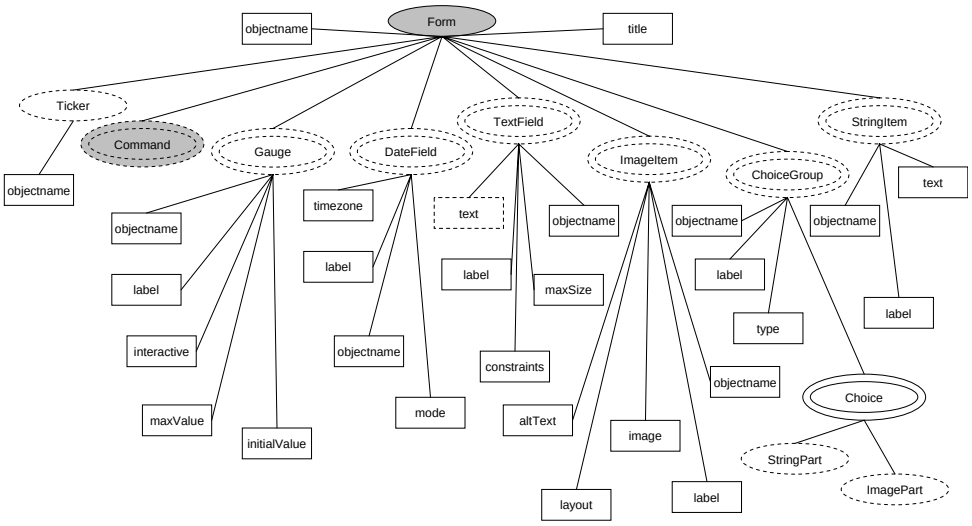


Figure 4-5
Form as XML document

(`Choice.MULTIPLE`). In case only the highlighted item can be chosen the *type* must be set to `Choice.IMPLICIT`. The attribute *title* refers to the list title displayed on the screen. Each of the list items can have a number of appended *Command*s.

- **Alert**
  An *Alert* is a pop-up like screen to display an information or error message for a short period of time. As a *List* item, an *Alert* has a *title* which will be displayed on the screen. An optional attribute is *type* which can be set to `ALARM`, `CONFIRMATION`, `ERROR`, `INFO` and `WARNING`. An *Alert* may include an *image* or a *text* which will be shown to the mobile phone user.
- **TextBox**
  A *TextBox* is an element, through which a user can enter and edit text. It is possible to define constraints on the size (maximum number of characters) and the type of text the *TextBox* supports. The attribute *constraints* can be set to `ANY`, `EMAILADDR`, `NUMERIC`, `PHONENUMBER`, `URI` and `DECIMAL`.
- **Form**
  A *Form* is a screen which can contain: a *TextField*, a *StringItem*, an *ImageItem*, a *ChoiceGroup*, a *Gauge* or a *DateField*. The attribute *title* will be displayed as title of the screen.

The structure of a *Form* element is shown in Figure 4-5. There are different items that can be added to a *Form*:

- **StringItem**
  A *StringItem* is a non editable item containing text to be displayed. It is defined by the attributes *objectname*, *label* and *text*.
- **ImageItem**
  An *ImageItem* holds a reference to an image. The attribute *altText* defines a String to be set instead of the image in case the image exceeds the capacity of the display. The attribute *layout* is to determine where the image has to appear on the screen.
- **TextField**
  A *TextField* has the same features as a *TextBox*.
- **DateField**
  A *DateField* is an editable component of a *Form* for presenting dates and time. Apart from the *label* and *objectname* attributes, there needs to be defined the time zone and the input mode. Three modes are supported: A mode which allows only date information to be set, a mode which allows only time information and one which allows both, time and date values.
- **ChoiceGroup**
  As indicated by the name, a *ChoiceGroup* allows to display a group of choices. Each choice is composed of a *StringPart* holding text and an *ImagePart* holding the reference to an image to be shown. The attribute *type* determines if a single choice has to be made (`Choice.EXCLUSIVE`) or if multiple choice are allowed as well (`Choice.MULTIPLE`).
- **Gauge**
  A component for displaying bars of different levels, it is often used in context of progress indication. A *Gauge* may be interactive or non-interactive; *interactive* has to be set true if the *Gauge* is interactive, to false otherwise. The attributes *initalValue* and *maxValue* represent integers indicating the the level of the lowest and highest bar of the *Gauge* object.

To interact with the user, a screen has appended commands. Commands are used to invoke an action and to change between screens. The module needs to know which function has to be invoked and which screen should be displayed next. For this reason, the XML element *command* has two subelements *nextscreen* and *action*, holding the next screen's object name and the name of the method to be called. The information about a *Command* object is encapsulated in the four XML attributes: *objectname* refers to the name of object, the *label* holds the command name which is visible on the screen. The attribute *type* specifies the type of command. Possible types are the following: `OK`, `BACK`, `CANCEL`, `EXIT`, `HELP`, `ITEM`, `STOP` and `SCREEN`. The first four are most commonly used. By *priority* it is possible to define a commands priority compared to the other commands appended on the same screen.
An other common object which can be appended to any screen is a *Ticker*. A *Ticker* is set in the upper part of a screen. The information it contains scrolls

continously across the display. For the definition of a *Ticker* object the *objectname* is needed.

Any XML document conform to *GUISchema.xsd* acts as input of the `GUIGenerator`, the class which parses the document and generates a user interface java class. The application `GUIGenerator` has to be invoked to generate a Java user interface class. The name of the XML document to parse as well as the class name of the generated java class are passed as command-line arguments.

4.2 Object Description Module

The goal of the object description module is to offer mechanisms to describe objects or persons accurately. The module is divided into two parts, a class library and a development tool.

4.2.1 Library

Profile

The main class for object description is `Profile`. It is useful for two purposes:

- **Object description**
  An object or a person can be described with an instance of `Profile`, which is a collection of `Attributes`. Each of them describes a character/ feature of the person/object.
- **Search description**
  A `Profile` describing a search is similar to an object description. Instead of `Attributes`, a search profile is composed of `SearchAttributes` which provide a more specific structure to describe a search coherently.

A valid `Profile`, regardless of the purpose it is used for, has to be composed of `Attributes` being within the ranges defined by a `ProfileTemplate`. For this reason, a `ProfileTemplate` has to be set up initally.

Attribute

Attributes are components of a `Profile`. An `Attribute` acts as container holding:

- an **ID** to identify the `Attribute` within the `Profile`.
- a **type** determing whether the `Attribute` is of type `String`, `Long` or `Integer`.
- a **value**. Contrary to the first two parameters, the value is not mandatory. Attributes without a value are called unlimited, which means that any value of the specified type can be used instead. In context of a `Profile`, an

unlimited `Attribute` expresses indifference. In BlueDating for instance, an unlimited `Attribute` can be used to state a BlueDaters indifference about a future partner's character.

SearchAttribute

A `SearchAttribute` extends the `Attribute` with two additional variables:

- an **operator**, used in combination with the attribute value. For an `Attribute`, the operator is "=" by default. For a `SearchAttribute` however, it is possible to set it to *<, > , ==* or *!=*.
- a **link operator** which specifies how to link the `SearchAttributes` of a `Profile` having the same ID. Logical *AND* and *OR* are allowed. In case the `Profile` is composed of `Attributes`, the link operator is by default set to *OR*.

ProfileTemplateAttribute

`ProfileTemplateAttributes` are the components of a `ProfileTemplate`. Compared to a regular `Attribute`, a `ProfileTemplateAttribute` allows the definition of an attribute value range. For `Integer` and `Long` typed attributes, it is possible to define a start and an end value of a range.

ProfileTemplate

A `ProfileTemplate` is a collection of `ProfileTemplateAttributes`. It targets at predefining the range of `Attributes` a `Profile` can contain.

4.2.2 Development Tool

A `ProfileTemplate` can be generated by means of the development tool included in the object description module. The tool works in a similar way as the user interface module described in Chapter 4.1: An XML document is parsed to generate a Java class. The XML document defines the `ProfileTemplateAttributes` and their possible value ranges. When `ProfileTemplateGenerator` is run, it parses the document and creates a `ProfileTemplate`.

The structure of the XML document to parse is illustrated in Figure 4-6. Oval boxes represent XML elements, square boxes stand for attributes, the dashed line signals that the element *value* is optional. The main XML element is *ProfileTemplate* which can be viewed as a collection of *ProfileTemplateAttribute* elements. A *ProfileTemplateAttribute* has two attributes and a choice of two subelements: *value* and *range*. The XML attribute *type* serves to define the type of the ProfileTemplateAttribute. The value can either be `String`, `Integer` or `Long`, *ID* refers to the ID of the ProfileTemplateAttribute. The element *value* is optional, it keeps a ProfileTemplateAttribute's value. If no value is set, the

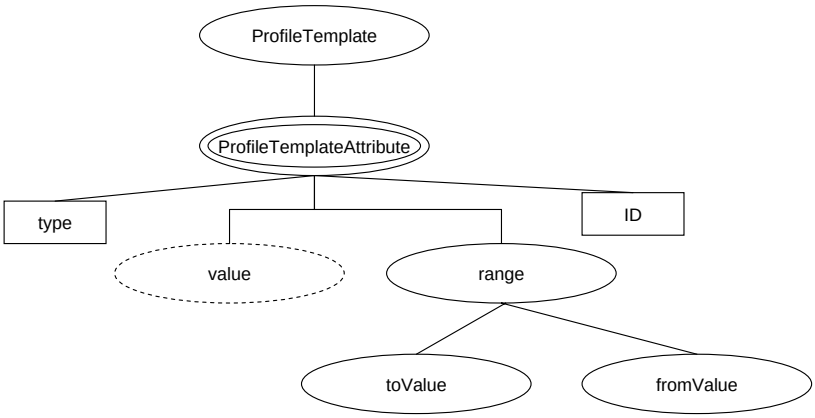


Figure 4-6
Structure of an XML document to generate a ProfileTemplate

ProfileTemplateAttribute will be unlimited. In case the ProfileTemplateAttribute defines a range of values, the start and the end value have to be provided in the XML elements *fromValue* and *toValue*.

4.3 Matching Module

The matching module aims at finding out if or to which degree the attributes of two `Profiles` correspond. The module provides two different matching algorithms: an exact and a partial match.

4.3.1 Exact Match

The exact matching algorithm compares a profile A with a profile B. Profile A is a search profile, a profile composed of search attributes. The method `exactMatch(Profile A, Profile B)` returns true if A and B match exactly. An exact match is accomplished if:

1. The number of attributes in profile B is at least the number of attributes in profile A.
2. All attributes of A are also defined in profile B.
3. Each attribute of A matches the corresponding attribute of B according to the operators defined in A.

To check these constraints, the method has to loop through all attributes of A. For every attribute, it is verified if its values and operators evaluate to true. If all

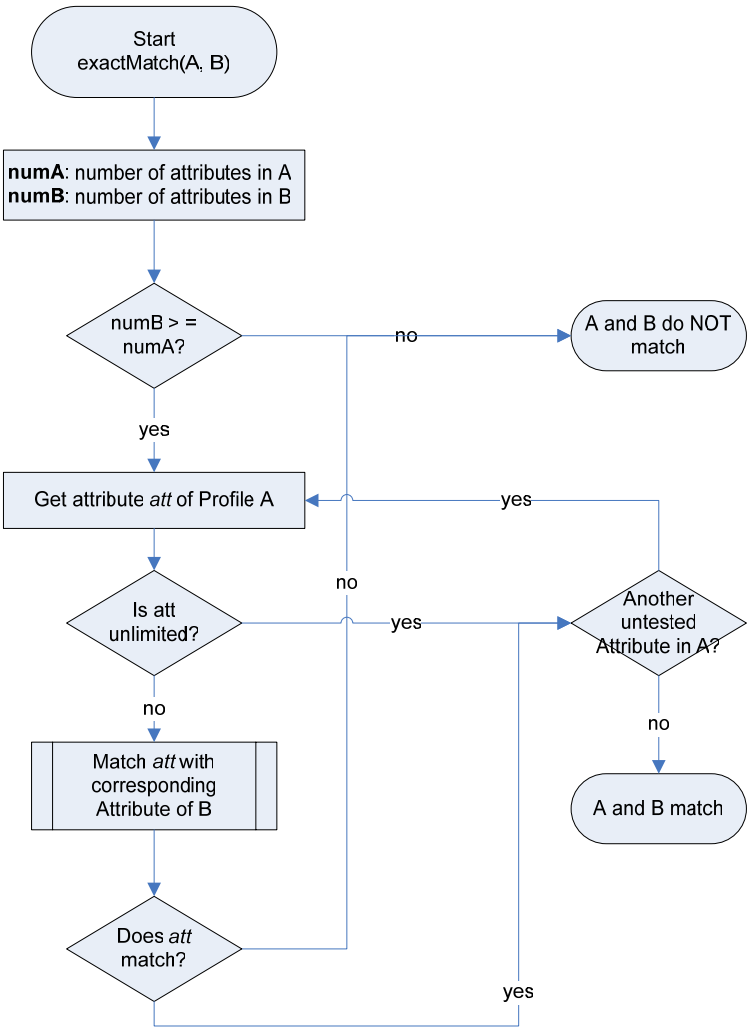


Figure 4-7
Exact match

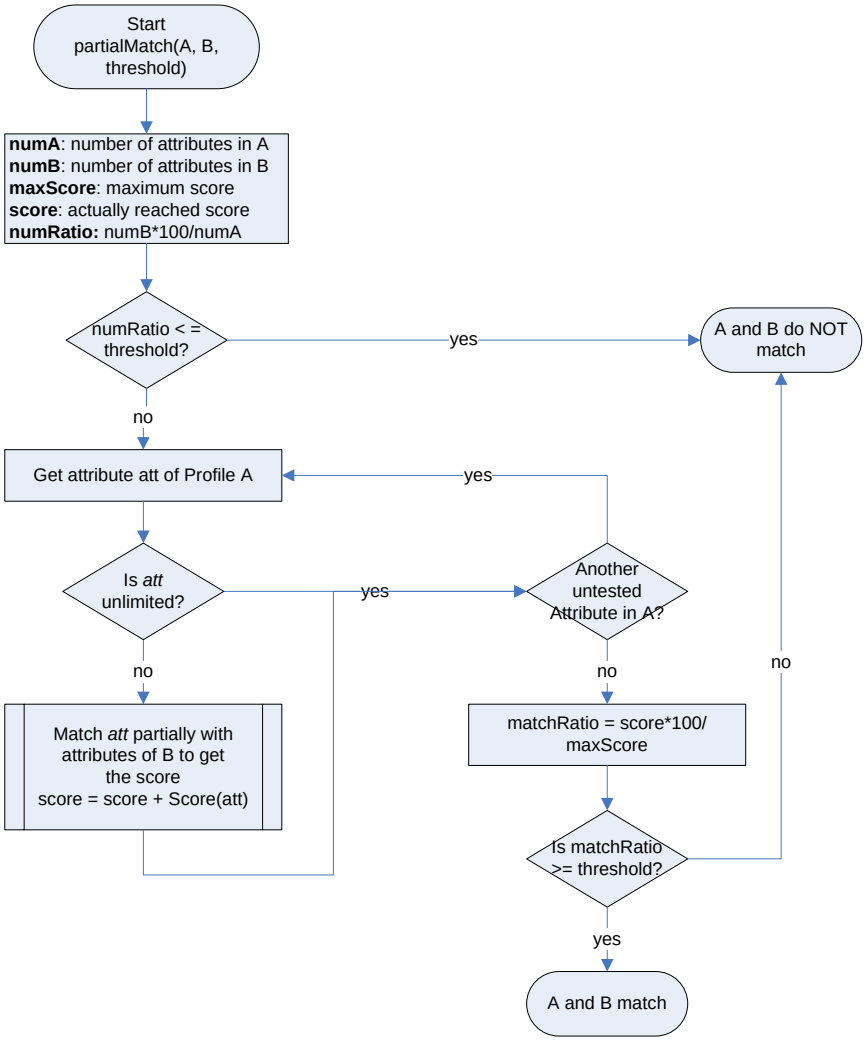


Figure 4-8
Partial match

attributes of *A* match according to the operators, an exact match is accomplished.

4.3.2 Partial Match

An alternative algorithm to the exact match explained in the previous section it the partial match. Compared to the exact match, the second method `partialMatch(Profile A, Profile B, int threshold)` compares the

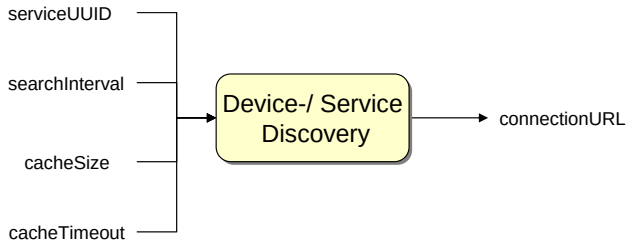


Figure 4-9
Discovery module

profiles in more detail and pays attention to the deviation of attribute values. It has an additional parameter *threshold*, an integer between 0 and 100, indicating the percentage of matching score, which still can be interpreted as match. As in the exact match, the attributes of A are compared with the attributes of B. Contrary to the previous matching algorithm, the partial matching algorithm assigns a score to each attribute. If an attribute of A and B match exactly, it will get the maximum score. Otherwise, the score obtained depends on the matching level of the attributes, the closer they are, the higher the score is. For unlimited attributes expressing an indifference, the score is evaluated on the basis of a standard deviation. The sum of the scores of each attribute of A are then divided by the sum of all maximum scores. The ratio must be equal or bigger the the *threshold* in order to return a partial match. A partial match of profile A with profile B is accomplished if:

1. The number of attributes in profile B is at least *threshold %* of the number of attributes in profile A.
2. The ratio of maximal score to the actually reached score is at bigger or equal *threshold %*.

The detailed procedure is illustrated in figure 4-8.

4.4 Discovery Module

The main objective of the discovery module is to locate all devices in proximity, which are running a certain service. As a result of this search, the discovery module returns a *connectionURL*, which is the information needed by the communication module to establish a Bluetooth connection. As stated in 3.1.4, the module should include a structure to remember recently discovered devices with their *connectionURL*. Already discovered devices are therefore stored in a cache.

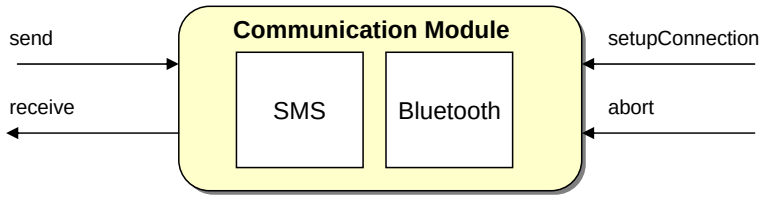


Figure 4-10
Communication module

This structure has the advantage to prevent from unnecessary repetitions. As illustrated in figure 4-9, the discovery module needs four input parameters:

- The *serviceUUID* identifies the service the module has to search for.
- The *searchInterval* (in seconds) determines the time interval before the discovery starts again.
- The *cacheSize* specifies the maximum number of cache entries.
- The *cacheTimeout* is the period of time a cache entry is stored before it expires.

The discovery module allows an application to run a device discovery in the background. The application will only be notified, if a Bluetooth device, which runs the searched service is nearby. All the functionality is provided by the class `DiscoveryModule`. To create a new instance of this class, the variables mentioned above are necessary. A `BluetoothStateException` is thrown to signal that Bluetooth is not available on the device running the application. Once the `DiscoveryModule` is instantiated, the discovery process can be started with the method `startDiscovery()` and will be terminated by a call of `stopDiscovery()`. The class includes two methods to retrieve the connection information of discovered Bluetooth devices. If `discoverServices(long timeout)` is invoked, the *connectionURL* of all discovered devices will be returned. If no device has been located so far, the method waits for a maximum delay of *timeout*. A call of `waitForServices()` waits until at least one device has been found and returns the corresponding *connectionURL* of all discovered devices.

4.5 Communication Module

Once the discovery module has found out all nearby devices with the services they offer, the communication module shown in Figure 4-10 can establish a Bluetooth

connection for data exchange. If a connection is set up, data can be exchanged by means of the methods for sending and receiving data. The module not only allows interaction based on a Bluetooth connection, it also offers data exchange via SMS. The API for sending and receiving data is the same, regardless of the underlying technology. For data exchange via SMS, an alternative method for discovering and registering devices (and in particular the phone number) would be needed.

4.5.1 Connection Setup

Before sending or receiving objects, the connection has to be set up and opened. Therefore, the method `setupConnection(Object settings)` has to be invoked. It creates a connection by passing a *settings* Object which provides all necessary details for the connection setup. Depending on the underlying technology, the *settings* is an instance of `BluettoothSettings` or of `SMSSettings`.

- `BluetoothSettings`
  Bluetooth follows a server client model for communication. A Bluetooth server is an application which provides some kind of service to a client device over a Bluetooth communication. A server listens to client requests, whereas a client takes on the active part and initiates the communication to a server. Connection setup on server side is different from the one on the client side. `BluetoothSettings` takes into account this difference and provides two methods for connection initalisation. With `setServerSettings(UUID serviceUUID, String serviceName)` a server side connection is setup. The *serviceUUID* refers to the unique idenfier of the service, the server offers; the second argument *serviceName* desribes the service in words. When `setClientSettings(String connectionURL)` is invoked, a client side connection will be established. The arguement `connectionURL` results from a device discovery via the discovery module.
- `SMSSettings`
  The wirless messaging protocol is based on a server client structure as well. The wireless messaging API (WMA) allows J2ME devices to run server applications which will automatically process and respond to incoming messages of client devices. Unlike Bluetooth servers, SMS servers are identified by their telephone number and a port number. SMS based applications should agree on a port for passing messages. The method `setPort(String port)` is used to set a server side, as well as a client side port for sms exchange. If no port is set, the incoming messages is handled by the mobile phone's inbox. Before sending data, the client has to set the recipient's phone number by calling `setPhoneNumber(String phoneNumber)`.

Connection establishment can fail for different reasons. A mobile phone might not support Bluetooth or wireless messaging, or it is attempted to communicate

over a port, which is not permitted. In both situations, the failure will be signaled with an `ConnectionSetupException`.

4.5.2 Data Exchange

Once a Bluetooth or an SMS connection has successfully been established, the devices are ready for data exchange. The functionality for sending and receiving data is implemented in the classes `BluetoothCommunicator` and `SMSCommunicator`. Both of them have the same interface:

- `send(Object object)`
  Method for sending an object to a mobile phone. The module allows to send a `Profile`, a list of `ProfileTemplateAttributes`, a `BlueMessage` , a `String`, an `Integer` or an `ErrorMessage` object. An `ErrorMessage` is used to send error codes. A `BlueMessage` is an object storing a text message and an ID of a `Profile` describing the message content.
  In case data is transmitted via SMS and exceeds the payload allowed, the data is split into several SMS. An attempt of sending a `null` object will result in a `NullPointerException`.
- `receive()`
  Method for receiving data. It supports reception of the objects mentioned above.

If no connection has been set up, a call of either of the methods will cause a `CommunicationException`. In case data transmission is not possible due to a failure while sending or receiving, a `CommunicationException` will be thrown.

4.6 Log Module

The log module is intended for debugging and monitoring purpose. It writes the state of the program at various stages of its execution to some repository and provides a detailed context for application behaviour and failures. The `Log` class represents a log and offers several methods to write in as well as to read from a data repository. A log statement includes the following information:

- The date the statement was written
- The class from which the statement was written
- The level to classify the statement whether it is an *INFO*, an *ERROR*, a *DEBUG*, a *FATAL* or a statement for the *USER*.
- The information to be logged.

As shown in figure 4-11the log module should support different data repositories. This functionality is implemented in the class `LogTarget`. It allows the the log statements to be written to a:

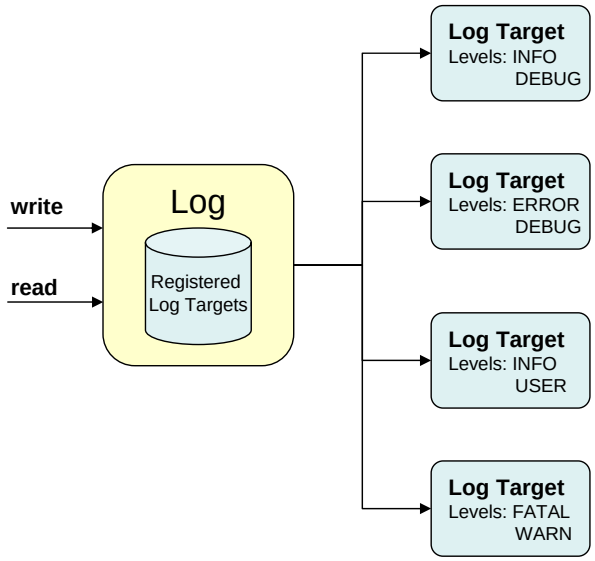


Figure 4-11
Log module

- **File**
  Log statements are written into a file stored on the mobile phone. This kind of log target is restricted to mobile phones supporting file systems. Neither *CLDC* nor *MIDP* include APIs for access to file systems on mobile phones. *The FileConnection Optional Package*, which is a part for JSR 75 [9], provides an API enabling J2ME applications to create, read and write files and directories.
- **RecordStore**
  MIDP provides a solution to persistently store data, the record management system (RMS). RMS contains multiple record stores, which can be accessed and managed by MIDlets. A record store can be viewed as a collection of records holding binary data. Record store access is restricted to one MIDlet only, it is not possible to view another applications data.
- **Screen**
  An alternative to keep log information for a short time but not persistently is to display the statements on the screen of the mobile phone.
- **Ticker**
  A ticker is another alternative to the screen log target. In a ticker, set in the upper part of the mobile phone screen, a log statement scrolls continuously across the display. This log target has the disadvantage that only one statement can be viewed at a time.

4.6.1 Write to a Log

In order to write log statements, the log has to keep track of all existing log targets. The method `registerLogTarget(LogTarget target, int[] levels)` of the class `Log` is thought to fulfill this purpose. The first parameter refers to the log target to register, the second parameter defines which level of log statements the log target stores. The following example demonstrates how to write to a log:

```
/** Log initialisation */
Log log = new Log();
LogTarget tt = new LogTarget(Log.TICKER,
                "tickerTarget", new int[]{Log.USER,Log.INFO});
LogTarget ft = new LogTarget(Log.FILE, "C:\\Nokia\log",
                new int[]{Log.DEBUG, Log.ERROR});

/** LogTarget registration with max number of log entries */
log.registerLogTarget(rst,20);
log.registerLogTarget(ft,35);

/** set the current display for the ticker LogTarget */
tt.setTickerDisplay(display);

/** add a log statement */
log.write(Log.INFO, "testClass", "this is a log statement");
```

As shown in the code example above, a `LogTarget` of type `TICKER` needs additional information. Before writing log entries, the current display has to be set.

4.6.2 Read from a Log

The class `Log` has different methods to read statements selectively. All methods take a `Display` object as argument to display the log statements to the user.

- `read(int level, Display logDisplay)` for instance, reads all log entries of the specified level from the log targets.
- `read(String class , Display logDisplay)` displays all statements written in *class*.
- `read(Date from, Date to, Display logDisplay)` scans the log targets for statements written in the time period between *from* and *to*.
- `read(Date from, Date to, String class , Display logDisplay)` allows a combination of the methods above. All log entries written in the given time period in the specified class are returned.

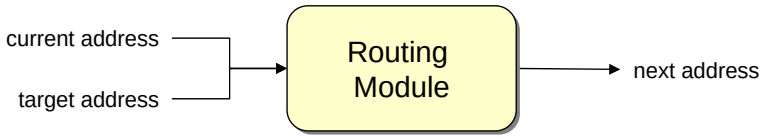


Figure 4-12
Routing module

- `read(Date from, Date to, int level , Display logDisplay)` is another combination of read methods. All log entries of a certain level written in the given time period are read.

4.7 Routing Module

The objective of the routing module is to provide the algorithms and data structures to route data from a sender to a target device. If data cannot directly be transmitted to the target, the next device to forward the information can be determined by means of the routing module. For this purpose a routing table storing the best routes to various devices is maintained. This routing table is a list structure storing address pairs. The routing module takes the address of the sender address and the receiver address as input and determines the address of the device, to which the data has to be transmitted next in order to finally reach the target device. The result of this look up is returned as Figure 4-12 illustrates.

4.8 Blacklist Module

The blacklist module prevents the application from receiving corrupt data. Once corrupt data is received, the sending device is put to a blacklist and is not contacted anymore. If the device recovered from its failure and does not send corrupt data anymore, there is a procedure to remove it again from the blacklist. The module illustrated in Figure 4-13 provides methods to add and to remove a device from either the blacklist or the whitelist. If a device is added to the blacklist it will automatically be removed from the whitelist.

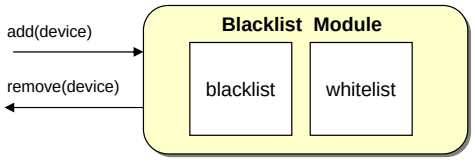


Figure 4-13
BlackList module

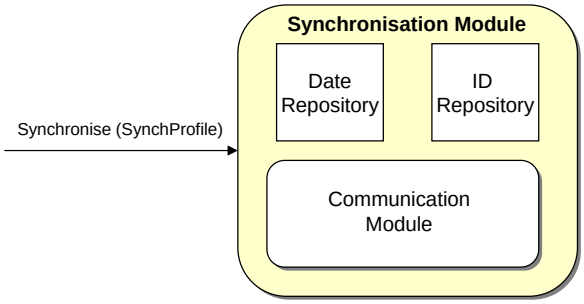


Figure 4-14
Synchronisation module

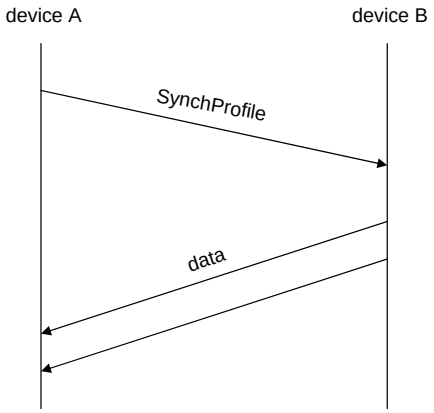


Figure 4-15
Data exchange in the synchronisation module

4.9 Synchronisation Module

As explained in Chapter 3.2.4, the synchronisation module is intended to synchronise data between two devices. For information transmission the synchronisation module profits of the methods of the communication module. Figure 4-14 shows the structure of this module. It provides two alternatives to keep track of the synchronisation status:

- The date repository is a list structure storing date and time of the last synchronisation with another device took place. The Recordstore Management System (RMS) provided in MIDP will be best for saving this information.
- Similar to the date repository, the ID repository is a list in the Recordstore Management System remembering which data has been sent to which device.

The device initialising a synchronisation invokes the method `synchronise(SynchProfile)`. The parameter `SynchProfile` is passed to indicate whether a synchronisation based on the ID or on the date takes place. Figure 4-15 illustrates data synchronisation between device *A* and device *B*. *A* is the one starting the process. Device *B* will respond will all information available according to the `SynchProfile`. Assuming the `SynchProfile` holds the date of the last synchronisation, device *B* will send all objects created after this specific date.

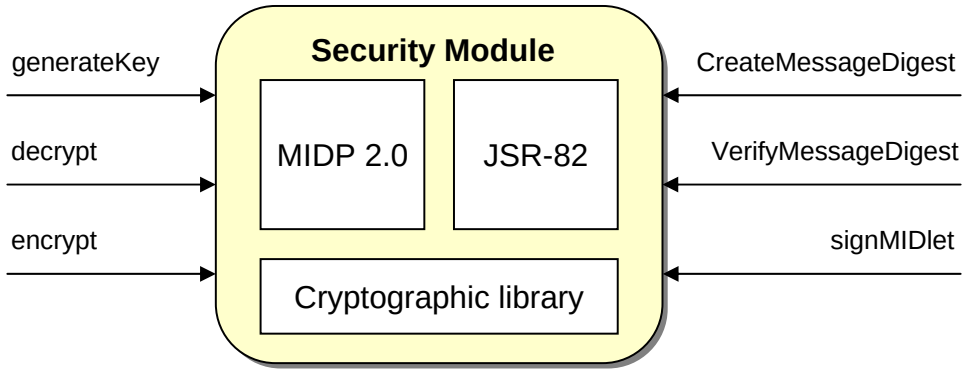


Figure 4-16
Security module

4.10 Security Module

Although J2ME is powerful for mobile application development, it provides only limited security. This originates from the fact that security functions require a lot of CPU and memory resources which is limited on devices running J2ME applications. The Bluetooth optional package however provides different levels of security.
Today's mobile phones support internet protocols like HTTP or WAP, they may have open GSM, GPRS or Bluetooth connection, which represent potential security hazards. According to [10], security for a mobile environment includes two aspects: The mobile device has to be protected against malicious mobile software and the mobile software must be protected against malicious mobile phones. As no general security solution for mobile applications is available, the security module combines the already existing ones and extends them with the implementation of missing security mechanisms. If BlueFramework wants to provide mechanisms to build up a secure mobile environment, it has to add the security issues shown in Figure 4-16:

- Encrypt data using asymmetric and symmetric cryptography
  The requirement analysis in Chapter 3.2.5 pointed out that that no mechanisms for encryption and decryption are available. As encryption of information is important to grant any security concept, BlueFramework provides a cryptographic library of functions used to encryt and decrypt data using symmetric and asymmetric cryptography. Instead of reimplementing the entire functionality, BlueFramework could take advantage of an already existing cryptographic library like the Bouncy Castle [11].The Legion of Bouncy Castle is an open source library providing a lightweight crypthog-

raphy API intended to work with J2ME.

- Create message digests
  In order to grant the integrity of data, the security module provides functions to create a message digest, a digital finger print of the data. This fingerprint could for example be generated by means of a one-way hash function. A one-way function is easy to calculate but hard to invert. Before data is transmitted, a digital fingerprint is calculated. The data in clear text will then be transmitted together with the message digest. For verifying data integrity the receiver simply has to recalculate the digest and comparing it to the original data. The security module should provide both: methods to create a message digest and methods to compare messages and their digests.
- Work with digital signatures
  In order to profit of the concept of trusted MIDlets introduced in MIDP 2.0, the security module includes mechanisms to work with digital signatures. The module provides methods to sign developed MIDlets as well as to validate a MIDlets certificate. Therefore the security module has also to include methods to safely store root certificates for MIDlet validation.

5

Implementation BlueFramework

This chapter takes a closer look at the implementation ot the modules introduced in the previous chapter. All modules which were necessary for the development of the BluePinbaord application have been implemented. This modules are: the user interface module, the object description module, the matching module, the discovery module, the communication module, the blacklist module and the log module.

5.1 User Interface Module

The development tool to generate new user interfaces for J2ME applications is implemented in the class `GUIGenerator`. It parses the XML document passed in the first command-line argument and creates a Java class with the name of the second command-line argument. The generated class implements the `GUI` interface of BlueFramework. For parsing the XML document, SAX [1] is used.
To obtain a SAX parser, the module uses a new instance of `SAXParserFactory` of the package `javax.xml.parsers`:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
```

As the parser reads an XML document, it invokes methods of a given `DefaultHandler` to handle events appropriately. The class `GUIGenerator` extends the `DefaultHandler` class of the package `org.xml.sax.helpers`. It overrides the callback methods for the following events:

- `startElement(String namespaceURI, String localName, String qName,Attributes attrs)`

[1] the Simple API for XML

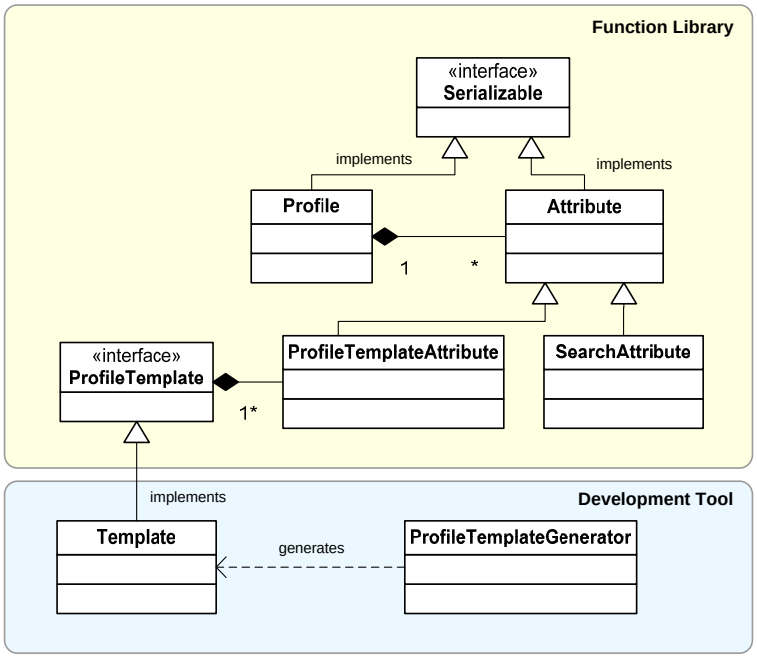


Figure 5-1
Class hierarchy object description module

As a documents content is parsed, the parser will invoke this method at the beginning of every element in the XML document. In `GUIGenerator`, the method reads the XML attributes which are needed to define the elements and commands of a screen.

- `endElement(String uri, String localName, String qName)`
  Method invoked by the parser at the end of each element in the XML document. `GUIGenerator` then writes the object definitions to the corresponding file.
- `characters(char[] ch, int start, int length)`
  Method to retrieve the characters between two XML elements.

5.2 Object Description Module

5.2.1 Library

The object description module consists of the classes `Attribute`, `SearchAttribute`, `ProfileTemplateAttribute` and the interface `ProfileTemplate`. `SearchAttribute` and `ProfileTemplateAttribute`

are derived from `Attribute` and provide more specific structures. Unfortunately, J2ME does not include the interface `java.io.Serializable`, which allows objects to be serialised and deserialised. Because serialisation will be necessary in the communication module, BlueFramework defines its own interface `Serializable` with the methods `readObject(InputStream in)` to read an Object from an InputStream and `writeObject(OutputStream out)` to write one to an OutputStream. The class hierarchy of the object description module is shown in figure 5-1. The blue box represents the development tool included in the object description module, the yellow box shows the classes and interfaces of the library.

5.2.2 Development Tool

In addition to the library for object description, the module includes a development tool. As mentioned in the previous section, a `ProfileTemplate` is only an interface. The actual implementation of `ProfileTemplate` is done through the development tool implemented in `ProfileTemplateGenerator`. This has the advantage that BlueFramework has not to be recompiled each time an new `ProfileTemplate` has been defined. As figure 5-1 demonstrates, the example class `Template` generated by `ProfileTemplateGenerator` implements the `ProfileTemplate` interface. The structure of `ProfileTemplateGenerator` is similar to the one of the user interface module. SAX is used for parsing the XML document as well.

5.3 Matching Module

The general use of the matching module has already been described in Chapter 4.3. This chapter focuses on the matching procedure and score calculation in more detail. The matching module is implemented in the class `MatchingModule`.

5.3.1 Exact Match

In order to accomplish a match between a profile *A* and a profile *B*, each attribute of *A* has to be defined in *B* and must match the the corresponding attribute in *B*. Figure 5-2 illustrates the process of matching an attribute *att* of profile *A* with the attributes of profile *B*.
First of all, all attributes having the same ID than *att* are retrieved from *A*. If *att* is the only, the method `evaluateExactly(att, B)` is called. True is returned if *att* is also defined in *B* and if it matches the corresponding attribute. In case profile *A* consists of more than one attribute having *att's* ID, they are stored in the list *attID*. In this case the match depends on *att's* link operation. If the attributes are linked with a logical *OR* only one of the attributes of the list *attID* must evaluate to true. If they are linked with a logical AND however, there must be an attribute in *B* which fulfills all contraints implied by the attributes of *attID*. The method `evaluateAllExactly(attID,B)` tests these contraints.

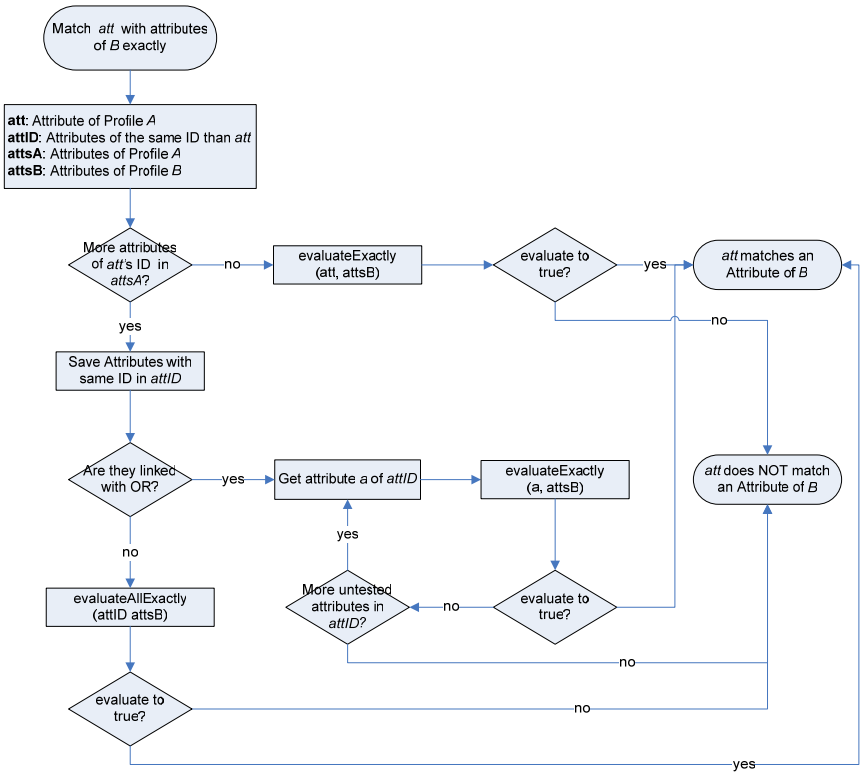


Figure 5-2
Attribute matching in an exact match

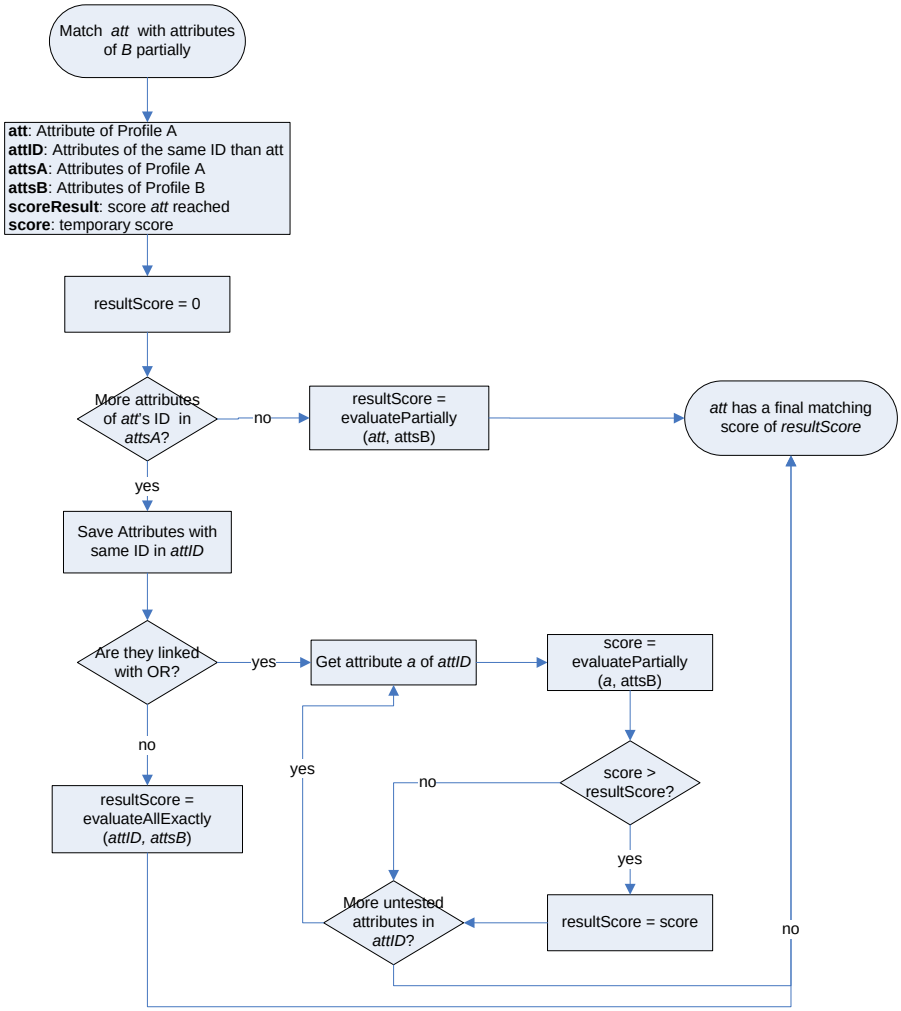


Figure 5-3
Attribute matching in a partial match

5.3.2 Partial Match

The partial match between a profile *A* and a profile *B* shown in figure 5-3 proceeds in a similar way than the exact match. There is also a distinction between AND and OR linked attributes. The difference to the previous algorithm lies in the evaluation of the attributes. Compared to an exact match, the partial match invokes `evaluatePartially` and `evaluateAllPartially`. Instead

of a `boolean` return value, those methods give back the score the attribute reached.
The method `evaluatePartially(Attribute attribute, Vector attributes)` returns the score *attribute* reached when matched with the ones in list *attributes*. For `String` typed attributes, the score is either 0 or the maximum score, it is not possible to calculate a matching level.
For attributes of type *Long* and *Integer* however it makes sense to compare their values and to express an eventual deviation in a reduced score. In case an attribute in not defined in *B*, it receives score 0. With `evaluateAllPartially` the AND linked attributes are evaluated.

5.4 Discovery Module

The module is composed of the classes `BTClient`, `ServiceCache`, `ServiceCacheEntry` and `DiscoveryModule`, where `DiscoveryModule` is the only one accessible from outside the BlueFramework.
When an application starts a discovery by a call of `startDiscovery()`, the thread in `BluetoothClient` starts running. It scans the environment for devices offering the desired service. Once one or more devices have been located, their *connectionURL*s are passed to the application over a shared object which is a list in the class `DiscoveryModule`. To make sure, both threads do not access this list simultaneously, the threads have to lock it when reading or writing its content. The discovery thread notifies the application thread when new *connectionURLs* are ready and the application thread notifies the discovery thread when the *connectionURLs* have been retrieved.
The functionality of each class involved in discovery process are explained in the following sections.

5.4.1 Service Cache

The service cache remembers discovered services for the time period defined by the argument `cacheTimeout` in `DiscoveryModule`. A `ServiceCacheEntry` represents a service in the cache. It consists of a `String` holding the connectionURL of the service and the date the cache entry expires. If the number of cache entries exceeds the `cacheSize`, the expired entries will be deleted. If none of the entries has expired, the oldest ones will be replaced.

5.4.2 Bluetooth Client

In every Bluetooth communication a server and a client are involved. A Bluetooth server runs a service which is made available to clients. The class `BTClient` takes on the client role in the discovery procedure. It is assumed that a server listening for connection requests is provided by the application using the framework.

The Bluetooth specification separates discovery of devices and discovery of services in two proceeding processes. A device discovery starts with the issue of an inquiry request to which other devices in the area respond with their Bluetooth address. The Bluetooth address is a 6-byte unique identifier assigned to each Bluetooth device by the manufacturer. `DiscoveryListener` and `DiscoveryAgent` provided by JSR-82 are important in this context. The following lines start the search for Bluetooth devices in range:

```
LocalDevice device = LocalDevice.getLocalDevice();
DiscoveryAgent agent = device.getDiscoveryAgent();
agent.startInquiry(DiscoveryAgent.GIAC, btClient);
```

When a new device is found, the system calls the method `deviceDiscovered()` and `inquiryCompleted()` at the end of the device inquiry process. Once a device has been discovered, the service discovery process asks a device whether it has the desired service to offer using the following method:

```
discoveryAgent.searchService(attrSet, uuidSet, rd, this);
```

The first two arguments describe the service requested, `rd` refers to the device inquired for services and `this` represents the `DiscoveryListener` used to handle service search events. For every device offering the desired service, `servicesDiscovered()` is called to receive a `ServiceRecord` to retrieve the connectionURL from. To verify if the service has been discovered lately, the client calls `ServiceCache.containsValidService(String service)`. If the service is known, the client continues the discovery. In case the service does not appear in the cache, the connection URL will be added to the service cache by calling the method `ServiceCache.addCacheEntry(entry)` and `DiscoveryModule.setFoundServices(urls)` to notify the module that the service discovery has been successful. When the search is over the method `serviceSearchCompleted()`is invoked.

5.5 Communication Module

The main classes of the communication module and their dependencies are illustrated in Figure 5-4. The interface `Communicator` forms the main part of the module. It is implemented by `BluetoothCommunicator` and `SMSCommunicator`. The advantages of this structure are:

- Abstraction from underlying technology. API and ExceptionHandling are similar for Bluetooth as well as for SMS based communication.
- The communication module can easily be extended by a third communication technology.

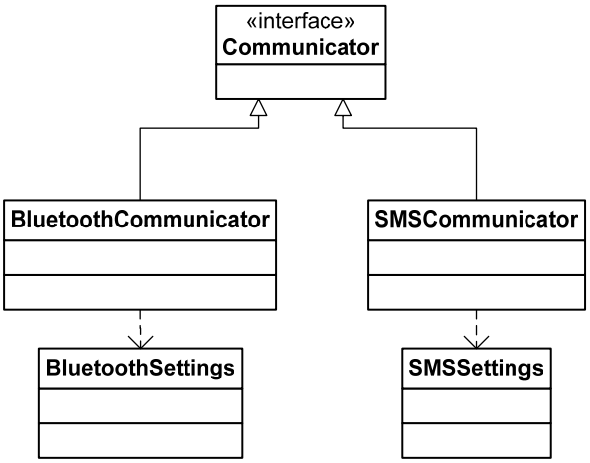


Figure 5-4
Classes in the communication module

5.5.1 Data Transfer Scheme

As already mentioned in Chapter 5.2, `Profiles` and `Attributes` implement the interface `Serializable`, which enables objects to be serialised and deserialised easily. The communication module takes advantage of the fact that the communication is stream based for both technologies. The first `int` written to or read from a stream stands for the type of data which will be transferred. The data followed by this definition depends on the object sent or received. The module makes us of a `DataInputStream` to read data and of a `DataOutputStream` to write data into a stream. These classes provide serialisation and deserialisation of primitive Java data types.

5.6 LogModule

The module for logging has been implemented in the classes `Log`, `LogTarget`, `LogStatement` `FileAccess`. Their functionalities are as follows:

- `Log`
  The main class accessible from outside the framework. It acts as a container storing the registered log targets. When a method to write to or to read from a log is called, it looks up the corresponding `LogTarget` and invokes the appropriate methods.
- `LogTarget`

The actual methods for writing and reading from a data repository are implemented in this class. It provides method for selective reading.

- `LogStatement`
  Object holding all information about a log entry: the date it was written, the class in which it was written, the level and the statement itself. In order to write a `LogStatement` easily into a recordstore or a file as well as to read it again from there, the class implements the BlueFramework's interface `Serializable`. One problem to handle in the context of a file `LogTarget` is the conversion of a `String` into a `Date` object. As J2ME offers only a reduced set of APIs, this conversion has been omitted. The class `LogStatement` provides the method `getDateFromString(String date)` to accomplish the conversion.
- `FileAccess`
  The `FileAccess` class provides methods to access the file system of a mobile phone if one is availabe. The API specifying this functionality is provided in an *optional package* [9].

6

BluePinboard

BluePinboard is a pin board application for Bluetooth enabled mobile phones. This application is intended to validate the BlueFramework functionality. It is programmed with the modules offered by BlueFramework.

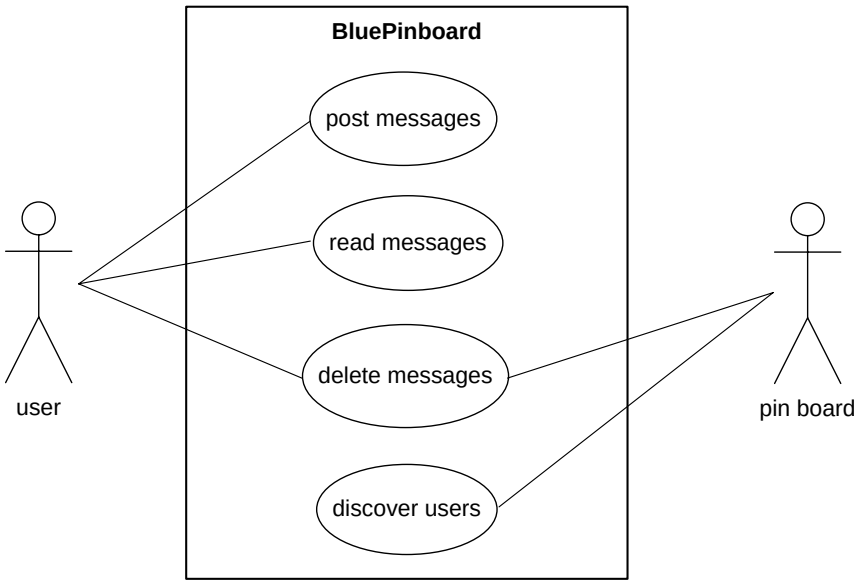


Figure 6-1
Use cases BluePinboard

6.1 Use Cases

In the BluePinboard application two different kind of Bluetooth devices interact: pin boards and users. Users post messages on the pin board and read messages from the pinboard. The message writer can define the expiration date of a message and restrict the readability to a group. With BluePinboard, the user has the possibility to retrieve only messages he is really interested in. As shown in figure 6-1, the application has the following use case scenarios:

6.1.1 Post a message to the pin board

A pin board is described by a pin board profile. The profile specifies which kind of messages can be attached and where the pin board is located. The user stores the messages ready for upload on his mobile phone in the upload list. When the pin board and the user device are close, the pin board checks if the user has messages in his upload list. To prevent from uploading the same message twice, the user device remembers the date the last update took place. On a subsequent upload, only messages created after this date will be transferred.

6.1.2 Read a message from the pin board

A user registers in a profile in which topics he is interested in and on which pin board locations he usually passes by. This profile is stored on his mobile phone. Every message attached to the pin board is also described by a message profile, saving the meta information of the message. Each time the user is in front of a pin board, the pinboard verifies if the user's profile matches the profile of one of its messages. If so, the message is downloaded to the users device. This interaction is based on a Bluetooth connection. As in the upload scenario it has to be made sure that the same message is not transferred twice. For this reason the pin board device remembers the date it encountered the user the last time for message download. The next time the user is in the pin boards area, only messages posted after this date are considered for exchange.

6.1.3 Delete messages

To reduce the risk of memory overflow, the messages saved on the pin board as well as on the user device have to be deleted once they expire. Pin board and user device store a certain number of messages and their profiles, which cannot be exceeded . If attempted to add another message after the maximum number has already been reached, the messages will be deleted in this order:

- Deletion of all expired messages and the corresponding profiles
- If none of the messages has expired, delete the oldest one and its profile

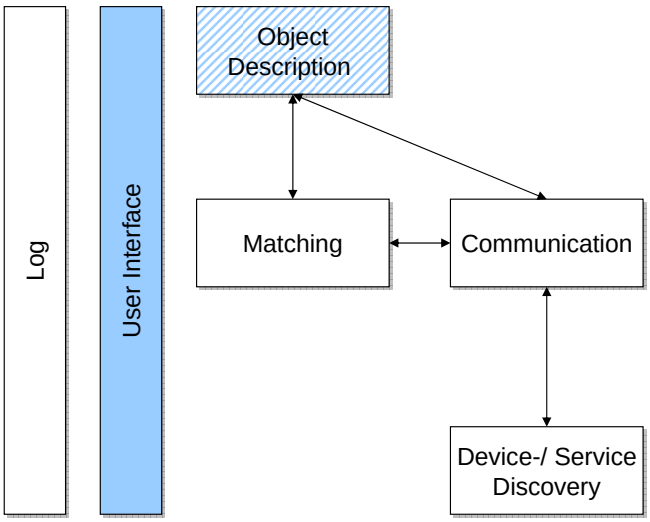


Figure 6-2
Modular design BluePinboard

6.1.4 Discover a user

As the application consists of two different kind of devices, both of them could be responsible for discovering other devices in the area. Because a user moves and not necessarily wants to discover a pin board all the time, it is easier for the fixed pin boards to discover the users.

6.2 Architecture

As mentioned in the previous section BluePinboard distinguishes two different types of devices: pin boards and users. From a technical point of view, it would be possible for a device to be a pin board and a user at the same time. Because a pin boards contains location related information, the BluePinboard application is divided into a user and a pin board MIDlet.
The following sections focus on the use of BlueFramework in BluePinboard. For a detailed description of each of the modules, refer to chapter 4.

6.2.1 User Interface

As there is a distinction between pin board and user device, a user interface for each kind of device has to be designed. Figure 6-3 gives an overview the screens in the pin board user interface. The user interface of the user device is shown in Figure 6-4.

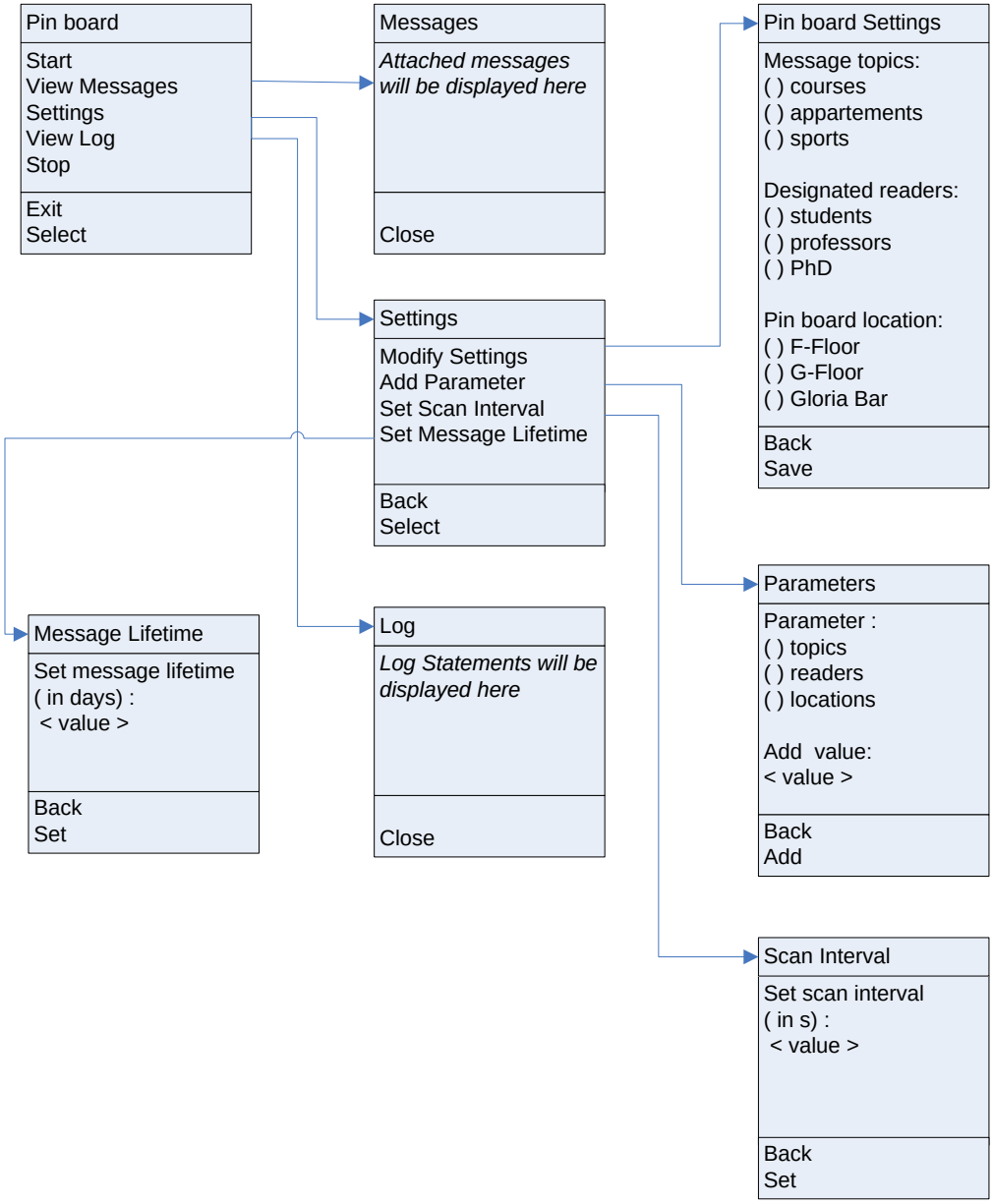


Figure 6-3
User interface pin board device

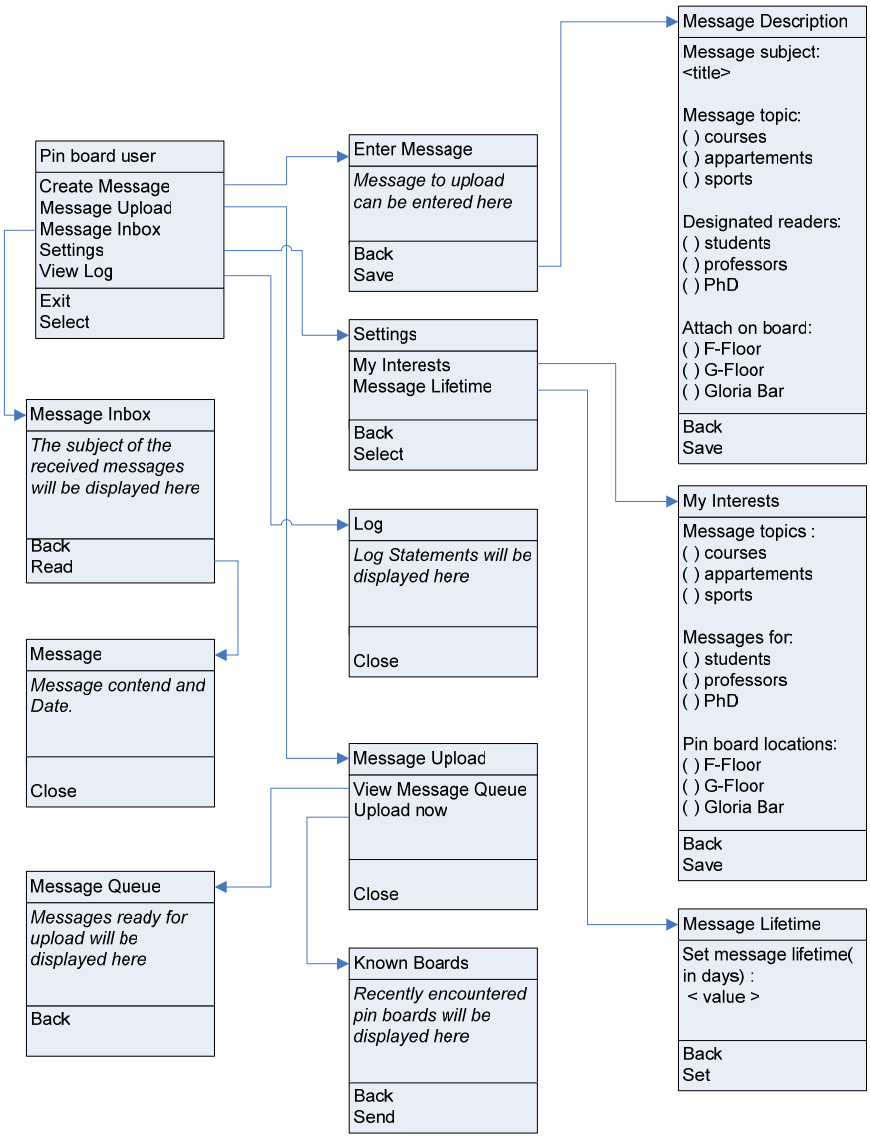


Figure 6-4
User interface user device

6.2.2 Object Description

In order to match messages with the user's interests and pin board definitions, different profiles are necessary:

- **Message profile** to store the meta information of a message.
- **User profile** to remember the users interests.
- **Pin board profile** to describe the pin board, which messages it has attached and where it is located.

Attribute ID	Attribute type	Attribute values
topic	String	[courses, events, apartments]
readers	String	[students, PhD students, professors]
location	String	[F-Floor, F-Floor, Gloriabar]
creation date	Long	*Long representation of creation date*
upload date	Long	*Long representation of upload Date*
expiry date	Long	*Long representation of expiry Date*
writer	String	*Bluetooth address of writer*

Table 6-1: possible attributes

All of the three profiles consist of the same attributes. Initially, BluePinboard uses the attributes listed in Table 6-1 in the above mentioned profiles. When a user creates a new message, he can describe its content with the attributes *topic*, *readers* and *location*. All other attributes are added by the application and do not depend on the users input. The attribute *creationDate* for instance refers to the date, the message was written, *uploadDate* remembers the date the message was attached on the pin board and expiry date holds the date, the message will expire. The attribute *writer* stores the Bluetooth address of the device, on which the message was written. This attribute is used to make sure, a user device does not receive its own messages back.
As shown in pin board user interface in Figure 6-3, there is a choice *add parameter* in the settings menu. Through this menu, a pin board administrator is allowed to add new attribute values. He could for example add a new message topic "sports".

6.2.3 Matching

Matching of profiles is done in two situations:

1. **Message upload**
   To guarantee that only messages which are conform to the pin boards description will be attached, the message profile and the pin board profile have to be compared using the matching module.
2. **Message download**
   As the message upload the message download process includes a matching

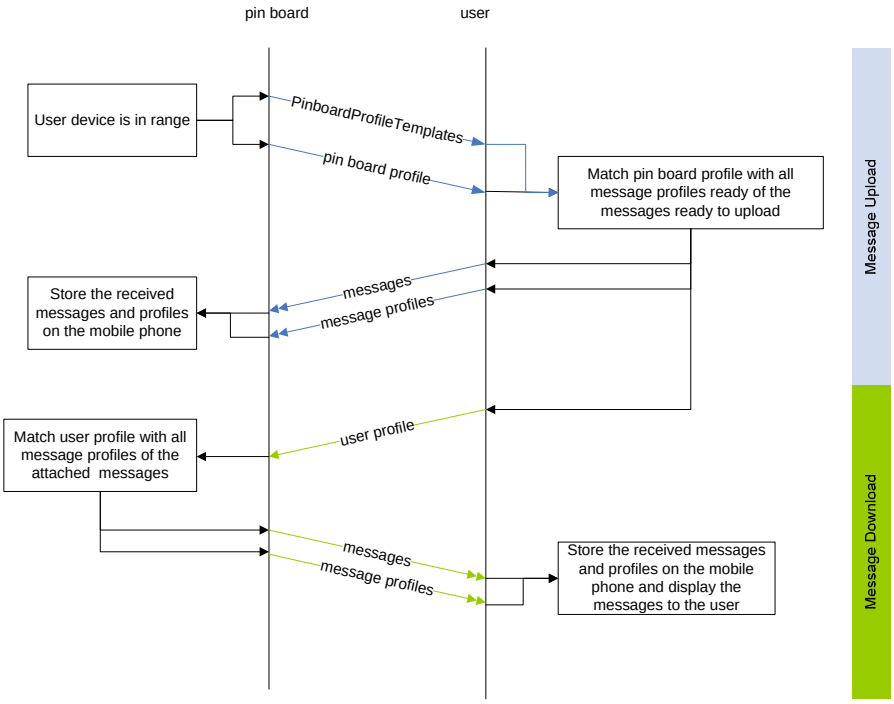


Figure 6-5
Message exchange in BluePinboard

as well. In this case, the user profile is matched with the profile of all attached messages. It is necessary to assure that the user receives only the messages of his interests.

Because a pin board profile and a user profile has the same structure, the same matching algorithm can be applied to both situations. An exact match will be sufficient for that purpose.

6.2.4 Device Discovery

Pin board devices scan their environment for user devices. The discovery module will take care of this task. With each user device within proximity, a message exchange is initiated. Already discovered devices are remembered in the cache, which prevents from unnecessary data transfer to the same device after a short time period.

6.2.5 Communication

The diagram in Figure 6-5 demonstrates the exchange of messages. After a pin board has discovered a user device, the data exchange starts with the message upload followed by the download. The pin board sends its `ProfileTemplateAttributes` to the discovered user device. On reception, the user device compares the received attributes with the ones contained in its `ProfileTemplate`. Unknown `ProfileTemplateAttributes`are added to the `ProfileTemplate`. The `ProfileTemplateAttributes` are followed by the pin board profile which describes the messages allowed to be attached on the pin board. The user device loads all messages ready for upload and matches their profiles with the received pin board profile. The matching messages are sent back to the pin board device together with the corresponding message profile. The pin board stores the messages and the profiles in a recordstore.

After all upload messages have been transferred, the user device sends the user profile. This is the beginning of the message download. The pin board matches the user's profile with all message profile of the attached messages. The ones matching the user's interests are sent back to the user. The user device stores the received messages and profiles and displays the messages to the user.

Both the pin board and the user device remember the date the last message exchange took place. When the devices are in reach for the next time, only messages attached or created after this date will be exchanged.

6.2.6 Log

The BluePinboard application uses a log to keep track of the programm's state as well as application failures during execution. In case communication fails for instance the cause will be written to the log. As log target BluePinboard uses the Recordstore Management System (RMS) provided by MIDP.

6.3 Implementation

Figure 6-6 gives an overview over the classes which implement the BluePinboard. The blue shaded ones represent classes which were generated by means of the provided development tools; the white ones are the ones which had to be implemented.
`UserMIDlet` is the MIDP application running on the user device. Its counter part it the `BoardMIDlet`, the application running on the pin board device. The `BoardMIDlet` creates a new instance of the discovery module and runs a discovery for user devices in the area. Both MIDlets include a user interface which allows the user to navigate through the application. The content of this user interface is displayed in Figure **??** and **??**. Classes responsible for the connection between a user and a pin board are `ConnectionToBoard` and

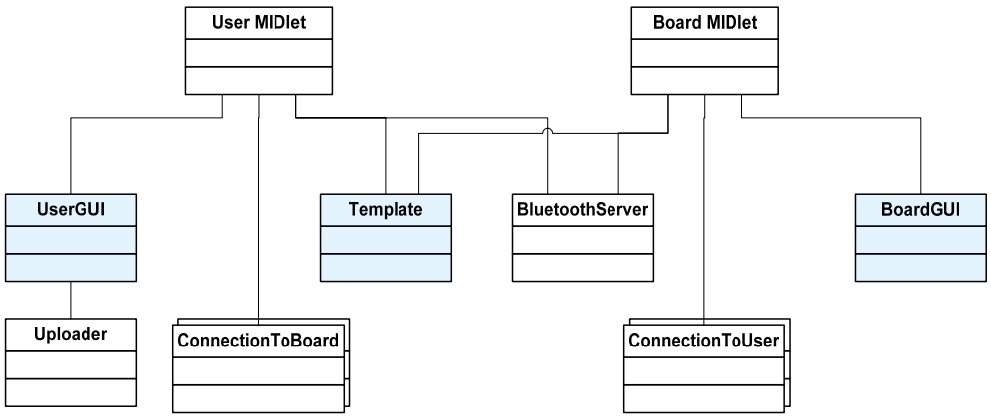


Figure 6-6
Survey BlueFramework

`ConnectionToUser`. The establish a Bluetooth connection and run a thread to exchange messages.

7

Evaluation of BlueFramework

The goal of this final chapter is to evaluate the functionality provided by BlueFramework. An evaluation is not trivial since no reference application is available. Ideally the BluePinboard application should have been implemented twice: once using the modules provided by BlueFramework and once without the framework. Only this situation would allow a quantitative evaluation.

Nevertheless, the following points express the advantages of application development by means of BlueFramework in numbers. Based on the experience gained in the development of BluePinboard, the following facts point out that complexity and time of application development can be reduced.

- **Number of generated classes**
  The BluePinboard application consists of nine classes. As shown in the class survey of BlueFramework introduced in Figure 6-6, three of those have been generated by means of a development tool offered by BlueFramework. The programming effort can therefore be reduced by one third.
- **Number of Input- and OutputStreams**
  In BluePinboard neither an InputStream nor an OutputStream has to be opened and closed manually. Methods for data serialisation and deserialisation are provided by BlueFramework. An application developer has not to care about object serialisation.
- **Number of Threads**
  With the use of BlueFramework, the BluePinboard application could be reduced by one thread. The thread for discovery is included and controlled by the framework. BluePinboard does not have to take care about controlling the discovery thread. Once a discovery has been initiated, the application will be notified if a desired device is in the area.

Furthermore, application programming by means of BlueFramework has the following qualitative advantages:

- **Reduction of development time**
  As a lot of functionality is already provided by BlueFramework, application development time can obviously be reduced by a significant factor. For a proper quantitative evaluation of implementation time, the same application should have been implemented twice, once with and without using the framework as stated above.
- **Ease of implementation**
  There is no need to get familiar with J2ME, MIDP and every optional package including Bluetooth communication in detail. It is sufficient to know the methods included in the libary and the development tools provided by BlueFramework to develop a mobile application.
- **Structured code**
  Frequently used functions for mobile applications are already implemented in BlueFramework. Application developers can access the modules of BlueFramework instead of reimplementing the same functions for every new J2ME applicaton. This also leads to a better structured and less complex code.
- **Exception handling**
  Exception handling is easier when the module of BlueFramework are used because most exceptions are already handled by BlueFramework.

8

Conclusion

8.1 Results

Former thesis developed lately within the scope of the BlueStar project have been analysed. Common functional blocks have been identified and described in a detailed concept. In order to provide a general library of functions to be used for future application development, the framework has been extended with additional modules.

The functionality of the framework has been validated with the development of BluePinboard, a pin board application intended for Bluetooth enabled mobile phones. All modules needed for the development of BluePinboard have been implemented. BluePinboard was tested successfully in the emulator [14] and on Nokia mobile phones [15].

Testing of BluePinboard on mobile phones has shown that the applications do not necessarily produce the same result on mobile phones as in an emulator. In addition to the limited resources available, a mobile phone is influenced by other Bluetooth devices in the area. If too many Bluetooth devices scan the environment at the same time, it might be possible that neither of them discovers a device although they are physically close.

8.2 FutureWork

Security concepts

The security issues discussed in this thesis have not been implemented yet. Data encryption was not necessary in the context of BluePinboard because no sensitive data had to be transferred. A digital pin board is not exposed to more threats

than a public pin boards with attached paper notices. In a future step, the BlueFramework security module has to be implemented.

Routing

Also routing was not of major importance in the BluePinboard application. The routing module still has to be implemented. This module will be of higher importance, once additional communication technologies have been added.

New communication technologies

Because BlueFramework follows a modular design it is easily extensible with new functionalites. The communication module for example could be extended with a third technology. It the mobile phone industry continues evolving they way it actually does, every mobile phone will soon be able to transmit data over wireless LAN and BlueFramework could provide the methods for transferring data over this technology.

BlueLocation

Other existing theses which were developed within the BlueStar project could be added to the BlueFramework. BlueLocation for instance provides means to determine the distance between two devices. It would be an advantage to include this location functionality in future application for mobile devices.

A

Assignment

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Sommer 2005

Masterarbeit

für

Nicole Hatt

Tutor: Matthias Bossardt
Co-Tutors: Vincent Lenders, Martin May

Ausgabe: 4.4.2005
Abgabe: 2.10.2005

BlueFramework - Application Framework for Bluetooth Enabled Mobile Phones

1 Introduction

Mobile phones of the latest generation feature a Java virtual machine (J2ME) and a Bluetooth stack for short range communications. In the context of the Blue* project [1], a number of networking applications for Bluetooth-based ad-hoc networks have been developed [3, 4, 2]. We noticed that such applications share a lot of common functionality. As consequence, the development of new applications could be simplified, if a framework implementing the common functional blocks and providing the corresponding development tools were available.

2 Assignment

2.1 Objectives

The goal of this thesis is to design and implement a framework for Bluetooth-based networking applications. The framework consists of a Java library implementing commonly used functional blocks and (possibly) a set of development tools. To validate this work, a new application,

BluePinboard, has to be implemented based on the proposed framework. BluePinboard consists of two different types of Bluetooth-enabled devices, one implements the pinboards, whereas the other type of devices is carried by the users. BluePinboard should work similarly to public pinboards like the ones found in the corridors of ETH.

2.2 Tasks

- Get familiar with the J2ME and related APIs provided by the latest models of mobile phones. In particular study the Bluetooth-API.
- Get familiar with the former semester theses on BlueDating [2] and Bluetella [3, 4].
- Identify common functional blocks among the different applications.
- Specify the functionality and interfaces of functional blocks for the framework.
- Implement the framework.
- Specify and design the BluePinboard application.
- Implement the BluePinboard application using the blocks of the framework.
- Define and set up a demonstrator of the BluePinboard. Preferably the demonstration should run on currently available mobile phones.
- Document your work in a detailed and comprehensive way. We suggest you to continually update your documentation. New concepts and investigated variants must be described. Decisions for a particular variant must be justified.

3 Deliverables and Organisation

- Generally students and advisor meet on a weekly basis to discuss progress of work and next steps. If problems/questions arise that can not be solved independently, the students may contact the advisor anytime.
- At the end of the third week, a detailed time schedule of the semester thesis must be given and discussed with the advisor.
- At half time of the semester thesis, a short discussion of 15 minutes with the professor and the advisor will take place. The student has to talk about the major aspects of the ongoing work. At this point, the student should already have a preliminary version of the table of contents of the final report. This preliminary version should be brought along to the short discussion.
- At the end of the semester thesis, a presentation of 15 minutes must be given during the TIK or the communication systems group meeting. It should give an overview as well as the most important details of the work. Furthermore, it should include a small demo of the project.

- The final report may be written in English or German. It must contain a summary written in either English or German, the assignment and the time schedule. Its structure should include an introduction, an analysis of related work, and a complete documentation of all used software tools. Related work must be referenced correctly. See http://www.tik.ee.ethz.ch/flury/tips.html for more tips. Three copies of the final report must be delivered to TIK.
- Documentation and software must be delivered on a CDROM.

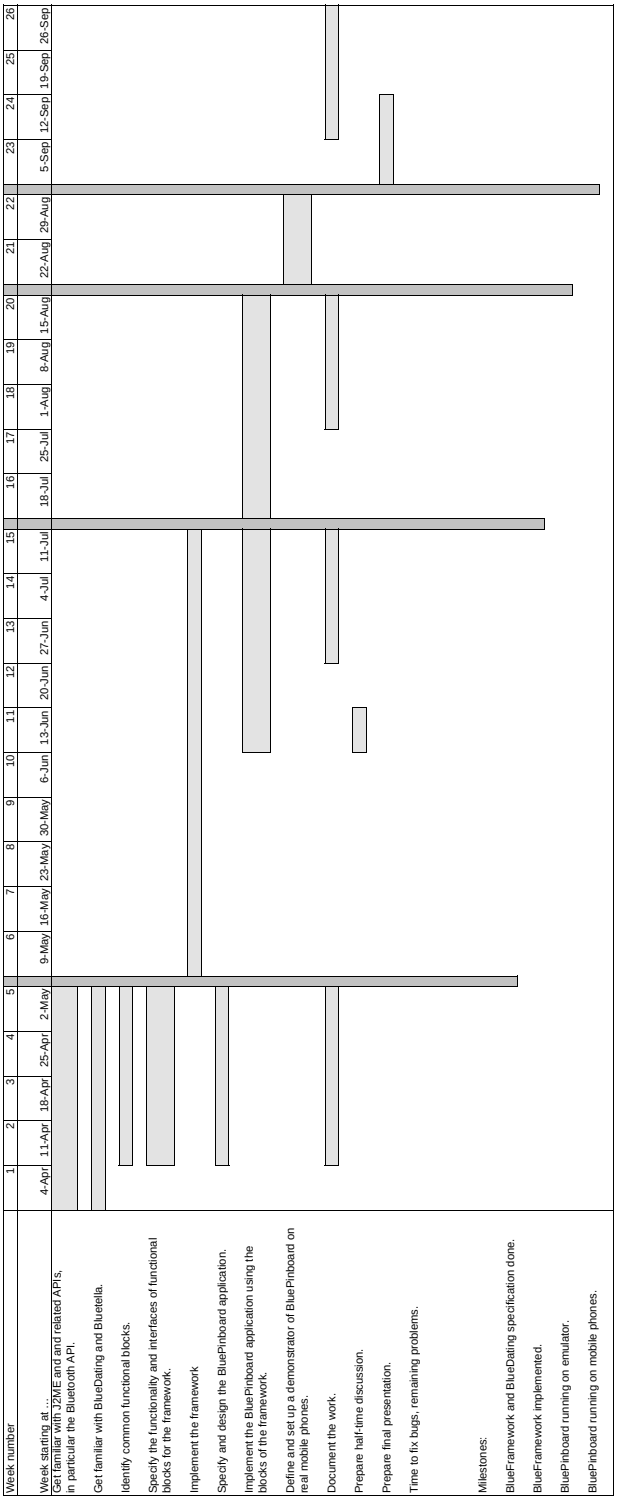
Literatur

[1] The BlueStar Project. `http://www.csg.ethz.ch/research/running/Blue_star`.
[2] Christian Braun and Sandro Schifferle. BlueDating - A Dating Application for Bluetooth Enabled Mobile Phones, February 2005.
[3] Ganymed Stanek. Bluetella: A Java application for new mobile phones, July 2003.
[4] Andreas Weibel and Lukas Winterhalder. Bluetella: A File Sharing Application for Bluetooth Enabled Mobile Phones, February 2005.

Zürich, den 3.5.2005

B

Time schedule


Week number
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
Week starting at ...
4-Apr 11-Apr 18-Apr 25-Apr 2-May 9-May 16-May 23-May 30-May 6-Jun 13-Jun 20-Jun 27-Jun 4-Jul 11-Jul 18-Jul 25-Jul 1-Aug 8-Aug 15-Aug 22-Aug 29-Aug 5-Sep 12-Sep 19-Sep 26-Sep
Get familiar with J2ME and and related APIs, in particular the Bluetooth API.
Get familiar with BlueDating and Bluetella.
Identify common functional blocks.
Specify the functionality and interfaces of functional blocks for the framework.
Implement the framework
Specify and design the BluePinboard application.
Implement the BluePinboard application using the blocks of the framework.
Define and set up a demonstrator of BluePinboard on real mobile phones.
Document the work.
Prepare half-time discussion.
Prepare final presentation.
Time to fix bugs, remaining problems.
Milestones:
BlueFramework and BlueDating specification done.
BlueFramework implemented.
BluePinboard running on emulator.
BluePinboard running on mobile phones.

C

XML Schema

C.1 ProfileTemplateSchema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="ProfileTemplate">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ProfileTemplateAttribute" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:choice>
                            <xs:element name="value" minOccurs="0"/>
                            <xs:element name="range">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="fromValue"/>
                                        <xs:element name="toValue"/>
                                    </xs:sequence>
                                </xs:complexType>
                            </xs:element>
                        </xs:choice>
                        <xs:attribute name="ID" type="xs:string" use="required"/>
                        <xs:attribute name="type" type="xs:string" use="required"/>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

C.2 GUISchema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="GUI">
        <xs:annotation>
            <xs:documentation>Comment describing your root element</xs:documentation>
        </xs:annotation>
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Screen" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:choice>
                            <xs:element name="List">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="Item" minOccurs="0" maxOccurs="unbounded">
                                            <xs:complexType>
                                                <xs:sequence>
                                                    <xs:element name="StringPart"/>
                                                    <xs:element name="ImagePart" type="xs:string" minOccurs="0"/>
                                                    <xs:element name="Command" minOccurs="0" maxOccurs="unbounded">
                                                        <xs:complexType>
                                                            <xs:sequence>
                                                                <xs:element name="Action" type="xs:string"/>
                                                                <xs:element name="NextScreen" type="xs:string"/>
                                                            </xs:sequence>
                                                            <xs:attribute name="objectname" type="xs:string" use="required"/>
                                                            <xs:attribute name="label" type="xs:string" use="required"/>
                                                            <xs:attribute name="type" type="xs:string" use="required"/>
                                                            <xs:attribute name="priority" type="xs:string" use="required"/>
                                                        </xs:complexType>
                                                    </xs:element>
                                                </xs:sequence>
                                            </xs:complexType>
                                        </xs:element>
                                        <xs:element name="Ticker" minOccurs="0">
                                            <xs:complexType>
                                                <xs:attribute name="objectname" type="xs:string" use="optional"/>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                                    <xs:attribute name="title" type="xs:string" use="required"/>
                                    <xs:attribute name="type" type="xs:string" use="required"/>
                                </xs:complexType>
                            </xs:element>
                            <xs:element name="Form">
                                <xs:complexType>
                                    <xs:sequence>
                                        <xs:element name="ChoiceGroup" minOccurs="0" maxOccurs="unbounded">
                                            <xs:complexType>
                                                <xs:sequence>
                                                    <xs:element name="Choice" maxOccurs="unbounded">
                                                        <xs:complexType>
                                                            <xs:sequence>
                                                                <xs:element name="StringPart" type="xs:string" minOccurs="0"/>
                                                                <xs:element name="ImagePart" type="xs:string" minOccurs="0"/>
                                                            </xs:sequence>
                                                        </xs:complexType>
                                                    </xs:element>
                                                </xs:sequence>
                                                <xs:attribute name="objectname" type="xs:string" use="required"/>
                                                <xs:attribute name="label" type="xs:string" use="required"/>
                                                <xs:attribute name="type" type="xs:string" use="required"/>
                                            </xs:complexType>
                                        </xs:element>
                                        <xs:element name="TextField" minOccurs="0" maxOccurs="unbounded">
                                            <xs:complexType>
                                                <xs:attribute name="objectname" type="xs:string" use="required"/>
                                                <xs:attribute name="label" type="xs:string" use="required"/>
                                                <xs:attribute name="text" type="xs:string" use="optional"/>
                                                <xs:attribute name="maxSize" type="xs:string" use="required"/>
                                                <xs:attribute name="constraints" type="xs:string" use="required"/>
                                            </xs:complexType>
                                        </xs:element>
                                        <xs:element name="StringItem" minOccurs="0" maxOccurs="unbounded">
                                            <xs:complexType>
                                                <xs:attribute name="objectname" type="xs:string" use="required"/>
                                                <xs:attribute name="label" type="xs:string" use="required"/>
                                                <xs:attribute name="text" type="xs:string" use="required"/>
                                            </xs:complexType>
                                        </xs:element>
                                        <xs:element name="ImageItem" minOccurs="0" maxOccurs="unbounded">
                                            <xs:complexType>
```

```
                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                    <xs:attribute name="label" type="xs:string" use="required"/>
                    <xs:attribute name="image" type="xs:string" use="required"/>
                    <xs:attribute name="layout" type="xs:string" use="required"/>
                    <xs:attribute name="altText" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="DateField" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                    <xs:attribute name="label" type="xs:string" use="required"/>
                    <xs:attribute name="mode" type="xs:string" use="required"/>
                    <xs:attribute name="timeZone" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Gauge" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                    <xs:attribute name="label" type="xs:string" use="required"/>
                    <xs:attribute name="interactive" type="xs:string" use="required"/>
                    <xs:attribute name="maxValue" type="xs:string" use="required"/>
                    <xs:attribute name="initialValue" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Command" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="Action" type="xs:string" minOccurs="0"/>
                        <xs:element name="NextScreen" type="xs:string" minOccurs="0"/>
                    </xs:sequence>
                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                    <xs:attribute name="label" type="xs:string" use="required"/>
                    <xs:attribute name="type" type="xs:string" use="required"/>
                    <xs:attribute name="priority" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
            <xs:element name="Ticker" minOccurs="0">
                <xs:complexType>
                    <xs:attribute name="objectname" type="xs:string" use="optional"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="objectname" type="xs:string" use="required"/>
        <xs:attribute name="title" type="xs:string" use="required"/>
    </xs:complexType>
</xs:element>
<xs:element name="Alert">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Command" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="Action" type="xs:string" minOccurs="0"/>
                        <xs:element name="NextScreen" type="xs:string" minOccurs="0"/>
                    </xs:sequence>
                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                    <xs:attribute name="label" type="xs:string" use="required"/>
                    <xs:attribute name="type" type="xs:string" use="required"/>
                    <xs:attribute name="priority" type="xs:string" use="required"/>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
        <xs:attribute name="objectname" type="xs:string" use="required"/>
        <xs:attribute name="title" type="xs:string" use="required"/>
        <xs:attribute name="text" type="xs:string" use="optional"/>
        <xs:attribute name="image" type="xs:string" use="optional"/>
        <xs:attribute name="type" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>
<xs:element name="TextBox">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="Command" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="Action" type="xs:string" minOccurs="0"/>
                        <xs:element name="NextScreen" type="xs:string" minOccurs="0"/>
                    </xs:sequence>
                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                    <xs:attribute name="label" type="xs:string" use="required"/>
                    <xs:attribute name="type" type="xs:string" use="required"/>
                    <xs:attribute name="priority" type="xs:string" use="required"/>
                </xs:complexType>
```

```
                                        </xs:element>
                                        <xs:element name="Ticker" minOccurs="0">
                                            <xs:complexType>
                                                <xs:attribute name="objectname" type="xs:string" use="required"/>
                                            </xs:complexType>
                                        </xs:element>
                                    </xs:sequence>
                                    <xs:attribute name="objectname" type="xs:string" use="required"/>
                                    <xs:attribute name="title" type="xs:string" use="required"/>
                                    <xs:attribute name="text" type="xs:string" use="optional"/>
                                    <xs:attribute name="maxSize" type="xs:string" use="required"/>
                                    <xs:attribute name="constraints" type="xs:string" use="required"/>
                                </xs:complexType>
                            </xs:element>
                        </xs:choice>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

D

CD Content

FOLDER	CONTENT
`documentation\report\`	Includes the final documentation of the thesis
`documentation\presentation\`	Includes the slides of the final presentation
`Framework\dist\`	Contains the compiled version of BlueFramework (JAR and JAD files)
`Framework\nbproject\`	Contains the Netbeans project used to develop BlueFramework
`Framework\src\`	Contains the Java source code of BlueFramework
`Pinboard\dist\`	Contains the compiled version of BluePinboard (JAR and JAD files)
`Pinboard\nbproject\`	Contains the Netbeans project used to develop BluePinboard
`Pinboard\src\`	Contains the Java source code of BluePinboard
`Pinboard\XML\`	Contains the XML documents used for the develoment tools
`Tools\GUI\bin\`	Contains the compiled version of the user interface module
`Tools\GUI\src\`	Contains the source code of the user interface module
`Tools\GUI\XML\`	Contains the XMLSchema defining the XML input of the user module
`Tools\Template\bin\`	Contains the compiled version of the object description tool
`Tools\Template\src\`	Contains the source code of the object description tool
`Tools\Template\XML\`	Contains the XMLSchema defining the XML input of the object description tool

Bibliography

[1] The BlueStar project.
`http://www.csg.ethz.ch/research/projects/Blue_star`
(30.09.2005)

[2] Christian Braun und Sandro Schifferle. *BlueDating - A Dating Application For Bluetooth Enabled Mobile Phones*, Semester Thesis TIK, ETH Zürich, February 2005.

[3] Andreas Weibel und Lukas Winterhalter. *Bluetella - A File Sharing Application for Bluetooth Enabled Mobile Phones*, Semester Thesis TIK, ETH Zürich, February 2005.

[4] Ganymed Stanek. *Bluetella - A Java Application For New Mobile Phones*, Semester Thesis TIK, ETH Zürich, July 2003.

[5] Sun Microsystems. *Mobile Information Device Profile 2.0 (JSR-118)*
`http://www.jcp.org/en/jsr/detail?id=118` (30.09.2005)

[6] Sun Microsystems. *Connected Limited Device Configuration 1.1 - Specification (JSR-139)*
`http://www.jcp.org/en/jsr/detail?id=139` (30.09.2005)

[7] Sun Microsystems. *Wireless Messaging API WMA(JSR-120)*
`http://www.jcp.org/en/jsr/detail?id=128` (30.09.2005)

[8] Sun Microsystems. Java™ APIs for Bluetooth™ Wireless Technology (JSR-82)
`http://www.jcp.org/en/jsr/detail?id=82` (30.09.2005)

[9] Sun Microsystems. PDA Optional Packages for the J2ME™ Platform (JSR-75 File Connection)
`http://www.jcp.org/en/jsr/detail?id=75` (30.09.2005)

[10] Otto Kolsi. *MIDP 2.0 Security Enhancements*, Helsinki University of Technology, 2004.
http://doi.ieeecomputersociety.org/10.1109/HICSS.2004.1265679 (30.09.2005)

[11] The Legend of Bouncy Castle. *Bouncy Castle Cryptographic API*
http://www.bouncycastle.org (30.09.2005)

[12] Forum Nokia.*MIDP 2.0: Tutorial On Signed MIDlets*
forum.nokia.com/ndsCookieBuilder?fileParamID=5075 (30.09.2005)

[13] B. Hopkins, R. Anthony. *Bluetooth and Java*, a!Press, 2004.

[14] The Netbeans IDE
http://www.netbeans.org (30.09.2005)

[15] *Nokia 6630*
http://www.nokia.com/nokia/0,8764,58711,00.html (30.09.2005)