MIDP: OBEX API Developer’s Guide

Version 1.0; June 22, 2006
Copyright © 2006 Nokia Corporation. All rights reserved.

Nokia and Forum Nokia are registered trademarks of Nokia Corporation. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. Bluetooth is a registered trademark of Bluetooth SIG, Inc.

Other product and company names mentioned herein may be trademarks or trade names of their respective owners.

Disclaimer

The information in this document is provided “as is,” with no warranties whatsoever, including any warranty of merchantability, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. Furthermore, information provided in this document is preliminary, and may be changed substantially prior to final release. This document is provided for informational purposes only.

Nokia Corporation disclaims all liability, including liability for infringement of any proprietary rights, relating to implementation of information presented in this document. Nokia Corporation does not warrant or represent that such use will not infringe such rights.

Nokia Corporation retains the right to make changes to this specification at any time, without notice.

License

A license is hereby granted to download and print a copy of this specification for personal use only. No other license to any other intellectual property rights is granted herein.
Contents

1  Introduction..............................................................................................................................................6
2  Bluetooth OBEX and IrOBEX ................................................................................................................... 7
   2.1  Headers...............................................................................................................................................7
   2.2  Protocol .............................................................................................................................................8
   2.3  Bluetooth OBEX ...............................................................................................................................10
3  JSR-82 OBEX ........................................................................................................................................... 12
   3.1  Client connection...............................................................................................................................13
   3.2  Server connection...............................................................................................................................14
   3.3  Security .............................................................................................................................................14
4  BCExchanger MIDlet ..............................................................................................................................16
   4.1  General description ..........................................................................................................................16
   4.2  Scenario ..........................................................................................................................................16
   4.3  Design ..............................................................................................................................................17
   4.4  Development environment .............................................................................................................20
   4.5  MIDlet, Storage, and AddressBook classes ....................................................................................21
      4.5.1  BCExchangerMIDlet ..................................................................................................................21
      4.5.2  Storage ......................................................................................................................................21
      4.5.3  AddressBook ............................................................................................................................21
   4.6  Comm package ................................................................................................................................22
      4.6.1  ExchangeListener ......................................................................................................................22
      4.6.2  ExchangerComm .......................................................................................................................22
      4.6.3  ExchangerCommImpl ...............................................................................................................22
      4.6.4  ExchangerState ........................................................................................................................25
      4.6.5  ExchangerStateParent ..............................................................................................................25
      4.6.6  IdleState .....................................................................................................................................26
      4.6.7  InquiryState ...............................................................................................................................28
      4.6.8  ServiceDiscoveryState ..............................................................................................................30
      4.6.9  ReceiveState ..............................................................................................................................35
      4.6.10 SendState ..................................................................................................................................38
   4.7  UI package ......................................................................................................................................40
      4.7.1  AddressBookScreen ..................................................................................................................40
      4.7.2  AlertMessage ..............................................................................................................................41
      4.7.3  MainScreen ...............................................................................................................................41
      4.7.4  ProgressScreen ..........................................................................................................................41
      4.7.5  Screen .......................................................................................................................................41
      4.7.6  ServiceListScreen .....................................................................................................................41
## Change history

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>June 22, 2006</td>
<td>1.0</td>
<td>Initial document release</td>
</tr>
</tbody>
</table>
1 Introduction

This document covers the JSR-82 OBEX API [JSR-82], the optional part of the Java™ API for Bluetooth Wireless Technology. This optional part provides an API to Bluetooth Object Exchange Protocol, which is specified in the Bluetooth IrDA Interoperability Specification [BTOBEX].

The OBEX protocol is a compact binary protocol for exchanging complex data in a simple, efficient manner. Originally it was created for infrared connectivity and was used in the Infrared Data Association (IrDA) protocol stack. Bluetooth SIG [BTSIG] has defined the use of the OBEX protocol over Bluetooth connections in the IrDA Interoperability Specification [BTOBEX].

The JSR-82 OBEX part describes the Java API for the OBEX protocol over RFCOMM, IrDA, or TCP/IP link. This document focuses on the OBEX protocol usage over the RFCOMM link only.

Note that the basics of Bluetooth wireless technology are not covered in this document. In order to follow the document and understand the source code example, it is recommended that you become familiar with such concepts of Bluetooth wireless technology as inquiry, service discovery, and service registration, as well as the Bluetooth protocol stack and connection procedures. This document assumes that the reader has knowledge of the JSR-82 mandatory part (javax.bluetooth). Basic information on Bluetooth wireless technology and JSR-82 is given in the documents Introduction to Developing Networked MIDlets Using Bluetooth [BTINTRO] and JSR-82: Java APIs for Bluetooth Wireless Technology [JSR-82]. In order to understand the document, it is also good to know how the above-mentioned Bluetooth procedures are implemented using the JSR-82 API.

The MIDP: Bluetooth OBEX Example - Business Card Exchanger [EXAMPLE] example application is closely related to this document and discussed in Chapter 4. The goal of the application is the exchange of business cards using the JSR-82 OBEX protocol over Bluetooth wireless technology. Apart from that, the application provides the ability to select your own business card and exchange business cards with other users. The application also uses the JSR-75 PIM API. Knowledge of this API is not essential, but can be helpful in understanding the example. The documents Introduction To The PIM API [PIM] and JSR 75: PDA Optional Packages for the J2ME Platform [JSR-75] are the primary sources of information on the PIM API.

In general, to properly benefit from this document, you should be familiar with Java programming and the basics of MIDP programming. The Forum Nokia document MIDP 1.0: Introduction to MIDlet Programming [MIDPPROG] is a good starting point for getting to know the subject.
2 Bluetooth OBEX and IrOBEX

OBEX is a session-layer protocol originally developed by the Infrared Data Association (IrDA) as IrOBEX. Its purpose is to support sending and receiving objects in a simple and spontaneous manner. The Bluetooth specification states: “in the Bluetooth system, the purpose of the OBEX protocol is to enable the exchange of data objects”. For example, push of business cards or synchronizing calendars on multiple devices are handled with this protocol.

A major use of OBEX is a “Push” or “Pull” application. However, OBEX is not limited to quick connect-transfer-disconnect scenarios — it also allows sessions in which transfers take place over a period of time, maintaining the connection even when it is idle.

OBEX performs a function similar to HTTP; however, OBEX can work on the devices that cannot afford the substantial resources required for an HTTP server. It also targets devices with different usage models from the Web. The OBEX protocol, in its simplest form, is quite compact and requires a small amount of code to implement. It can reside on top of any reliable transport.

The OBEX session protocol allows the higher layers of the stack to work with logical elements of a higher layer of abstraction than that of the packet formats used by the transport protocols, for example, RFCOMM. As the IrDA OBEX specification states, this is ensured by two major elements of the OBEX protocol:

- A model for representing objects and information that describes the objects.
- A session protocol that provides a structure for the “conversation” between devices.

The first, the object model, carries information about the objects being sent and contains the objects themselves. The object model is built entirely with parsable headers, similar to the headers in HTTP. The second, the session protocol, structures the dialogue between two devices. The session protocol uses a binary packet-based client-server request-response model.

2.1 Headers

The headers allow transferring metadata together with the objects. By using headers OBEX peers can transmit different kind of information related to a transmitted object, session, etc. For example, the name of the object, its size, a description, and the type of object can be defined in the headers. According to the IrOBEX specification the header consists of two elements:

- HI, the header ID
- HV, the header value

HI, the header ID, is an unsigned one-byte integer that identifies what the header contains and how it is formatted. HV consists of one or more bytes in the format and meaning specified by HI. Table 1 provides information about the OBEX headers. Full information on each header can be found in IrOBEX.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC0</td>
<td>Count</td>
<td>Number of objects (used by Connect)</td>
</tr>
<tr>
<td>0x01</td>
<td>Name</td>
<td>Name of the object (often a file name)</td>
</tr>
<tr>
<td>0x42</td>
<td>Type</td>
<td>Type of object, e.g. text, html, binary, manufacturer-specific</td>
</tr>
<tr>
<td>0xC3</td>
<td>Length</td>
<td>The length of the object in bytes</td>
</tr>
<tr>
<td>ID</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>0x44</td>
<td>Time</td>
<td>Date/time stamp – ISO 8601 version - preferred</td>
</tr>
<tr>
<td>0xC4</td>
<td></td>
<td>Date/time stamp – 4 byte version (for compatibility only)</td>
</tr>
<tr>
<td>0x05</td>
<td>Description</td>
<td>Text description of the object</td>
</tr>
<tr>
<td>0x46</td>
<td>Target</td>
<td>Name of the service that an operation is targeted to</td>
</tr>
<tr>
<td>0x47</td>
<td>HTTP</td>
<td>An HTTP 1.x header</td>
</tr>
<tr>
<td>0x48</td>
<td>Body</td>
<td>A chunk of the object body</td>
</tr>
<tr>
<td>0x49</td>
<td>End of Body</td>
<td>The final chunk of the object body</td>
</tr>
<tr>
<td>0x4A</td>
<td>Who</td>
<td>Identifies the OBEX application, used to tell if it is talking to a peer</td>
</tr>
<tr>
<td>0xCB</td>
<td>Connection Id</td>
<td>An identifier used for OBEX connection multiplexing</td>
</tr>
<tr>
<td>0x4C</td>
<td>App. Parameters</td>
<td>Extended application request &amp; response information</td>
</tr>
<tr>
<td>0x4D</td>
<td>Auth. Challenge</td>
<td>Authentication digest-challenge</td>
</tr>
<tr>
<td>0x4E</td>
<td>Auth. Response</td>
<td>Authentication digest-response</td>
</tr>
<tr>
<td>0x4F</td>
<td>Object Class</td>
<td>OBEX Object class of object</td>
</tr>
<tr>
<td>0x10 to 0x2F</td>
<td></td>
<td>Reserved</td>
</tr>
<tr>
<td>0x30 to 0x3F</td>
<td></td>
<td>User–defined</td>
</tr>
</tbody>
</table>

Table 1: List of OBEX headers. Source: [IrOBEX]

Note that the OBEX specification denotes that all headers are optional.

### 2.2 Protocol

The OBEX protocol is a client-server protocol. The protocol defines the notion of the client, which sends a request, and the server, which replies with a response. The client and server exchange the requests and responses during a connection session. Table 2 provides the exact definitions of the OBEX terms.

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBEX Client</td>
<td>An OBEX Client is the entity that initiates the underlying transport connection to an OBEX server and initiates OBEX operations.</td>
</tr>
<tr>
<td>OBEX Server</td>
<td>An OBEX Server is the entity that responds to OBEX operations. The OBEX server waits for the OBEX client to initiate the underlying transport connection.</td>
</tr>
<tr>
<td>OBEX Connection</td>
<td>An OBEX Connection is a virtual binding between two applications or services. An OBEX connection is initiated by sending an OBEX CONNECT packet. Once a connection is established, all operations sent over the connection are interpreted in a continuous context.</td>
</tr>
<tr>
<td>Application</td>
<td>An OBEX application communicates using a proprietary method known only by the manufacturer. Such applications can only expect to be understood by exact peers. Alternatively, an application may be a service with proprietary extensions. In this case the application must know if it is communicating with a service or application peer.</td>
</tr>
</tbody>
</table>

Table 2: OBEX definitions. Source: [IrOBEX]
The OBEX protocol follows a client-server request-response paradigm for the conversation format. The request-response pair is referred to as an operation. The OBEX protocol defines two operations for controlling the connection, CONNECT and DISCONNECT. After the connection session is established, the client can perform various operations on the server. Two basic operations are the GET operation for downloading an object from the server and the PUT operation for uploading objects. During the connection session, these operations can be performed as many times as needed in any order.

The maximum size of the object that can be transmitted in the single PUT or GET operation is 65535 bytes. Maximum object size for the session is negotiated during the connection procedure. Both peers exchange their maximum allowed sizes and the smaller value is selected. When PUT and GET operations are requested to transmit large objects, the protocol handles transmission of the object in several packages. It helps to maintain good synchronization when requests and responses are broken into multiple OBEX packets that are limited to a size specified at connection time. Each request packet is acknowledged by the server with a response packet. The whole procedure is transparent to an application, so PUT or GET appears as one lengthy operation (for example, by using JSR-82 API an application developer can transmit large objects without bothering to split the objects into packages).

In order to cancel such lengthy operations an ABORT command is defined. The ABORT operation may come in the middle of a request-response sequence. If the ABORT command is invoked while running an operation, the data transmission is canceled.

Request and response general formats are shown in Table 3 and Table 4.

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Bytes 1, 2</th>
<th>Bytes 3 to n</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode</td>
<td>packet length</td>
<td>headers or request data</td>
</tr>
</tbody>
</table>

Table 3: General form of a request packet. Source: [IrOBEX]

<table>
<thead>
<tr>
<th>Byte 0</th>
<th>Bytes 1, 2</th>
<th>Bytes 3 to n</th>
</tr>
</thead>
<tbody>
<tr>
<td>response code</td>
<td>response length</td>
<td>response data</td>
</tr>
</tbody>
</table>

Table 4: General form of a response packet. Source: [IrOBEX]

Note: Each peer may implement both the client and the server, and thereby create a peer-to-peer relationship between applications by using a pair of OBEX sessions, one in each direction.

The aforementioned operations are the basic ones, but not the only ones defined in the OBEX protocol. The full list is given in Table 5.

<table>
<thead>
<tr>
<th>Opcode (w/high bit set)</th>
<th>Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80 (high bit always set)</td>
<td>Connect</td>
<td>Choose your partner, negotiate capabilities</td>
</tr>
<tr>
<td>0x81 (high bit always set)</td>
<td>Disconnect</td>
<td>Signal the end of the session</td>
</tr>
<tr>
<td>0x02 (0x82)</td>
<td>Put</td>
<td>Send an object</td>
</tr>
<tr>
<td>0x03 (0x83)</td>
<td>Get</td>
<td>Get an object</td>
</tr>
<tr>
<td>0x04 (0x84)</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>0x85 (high bit always set)</td>
<td>SetPath</td>
<td>Modifies the current path on the receiving side</td>
</tr>
<tr>
<td>0xFF (high bit always set)</td>
<td>Abort</td>
<td>Abort the current operation</td>
</tr>
</tbody>
</table>

1 High bit indicates the final packet of the request. More information is given in [IrOBEX]
<table>
<thead>
<tr>
<th>Opcode (w/high bit set)</th>
<th>Definition</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x06 to 0x0F</td>
<td>Reserved</td>
<td>Not to be used without extension to [IrOBEX] specification</td>
</tr>
<tr>
<td>0x10 to 0x1F</td>
<td>User definable</td>
<td>Use as you wish with peer application</td>
</tr>
</tbody>
</table>

Bits 5 and 6 are reserved and should be set to zero. Bit 7 of the opcode means Final packet of request.

Table 5: OBEX operations. Source: [IrOBEX]

More detailed definition of each operation can be found in [IrOBEX].

IrDA OBEX Exchange Protocol [IrOBEX] defines the application framework concept, which is built on top of the OBEX protocol. It ensures interoperability between devices. For example, it allows sending messages to the Inbox of a device.

2.3 Bluetooth OBEX

The Bluetooth specification reflects that the OBEX protocol is originated from IrDA. The OBEX protocol usage in Bluetooth wireless technology is defined in the specification called IrDA interoperability [BTOBEX]. This term does not mean that devices with Bluetooth wireless technology can communicate directly with IrDA devices, but rather it refers to protocols that enable applications to use either form of wireless communications. OBEX is based on the client-server model and is independent of the actual transport mechanism. This document covers only the case when RFCOMM is used as the transport layer for OBEX as Figure 1 illustrates, but the OBEX protocol can be used over TCP/IP or infrared as well.

![OBEX protocol in the Bluetooth stack](MIDP: OBEX API Developer's Guide)

The OBEX protocol is the basis for several profiles, which are meant to be commonly implemented on devices that employ the Bluetooth wireless technology. The profiles are:

- General Object Exchange Profile (GOEP) — the general profile that ensures generic interoperability for the application profiles using OBEX.
• File Transfer Profile — the profile that defines the ability to transfer a file considering the file system of both devices.

• Object Push Profile — the profile that allows a Bluetooth device to push an object to the Inbox of another Bluetooth device.

• Synchronization Profile — the profile that defines the requirements for the protocols and procedures to provide device-to-device synchronization of such information as phonebook, calendar, notes, etc.

All these profiles are based on the principle that certain objects are pushed to and pulled from a device, and aim to allow rapid communications and interoperability among portable devices. More information on the profiles can be found in the Bluetooth Specification [GOEP], [OBEXPUSH], [OBEXFTP], [OBEXSYNC].
3 JSR-82 OBEX

The JSR-82 API consists of two parts: the core Bluetooth API (javax.bluetooth) and the OBEX API (javax.obex). This document does not provide any information regarding the core part, but focuses on the OBEX part instead.

JSR-82 OBEX API uses the Generic Connection Framework (GCF) for networking (more details on GCF in [CLDC]). In practice it means that some classes and interfaces from javax.microedition.io package are used as illustrated in Figure 2.

![Diagram showing OBEX in the Generic Connection Framework. Source: [JSR-82]](image)

The GCF is used to establish the transport layer communication. Once the connection is established, peers can start exchanging OBEX commands. Chapter 2, “Bluetooth OBEX and IrOBEX,” has introduced the OBEX protocol operations. JSR-82 supports the following operations:

CONNECT, PUT, GET, SETPATH, ABORT, CREATE-EMPTY, PUT-DELETE, DISCONNECT

Also JSR-82 gives access to the following OBEX headers:

- **Count**
- **End of Body**
- **Name**
- **Who**
- **Type**
- **Connection ID**
- **Length**
- **Application Parameters**
- **Time**
- **Authentication Challenge**
- **Description**
- **Authentication Response**
- **Target**
- **Object Class**
HTTP
User defined
Body

The class `HeaderSet` is used to manipulate most of the headers. Exceptions are:

- Body and End of Body headers, which are accessible using the `Operation` class.
- Connection ID header, which is accessible using the `ClientSession` class.
- Authentication Challenge and Authentication Response headers, which are accessible using both the `HeaderSet` and the `Authenticator` classes.

3.1 Client connection

Firstly, the client establishes the Bluetooth transport layer connection by using the GCF `Connector.open()` with OBEX client URL. The JSR-82 specification defines three types of URL for using OBEX over RFCOMM (Bluetooth), TCP/IP, and IrDA. For example, an OBEX client connection URL over RFCOMM may look like:

```
btgoep://00A3920B2C22:12
```

- `btgoep` is the protocol (Bluetooth Generic Object Exchange Profile)
- `00A3920B2C22` is the Bluetooth Address of the target device
- `12` is the RFCOMM server channel identifier (each OBEX service has its own identifier)

This information is obtained during device discovery and service discovery. The OBEX client URL is similar to other JSR-82 Bluetooth URL formats and allows using various optional parameters, as defined in the JSR-82 specification. Device discovery and service discovery procedures are rather complicated processes and they are well described in the document *Introduction to Developing Networked Midlets Using Bluetooth* [BTINTRO].

In case an application wants to connect to a standard OBEX service (for example, send the message to the Inbox or browse the file system of the remote device), it must use the UUID defined in the Bluetooth SIG in Bluetooth Assigned Numbers [BTNUMB]. For example, the short UUID `0x1105` is used for searching the standard Inbox service and the short UUID `0x1106` for Bluetooth OBEX File Transfer Profile services.

**Note:** Nokia Prototype SDK 3.0 / 4.0 for Java™ Platform, Micro Edition used for developing the example application supports only OBEX over RFCOMM (`btgoep`).

The `Connector.open()` method returns a `ClientSession` object. The `ClientSession` interface provides methods for accessing the client part of the OBEX protocol, which is responsible for sending requests to a server; for example, in order to issue the OBEX CONNECT, call method `connect()` of the `ClientSession` interface. GET and PUT requests are issued by invoking `get()` and `put()` methods respectively.

In order to create headers, the method `createHeaderSet()` of the interface `ClientSession` is used. Using the returned `HeaderSet` object you can define a set of headers, which can be transmitted during such operations as CONNECT, PUT, GET, etc.

Methods `get()` or `put()` return an `Operation` object representing the current PUT or GET operation. In practice, using an `Operation` object a developer can access transmitted data via input and output streams. In case of lengthy operations the `Operation.abort()` method is used to stop the current operation. However, the `Operation.abort()` method is not affected by the connection session and new operations can be executed after the abortion.
Other OBEX commands are accessible via the `ClientSession` object in a similar manner and are thoroughly described in the JSR-82 specification.

### 3.2 Server connection

In order to create a server connection, the `GCF` class `Connector` is used as well. The `Connector.open()` method is called with a server URL as a parameter. The server URL for OBEX over RFCOMM may look like:

```
btgoep://localhost:ed495afe28ed11da94d900e08161165f
```

- `btgoep` – is the protocol (Bluetooth Generic Object Exchange Profile)
- `localhost` – identifies the server URL
- `ed495afe28ed11da94d900e08161165f` – is the service UUID

The OBEX server URL is similar to other JSR-82 Bluetooth URL formats and allows using various optional parameters, as defined in the JSR-82 specification. The service UUID is application specific. However, if an application wants to support a particular Bluetooth profile, it is responsible for adding the corresponding profile UUID to the service record. This process is described in *Introduction to Developing Networked MIDlets Using Bluetooth* [BTINTRO] and *JSR-82: Java APIs for Bluetooth Wireless Technology* [JSR-82]. All the UUID values for standard Bluetooth profiles are defined in the Bluetooth Assigned Numbers [BTNUMB].

The `Connector.open()` returns a `SessionNotifier` object. The method `acceptAndOpen()` of the `SessionNotifier` starts waiting for incoming transport layer connections. This follows the JSR-82 server connection scheme and allows modifications in the service records (more details in [JSR-82] and [BTPROG]). However, the difference between the OBEX connection and other types of JSR-82 connections (RFCOMM and L2CAP) is that the `acceptAndOpen()` method takes a `ServerRequestHandler` object as a parameter. The `ServerRequestHandler` class is a callback class that has methods corresponding to incoming OBEX requests.

After the transport layer connection is established, the `acceptAndOpen()` method returns a `Connection` object, which represents the connection to a single client. The server communicates with the remote peer by two means: the `Connection` object and the `ServerRequestHandler` callback methods that are called on incoming OBEX requests. A server application usually creates its own class that extends the `ServerRequestHandler` class and implements the methods corresponding to requests the OBEX server supports. For example, if the server supports only GET requests, it has to implement the `onGet()` method.

Methods `onGet()` and `onPut()` receive an `Operation` object as a parameter. The application uses an `Operation` object to read and write data as described above. After the request is handled, the method must return a response code. All response codes are defined in the `ResponseCode` class.

**Note:** A server application should not call the `Operation.abort()` method.

### 3.3 Security

The JSR-82 OBEX API complies with the MIDP 2.0 Security model and defines the following permissions and sensitive APIs.

<table>
<thead>
<tr>
<th>Permission</th>
<th>Permitted API calls</th>
<th>Function group</th>
</tr>
</thead>
</table>
Table 6: OBEX-related security permissions and API calls. Source: [MIDP 2.0].

The JSR-82 specification allows performing the OBEX authentication procedure of the remote peer. The authentication is based on the shared secret or password and follows a challenge-response scheme. The authentication was not used in the MIDP: Bluetooth OBEX Example - Business Card Exchanger [EXAMPLE] and is not described in this document. Details are given in [JSR-82].

Also basic Bluetooth security mechanisms (authorization, encryption, and authentication) are available for OBEX connections. They are accessible via URL parameters and API calls as described in [BTPROG].
4 BCExchanger MIDlet

4.1 General description

The OBEX protocol is mostly suitable for transfer of such information as business cards and calendar items. The purpose of the **MIDP: Bluetooth OBEX Example - Business Card Exchanger** [EXAMPLE] is to exchange business cards. A typical scenario of using the application goes as follows: When the MIDlet is started for the first time, the user selects his own card from the address book. When two users meet, they start the MIDlet, which exchanges the business cards. New business cards are saved in the phone book on each device.

**Note:** The device must have at least one contact (business card) in the phone book. Otherwise the MIDlet will not run correctly. The business cards are delivered directly to the application (not to the Inbox). Therefore, sending business cards to a device requires that the BCExchanger MIDlet is running on it.

4.2 Scenario

A high-level scenario of the BCExchanger application is illustrated in in Figure 3. Right after the application is started, the user selects his or her own card and proceeds to the main menu. If the card was already selected, the application starts directly from the main menu. While the user is in the main menu, it is possible to select another card, initiate card exchange, accept card exchange, or exit the application.

![High-level scenario of the BCExchanger application](image)

**Figure 3:** High-level scenario of the BCExchanger application.

Figure 4 illustrates the “Initiate card exchange” subscenario. When a user initiates the card exchange from the menu, the application starts a series of Bluetooth procedures: inquiry, service discovery, sending own card, and receiving and saving the remote business card. Also if several Bluetooth services declared by the BCExchanger MIDlet are discovered, the application displays the list of found services and asks the user to select one. Each procedure can be canceled or it can end with an error. Both cases should end the execution of the subscenario and return the application to the main menu.
Figure 4: Initiate card exchange subscenario

Figure 5 shows the Accept card exchange subscenario. When the application acts as an accepting peer, it simply receives the card, saves it, and sends the user’s own card. In addition, each activity can be canceled or it can fail. The party that starts the exchange operation receives a notification when the exchange operation goes successfully or fails.

Figure 5: Accept card exchange sub-scenario

These high-level scenarios are reflected in the design of the application.

4.3 Design

The BCExchanger architecture can be divided into several parts as shown in Figure 6. The MIDlet is the central architecture element. It represents the application execution and owns and manages other architecture components.

- The ui module is responsible for the user interface of the application. In practice, it contains classes implementing various MIDlet screens and transitions between screens.
The comm module is responsible for communication, that is, all operations related to Bluetooth OBEX.

The Address Book module provides access to the phone book of the device.

The Storage module is responsible for saving information about the card chosen as the own card.

In practice, the Storage module and the Address Book module contain only one class each. The class in the Storage module uses MIDP 2.0 Record Management System (RMS) and the class in the Address Book module uses the PIM API [JSR-75]. Both classes are quite straightforward in the implementation.

The modules ui and comm are more sophisticated. The comm module is responsible for various Bluetooth and OBEX operations. The ui module has to represent the sophisticated UI of the application. Each module contains the classes, which implement a state machine. The design pattern State [PATTERNS] has been used to implement both ui and comm modules.

Figure 7 illustrates the state machine diagram of the comm module. When the application initiates the exchange of cards, the normal state change flow is IdleState – InquiryState – ServiceDiscoveryState – SendState – ReceiveState – IdleState. However, each operation can fail or it can be canceled, which leads to the cancellation of the whole exchange process and returning to the IdleState.

2 The ui and comm modules must take all possible error situations into account and allow the cancellation of each procedure. This explains the complexity of both modules.
Accepting connection is implemented as serving OBEX GET and PUT requests and therefore it is done without a change of the state. However, note that the state machine has to be in the IdleState to serve the OBEX requests. In other words, the application is always ready to accept the card exchange from a remote device if it is not performing active exchange itself.

The ui module has a design similar to the one shown in Figure 8. Each state of the UI state machine represents a separate screen. When starting the application, the initial state is the AddressBookScreen (if it is the first application start) or the MainScreen (if the application has been started before). While the application is in the MainScreen state, it can display the AddressBookScreen if the user needs to change his or her own card, and it then transfers back to the MainScreen after successful selection of the card. Note that the application can exit only if it displays the MainScreen or the AddressBookScreen.

![UI module state diagram]

Figure 8: UI module state diagram

When the user initiates the exchange of the cards from the MainScreen, the state machine performs a change of various ProgressScreens as shown in Figure 8. Each ProgressScreen is transferred to the next ProgressScreen upon the successful completion, or to the MainScreen in case of errors or cancellations. The only exception is the ProgressScreen (service search) — if more than one service is found, the UI is changed to ServiceListScreen. It allows the user to select one single service.

In practice, one more screen is used in the application — AlertMessageScreen. However, it is used to display the results of an operation (for example, Business card is sent, Inquiry is canceled, etc) and is shown for a limited period of time (using MIDP 2.0 Alert class). Therefore is not considered as a state of the UI machine.

Examples of some screens are shown in Figure 9. The screen shots are from the Nokia Prototype SDK, S60 device.
4.4 Development environment

This MIDlet application has been developed using the following tools:

- Eclipse 3.1
- Nokia Prototype SDK 3.0 for Java™ Platform, Micro Edition and Nokia Prototype SDK 4.0 for Java™ Platform, Micro Edition
- Nokia Carbide.j 1.0
- Nokia Connectivity Framework (NCF) 1.2

In order to create the working environment, the applications should be installed in the order they are listed above. Installation and integration instructions are delivered with each product.

Eclipse is used as the main development environment and the Nokia Prototype SDK is used for compilation and execution of the example in the emulator. Nokia Carbide.j (former Nokia Developer’s Suite) provides integration of the Nokia Prototype SDK with Eclipse IDE, which allows executing the application in the emulator directly from the Eclipse IDE. Nokia Connectivity Framework provides an emulation of phone connectivity. It allows emulating Bluetooth connections between several emulators running on the same machine.

Note: It is possible to use the S60 2nd Edition SDK for Symbian OS, Supporting Feature Pack 3, for MIDP as well. However, in this case it will not be possible to use NCF for Bluetooth communication emulation. This example has also been tested on Nokia Prototype SDK 3.0 for Java™ Platform, Micro Edition, and Sun Java Wireless Toolkit 2.3 Beta.

Nokia Carbide.j 1.0 contains NCF Lite. It means that, strictly speaking, there is no need to separately install NCF 1.2. However, NCF 1.2, which is available as a separate application, contains a Full version and allows tracing Bluetooth packets sent between applications.

Tip: It is possible to use Bluetooth hardware with NCF 1.2. Please refer to the NCF documentation for more details.

After applications are properly installed, integrated, and configured, the BCExchanger example can be loaded into the Eclipse environment. Start several emulator instances by using the menu command Run | Run.
JAR files can be created and deployed on the device with Carbide.j.

**Tip:** On some devices (such as the Nokia N90) you need to change the suite settings to allow the application to use PIM data. Otherwise, you will receive an “Error during receiving” message when exchanging business cards. You can change the “Edit user data” value to “Ask every time” from Tools | Manager | MIDlet suite (BCExchanger) | Options | Suite settings.

### 4.5 MIDlet, Storage, and AddressBook classes

#### 4.5.1 BCExchangerMIDlet

This is the main application class which extends a MIDlet class. It serves as a parent to UI states (package bcexchanger.ui) and listens to communication events by implementing the ExchangeListener interface.

#### 4.5.2 Storage

This class contains methods for abstracting access to RMS. It is needed for saving the ID of the business card that the user has selected as his or her own card.

#### 4.5.3 AddressBook

This class provides an abstraction of the phone book to application. It implements access to the phone book of the device and provides information in the form the application needs.
This package contains classes which are related to Bluetooth OBEX communication. The classes from the `comm` package include classes realizing the communication state machine and other auxiliary classes and interfaces. Package `comm` contains the interfaces `ExchangeListener`, `ExchangerComm`, `ExchangeStateParent`, `IdleState`, `InquiryState`, `ReceiveState`, `SendState`, `ServiceDiscoveryState`, and the abstract class `ExchangerState`.

### 4.6.1 ExchangeListener

The `ExchangeListener` is a callback interface. By means of the interface methods the OBEX communication module signals to a listener about various networking events. This interface is implemented by the `BCExchangerMIDlet`.

### 4.6.2 ExchangerComm

This interface contains methods representing functionality of the OBEX communication module to other classes of the application. The OBEX communication module implemented by the classes from the `bcexchanger.comm` package realizes the exchange of the business card over JSR-82 OBEX API. The control of the module is abstracted in this interface.

The `BCExchangerMIDlet` calls the methods of this interface implemented by the `ExchangerCommImpl` class to control OBEX communication.

### 4.6.3 ExchangerCommImpl

This class is the central class in the OBEX communication module (package `bcexchanger.comm`). The class implements the interface `ExchangerComm` and realizes the methods for controlling the process of sending and receiving business cards. It is also a parent for states of the communication state machine. It keeps the current state of the state machine and implements the interface `ExchangeStateParent`. Using this interface, state classes can access the required functionality of the parent.

This class waits for an incoming Bluetooth connection in a separate thread and therefore this class implements the `Runnable` interface. The class also works as an OBEX server. In order to serve OBEX requests, it extends the `ServerRequestHandler` class and overrides some of its methods.

```java
package bcexchanger.comm;
import java.io.IOException;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.LocalDevice;
import javax.microedition.io.Connection;
import javax.microedition.io.Connector;
import javax.obex.HeaderSet;
import javax.obex.Operation;
import javax.obex.ServerRequestHandler;
import javax.obex.SessionNotifier;
public class ExchangerCommImpl extends ServerRequestHandler
    implements ExchangerStateParent, ExchangerComm, Runnable {
    // Instance variables
    final private String uuid = "ed495afe28ed11da94d900e08161165f";
    final private String serverURL = "btgoep://localhost:" + uuid;
```
private boolean cancelWaitingInvoked = false; // becomes true if
    // cancelWaiting() is
    // called
private Thread waitingThread;
private ExchangeListener listener;
private ExchangerState currentState;
private SessionNotifier notifier = null;
private Connection con = null;

/**
   * Constructor
   * <p>
   * description
   *
   * @param _listener -
   * listener of the communication module events
   * @exception
   * @see
   */
public ExchangerCommImpl(ExchangeListener _listener)
{
    listener = _listener;
    startWaiting();
    setState(new IdleState(this));
}

public void setState(ExchangerState state) {
    currentState = state;
}
public ExchangeListener getListener() {
    return listener;
}
public ExchangerState getState() {
    return currentState;
}
public void startSending(int oper) throws Exception {
    getState().startSending(oper);
}
public void startSending() throws Exception {
    getState().startSending(0);
}

public void cancelSending()
{
    getState().cancelSending();
}

public void startWaiting()
{
    cancelWaitingInvoked = false;
    waitingThread = new Thread(this);
    waitingThread.start();
}

public void cancelWaiting()
{
    cancelWaitingInvoked = true;
}
try {
    notifier.close(); // indicate to acceptAndOpen that it is
    // canceled
} catch (IOException e) {
    // Ignore, we are closing anyways
}

public synchronized void run() {

    // to initialize stack and make the device discoverable
    try {
        LocalDevice local = LocalDevice.getLocalDevice();
        local.setDiscoverable(DiscoveryAgent.GIAC);
    } catch (Exception e) {
        // catching notifier exception
        listener.onServerError();
        return;
    }

    try {
        notifier = (SessionNotifier) Connector.open(serverURL);
    }
    catch (IOException e2) {
        // the cycle stops only if cancelWaiting() was called
        while (!cancelWaitingInvoked) {
            try {
                try {
                    con = notifier.acceptAndOpen(this);
                    wait(); // wait until the remote peer disconnects
                    try {
                        con.close();
                    } catch (IOException e0) {
                    }
                } catch (Exception e1) {
                    listener.onServerError();
                    return;
                }
            } catch (Exception e) {
                listener.onServerError();
                return;
            }
        }
    }

    /*
    * This method is related to OBEX server functionality. This method
    * delegates this execution to the current state
    *
    * @see javax.obex.ServerRequestHandler#onGet()
    */
public int onGet(Operation op) {
    return getState().onGet(op);
}

/*
 * This method is related to OBEX server functionality. This method
 * delegates this execution to the current state
 * @see javax.obex.ServerRequestHandler#onPut()
 */
public int onPut(Operation op) {
    return getState().onPut(op);
}

/*
 * This method is related to OBEX server functionality. This method
 * handles OBEX DISCONNECT command from the remote device.
 * @see javax.obex.ServerRequestHandler#onDisconnect()
 */
public synchronized void onDisconnect(HeaderSet request,
                                          HeaderSet reply) {
    super.onDisconnect(request, reply);
    notify(); // stops waiting in run()
}

public String getUUID() {
    return uuid;
}

4.6.4 ExchangerState

This is the abstract base class for all the states of the communication state machine. Each state
implements only those methods which can be executed in that particular case.

Classes IdleState, InquiryState, ServiceDiscoveryState, ReceiveState, and
SendState inherit from this class.

4.6.5 ExchangerStateParent

This interface contains the function used by the state to address the functionality of the parent class
— ExchangerCommImpl. It contains the methods which the states of the communication state
machine use to get certain functionality from the parent.

This interface is implemented by the ExchangerCommImpl class.

package bcexchanger.comm;

public interface ExchangerStateParent {

    /**
     * Current state setter
     * @param state - state of communication machine
     */
    public void setState(ExchangerState state);
/**
 * Listener getter
 * <p>
 * Returns the communication event listener which is stored in the
 * parent
 * @return communication event listener interface
 */
public ExchangeListener getListener();

/**
 * Communication state getter
 * <p>
 * Returns current state of the communication machine which is kept
 * in the parent
 * @return current state of the communication machine
 */
public ExchangerState getState();

/**
 * Returns "Business Card Exchanger Service" UUID
 * @return string containing UUID of the Bluetooth OBEX server
 * connection
 */
public String getUUID();

4.6.6  IdleState

This class implements the idle state of the communication state machine. In idle state the machine
waits for external events (such as incoming Bluetooth connections or a user command to start
business card exchange). The IdleState class extends the ExchangerState abstract class.

package bcexchanger.comm;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.InputStream;
import java.io.OutputStream;
import javax.obex.Operation;
import javax.obex.ResponseCodes;

public class IdleState extends ExchangerState {
    /**
     * Constructor
     * @param _parent -
     * class which nests the current state of state
     * machine
     */
    public IdleState(ExchangerStateParent _parent) {
        super(_parent);
    }

    public int onGet(Operation op) {
        try {
            OutputStream out = op.openOutputStream();
            byte[] vCard = parent.getListener().getOwnBC(); // getting
            // own card
int vlen = vCard.length;
byte[] tmpBuf = new byte[vlen + 4];
System.arraycopy(vCard, 0, tmpBuf, 4, vlen);
tmpBuf[0] = (byte) ((vlen >>> 24) & 0xff);
tmpBuf[1] = (byte) ((vlen >>> 16) & 0xff);
tmpBuf[2] = (byte) ((vlen >>> 8) & 0xff);
tmpBuf[3] = (byte) ((vlen >>> 0) & 0xff);
out.write(tmpBuf); // sending data
op.close();
return ResponseCodes.OBEX_HTTP_OK;
} catch (Exception e) {
    return ResponseCodes.OBEX_HTTP_INTERNAL_ERROR;
}

public int onPut(Operation op) {
    try {
        InputStream in = op.openInputStream();
        byte[] fullResult = null;
        byte[] buf = new byte[256];
        ByteArrayOutputStream bout = new ByteArrayOutputStream(2048);
        for (int len = in.read(buf); len >= 0; len = in.read(buf))
            bout.write(buf, 0, len);
        fullResult = bout.toByteArray();
        ByteArrayInputStream bin = new ByteArrayInputStream(fullResult);
        DataInputStream din = new DataInputStream(bin);
        int size = din.readInt();
        byte[] vCard = new byte[size];
        din.read(vCard);
        // card is received
        op.close();
        parent.getListener().onReceiveBC(vCard);
        return ResponseCodes.OBEX_HTTP_OK;
    } catch (Exception e) {
        return ResponseCodes.OBEX_HTTP_INTERNAL_ERROR;
    }
}

public void startSending(int oper) throws Exception {
    parent.setState(new InquiryState(parent, oper));
}

public void cancelSending() {
    // Internal error, but not fatal
4.6.7 InquiryState

This class represents the state of running inquiry. In this state an active device discovery is performed. No OBEX server commands are handled. This class extends the `ExchangerState` abstract class. This class also implements the `DiscoveryListener` interface to receive Bluetooth inquiry callbacks.

```java
package bcexchanger.comm;

import java.io.IOException;
import java.util.Vector;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;
import javax.obex.Operation;
import javax.obex.ResponseCodes;

public class InquiryState extends ExchangerState
    implements DiscoveryListener {
    private DiscoveryAgent agent;
    private Vector remoteDevices; // vector of found devices
    private int operation = ServiceDiscoveryState.GET;

    /**
     * Constructor
     *
     * @param _parent - the class which nests the current state of the state machine
     */
    public InquiryState(ExchangerStateParent _parent, int oper)
        throws IOException {
        super(_parent);
        operation = oper;
        remoteDevices = new Vector();
        // initiate Bluetooth
        LocalDevice local = LocalDevice.getLocalDevice();
        agent = local.getDiscoveryAgent();
        // start Bluetooth inquiry
        agent.startInquiry(DiscoveryAgent.GIAC, this);
    }

    /**
     * Inquiry state does not allow the starting of any other business card exchange process.
     * @see bcexchanger.comm.ExchangerState#startSending()
     */
    public void startSending(int op) throws IOException {
        throw new IOException(
            "Inquiry is in progress. Inquiry has to be canceled before starting new sending process");
    }

    /**
     * InquiryState allows cancelling of the inquiry process.
     * @see bcexchanger.comm.ExchangerState#cancelSending()
     */
    public void cancelSending() {
        agent.cancelInquiry(this);
    }
}
```
public void deviceDiscovered(RemoteDevice dev, DeviceClass devClass) {
    remoteDevices.addElement(dev);
}

public void inquiryCompleted(int code) {
    try {
        int completionCode = ExchangerComm.ERROR;
        // convert the inquiry completion code to application completion
        // code
        switch (code) {
            case DiscoveryListener.INQUIRY_COMPLETED:
                completionCode = ExchangerComm.DONE;
                break;
            case DiscoveryListener.INQUIRY_TERMINATED:
                completionCode = ExchangerComm.CANCELED;
                break;
            case DiscoveryListener.INQUIRY_ERROR:
                completionCode = ExchangerComm.ERROR;
                break;
        }
        parent.getListener().onInquiryComplete(completionCode); // signal
        // that
        // inquiry
        // is
        // done
        if (code == DiscoveryListener.INQUIRY_COMPLETED) { // no errors
            // or
            // cancelations
            parent.setState(new ServiceDiscoveryState(parent,
                remoteDevices, operation));
        } else {
            parent.setState(new IdleState(parent));
        }
    } catch (Exception e) {
        parent.setState(new IdleState(parent));
    }
}

/*
 * Service discovery callbacks are not handled and not supposed to
 * occur, since service discovery process is not started
 * @see javax.bluetooth.DiscoveryListener#servicesDiscovered(int,
 *      javax.bluetooth.ServiceRecord[])
 */
public void servicesDiscovered(int arg0, ServiceRecord[] arg1) {
    throw new RuntimeException(
        "Internal error #4: InquiryState.servicesDiscovered() should not
        be called");
}

/*
 * Service discovery callbacks are not handled and not supposed to
 * occur, since service discovery process is not started
 * @see javax.bluetooth.DiscoveryListener#serviceSearchCompleted(int,
 *      int)
 */
public void serviceSearchCompleted(int arg0, int arg1) {
    throw new RuntimeException(
        "Internal error #5: InquiryState.serviceSearchCompleted() should
        not be called");
}
/*
 * Serving OBEX GET operation is supported only in IdleState
 * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
 */
public int onGet(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}

/*
 * Serving OBEX GET operation is supported only in
 * IdleState
 * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
 */
public int onPut(Operation op) {
    // onPut is supported only in IdleState
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}
}

4.6.8 ServiceDiscoveryState

This class represents the state of the running service discovery. In this state an active service discovery
is performed and no OBEX server commands are handled.

This class extends the ExchangerState abstract class. It also implements the
DiscoveryListener interface to receive Bluetooth service discovery callbacks. Service discovery
procedures are run from a separate thread and therefore the class implements the Runnable
interface.

package bcexchanger.comm;
import java.io.IOException;
import java.util.Enumeration;
import java.util.Vector;
import javax.bluetooth.DeviceClass;
import javax.bluetooth.DiscoveryAgent;
import javax.bluetooth.DiscoveryListener;
import javax.bluetooth.LocalDevice;
import javax.bluetooth.RemoteDevice;
import javax.bluetooth.ServiceRecord;
import javax.bluetooth.UUID;
import javax.obex.Operation;
import javax.obex.ResponseCodes;
public class ServiceDiscoveryState extends ExchangerState implements
    DiscoveryListener, Runnable {
    // Instance variables
    private Thread serviceDiscoveryThread;
    private Vector services;
    private DiscoveryAgent agent;
    private int serviceDiscoveryID;
    private Vector devices;
    private boolean canceled = false;
    public static int GET = 0;
    public static int PUT = 1;
/**
 * Constructor
 *
 * @param _parent - 
 * the class which nests the current state of 
 * the state machine
 * @param _devices - 
 * a vector of RemoteDevice object representing 
 * devices found during inquiry
 */
public ServiceDiscoveryState(ExchangerStateParent _parent,
 Vector _devices,int oper)
    throws IOException {
        super(_parent);
        canceled = false;
        services = new Vector();
        devices = _devices;
        // initiate Bluetooth
        LocalDevice local = LocalDevice.getLocalDevice();
        agent = local.getDiscoveryAgent();
        serviceDiscoveryThread = new Thread(this);
        serviceDiscoveryThread.start();
    }

    /*
    * ServiceDiscoveryState does not allow the starting of any other
    * business card exchange process.
    * @see bcexchanger.comm.ExchangerState#startSending()
    */
    public void startSending(int op) throws Exception {
        throw new IOException(
            "Service discovery is in progress. Service discovery has 
            to be canceled before starting new sending process");
    }

    /*
    * ServiceDiscoveryState allows cancelling of the discovery process.
    * @see bcexchanger.comm.ExchangerState#cancelSending()
    */
    public void cancelSending() {
        canceled = true;
        agent.cancelServiceSearch(serviceDiscoveryID);
    }

    /*
    * Inquiry callbacks are not handled and not supposed to occur, since
    * inquiry process is not started
    * @see javax.bluetooth.DiscoveryListener#deviceDiscovered(javax.
    *      bluetooth.RemoteDevice,javax.bluetooth.DeviceClass)
    */
    public void deviceDiscovered(RemoteDevice arg0, DeviceClass arg1) {
        throw new RuntimeException(
            "Internal error #8: ServiceDiscoveryState.deviceDiscovered()
            should not be called");
    }
    */
* Inquiry callbacks are not handled and not supposed to occur, since
* inquiry process is not started
* @see javax.bluetooth.DiscoveryListener#inquiryCompleted(int)
*/
public void inquiryCompleted(int arg0) {
    throw new RuntimeException("Internal error #9: ServiceDiscoveryState.inquiryCompleted()
        should not be called");
}

public void servicesDiscovered(int id, ServiceRecord[] _services) {
    for (int i = 0; i < _services.length; i++) {
        services.addElement(_services[i]);
    }
}

public synchronized void serviceSearchCompleted(int arg0, int arg1) {
    notify();
}

/*@ (non-Javadoc)
* This method implements logic of the service discovery process
* @see java.lang.Runnable#run()
*/
public synchronized void run() {
    Enumeration e = devices.elements();
    // Business card exchanger service UUID
    UUID[] uuids = new UUID[1];
    uuids[0] = new UUID(parent.getUUID(), false);
    // proceeded with all devices if not canceled
    while (e.hasMoreElements() && !canceled) {
        RemoteDevice device = (RemoteDevice) e.nextElement();
        try {
            serviceDiscoveryID = agent.searchServices(null, uuids, device,
                this);
        } catch (Exception e0) {
            // signal error to the MIDlet
            parent.getListener().onServiceDiscoveryComplete(
                ExchangerComm.ERROR);
            parent.setState(new IdleState(parent));
            return;
        }
        try {
            wait(); // wait until service search is done on this device
        } catch (Exception e1) {
            // Ignore
        }
    }
    if (canceled) { // handle user's cancelation
        try {
            parent.getListener().onServiceDiscoveryComplete(
                ExchangerComm.CANCELED);
            parent.setState(new IdleState(parent));
        }
} catch (Exception e1) {
    throw new RuntimeException("Internal error #10: ServiceDicoveryState.run()");
}
} else
{
    // not canceled
    if (services.isEmpty()) { // no services on devices
        try {
            parent.getListener().onServiceDiscoveryComplete(
                ExchangerComm.NO_RECORDS);
            parent.setState(new IdleState(parent));
        } catch (Exception e1) {
        }
    } else if (services.size() == 1)
    { // one service found,
        // connect to it
        try {
            ServiceRecord serviceRecord = (ServiceRecord) services
                .firstElement();
            parent.getListener().onServiceDiscoveryComplete(
                ExchangerComm.DONE);
            SendState state = new SendState(parent, serviceRecord
                .getConnectionURL(
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                false));
            parent.setState(state);
            state.doSend();
            state = null;
            Thread.sleep(new Long(5000).longValue());
            ReceiveState rstate = new ReceiveState(parent,
                serviceRecord.getConnectionURL(
                ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                false));
            parent.setState(rstate);
            rstate.doGet();
            rstate = null;
            Thread.sleep(new Long(5000).longValue());
        } catch (Exception e2) {
        }
    }
    else
    { // several services found, let user choose
        try {
            // list of friendly names of devices which
            // contain services
            Vector friendlyNames = createFriendlyNamesList(services);
            int index = parent.getListener().resolveMultipleServices(
                friendlyNames);
            if (!canceled)
            { // if not canceled during resolving
                ServiceRecord serviceRecord = (ServiceRecord) services
                    .elementAt(index);
                parent.getListener().onServiceDiscoveryComplete(
                    ExchangerComm.DONE);
                SendState state = new SendState(parent,
                    serviceRecord.getConnectionURL(
                    ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                    false));
            } else
            { // not canceled during resolving
                ServiceRecord serviceRecord = (ServiceRecord) services
                    .elementAt(index);
                parent.getListener().onServiceDiscoveryComplete(
                    ExchangerComm.DONE);
                SendState state = new SendState(parent,
                    serviceRecord.getConnectionURL(
                    ServiceRecord.NOAUTHENTICATE_NOENCRYPT,
                    false));
        } catch (Exception e3) {
        }
    }
}
public int onGet(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}

private Vector createFriendlyNamesList(Vector _services) {
    Vector friendlyNames = new Vector();
    Enumeration e = _services.elements();
    while (e.hasMoreElements()) {
        ServiceRecord serviceRecord = (ServiceRecord) e.nextElement();
        RemoteDevice device = serviceRecord.getHostDevice();
        try {
            friendlyNames.addElement(device.getFriendlyName(false));
        } catch (IOException e1) {
            // If there is an exception when getting the friendly name
            // use the address
            friendlyNames.addElement(device.getBluetoothAddress());
        }
    }
    return friendlyNames;
}

/*
 * This method creates a list of the friendly names of devices
 * that contain the services
 * @param _services - vector of ServiceRecord objects
 * @return - vectors of strings containing friendly names of devices
 */
public int onPut(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}

4.6.9 ReceiveState

This class represents the state when a business card is received from the remote device. In this state the client OBEX GET operation is performed. No OBEX server commands are handled.

package bcexchanger.comm;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.IOException;
import java.io.InputStream;
import javax.microedition.io.Connector;
import javax.obex.ClientSession;
import javax.obex.HeaderSet;
import javax.obex.Operation;
import javax.obex.ResponseCodes;
public class ReceiveState extends ExchangerState{
    // Instance variables
    private ClientSession connection = null;
    private Operation operation = null;
    String url = null;
    boolean cancelInvoked = false; // true after cancelSending is called

    /**
     * Constructor
     * @param _parent - the class which nests the current state of the state machine
     * @param _url - a URL of the remote OBEX service. This state is responsible for establishing the connection
     * @param _exception
     * @param _see
     */
    public ReceiveState(ExchangerStateParent _parent, String url)
    {
        super(_parent);
        this.url = url;
    }

    /**
     * ReceiveState does not allow the starting of any other business card exchange process.
     * @see bcexchanger.comm.ExchangerState#startSending()
     */
    public void startSending(int to) throws IOException {
    }
throw new IOException(
    "Receiving is in progress. Receiving has to be canceled before
    starting new sending process");

}/*
 * ReceiveState allows cancelling of the receiving process.
 * @see bcexchanger.comm.ExchangerState#cancelSending()
 */
public void cancelSending() {
    cancelInvoked = true;
    try { // cancel any active operation running
        operation.abort();
    } catch (IOException e) {
        // Ignore, aborting operation anyway
    }
    try { // send DISCONNECT command to the remote peer
        connection.disconnect(null);
    } catch (IOException e1) {
        // Ignore, disconnecting anyway
    }
}

/*
 * This method implements the logic for
 * receiving a business card
 */
public void doGet()
{
    try 
    {
        connection = (ClientSession) Connector.open(url);
        HeaderSet response = connection.connect(null);
        operation = connection.get(null);
        InputStream in = operation.openInputStream();
        byte[] fullResult = null;
        {
            byte[] buf = new byte[256];
            ByteArrayOutputStream bout = new ByteArrayOutputStream(2048);
            for (int len = in.read(buf); len >= 0; len = in.read(buf))
            {
                bout.write(buf, 0, len);
            }
            fullResult = bout.toByteArray();
        }
        ByteArrayInputStream bin = new ByteArrayInputStream(fullResult);
        DataInputStream din = new DataInputStream(bin);
        int size = din.readInt();
        byte[] vCard = new byte[size];
        din.read(vCard);
        parent.getListener().onReceiveBC(vCard);
        // End the transaction
        in.close();
        int responseCode = operation.getResponseCode();
        operation.close();
        if (cancelInvoked) {
            throw new Exception(
                "Cancel did not invoke any exception; throw exception then");
        }
    } catch (IOException e) { // Ignore, aborting operation anyway
    } catch (Exception e1) { // Ignore, disconnecting anyway
    }
}
try {
    parent.setState(new IdleState(parent));
    parent.getListener().onReceiveComplete(ExchangerComm.DONE);
}
catch (Exception e) {
}

catch (Exception e1) {
    if (cancelInvoked) {
        // if exception was caused by canceling
        parent.setState(new IdleState(parent));
        parent.getListener().onReceiveComplete(ExchangerComm.CANCELED);
    } else {
        // if exception was caused by error
        parent.setState(new IdleState(parent));
        parent.getListener().onReceiveComplete(ExchangerComm.ERROR);
    }
}
finally {
    try {
        // finalizing operation
        operation.close();
    } catch (IOException e3) {
    }

    try {
        // sending DISCONNECT command
        connection.disconnect(null);
    } catch (IOException e2) {
        // Ignore, disconnecting anyway
    }
}

/*
 * Server OBEX GET command is supported only in
 * IdleState
 * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
 */
public int onGet(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}

/*
 * Server OBEX PUT command is supported only in
 * IdleState
 * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
 */
public int onPut(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}
4.6.10 SendState

This class represents the state when a business card is sent to the remote phone. In this state the client OBEX PUT operation is performed. No OBEX server commands are handled. This class extends the ExchangerState abstract class.

```java
package bcexchanger.comm;
import java.io.IOException;
import java.io.OutputStream;
import javax.microedition.io.Connector;
import javax.obex.ClientSession;
import javax.obex.HeaderSet;
import javax.obex.Operation;
import javax.obex.ResponseCodes;

public class SendState extends ExchangerState {

    // Instance variables
    private ClientSession connection = null;
    private String url = null;
    private Operation operation = null;
    private boolean cancelInvoked = false;

    /** Constructor
     * @param _parent - the class which nests the current state of the state machine
     * @param _url - a URL of the remote OBEX service. This state is responsible for establishing the connection
     */
    public SendState(ExchangerStateParent _parent, String _url) {
        super(_parent);
        url = _url;
    }

    /** SendState does not allow the starting of any other business card exchange.
     * @see bcexchanger.comm.ExchangerState#startSending()
     */
    public void startSending(int o) throws IOException {
        throw new IOException("Sending is in progress. Sending has to be canceled before starting new sending process");
    }

    /** SendState allows the cancelling of the sending process.
     * @see bcexchanger.comm.ExchangerState#cancelSending()
     */
    public void cancelSending() {
    }
}
```

MIDP: OBEX API Developer's Guide
cancelInvoked = true;

try { // cancel any active OBEX operation
    operation.abort();
} catch (Exception e) { // catch NullPointerException exception also
    // Ignore, aborting anyway
}

try { // send DISCONNECT to the remote peer
    connection.disconnect(null);
} catch (Exception e1) { // catch NullPointerException exception also
    // Ignore, disconnecting anyway
}

try { // close the connection
    connection.close(); //
} catch (Exception e2) { // catch NullPointerException exception also
    // Ignore, closing anyway
}

}

/*
 * This method implements the logic for sending a business card
 */
public void doSend()
{
    try
    {
        connection = (ClientSession) Connector.open(url);
        HeaderSet response = connection.connect(null);

        // Initiate the PUT request
        operation = connection.put(null);
        OutputStream out = operation.openOutputStream();
        //getting the own card
        byte[] vCard = parent.getListener().getOwnBC();
        int vlen = vCard.length;
        byte[] tmpBuf = new byte[vlen + 4];
        System.arraycopy(vCard, 0, tmpBuf, 4, vlen);
        tmpBuf[0] = (byte) ((vlen >>> 24) & 0xff);
        tmpBuf[1] = (byte) ((vlen >>> 16) & 0xff);
        tmpBuf[2] = (byte) ((vlen >>> 8) & 0xff);
        tmpBuf[3] = (byte) ((vlen >>> 0) & 0xff);
        //sending data
        out.write(tmpBuf);
        out.close();
        int responseCode = operation.getResponseCode();
        operation.close();

        if (cancelInvoked) {
            throw new Exception("Cancel did not invoke any exception; throw exception then");
        }

        // send is done
        try {
            parent.setState(new IdleState(parent));
        } catch (Exception e)
        {
        }
    }
}
catch (Exception e1)
{
    parent.setState(new IdleState(parent));
    if (cancelInvoked)
    {
        // if exception is caused by cancelation
        parent.getListener().onSendComplete(ExchangerComm.CANCELED);
    } else
    {
        // if exception is caused by error
        parent.setState(new IdleState(parent));
        parent.getListener().onSendComplete(ExchangerComm.ERROR);
    }
}
finally
{
    try { // finalizing operation
        operation.close();
    }
    catch (IOException e3) {
    }
    try { // sending DISCONNECT command
        connection.disconnect(null);
    }
    catch (IOException e2)
    {
        // Ignore, disconnecting anyway
    }
}

/*
 * Server OBEX GET command is supported only in IdleState
 * @see bcexchanger.comm.ExchangerState#onGet(javax.obex.Operation)
 */
public int onGet(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}

/*
 * Server OBEX PUT command is supported only in IdleState
 * @see bcexchanger.comm.ExchangerState#onPut(javax.obex.Operation)
 */
public int onPut(Operation op) {
    return ResponseCodes.OBEX_HTTP_CONFLICT;
}

4.7 UI package

The package bcexchanger.ui contains the classes which implement the UI of the application. The UI is implemented as a state machine, where each screen is a state. Package ui contains the abstract class Screen and classes AddressBookScreen, AlertMessage, MainScreen, ProgressScreen, and ServiceListScreen.

4.7.1 AddressBookScreen

This class is responsible for displaying the content of the phone book and allowing user to choose his or her own business card. The user can either choose the contact or exit the application. The class extends the Screen (bcexchanger.ui) class.
4.7.2 AlertMessage

This class represents a screen displaying an alert message. The class extends the Screen (bcexchanger.ui) class.

4.7.3 MainScreen

This is the idle state screen. This class implements the main screen of the application. Using this screen the user can initiate an exchange of business cards, change own business card, and exit the application. The class is extends the Screen (bcexchanger.ui) class.

4.7.4 ProgressScreen

This class implements the screen, which indicates that some operation is in progress. For example, when inquiry, service discovery, or business card sending and receiving is running, this screen is displayed. With this screen the user can cancel the process. The class is extends the Screen (bcexchanger.ui) class.

4.7.5 Screen

This class is an abstract base class for all the UI screens in this application. This class implements the CommandListner interface.

4.7.6 ServiceListScreen

This class is used to display the list of devices which provide services (in case there is more than one "Business Card Exchanger Service" in the vicinity) and to provide the means for choosing only one service or canceling the process.
## 5 Terms and abbreviations

<table>
<thead>
<tr>
<th>Term or abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLDC</td>
<td>Connected Limited Device Configuration</td>
</tr>
<tr>
<td>GOEP</td>
<td>General Object Exchange Profile</td>
</tr>
<tr>
<td>IrDA</td>
<td>Infrared Data Association</td>
</tr>
<tr>
<td>JSR</td>
<td>Java Specification Request</td>
</tr>
<tr>
<td>JSR-82</td>
<td>Java APIs for Bluetooth wireless technology</td>
</tr>
<tr>
<td>MIDP</td>
<td>Mobile Information Device Profile</td>
</tr>
<tr>
<td>NCF</td>
<td>Nokia Connectivity Framework</td>
</tr>
<tr>
<td>Carbide.j</td>
<td>Software development tool (formerly Nokia Developer’s Suite for Java™ Platform Micro Edition)</td>
</tr>
<tr>
<td>OBEX</td>
<td>Object Exchange protocol</td>
</tr>
<tr>
<td>PIM</td>
<td>Personal Information Management</td>
</tr>
</tbody>
</table>
# 6 References

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BTINTRO</strong></td>
<td>Introduction to Developing Networked MIDlets Using Bluetooth</td>
</tr>
<tr>
<td><strong>BTSIG</strong></td>
<td>Bluetooth Special Interest Group (SIG)</td>
</tr>
<tr>
<td><strong>MIDPPROG</strong></td>
<td>MIDP 1.0: Introduction to MIDlet Programming</td>
</tr>
<tr>
<td><strong>MULLER</strong></td>
<td>Muller, Nathan J. “Bluetooth demystified”</td>
</tr>
<tr>
<td><strong>OBEXFTP</strong></td>
<td>File Transfer Profile, Bluetooth Profiles, v1.1</td>
</tr>
<tr>
<td><strong>OBEXPUSH</strong></td>
<td>Object Push Profile, Bluetooth Profiles, v1.1</td>
</tr>
<tr>
<td><strong>OBEXSYNC</strong></td>
<td>Synchronization Profile, Bluetooth Profiles, v1.1</td>
</tr>
<tr>
<td><strong>PATTERNS</strong></td>
<td>Gamma et. al., “Design Patterns: Elements of Reusable Object-Oriented Software”</td>
</tr>
<tr>
<td><strong>PIM</strong></td>
<td>Introduction to the PIM API</td>
</tr>
</tbody>
</table>
7 Evaluate this resource

Please spare a moment to help us improve documentation quality and recognize the resources you find most valuable, by rating this resource.