Java Smart Ticket Demo Application Scrutinized

Dominik Gruntz and René Müller

University of Applied Sciences, Aargau, Switzerland
{d.gruntz, rene.mueller}@fh-aargau.ch

Abstract. For the Java component model J2EE, Sun has published guidelines which show how to use their technology effectively to create enterprise applications. These guidelines are illustrated in the two example enterprise application blueprints *Pet Store* and *Smart Ticket*. Since our group is working on a wireless project, we investigated the Smart Ticket demo application. This paper describes our experiences until we had the application running on real Java cell phones and shows weaknesses and bugs in the demo application.

1 Wireless Enterprise Applications

The Smart Ticket is a sample application which illustrates the guidelines to build J2EE applications. The sample application allows users to buy movie tickets through a mobile device such as a cell phone. Users can select a movie, choose a movie theater, a show-time, select one or more seats and then purchase the tickets. Figure 1 shows a seating plan where three seats in the first row have been selected. The demo application is available from Sun on the Web [1, 2].

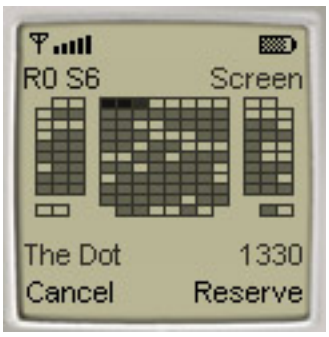


Fig. 1. Seating Plan

In our group, we are working on a similar project. The idea is to purchase tickets for parking lots via cell phones. Our server can either be accessed through a Java cell phone (as in the Smart Ticket sample application) or through a SMS

or USSD gateway. The reservation may also be forwarded to the parking ticket machine which prints a ticket [3].

As we wanted to apply the J2EE infrastructure appropriately, we carefully studied the Smart Ticket blueprint application (version 1.1.1) and installed it on several application servers. It was very instructive to see how the J2EE design patterns are applied in practice. While studying the code of the Smart Ticket application we also found (and fixed) some bugs which was very instructive too.

The goal of this paper is to share our experiences and to point out potential pitfalls programmers may face with enterprise applications. However we want to emphasize that we consider the blueprint initiative (and the Smart Ticket application) as very valuable, although the critical remarks may outweigh the positive ones in this paper. We hope, that our comments help to improve the application and that the readers can also benefit from the problems discussed.

We also installed the midlet part of the Smart Ticket application on real Java cell phones (Siemens SL45i, Nokia 3410, and Motorola Accompli 008). The application did not run on those three cell phones right away. Regardless of these experiences, we worked around the problems these cell phones still have[1]. The application is now running reasonably on the Siemens SL45i and on the Accompli 008, but memory restrictions of the cell phones are still the most limiting factor.

The paper is organized as follows. In chapter 2 we describe the architecture of the Smart Ticket application, and in chapters 3 and 4 we describe the server part in more detail and discuss the encountered problems. Chapter 5 describes the problems we met when we installed the client part on real cell phones and the workarounds we used. The last chapter contains our concluding remarks.

2 Architecture of the Smart Ticket Demo Application

The Smart Ticket application is designed as a multi-tier architecture (see Figure 2) and follows the MVC pattern. The business logic is implemented in EJBs on top of the database. The EJBs are accessed from a servlet which accepts requests from MIDP clients over HTTP. The MIDP client is responsible for the data presentation.

The client tier consists of a MIDP/CLDC application providing the user interface. The data is presented in forms, lists or on canvas objects and the application state is controlled with a state machine. Personal preferences (like user name, password) and all messages of a particular locale setting (internationalization support) are stored in record stores on the cell phone.

The web tier handles the HTTP POST requests from the midlet. A simple message format is used to specify the commands. The web tier is also responsible for the session management. Sessions are controlled with a session listener.

[1] Btw, we can completely agree with a quote by Lars Bak (Sun microsystems). At his presentation at JavaOne 2002 he emphasized that "we work on things that you can throw against the wall."

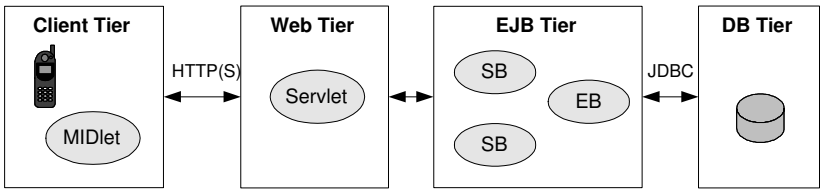


Fig. 2. Smart Ticket architecture

The EJB tier implements the business logic. Customer data is represented by an entity bean and access to this data is provided through value objects. Requests for movie information and the reservation process are handled by session beans.

The database tier consists of nine tables which store information about movies and shows. All pending reservations are stored in a separate table until they are confirmed or canceled. Two tables are used for internationalization support. The demo application comes with two predefined language settings, but others can easily be added.

Further information on the Smart Ticket application can be found in [4, 5]. In the following sections we want to focus on the EJB and the Web tiers.

3 EJB Tier

The access to the database is handled by a series of EJBs. The *CustomerEJB* is an entity bean which represents a customer and his or her account. All other EJBs are session beans. *MovieInfoEJB* and *LocaleInfoEJB* are stateless session beans for retrieving movie information (title, location, showtimes) and locale information (available language locales and messages) respectively. *TicketSalesEJB* is a stateful session bean which is used to reserve and cancel reservations for a particular user in a particular show.

In the following two subsections we discuss the implementations of *CustomerEJB* and *TicketSalesEJB*. The first one contains errors regarding the EJB specification, and the second one makes assumptions on the parameter types and on the order the methods have to be invoked which make it difficult to reuse this bean in another context. Both aspects show, that it is difficult to design, specify and implement beans correctly.

3.1 Customer Bean

The customer bean represents the customer information (name, password, zip code, credit card info) and is implemented as a BMP entity bean. Its remote interface contains the method *getInformation* which returns the customer information of a particular customer as a value object. The transaction attribute for this method is set to *Required*.

```
public interface Customer extends EJBObject {
   public CustomerInformation getInformation() throws RemoteException;
}
```

This bean is implemented in class *CustomerEJB*. We discuss two methods of this class and show that they do not conform with the EJB specification. These inconformities are rather subtle and do not show up when the application is deployed and executed on the J2EE reference container implementation. One bug was only discovered when we installed the beans on a second J2EE application server.

ejbCreate

We could successfully run the application with the J2EE reference container implementation from Sun. When we moved the EJB Layer to the JBoss application server (versions 2.4.4 and 3.0.0) the *getInformation* method returned *null* (which led to a *NullPointerException* when this object was accessed). The implementation of *getInformation* simply returns the customer information value object as it is stored in a private field, and this field is initialized in the bean's *ejbLoad* method.

On first sight, this is a bug in the JBoss application server, as the *ejbLoad* method was not called on the bean instance after it was initialized with *ejbCreate* and before *getInformation* was called. The following table shows the callback methods which are invoked from the J2EE reference container implementation when a new customer EJB is created and the *getInformation* method is called (`<init>` stands for the invocation of *CustomerEJB*'s constructor):

MIDPService.createUser	**CustomerEJB**
`c = (Customer)home.create(user, pw, zip, cc);`	`<init>`
	`setEntityContext`
	`ejbCreate`
	`ejbPostCreate`
	`ejbStore`
`customerInfo = c.getInformation();`	`ejbLoad`
	`getInformation`
	`ejbStore`

Before *getInformation* is called on *CustomerEJB*, the data is loaded from the database with *ejbLoad* as the data might already have been changed in the database by another application. Invocation of *ejbLoad* is therefore necessary and guaranteed as the *Required* transaction attribute is set for the *getInformation* method.

However, if we call the *create* and the *getInformation* methods in the context of the same transaction, then the container does not have to reload the data. This could be enforced if the *create* and *getInformation* methods were wrapped by explicit *begin()* and *commit()* calls on a user transaction. The callback methods

called by the J2EE reference container implementation for this situation are shown in the following table:

MIDPService.createUser	**CustomerEJB**
`ut.begin();`	
`c = (Customer)home.create(user, pw, zip, cc);`	`<init>`
	`setEntityContext`
	`ejbCreate`
	`ejbPostCreate`
`customerInfo = c.getInformation();`	`getInformation`
	`ejbStore`
`ut.commit();`	

Before *getInformation* is called, the data is not reloaded with *ejbLoad* and the subsequent *getInformation* method call would return *null*. This shows, that the *CustomerEJB* class is not implemented correctly. The *ejbCreate* method has to completely initialize the bean instance.

This bug can be fixed by either initializing the private customer information field lazily or by initializing the customer data in the creation phase. The code for the latter solution is shown in the following listing.

```
public String ejbCreate(String u, String p, String zc, String cc)
      throws EJBException, DuplicateKeyException, CreateException {
   String ret = dbInsertCustomer(u, p, zc, cc);
   if (ret != null) {
      // initialize information field
      information = new CustomerInformation(u, p, zc, cc);
      return ret;
   }
   else {
      throw new CreateException(u);
   }
}
```

ejbRemove

The customer bean is implemented incompletely as the implementation of *ejbRemove* in class *CustomerEJB* is left empty. For entity beans, *ejbRemove* has to remove the entry from the database. With the given implementation Customer data can never be removed from the database through the EJB Layer. A correct implementation of *ejbRemove* is shown below. With this change however, the Smart Ticket application no longer works properly. The reason is, that the *remove* method is actually called on customer beans in the controller servlet in the class *MIDPService*. This call has to be removed.

```
public void ejbRemove() {
   dbRemoveCustomer();
}
```

```
private void dbRemoveCustomer() {
   // Remove the customer.
   try {
      Connection c = dataSource.getConnection();
      PreparedStatement ps = c.prepareStatement(
                    "delete from userprefs where username = ?");
      String username = (String) entityContext.getPrimaryKey();
      ps.setString(1, username);
      ps.executeUpdate();
      ps.close();
      c.close();
   }
   catch (SQLException e) {
      throw new EJBException(e);
   }
}
```

3.2 Ticket Sales Bean

The Ticket Sales bean handles the movie ticket sales. It is implemented as a stateful session bean. Its state is the name of the customer who wants to make reservations, the identification number of the show the customer is interested in and of course reservations of seats.

With method *getSeatingPlan* the seating plan of a show can be fetched. The seating plan is represented as a rectangular matrix (zero-based indices). Each seat is either marked as available or unavailable or as an aisle (no seat at all). The seating plan of a show is stored row by row as string in the database. The following listing shows the remote and the home interface of the *TicketSales* bean.

```
public interface TicketSales extends EJBObject {
   public SeatingPlan getSeatingPlan() throws RemoteException;
   public boolean reserveSeats(Set seats)
                  throws RemoteException, UnavailableSeatsException;
   public boolean cancelSeats() throws RemoteException;
   public boolean confirmSeats() throws RemoteException;
}

public interface TicketSalesHome extends EJBHome {
   public TicketSales create(String customerID, int showID)
                  throws RemoteException, CreateException;
}
```

The implementation of the *TicketSales* bean works correctly in cooperation with the *MIDPService* controller of the Smart Ticket application. As this enterprise Java bean is a component, it should also be reusable in other contexts, e.g. from a web client or a stand-alone client application. It is therefore necessary that the interface is clearly specified and that the implementation does not

make additional assumptions on how the component has to be used (unless such assumptions are specified in the component contract). We claim, that some of the methods in the *TicketSales* EJB make such additional assumptions and are too tightly coupled to the way they are used in the Smart Ticket application.

Parameter *seats* in reserveSeats
The method *reserveSeats* in the *TicketSales* interface takes a set of seats that have to be reserved. This set must contain *Seat*[2] objects, and, as it is passed remotely, it must be serializable. We called this method from a test program and since we only wanted to reserve a single seat, we passed a singleton set. The invocation of the session bean looked as follows (*home* is a reference to a *TicketSalesHome*):

```
// create a ticket sales bean for user gruntz and show #7
TicketSales sales = home.create("gruntz", 7);
// create a singleton set for seat #9 in row #5
Set set = Collections.singleton(new Seat(5, 9));
// reserve this seat and confirm the reservation
sales.reserveSeats(set);
sales.cancelSeats();
```

As a result we received a *RemoteException* whose nested exception was an *UnsupportedOperationException*. Investigation of the code showed, that both the *cancelSeats* and the *confirmSeats* method clear the seat set which was passed. However, the singleton set we used is immutable.

When the *TicketSales* bean is invoked from the *MIDPService* controller, a mutable *HashSet* instance is passed and no exception is thrown. Either the code should support immutable sets or the interface contract should specify that the set must be mutable (together with an indication on how the set is modified).

There are two ways how to improve the implementation so that arbitrary set implementations can be passed. One solution is to make a copy of the set argument in the *reserveSeats* method. However, as the method is always called remotely, the argument which is passed is already a copy and making a second copy would be inefficient. Note, that copying the argument would be necessary if the method were called through a local interface. The other solution is to change the statements where the set is modified. The only modification which is performed is clearing the set. Instead of calling *seats.clear*, the reference which holds this set could also be assigned a reference to an empty set (e.g. *Collections.EMPTY_SET*).

Ticket Sales States
Objects are state machines. Every method call changes the state of an object, and an interesting question is, whether methods can be invoked in an arbitrary order

[2] Seat is a serializable class which represents a seat in a theater and encapsulates the row and the seat number.

on the object. For the *TicketSales* bean, no restrictions are specified, and it is actually possible to call *confirmSeats* or *cancelSeats* without having reserved any seats. This works, as the private field which stores the reserved sets is initialized with an empty set, and both the *confirmSeats* and *cancelSeats* methods iterate over this set (and thus for an empty reservation set, no action is performed). Since both *confirmSeats* and *cancelSeats* clear the set of reserved seats, it is also possible to call *confirmSeats* or *cancelSeats* twice (this does not hold if the seats are confirmed or cancelled through the *MIDPService* controller as the controller removes the session bean after confirmation or cancellation).

However, if *reserveSeats* is called several times without confirming or removing previous reservations, then the database becomes inconsistent. Consider the following statements (where *home* is again a reference to the *TicketSalesHome* interface):

```
// create a ticket sales bean for user gruntz and show #7
TicketSales sales = home.create("gruntz", 7);
// reserve seat # 0 in row 6
Set s1 = new HashSet();
s1.add(new Seat(6, 0));
sales.reserveSeats(s1);
// reserve seat # 1 in row 6
Set s2 = new HashSet();
s2.add(new Seat(6, 1));
sales.reserveSeats(s2);
// cancel the reservations
sales.cancelSeats(); // => only the latter reservation is cancelled
```

Upon the first invocation of *reserveSeats*, the seat (6,0) is marked as unavailable in the seating table in the database. Moreover, the reservation is stored in the reservations table and in the seating field of the session bean. When *reserveSeats* is called the second time, the seat (6,1) is also marked as unavailable and the reservation is inserted in the reservations table too, but the set of reserved seats which is stored in the session bean is overwritten with the new seat set. The *cancelSeats* operation therefore only cancels the seats of the second reservation. The seat (6,0) remains marked unavailable in the seating table and cannot be booked by another client. Actually, as all pending reservations are stored in the database, a service could be implemented which removes pending reservations of a customer, but we think it is easier to update the database from the ticket sales session bean directly.

In order to solve this problem, one has to decide what it should mean if *reserveSeats* is called several times before the reservation process is completed with either *confirmSeats* or *cancelSeats*.

One possible interpretation would be, that a former reservation is automatically cancelled. Before the new set of seats is stored in the session bean, the *cancelSeats* method would be called[3].

[3] This method call is only necessary if a reservation is pending. Additionally, loading of the seating plan should be factored out for efficiency reasons.

```
public boolean reserveSeats(Set s) throws UnavailableSeatsException {
   try {
      Connection c = dataSource.getConnection();
      if(seats.size() > 0) // cancel any pending reservation
          cancelSeats();
      seats = s;
      ....
```

A better semantic is, that all *reserveSeats* invocations extend an already pending reservation set (provided that all seats of the extension set are available). The set of the reserved seats would have to be added to the existing seat set after all changes have been processed in the database (remember, that *reserveSeats* requires a transaction context). This solution is implemented in the Smart Ticket application version available from our web site [6].

ejbRemove
A similar problem occurs if a session is removed before pending reservations are completed. The termination of the session could either be initiated explicitly with the *remove* method or implicitly over a session bean time-out. Currently, pending reservations are not removed in the *ejbRemove* method and the database is left in an inconsistent state (to be precise, the seats remain marked as unavailable and the reservation remains in the reservations table). The following example code causes such a situation.

```
// create a ticket sales bean for user gruntz and show #7
TicketSales sales = home.create("gruntz", 7);
// reserve seat #3 in row 6
sales.reserveSeats(Collections.singleton(new Seat(6, 3)));
// terminate session
sales.remove();
```

To fix this problem, pending reservations should be cancelled in the *ejbRemove* method.

```
public void ejbRemove() {
   if(seats.size() > 0) // cancel pending reservations
      cancelSeats();
   dataSource = null;
}
```

All these problems have been reported to Sun, and it is planned that they will be incorporated into the Smart Ticket blueprint version 1.2.

4 Web Tier

In addition to the communication with the midlet, the web tier also takes control over the lifetime of a servlet session. This is done by a class that implements the

HttpSessionListener interface. This interface contains the two methods *sessionCreated* and *sessionDestroyed*.

The *sessionCreated* method creates a new controller object and stores it in the parameter map of the session. *sessionDestroyed* is called when a session was invalidated. A session can either be invalidated by an *invalidate* call on the session or by a time-out. The *MIDPController* servlet explicitly invalidates a session after confirmation of a seat reservation. The timeout for the implicit sessions invalidation is defined in the *web.xml* file as shown below[4]. The timeout could also be set directly on a HTTP session with the *setMaxInactiveInterval* method.

```
<session-config>
   <session-timeout>54</session-timeout>
</session-config>
```

In the *sessionDestroyed* method all resources which were used by the session should be freed. On (stateful) session beans the *remove* method should be called. This implies, that access to those resources must be granted in the *sessionDestroyed* method. However, since most session methods throw an *InvalidStateException* on a invalidated session, the controller cannot be accessed over the session attributes as this is done in the *MIDPSessionListener* class. We suggest to store the controller objects in a separate map. New controller objects have to be inserted in this map in the *sessionCreated* method. The updated *MIDPSessionListener* class is shown below.

```
public class MIDPSessionListener implements HttpSessionListener {
   private Map services = new HashMap();

   public void sessionCreated(HttpSessionEvent se) {
      MIDPService midpService = null;
      try { midpService = new MIDPService(); }
      catch (MIDPException ex) { }
      se.getSession().setAttribute("midpService", midpService);
      services.put(se.getSession(), midpService);
   }

   public void sessionDestroyed(HttpSessionEvent se) {
      MIDPService midpService =
         (MIDPService) services.remove(se.getSession());
      try { midpService.removeMovieInfo(); }
      catch (MIDPException ex) { }
   }
}
```

The *removeMovieInfo* method which is called should not only remove the movie info session bean but in particular also the (stateful) ticket sales session bean.

[4] The session-timeout element defines the default session timeout interval for all sessions created and is specified in minutes (in contrast to the label in the deployment tool which indicates that the argument has to be specified in seconds).

```
public void removeMovieInfo() throws MIDPException {
   try {
      if(movieInfo != null) movieInfo.remove();
      if(ticketSales != null) ticketSales.remove();
   }
   catch (RemoteException rem) { serverError(rem); }
   catch (RemoveException rev) { serverError(rev); }
   finally {
      movieInfo = null;
      ticketSales = null;
      customerInfo = null;
   }
}
```

5 Running Smart Ticket on real cell phones

We also wanted to test the client part of the Smart Ticket application on a real cell phone. We used a Siemens SL45i (with firmware version 49), a Motorola Accompli 008 and a Nokia 3410. We could install the midlet on all three cell phones, but on none of them the application did run right away. The reasons were erroneous network implementations, problems with the display and memory restrictions. However, we found solutions to work around these problems and the application is now running (still with restrictions) on all three Java cell phones. We want to share our experiences here so that the reader may get some hints on how similar problems might be fixed when the Smart Ticket application is installed on other cell phones.

5.1 HTTP POST

On all three cell phones we could test the HTTP-POST request protocol (in particular HTTP 1.1) is not properly implemented[5]. As a consequence we decided to move to HTTP-GET requests.

For this change we had to adapt the *MIDPController* servlet which now also accepts GET requests. The command is passed as a GET parameter in the request URL. Moreover, the midlet now has to append the command to the servlet URL and it has to URL-encode the arguments. Unfortunately, the midlet classes do not support URL encoding.

5.2 Color Display

The SL45i and the Nodia 3410 supports only black-and-white (2 gray levels) graphics. However, the seating canvas which presents the unavailable, available and booked seats uses five colors. We had to change this code so that the functionality can also be accessed on a black-and-white screen. For booked seats we used a cross instead of a separate color (see Figure 3).

[5] According to the Siemens developer forum the problems with the HTTP-POST request are known and will be fixed in a future firmware release.

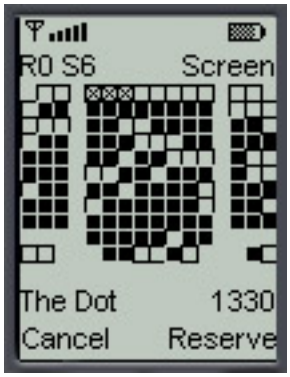


Fig. 3. Seating plan on a black-and-white screen

5.3 Character Mapping

The command which is sent from the midlet to the servlet follows a simple format. The command identifier is specified with an integer, followed by a circumflex sign ^ and the arguments which are separated by commas. For example, the login command is cunstructed in the midlet as follows:

```
String message = MessageConstants.LOGIN_USER + "^" +
                                          userID + "," + password;
```

The servlet parses this command and first looks for the circumflex sign (character code 94/0x5E). Unfortunately, on the Nokia 3410 this sign was translated into the latin capital letter eth (character code 208/0xD0) and the servlet could not parse the command string. We fixed this problem in the servlet. For the separation between the command identifier and the arguments any non-digit character is accepted.

5.4 Memory Problems

On the Siemens phone the application works fine as long as the poster preview mode is disabled. However, when the poster of a movie is loaded and tried do be displayed, the Siemens cell phone runs out of memory.

We tried to reduce the size of the midlet by using obfuscation and compactation techniques. Our intention was to reduce the verbose method and variable names Java keeps within its class files. We used the jarg [7] obfuscator and were able to reduce the size of the final JAR file by 9.7%[6]. However this did not help us any further.

We were able to pin down the occurrence of the "Out of heap memory" exception to a line where a byte array buffer is allocated into which the image data

[6] In order to achieve a maximum reduction, we modified the public member attributes of all midlet classes and methods to package level visibility where possible.

is loaded from the network. Although 61 kB were still available[7], the memory manager was not able to allocate a consecutive block of 8 kB. Given the fact that the Siemens' implementation does not use a compacting garbage collector we assume that the memory was already too fragmented at this point.

Preallocation of a byte buffer worked, but the application hung at the point where the byte array is handed over to *Image.createImage*. This method copies the byte buffer and obviously needs some contiguous memory as well. Again, this problem could be fixed with another preallocated buffer, but only one picture could be displayed this way as the buffers could not be reclaimed. At this point we decided to give up because it seems that we have reached the physical limits of the Siemens device.

On the Motorola Accompli 008 (which has more main memory) the application can be run in preview mode, and the Nokia cell phone can only display grayscale posters (as it has problems with the color mapping).

6 Summary

In this article we have discussed the successful installation of the Smart Ticket blueprint application on several Java cell phones. We had to solve problems in the Web and EJB tiers and to overcome restrictions of the cell phones. During this process, we gained a deep understanding of wireless enterprise applications based on J2EE and J2ME in general and of the Smart Ticket application in particular. We are convinced that we can apply these experiences in our parking project and we are very grateful to Sun for the publication of the J2EE blueprints.

This application however is only a starting point and needs further development. For the seating plan canvas (shown in figure 3) a scrolling strategy should be implemented for large movie theaters or for cell phones with a small screen resolution. Moreover, the credit card information should be encrypted before it is sent over the HTTP connection. Crypto packages for J2ME are available [8].

Our discussion also showed, that it may be difficult to implement beans completely compliant with the EJB specification. Tools should be provided which can test whether a given bean is EJB compliant.

The J2EE design patterns and the blue prints support the designer on how the EJB technology is best applied, but we have seen, that these guidelines do not automatically lead to reusable components. It is very demanding to implement components which are reusable in arbitrary contexts. Actually reusing a component in different contexts may improve the quality of a component, as well as code reviews do (that is what we have done with the Smart Ticket application). The real problem however is, that it is difficult to specify a component interface contract and to guarantee that this contract is also met by a component implementation. Hopefully, we will see progress in this field in the future.

[7] Runtime.freeMemory returns the amount of free memory in the system.

References

1. Java BluePrints Online, Wireless Blueprints.
   http://java.sun.com/blueprints/wireless/index.html
2. Java Smart Ticket Demo 1.1.1 Release, April 2002.
   http://developer.java.sun.com/developer/releases/smarticket/
3. Mobile Enterprise (Handy Parking) Project, Aug 2002.
   http://www.cs.fh-aargau.ch/~gruntz/projects/parking/
4. Eric D. Larson: Developing an End to End Wireless Application Using Java Smart Ticket Demo, Java Wireless Developer Site, Mai 2002.
   http://wireless.java.sun.com/midp/articles/smartticket/
5. Steve Meloan: The Java Blueprints for Wireless Program: Charting the Wireless Way, Nov 2001.
   http://java.sun.com/features/2001/11/j2eebluprnts.html
6. Sources of the fixed Smart Ticket Demo Application, Sept 2002.
   http://www.cs.fh-aargau.ch/~gruntz/projects/ejb/smarticket111fixed.zip
7. Java Archive Grinder (jarg), Version 0.9.12, July 2002.
   http://sourceforge.net/projects/jarg/
8. Bouncy Castle Crypto APIs, July 2002.
   http://www.bouncycastle.org/