EJBMaker version 2.0

New in version 2.0:

- Support for J2EE and WebSphere 4.0
- Complete EJB layer support through generated helper classes
- Immutable data replaced by fast, non-persistent adaptor classes.
- Encryption support for sensitive data

New in version 1.1

- Support for Java 1.2 and WebSphere 3.5.
- Database scripts for indexing, importing and exporting EJB data from tables

1. Overview

As the Enterprise JavaBeans (EJB) technology becomes more mature, several design patterns are gaining in popularity. Patterns are reusable across projects and often exhibit regularities that make them easy to automate. The latest version of the EJBMaker takes advantage of this fact and gives you support for several common patterns.

The EJBMaker simplifies writing CMP (Container Managed Persistence) entity EJBs for the IBM WebSphere 4.x and 3.x application server. The tool has many features, including coarse-grained data operations using generated "adaptor" classes, persistent representation of one-to-many and many-to-many relationships between EJBs, and the ability to generate efficient immutable data structures for read-only data. Unlike the "dependent value objects" technology described in the EJB 2.0 [1] specification, the code generated doesn't require the container manager to perform any special complex persistence management. The code relies on the techniques supported by any EJB 1.1 compliant application server (e.g. IBM WebSphere 4.0).

All CMP EJBs used in a project are described in a single XML file. The tool generates the following:

1. EJB source code for the CMP entity EJBs
2. XML deployment descriptors (both individual for each EJB and a merged deployment descriptor for all the EJBs).
3. Database script for creating the required persistence tables.
4. XMLAdaptors. XMLAdaptor classes provide wrappers for EJBs on the client side. By using wrappers instead of remote EJB stubs, expensive remote methods invocations can be avoided.
5. Helper classes. Helper classes are used to create, update and remove EJBs, as well as for quick retrieval of EJB data.
6. Home interface pool. Pooling class for EJB home interfaces.

Source files are not overwritten unless they have changed, making the EJBMaker tool suitable within a build environment. Finder methods for each persistent field in the EJB are automatically generated, plus the *findAllInstances*() method returning all instances of an EJB. The tool can also produce EJBs representing relationships, and automatically generates the code and finder methods required. In addition, the remote classes of the EJBs has corresponding adaptor classes, entitled *XMLAdaptors* due to their ability to import or export EJB data as XML. XMLAdaptors are used as wrappers for remote EJB instances (cf. "value objects" [2]) to reduce expensive network calls and improve performance.

The tool is invoked using the following syntax:

Usage: *java com.ibm.almaden.ejbmaker.EJBSourceGenerator [-d <java source folder>] [-p <package name>] [-db <database folder>][-e] –v [3 | 4] <xml ejb descriptor>*

-d <dir>	Specifies the folder where the source code is saved. Subfolders for package names are created automatically.
-p <package>	Specifies the root package name. Subpackages are created for each EJB. For instance, using -p com.acme.ejb and an EJB called customer, the customer EJB is put in package com.acme.ejb.customer.
-db <dir>	Specifies the folder where database scripts are stored.
-e	Set this flag is encryption should be enabled. Encryption is supported using JCE.
-v [3 | 4]	Generate code for WebSphere 3.x or 4.x. Default is 4.

Example:

java com.ibm.almaden.ejbmaker.EJBSourceGenerator –d C:\MyProject\Source –p com.acme.ejb –db C:\MyProject\db ejbs.xml

Important: Make sure that the *ejbmaker.jar* file and the *xerces.jar* file (both are contained in the lib folder) is in your CLASSPATH variable when invoking the command.

2. The XML EJB descriptor

A single XML file contains all EJB used in a project. The first two lines are the standard XML header and a *doctype* definition (see the appendix for the ejbmaker.dtd file):

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

The root element is always the ejbmaker element:

```
<ejbmaker>
```

This is followed by one or several EJB descriptors (using the *bean* element), constant descriptors (using *constan-bean* elements) or relation descriptors (*relation* element). Global transaction attributes and re-entrant specifications can be specified here, e.g.

```
<transaction-attr>Required</transaction-attr>
<re-entrant>false</re-entrant>
```

These attributes represent the defaults for all EJBs in the project.

2.1. The bean element

A CMP EJB is defined as follows:

```
<bean name="..." type = "...">
```

The two attributes are:

name	This must always be specified. It is the JNDI name for the EJB, used by the application server to create and locate instances.

type — The type attribute is required. It can either be set to ‘entity’ or ‘topic’. In most cases set it to entity. The topic alternative generates an entity EJB with an additional *onMessage* method used to provide JMS messaging support in WebSphere. However, JMS support is currently not part of the ejbmaker distribution.

The *bean* element contains any number of persistent field elements or finder method elements. A persistent field is defined as follows:

<persistent-field dt="..." col-dt="..." fkey = "..." default=”…” encrypt=”…”>…</persistent-field>

The text enclosed by the element is the name of the persistent field, and the attributes are:

1. **dt** — Required. Java class for the field, e.g. *java.lang.String* or *java.sql.Timestamp*. Use the fully qualified name – including packages – for all classes. Primitive classes are accepted too.

2. **col-dt** — Required. The DB2 data type used to store the field persistently in a database. Examples "VARCHAR(250)" or "TIMESTAMP"

3. **fkey** — Not required. If the field is containing a foreign key of another EJB, specify the name of the foreign EJB using the fkey attribute.

4. **default** — Not required. Default value assigned to the field during creation. If not specified, the initial value of the field is undefined.

5. **encrypt** — Not required. If ’true’, the field is stored encrypted in the database. If ‘false’ or not specified, no encryption is performed. Observe that the ‘-e’ flag must be specified in the ejbmaker tool for encryption to be turned on.

The *bean* element can also contain custom finder methods as specified using the *finder-method* element. Note however that the *getAllInstances*() finder method, returning all existing instances of an EJB, is automatically generated, in addition to finder methods for each persistent field in the EJB and for any relationships defined (see examples below).

The syntax for a custom finder method is:

<finder-method name="...">

The name attribute is required; it specifies the method name for the finder method generated (e.g. *‘findByCustomerIdSorted’*). The *finder-method* elements contain any number of *arg-type* elements and a single required *sql* element. The *arg-type* elements list the Java data types of the arguments to the finder method, e.g.

<arg-type>java.lang.String</arg-type>
<arg-type>java.sql.Timestamp</arg-type>

Observe that the order is important; it must match the order in the method signature.

The *sql* element contains SQL code for the finder method, e.g.

*<sql> select * from customer where custid=?</sql>*

Observe that reserved characters must be escaped in XML, e.g. ‘<’ must be written ‘<’.

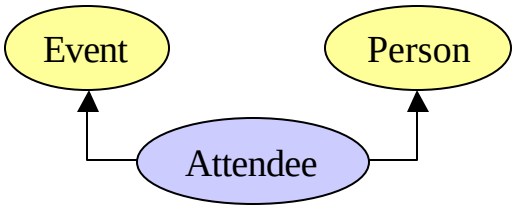

Figure 1. Relationship example

2.2. The constant-bean element

Storing immutable, read-only data using EJBs introduces a lot of unnecessary overhead in an application. EJBMaker allows you to create immutable instances of objects through the *constant-bean* element:

<constant-bean name="…">

The EJBMaker tool generates an XML adaptor class similar to the ones used by the CMP EJBs, but without creation, deletion and setter methods, and the data is not persisted in a database.

The name attribute is required. The *constant-bean* element encloses one or more *field* elements containing immutable data:

<field dt="…">…</field>

The *field* element contains the name of the field and the *dt* attribute specifies the primitive or non-primitive java class used to hold values of the field. If the class is non-primitive, *dt* must be the fully qualified Java class name, e.g. *java.util.Date*.

Constant, immutable instances are entered using *instance* elements:

<instance>

followed by initializers:

<initalize field="…">…</initialize>

The *field* attribute of the initialize element contains the name of the field being initialized, and the text of the element is the immutable value. The text is passed as a *java.lang.String* to the constructor of the field's class.

Constant XMLAdaptor instances are also associated with a primary key (just like regular EJBs), and the value of the key is determined when the source code is generated. The value changes when either the name, value or the class of a constant field changes. Primary keys for constants can be used as foreign keys in the CMP EJBs, so make sure to update such fields in the database whenever a constant has changed.

2.3. The relation element

A relation is used to define a many-to-many relationship between two EJBs (foreign keys are used for 1-many relationships). For example, given an entity EJB representing a calendar event (called Event) and another entity EJB representing a person (called Person), a relationship called Attendee specifies the particular events a person is attending (see Figure 1). A person can attend many events, and an event can host many attendees. The relationships are stored in a separate entity EJB containing two fields (strictly

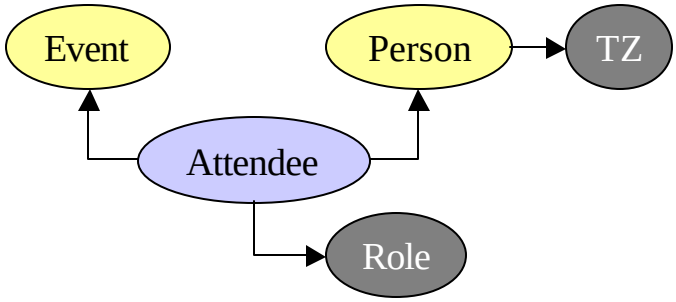

Figure 2. Representation of Persons and the calendar Events they participate in. **Event**, **Person** and **Attendee** are persistent CMP EJBs, while **TZ** and **Role** are immutable constants. Role represents the role type of attendee, e.g. 'Chair' or 'Participant', while TZ is the home time zone of the Person ('PST', 'EST' etc.)

speaking three fields if we count the primary key field for the relationship EJB itself). The first field contains foreign key of an Event instance and the second field has a foreign key of a Person.

A relation is defined in the following manner

<relation name="...">

The name attribute is required; it defines the JNDI name for the EJB. In the case above, the name is Attendee.

The relation element contains two other elements, *bean-ref-1* and *bean-ref-2* containing the JNDI names for the EJBs being associated. Our example looks like this:

```
<relation name="Attendee">
        <bean-ref-1>Event</bean-ref-1>
        <bean-ref-2>Person</bean-ref-2>
</relation>
```

In addition, a relationship can be further quantified by *persistent-field* declarations (see 2.1).

3. Source code and example

The example shown in Listing 1 will be used as a background for the sake of discussion. The example – complex enough to cover a majority of the features provided by the tool – represents the objects and relationships shown in Figure 2.

The *Event* EJB has three persistent fields, SUMMARY, DTSTART and DTEND. One finder method is declared, *findByDateInterval*. Observe how XML reserved characters such as '<' or '>' must be escaped in the SQL code. The second EJB, *Person*, has four fields: ID, FIRSTNAME, LASTNAME and TZ. The TZ field maps to a constant *Timezone* instance through the *fkey* attribute. Finally, there's a relationship defined by the *Attendee* relation, binding Persons to Events and Events to Persons. The *Role* field, mapping to a constant *Role* instance, further quantifies the relationship.

```
<?xml version="1.0"?>
<!DOCTYPE ejbmaker SYSTEM "ejbmaker.dtd">
<ejbmaker>
        <!-- default attributes for beans -->
        <transaction-attr>Required</transaction-attr>
        <re-entrant>false</re-entrant>

        <!-- Event Bean -->
        <bean name = "Event" type="entity">
```

```
                <persistent-field dt="java.lang.String" col-dt="VARCHAR(251)">SUMMARY</persistent-field>
                <persistent-field dt="java.sql.Timestamp" col-dt="TIMESTAMP">DTSTART</persistent-field>
                <persistent-field dt="java.sql.Timestamp" col-dt="TIMESTAMP">DTEND</persistent-field>
                <finder-method name="findByDateInterval">
                        <arg-type>java.sql.Timestamp</arg-type>
                        <arg-type>java.sql.Timestamp</arg-type>
                        <sql>SELECT * FROM EVENT WHERE DTSTART &gt;= ? AND DTSTART &lt;= ? ORDER BY
DTSTART</sql>
                </finder-method>
        </bean>

        <!-- Person Bean -->
        <bean name = "Person" type="entity">
                <persistent-field dt="java.lang.String" col-dt="VARCHAR(251)">ID</persistent-field>
                <persistent-field dt="java.lang.String" col-dt="VARCHAR(251)">FIRSTNAME</persistent-field>
                <persistent-field dt="java.lang.String" col-dt="VARCHAR(251)" >LASTNAME</persistent-field>
                <persistent-field dt="java.lang.String" col-dt="VARCHAR(56)" fkey="Timezone" >TZ</persistent-field>
        </bean>

        <!-- Relationships follows here.  -->

        <!--Attendee -->
        <relation name = "Attendee">
                <bean-ref-1>Event</bean-ref-1>
                <bean-ref-2>Person</bean-ref-2>
                <persistent-field dt="java.lang.String" col-dt="VARCHAR(56)"  fkey="Role">
                Role
                </persistent-field>
        </relation>

        <!-- Constant declarations .  -->

        <constant-bean name = "Role">
                <field dt="java.lang.String">RoleType</field>
                <instance>
                        <initialize field = "RoleType">Chair</initialize>
                </instance>
                <instance>
                        <initialize field = "RoleType">Attendee</initialize>
                </instance>
        </constant-bean>

        <constant-bean name = "Timezone">
                <field dt="java.lang.String">TZ</field>
                <instance>
                        <initialize field = "TZ">PST</initialize>
                <instance>
                </instance>
                        <initialize field = "TZ">EST</initialize>
                </instance>
                <!—More time zones follows here .  -->
        </constant-bean>
</ejbmaker>
```

Listing 1. Sample XML EJB description for the objects shown in Figure 2.

Figure 3 shows the application architecture for objects created by the XML file in Listing 1. XMLAdaptor classes provides client-side wrappers around EJBs. The immutable data contained in *Role* and *Timezone* is stored the XMLAdaptor classes themselves and initialized upon JVM startup. Observe that a *Client* can pretty much mean anything accessing EJBs, including session EJBs, servlets or even other entity EJBs.

4. Generated source

After having compiled and deployed the generated code (see below for more details), we can start writing code to access the EJBs. The home interfaces are retrieved using JNDI lookup on the application server.

```
public static void main(String args[])
```

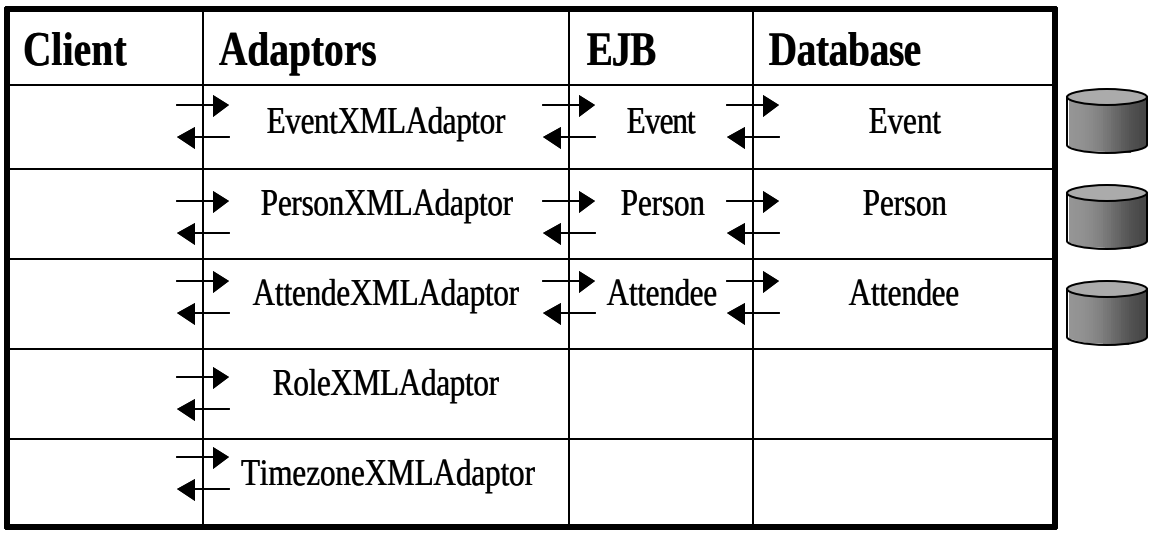

Figure 3. Application architecture for the objects generated by the XML file in Listing 1.

```
{
        try {
                Properties p = new Properties();
                p.put(Context.INITIAL_CONTEXT_FACTORY,
                        "com.ibm.websphere.naming.WsnInitialContextFactory");
                p.put(Context.PROVIDER_URL, "IIOP:///");
                InitialContext ic = new InitialContext(p);
                Object eobj = ic.lookup("Event");
                Object pobj = ic.lookup("Person");
                EventHome eHome = (EventHome)javax.rmi.PortableRemoteObject.narrow(eobj,
                        EventHome.class);
                PersonHome pHome = (PersonHome)javax.rmi.PortableRemoteObject.narrow(pobj,
                        PersonHome.class);
```

Retrieving existing EJB instances is done through the home interfaces, e.g.

```
Event evt  = eHome.findByPrimaryKey(new EventKey("123"));
```

or by using a finder method:

```
Enumeration en = eHome.findBySUMMARY("Meeting");
if(en.hasMoreElements()) {
        Event evt = javax.rmi.PortableRemoteObject.narrow(
                en.nextElement, Event.class);
```

Instead of using remote instances of the EJBs on the client side, use the corresponding XMLAdaptor classes. An XMLAdaptor class is initialized using an EJB remote instance, e.g.:

```
EventXMLAdaptor eventAdaptor = new EventXMLAdaptor(evt);
```

XMLAdaptor classes are serializable and returnable by the business methods of a session EJB. If a collection of XMLAdaptors is returned (for instance the result of a finder method), store the XMLAdaptor instances in a **SerializableEnumeration** instance and return it instead, e.g.

```
java.util.Enumeration findBySummary(String sum) {
        Enumeration en = eHome.findBySUMMARY(sum);
        SerializableEnumeration se = new SerializableEnumeration();
        while(en.hasMoreElements()) {
```

```
                    Event evt = javax.rmi.PortableRemoteObject.narrow(
                            en.nextElement, Event.class);
                    EventXMLAdaptor adaptor = new EventXMLAdaptor(evt);
                    se.add(adaptor);
                }
                return se;
        }
```

4.1. EJB Home interfaces

The home interfaces have several methods. Starting with the Event EJB, the following methods have been generated:

1. **Event create(EventKey key)** Creates a new instance of the EJB
2. **Event findByPrimaryKey(EventKey key)** Retrieve an EJB instance using its primary key
3. **Enumeration findAllInstances()** Retrieve all existing instances of the EJB as an enumeration
4. **Enumeration findBySUMMARY(java.lang.String s)** Find Event instances by the value of the SUMMARY persistent field.
5. **Enumeration findByDTSTART(java.sql.Timestamp s)** Retrieve Events by the value of the start time of the event
6. **Enumeration findByDTEND(java.sql.Timestamp s)** Retrieve Events by the value of the end time of the event
7. **Enumeration findByAttendee(java.lang.String ptr)** Retrieve Events associated with a Person. The java.lang.String is a primary key for a particular Person.
8. **Enumeration findByDateInterval(java.sql.Timestamp arg0, java.sql.Timestamp arg1)** User defined finder method (see Listing 1). Find events starting within a date interval.

Similarly, the home interface for the Person EJB has the following methods:

1. **Event create(EventKey key)** Creates a new instance of the EJB
2. **Event findByPrimaryKey(EventKey key)** Retrieve a EJB instance using its primary key
3. **Enumeration findAllInstances()** Retrieve all existing instances of the EJB as an enumeration
4. **Enumeration findByID(java.lang.String s)** Find Person instances by the value of the ID field.
5. **Enumeration findByFIRSTNAME(java.lang.String s)** Find Person instances by the value of the FIRSTNAME field.
6. **Enumeration findByLASTNAME(java.lang.String s)** Find Person instances by the value of the LASTNAME field.
7. **Enumeration findByAttendee(java.lang.String ptr)** Retrieve Persons associated with an Event. The java.lang.String contains a primary key for a particular Event.
8. **Enumeration findByTZ(java.lang.String ptr)** Retrieve Person EJBs matching the given primary key for a constant TIMEZONE instance.

The relationship EJB Attendee has the following methods in its home interface:

1. **Attendee create(AttendeeKey key)** Create new instance
2. **Attendee findByPrimaryKey(AttendeeKey key)** Retrieve using a primary key
3. **Enumeration findAllInstances()** Retrieve all instances as an enumeration
4. **Enumeration findByEventRef(String key)** Find instances matching the given Event primary key
5. **Enumeration findByPersonRef(String key)** Find instances matching the given Person primary key.
6. **Enumeration findByEventRefAndPersonRef(String key1, String key2)** Find instances matching both the Event primary key and the Person primary key

4.2. EJB remote interfaces

The remote instance of the *Event* EJB has the following methods:

1. Getter and setter methods for the persistent fields (SUMMARY, DTSTARTand DTEND)
2. **initialize(Document)** Initialize from document
3. **initialize**(EventXMLAdaptor) Initializes the EJBs fields from an XMLAdaptor instance
4. **getXML().** Returns EJB data in XML
5. **String getKey()** Return primary key as a string
6. **ejbActivate()** Required for entity EJBs.
7. **ejbCreate(EventKey)** Initializes EJB default values
8. **ejbLoad()** Requred for entity EJBs
9. **ejbPassivate()** Required for entity EJBs
10. **ejbPostCreate()** Required for entity EJBs
11. **ejbStore()** Required for entity EJBs
12. **ejbRemove()** Required for entity EJBs
13. **setEntityContext(EntityContext)** Required for entity EJBs
14. **unsetEntityContext()** Required for entity EJBs

The remote interface for the Person EJB is very similar:

1. Getter and setter methods for the persistent fields (ID, FIRSTNAME, LASTNAME and TZ)
2. **initialize(Document)** Initialize from XML document
3. **initialize**(PersonXMLAdaptor) Initializes the EJB fields from an XMLAdaptor instance
4. **getXML().** Returns EJB data in XML
5. **String getKey()** Return primary key as a string
6. **ejbActivate()** Required for entity EJBs.
7. **ejbCreate(PersonKey)** Initializes EJB default values
8. **ejbLoad()** Requred for entity EJBs
9. **ejbPassivate()** Required for entity EJBs
10. **ejbPostCreate()** Required for entity EJBs
11. **ejbStore()** Required for entity EJBs
12. **ejbRemove()** Required for entity EJBs
13. **setEntityContext(EntityContext)** Required for entity EJBs
14. **unsetEntityContext()** Required for entity EJBs

The attendee relationship remote interface:

1. **String getEventRef()** Return Event foreign key
2. **String getPersonRef()** Return Person foreign key
3. **setEventRef(String)** Set the Event foreign key
4. **setPersonRef(String)** Set the Person foreign key
5. **String getRole()** Return the role for the attendee relationship
6. **setRole(String)** Set the role for the attendee relationship
7. **String getKey()** Return primary key as a string
8. **initialize(Document)** Initialize from document
9. **initialize(AttendeeXMLAdaptor)** Initialize from XMLAdaptor
10. **getXML()** Return attendee relationship as XML
11. **ejbCreate(AttendeeKey)** Initializes EJB default values
12. **ejbLoad()** Requred for entity EJBs
13. **ejbPassivate()** Required for entity EJBs
14. **ejbPostCreate()** Required for entity EJBs
15. **ejbStore()** Required for entity EJBs
16. **ejbRemove()** Required for entity EJBs
17. **setEntityContext(EntityContext)** Required for entity EJBs
18. **unsetEntityContext()** Required for entity EJBs

4.3. XMLAdaptor classes

An XMLAdaptor instance wraps EJB data and provides an interface very similar to the EJB remote interface. The EventXMLAdaptor in the example above has the following public methods:

1. getter and setter method for wrapped fields (SUMMARY, DTSTART, DTEND)
2. **EventXMLAdaptor(Document)** Construct from an XML document
3. **EventXMLAdaptor(EJBObject)** Construct using an EJB remote object
4. **getXML()** Return XMLAdaptor data in XML
5. **initialize(XMLAdaptor)** Initialize XMLAdaptor from another XMLAdaptor instance
6. **initialize(Document)** Initialize XMLAdaptor from XML document
7. **initialize(EJBObject)** Initialize XMLAdaptor from EJB remote object
8. **java.lang.String getPrimaryKey()** Return primary key of the wrapped EJB
9. **AttendeeXMLAdaptor addAttendee(java.lang.String)** Add a new association between the Event and a Person. The String argument is the primary key for a Person EJB.
10. **Iterator getAttendee()** Return an iteration of AttendeeXMLAdaptor instances associated with the Event
11. **synchronize()** Synchronize the state of the EventXMLAdaptor with its associated EJB and subsequently the database. The method should be used inside business methods of session EJBs.

The getAttendee() methods returns an empty iteration until the EventXMLAdaptor has been expanded to at least level 1 using the expand(…) method of the EventHelper class (see below).

The PersonXMLAdaptor class has the following public interface:

1. getter and setter method for wrapped fields (ID, FIRSTNAME, LASTNAME, TZ)
2. **PersonXMLAdaptor(Document)** Construct from an XML document
3. **PersonXMLAdaptor(EJBObject)** Construct using an EJB remote object
4. **getXML()** Return XMLAdaptor data in XML
5. **initialize(XMLAdaptor)** Initialize XMLAdaptor from another XMLAdaptor instance
6. **initialize(Document)** Initialize XMLAdaptor from XML document
7. **initialize(EJBObject)** Initialize XMLAdaptor from EJB remote object
8. **java.lang.String getPrimaryKey()** Return primary key of the wrapped EJB
9. **AttendeeXMLAdaptor addAttendee(java.lang.String)** Add a new association between the Person and an Event. The String argument is the primary key for an Event EJB.
10. **getTZAdaptor()** Return TimeZoneXMLAdaptor associated with the Person instance.
11. **Iterator getAttendee()** Return an iteration of AttendeeXMLAdaptor instances associated with the Person
12. **synchronize()** Synchronize the state of the EventXMLAdaptor with its associated EJB and subsequently the database. The method should be used inside business methods of session EJBs.

getAttendee() and getTZAdaptor() returns nothing (i.e. empty iteration and null) until the PersonXMLAdaptor has been expanded to level 1 or deeper using the PersonHelper class (see below).

AttendeeXMLAdaptors are created using the **addAttendee(String …)** method of the EventXMLAdaptor or PersonXMLAdaptor class. The interface of this class is:

1. **setEvenRef(String)** Set the Event foreign key of the attendee relationship
2. **String getEventRef()** Get the Event foreign key of the attendee relationship
3. **setPersonRef(String)** Set the Person foreign key of the attendee relationship
4. **String getPersonRef()** Get the Person foreign key of the attendee relationship
5. **setRole(String)** Set the role foreign key of the relationship
6. **String getRole()** Get the role foreign key of the relationship

7. **AttendeeXMLAdaptor(Document)** Construct from an XML document
8. **AttendeeXMLAdaptor(EJBObject)** Construct using an EJB remote object
9. **getXML()** Return AttendeeXMLAdaptor data in XML
10. **initialize(XMLAdaptor)** Initialize XMLAdaptor from another XMLAdaptor instance
11. **initialize(Document)** Initialize XMLAdaptor from XML document
12. **initialize(EJBObject)** Initialize XMLAdaptor from EJB remote object
13. **java.lang.String getPrimaryKey()** Return primary key of the wrapped EJB
14. **getRoleAdaptor()** Return RoleXMLAdaptor associated with the Attendee instance.
15. **synchronize()** Synchronize the state of the EventXMLAdaptor with its associated EJB and subsequently the database. The method should be used inside business methods of session EJBs.

The interfaces for the immutable RoleXMLAdaptor and TimezoneXMLAdaptor are straightforward:

1. **String getRoleType()** Get the value of the role type
2. **static RoleXMLAdaptor findByPrimaryKey(String)** Retrive a RoleXMLAdaptor using its primary key. There's no home interface for immutable data, so the equivalent of EJB finder methods are represented as static methods in the XMLAdaptor class.
3. **static Enumeration findByRoleType(String)** Return an enumeration of RoleXMLAdaptors matching the given role type
4. **static Enumeration findAllInstances()** Return an enumeration of all instances of the RoleTypeXMLAdaptor.
5. **getPrimaryKey()** Return primary key associated with the RoleTypeXMLAdaptor. Because there's no actual EJB wrapped by the RoleTypeXMLAdaptor, the primary key is constructed when the source code is generated.
6. **Document getXML()** Return RoleXMLAdaptor as an XML document

Similarly for the TimezoneXMLAdaptor:

7. **String getTZ()** Get the TZ value
8. **static TimezoneXMLAdaptor findByPrimaryKey(String)** Retrive a TimezoneXMLAdaptor using its primary key.
9. **static Enumeration findByTZ(String)** Return an enumeration of TimezoneXMLAdaptors matching the given timezone
10. **static Enumeration findAllInstances()** Return an enumeration of all instances of the TimezneTypeXMLAdaptor.
11. **getPrimaryKey()** Return primary key associated with the TimezoneXMLAdaptor.
12. **Document getXML()** Return TimezoneXMLAdaptor as an XML document

The primary key values for immutable constants are generated when the source code is generated, and the value changes if one of the following changes:

1. The name of one of the immutable fields is changed
2. The type of an immutable fields is changed
3. The value of an immutable field for the instance is changed
4. The name of the immutable XMLAdaptor is changed

Primary keys for immutable data can be used as foreign keys in the persistent CMP EJBs, but care should be taken whenever a primary key is changed so that data consistency is maintained.

4.4. Helper classes

Helper classes are meant to sit between session EJBs and the CMP entity EJBs in the architecture (see Figure 4). They provide common methods for creating, deleting and updating entity EJBs and the relationships between EJBs. In the example, the EventHelper class has the following interface:

1. **static EventHelper getInstance()** Return the singleton instance of the EventHelper class
2. **EventXMLAdaptor expand(EventXMLAdaptor, int)** Expands an EventXMLAdaptor instance to the desired level (specified by the integer). If the integer level is set to 0, the fields of the XMLAdaptor are simply refreshed from the associated EJB cmp fields. An expansion to level 1 will retrieve all relationships between the XMLAdaptor and other entities, that is one-to-many relationships represented using foreign key fields and many-to-many relationships stored in separate relationship tables. The relationships instances are saved in the XMLAdaptor so that they can be queried later (for instance by the getXML method). The method will recursively call expand on the relationship EJBs and the related EJBs until the desired level number is reached (as specified by the passed integer).

3. **remove(EventXMLAdaptor)** Removes the EventXMLAdaptor and the relationships between the EventXMLAdaptor and other EJBs.
4. **EventXMLAdaptor create(EventXMLAdaptor)** Create a new instance of the EventXMLAdaptor, including relationships between the EventXMLAdaptor and other EJBs. The returned XMLAdaptor has the primary key field set.
5. **EventXMLAdaptor update(EventXMLAdaptor E, EventXMLAdaptor U).** Update the first EventXMLAdaptor **E** using information from the second EventXMLAdaptor (**U**). Fields that are set in **U** are updated in **E**, and all relationships are removed from **E** and replaced with relationships contained in **U**.
6. **EventXMLAdaptor updateFields(EventXMLAdaptor E, EventXMLAdaptor U)** Update **E** using the fields set in **U**. No relationships are changed.

The PersonHelper class is identical to the EventHelper class. For relationships, the methods are (using Attendee as an example):

1. **static AttendeeHelper getInstance()** Return singleton instance of the helper class
2. **AttendeeXMLAdaptor expand(AttendeeXMLAdaptor, int)** Expand the AttendeeXMLAdaptor

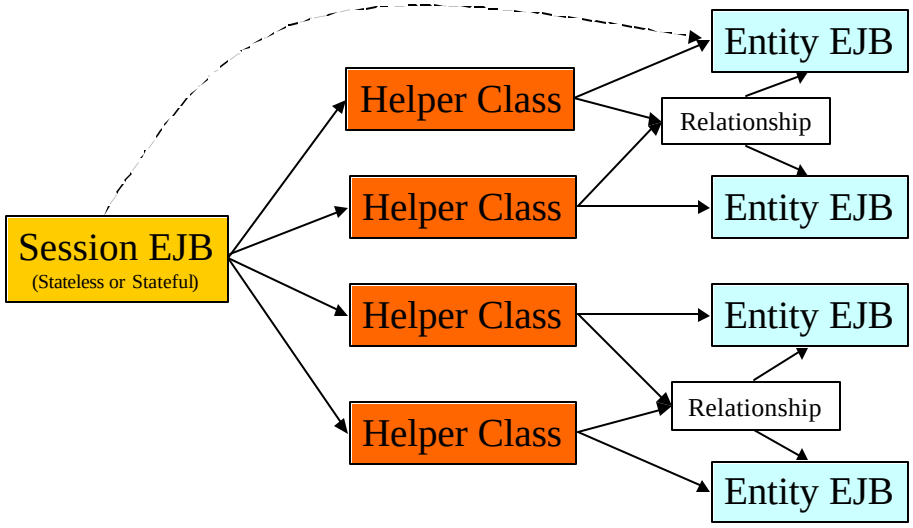

Figure 4. Recommended application architecture. One (or several) session EJBs provides the public interface to the application. The session EJB uses generated helper classes for creating, modifying or deleting entity EJBs. A helper class for a particular entity EJB only access data associated with that EJB and its relationship to other EJBs. A session EJB can also access the entity EJBs directly when needed (as indicated by the dotted line).

instance to the desired level. The expand() method of the two foreign EJBs are called recursively until the desired level is reached.

3. **remove(AttendeeXMLAdaptor)** Remove the given XMLAdaptor (and it's associated EJB).
4. **AttendeeXMLAdaptor create(AttendeeXMLAdaptor)** Create a new attendee relationship. The returned XMLAdaptor has the primary key field set.

This following code performs creation, updates and deletion of EJBs instances using generated Helper classes:

```
// Retrieve singleton helper instance for Person
PersonHelper personHelper = PersonHelper.getInstance();
// Create new PersonXMLAdaptor instance
PersonXMLAdaptor adaptor = new PersonXMLAdaptor();
// Initialize some fields
adaptor.setFIRSTNAME("Joe");
adaptor.setLASTNAME("Schmoe");
// … create it. The primary key of the new Person instance is a random string
PersonXMLAdaptor newPerson = personHelper.create(adaptor);

// Similarly for a new event:

EventHelper eventHelper = EventHelper.getInstance();
EventXMLAdaptor eventAdaptor = new EventXMLAdaptor();
eventAdaptor.setSUMMARY("This is the summary");
eventAdaptor.setDTSTART(…);eventAdaptor.setDTEND(…);
// Add person as an attendee of the event
eventAdaptor.addAttendee(newPerson.getPrimaryKey());
// create event with associated relationship
EventXMLAdaptor newEvent = eventHelper.create(eventAdaptor);

// Change the summary of the event:
EventXMLAdaptor updateEvent = new EventXMLAdaptor();
updateEvent.setSUMMARY("Updated summary");
newEvent = eventHelper.updateFields(newEvent, updateEvent);

// Retrieve another person from database:

Person person = PersonHome.findByPrimaryKey("…");
PersonXMLAdaptor person2XMLadaptor = new PersonXMLAdaptor(person);

// Add the person as an attendee of the event we just created

AttendeeHelper att_helper = AttendeeHelper.getInstance();
AttendeeXMLAdaptor attendeeXMLAdaptor = new AttendeeXMLAdaptor();
attendeeXMLAdaptor.setEventRef(newEvent.getPrimaryKey());
attendeeXMLAdaptor.setPersonRef(person2XMLAdaptor.getPrimaryKey());
att_helper.create(attendeeXMLAdaptor);

// Delete the event. This will also delete the associated relationship between the event and
// the person.

eventHelper.remove(newEvent);

// Remove a person
personHelper.remove(newPerson);
```

5. Home interface pooling class

Home interfaces for EJBs are used when there's a need to create new instances of EJBs or for retrieving existing instances using finder methods. The generated helper classes uses a home interface pooling class to retrieve home interfaces, and the same class can also be used in other parts of the code.

The HomeObjectPool class is generated in the same folder as the CMP EJBs, under the HomePool directory. To retrieve the singleton instance of the HomeObjectPool class, use the following:

```
HomeObjectPool pool = HomeObjectPool.getInstance(); // uses default number of home interface instandces / ejb in the pool (=50)
```

or to tailor the pool using a different number of home interface objects, use:

```
HomeObjectPool pool = HomeObjectPool.getInstance(size);
```

where size is an integer representing the number of home interfaces / ejb created initially in the pool. Now to retrieve the home interface for a particular EJB, call 'get' passing the JNDI name of the home interface you want to retrieve:

```
EJBHome home = pool.get("jndi-name");
```

When you're done using the home interface, remember to return it to the pool so that others can use it later.

```
pool.release(home);
```

The size of the pool grows dynamically when needed.

6. XML schemas for EJB data

All entity Enterprise JavaBeans created by EJBMaker can export themselves as XML using its XMLAdaptor class. Example:

```
EventXMLAdaptor event = new EventXMLAdaptor(event_ejb);
Document d = event.getXML();
```

The element name of the top element in the XML document is identical to the JNDI name of the EJB. One attribute, 'key', contains the primary key of the EJB. The top element encloses one subelement for each cmp field in the EJB, and the value of the subelement is the string value of the field (retrived by calling toString() method of the instance). Example from the Person EJB described above:

```
<?xml version="1.0"?>
<Person key="P1">
        <ID>Joe</ID>
        <FIRSTNAME>Joe</FISTNAME>
        <LASTNAME>Schmoe</LASTNAME>
        <TZ>T1</TZ>
</Person>
```

To retrieve more detailed XML, an XMLAdaptor instance can be extended using the expand() method of its helper class. Example

```
PersonXMLAdaptor expandedPerson = PersonHelper.getInstance().expand(person, 1);
```

If the TZ reference in the XML document above refers to the following Timezone data:

```
<Timezone key = “T1”>
        <TZ>PST</TZ>
</Timezone>
```

calling getXML() on the expandedPerson instance returns the following:

```
<?xml version=”1.0”?>
<Person key=”P1”>
        <ID>Joe</ID>
        <FIRSTNAME>Joe</FISTNAME>
        <LASTNAME>Schmoe</LASTNAME>
        <TZ>
                <Timezone key = “T1”>
                        <TZ>PST</TZ>
                </Timezone>
        </TZ>
</Person>
```

Many to many relationships is also expanded using the expand method of the helper class. The relationships are stored in list elements named after the name of the relationship followed by ‘-list’. For instance, an Event EJB related to two Persons instances via the Attendee relationship looks like this before expand is called:

```
<?xml version = “1.0”>
<Event key = “E1”>
        <SUMMARY>Some summary</SUMMARY>
        <DTSTART>July 3, 2001 14:00</DTSTART>
        <DTEND>July 3, 2001, 16:00</DTEND>
</Event>
```

Calling expand() using level 1 on its helper class results in the following XML:

```
<?xml version = “1.0”>
<Event key = “E1”>
        <SUMMARY>Some summary</SUMMARY>
        <DTSTART>July 3, 2001 14:00</DTSTART>
        <DTEND>July 3, 2001, 16:00</DTEND>
        <Attendee-list>
                <Attendee key=”A1”>
                        <PersonRef>P1</PersonRef>
                        <EventRef>E1</EventRef>
                        <Role>R1</Role>
                </Attendee>
                <Attendee key = “A2”>
                        <PersonRef>P2</PersonRef>
                        <EventRef>E1</EventRef>
                        <Role>R2</Role>
                </Attendee>
        </Attendee-list>
</Event>
```

If we go one step further and use the helper class to extend the Event to level 2, we get the following:

```
<?xml version = “1.0”>
<Event key = “E1”>
        <SUMMARY>Some summary</SUMMARY>
        <DTSTART>July 3, 2001 14:00</DTSTART>
        <DTEND>July 3, 2001, 16:00</DTEND>
```

```
        <Attendee-list>
                <Attendee key=”A1”>
                        <PersonRef>
                                <Person key=”P1”>
                                        <ID>Joe</ID>
                                        <FIRSTNAME>Joe</FISTNAME>
                                        <LASTNAME>Schmoe</LASTNAME>
                                        <TZ>T1</TZ>
                                </Person>
                        </PersonRef>
                        <EventRef>E1</EventRef>
                        <Role>
                                <Role key=”R1”>
                                        <RoleType>Chair</RoleType>
                                </Role>
                        </Role>
                </Attendee>
                <Attendee key = “A2”>
                        <PersonRef>
                                <Person key=”P2”>
                                        <ID>Tim</ID>
                                        <FIRSTNAME>Tim</FISTNAME>
                                        <LASTNAME>Berners-Lee</LASTNAME>
                                        <TZ>T1</TZ>
                                </Person>
                        </PersonRef>
                        <EventRef>E1</EventRef>
                        <Role>
                                <Role key=”R2”>
                                        <RoleType>Attendee</RoleType>
                                </Role>
                        </Role>
                </Attendee>
        </Attendee-list>
</Event>
```

As can be seen by the example above, the size of the XML document grows quickly when an XMLAdaptor is expanded several levels deep.

7. Installing the sample application.

The following instructions are for WebSphere 4.0 installation. For WebSphere 3.5, see the EJBMaker 1.0 documentation.

The EJBMaker 2.0 distribution includes a sample application to get you up and running quickly. Follow these instructions to install the application:

1. Change the current directory to the ‘exa mple’ folder under the EJBMaker installation root (e.g. C:\EJBMaker2\example).
2. Edit the ‘buildall’ batch file and change the WASDIR and the EJBMAKERDIR to point to the correct installation locations for your environment.
3. Invoke the ‘buildall’ batch file. This will generate EJB source code, compile it, package it into a jar file and generate the deployment code. The deployable application (‘DeployableExample.jar’) is stored in under the install/classes folder in your installation directory. Copy it to the installableApps folder under WebSphere.

7.1. Configuring the database

Create a database by simply issuing the following command at the DB2 CLI prompt:

create database example

(make sure you're executing the command under a user id with database creation permission).

Connect to the database using:

connect to example

Now exit the CLI environment by typing '*exit*' and change directory to the 'example' folder under your EJMaker2 installation directory. Issue the following command at the DOS prompt:

db2 –f CreateTables.ddl

This will create the tables required for your CMP EJBs using the DB2 data types defined in the example.xml file.

7.2. Deployment in WebSphere 4

First create a JDBC datasource in WebSphere by opening up the Resources node, followed by the DB2JdbcDrivers node and clicking on Data Sources. If you don't have the DB2JdbcDriver node, follow the 'Database config' instructions in the sample applications shipped with WebSphere.

Now click on 'New' and enter the following information:

Name: Example
JNDI Name: jdbc/Example
Description: Example database
Category: Samples
Database Name: example
Default user id: <your db id>
Default password: <password>

Click OK and save your configuration.

Install the application (DeployableExample.jar) in WebSphere 4 using the Administrator's console. When asked for the JNDI names of the three EJBs, enter 'Event', 'Person', 'Attendee' and 'Session'. Enter 'jdbc/Example' as data source and make sure to enter an authorized userid and password for the data source. Uncheck the option that enables you to regenerate the deployment code for the EJBs (this has already been done).

If the application server gives an IO Exception during the deployment stage it is safe to ignore it (this is a bug). When the application is installed, save your configuration and restart the application server.

7.3. Running the sample client

Change directory to the 'example' folder under your EJBMaker2 installation. Edit the 'run.bat' file and change the WASDIR and the EJBMAKERDIR variables to your environment settings. Save the file and invoke the run batch file.

The sample client does the following:

1. It uses the Person helper class to create a new instance of a person object
2. Creates a new Event instance and adds the Person instance created in step 1 as an attendee of the Event
3. It expands the Event instance to level 1, 2 and 3 using the Event helper class and displays the resulting XML.
4. It removes all the EJBs created using the helper classes

8. Deployment descriptor merger tool

An application consist of entity EJBs and session EJBs, but the deployment descriptors generated by the EJBMaker source generator only contains descriptors for the CMP entity EJBs in a project. Instead of manually having to merge this file with session EJB descriptors, the DDMerger tool does it automatically for you. To invoke the tool, use the following:

java com.ibm.almaden.ejbmaker.DDMerger <DD1> <DD2> <target DD>

where

DD1 is the first XML deployment descriptor being merged
DD2 is the second XML deployment descriptor
target DD is the output filename for the merged deployment descriptor

The tool merges all EJBs described in *DD2* into the *DD1* deployment descriptor.

9. Encryption support

Sensitive data, such as passwords, can be flagged encrypted. To enable encryption, the '–e' flag must be passed to the ejbmaker tool, and the sensitive field must be marked encrypted by specifying the *encrypt = 'true'* attribute in the persistent field element. Encryption is performed using the DES algorithm, and keys are generated randomly. Currently, only strings can be encrypted.

Encryption is handled using Sun's Java Cryptography Extension (JCE) reference implementation.

References

[1] *Enterprise JavaBeans Specification version 2.0* http://java.sun.com/products/ejb/docs.html
[2] *Sun Java Center J2EE Patterns, Value Object*
http://developer.java.sun.com/developer/restricted/patterns/ValueObject.html

Appendix

The EJBMaker.dtd file

```
<!ELEMENT ejbmaker (transaction-attr? , isolation-level? , run-as-mode? , re-entrant?, ( bean | relation )*) >
<!ELEMENT bean (persistent-field | finder-method)*>
<!ELEMENT ejbmaker (transaction-attr? , isolation-level? , run-as-mode? , re-entrant?, ( bean | relation |
constant-bean )*) >
<!ELEMENT bean (persistent-field | finder-method)*>
<!ATTLIST bean name CDATA #REQUIRED>
<!ATTLIST bean type (entity | topic) #REQUIRED>
```

```
<!ELEMENT persistent-field (#PCDATA)>
<!ATTLIST persistent-field dt CDATA #REQUIRED>
<!ATTLIST persistent-field col-dt CDATA #REQUIRED>
<!ATTLIST persistent-field fkey CDATA #IMPLIED>
<!ATTLIST persistent-field default CDATA #IMPLIED>
<!ATTLIST persistent-field encrypt CDATA #IMPLIED>
<!ELEMENT finder-method (arg-type*, sql)>
<!ATTLIST finder-method name CDATA #REQUIRED>
<!ELEMENT arg-type (#PCDATA)>
<!ELEMENT sql (#PCDATA)>
<!ELEMENT relation (bean-ref-1 , bean-ref-2, persistent-field*)>
<!ATTLIST relation name CDATA #REQUIRED>
<!ELEMENT bean-ref-1 (#PCDATA)>
<!ELEMENT bean-ref-2 (#PCDATA)>
<!ELEMENT constant-bean (field | instance)*>
<!ATTLIST constant-bean name CDATA #REQUIRED>
<!ELEMENT field (#PCDATA)>
<!ATTLIST field dt CDATA #REQUIRED>
<!ELEMENT instance (initialize)*>
<!ELEMENT initialize (#PCDATA)>
<!ATTLIST initialize field CDATA #REQUIRED>
<!ELEMENT transaction-attr (#PCDATA)>
<!ELEMENT isolation-level (#PCDATA)>
<!ELEMENT run-as-mode (#PCDATA)>
<!ELEMENT re-entrant (#PCDATA)>
```