



Mapping Platform Independent Models to Entity Java Beans 3.0

Carolyn Duby

Version 1.10
April 22, 2009

Pathfinder Solutions LLC
33 Commercial Street, Suite 2
Foxboro, MA 02035 USA
www.PathfinderMDA.com
888-662-7284

Table Of Contents

1. Introduction	4
2. Target Document Set	4
3. PIM to EJB 3.0 Mapping Rules	4
Classes	4
Attributes	4
Temporal Data Types	6
Enumerated Types	6
Binary Associations	6
Inheritance Relationships	18
Behavior	19
4. Deployment	26
Specifying the Data Source	26
JNDI Names for Session Beans	26
Table Name Length Checking	26
5. Design Decisions	27
Annotations vs. XML Deployment Descriptor	27
Representing Composite Primary Keys	27
Multitable Mappings	27
Collection Types for Many Associations	27
FIFO and LIFO Instance Ordering	27
Cascading	28
Inheritance Strategy	28
Association Classes	28
6. Markings	28
System	28
Domain	29
Domain Service	30
Class	31
Attribute	31
Participant	32

Association.....	33
User Defined Type.....	33
7. Annotation to Marking Index	33
8. Model Restrictions.....	34
9. References	35
10. Appendix.....	35
How to Specify Stereotypes for Model Elements.....	35

1. Introduction

This document describes how to translate Platform Independent Models into Entity Java Beans (EJB) 3.0. It describes the design patterns and markings used by the transformation.

2. Target Document Set

The EJB Transformation Map will generate annotated Java and the persistence.xml deployment descriptor.

3. PIM to EJB 3.0 Mapping Rules

This section describes mapping rules that will be added to the PathMATE Java Transformation Map to support EJB 3.0.

Classes

Classes with the EjbEntityBean marking equal to "T" map to Entity Beans. Entity Beans will implement the java.io.Serializable interface. Because the entire population of entity beans is stored in the database, the entity bean does not need an instance list.

EJB divides logic from data. Logic is contained in a session bean and data is stored in an entity bean. This is contrary to OO modeling which combines logic and data into the same class abstraction. For each class in the PIM marked as an entity bean, the transformation rules will create two classes. A class called <class_name>Logic to handle the operations and one called <class_name> which is the entity bean itself. We use a plain java class for the Logic class since it is not used outside the domain. The entity bean itself will contain only the data for the class. The operations and state actions will be in the Logic class.

An optional EjbTable marking allows the user to specify the table where the entity will be stored.

Some databases such as Oracle restrict the length of table names. Class names generated from the default EJB persistence mappings are often too long. If the table is not specified, the table name will be set to EjbTablePrefix property for the domain followed by an underscore followed by the identifier of the class.

Classes with EjbEntityBean marking equal to "F" will have the default Java mappings.

Attributes

Attributes will map to private data of the class.

For entity beans, a get and set methods will be generated for each attribute. The get and methods will be annotated with the Column annotation if the EjbColumn marking is set on the attribute.

If an entity bean attribute has the <<Identifier>> stereotype applied, the Id annotation will be inserted before the get and set methods. For more information on how to specify a stereotype see How to Specify Stereotypes for Model Elements on page 35. The EjbldGenerationStrategy marking determines if an identifier is automatically generated. The strategy may be either AUTO, IDENTITY, TABLE or SEQUENCE. If the strategy is AUTO, the persistence provider will generate a unique identifier automatically. If the strategy is IDENTIFIER, a special identifier column type will be used.

If the strategy is TABLE, the EjbldGeneratorProperties marking specifies the properties of the TableGenerator annotation excluding the name. The templates will construct the annotation as follows:

```
@TableGenerator(name="<class_name>_GENERATOR"
<EjbldGeneratorProperties marking>)
```

If the strategy is SEQUENCE, the EjbldGeneratorProperties marking specifies the properties of the SequenceGenerator annotation excluding the name. The templates will construct the annotation as follows:

```
@SequenceGenerator(name="<class_name>_GENERATOR"
<EjbldGeneratorProperties marking>)
```

If the class has compound identifiers, a primary key class is generated. The primary key class is called <class_name>PK. It implements the Serializable interface, has a public no argument constructor, and has fields to represent each attribute in the compound identifier. Get and set methods are provided for each of the identifier attributes. The equals and hashCode methods are implemented. The IdClass annotation is added to the entity bean.

If the class has no identifier attribute, an integer attribute named generatedId will be added to the entity bean with an automatically generated value. To use the sequences for automatically generated identifiers, set the system property EjbGenSequenceGenerators to T. A GeneratedValue and SequenceGenerator annotation are added to the generated identifier annotations as follows:

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
generator = "SQ_<class_table_name>")
@SequenceGenerator(name = "SQ_<class_table_name>",
sequenceName = "SQ_<class_table_name>",
initialValue = 1, allocationSize = 1)

public int getGeneratedId() { return generatedId; }
public void setGeneratedId(int generatedId)
{ this.generatedId = generatedId; }
```

Generate the file generate_sequences.sql containing the SQL to create the sequences.

If an attribute is marked with `EjbTransient = T`, the `Transient` annotation will be applied to the attribute's get and set methods.

Entity beans that have state models will have an implicit `currentState` attribute to persist the current state of the bean.

Temporal Data Types

New system level user defined primitive types represent time stored in entity beans:

`sys_date_t`

`sys_time_t`

`sys_timestamp_t`

All of these types will map to the `java.util.Calendar` type. Each one will have a different `@Temporal` annotation that affects the data type used to store the attribute in the database.

User Defined Type	Temporal Annotation
<code>sys_date_t</code>	<code>@Temporal(TemporalType.DATE)</code>
<code>sys_time_t</code>	<code>@Temporal(TemporalType.TIME)</code>
<code>sys_timestamp_t</code>	<code>@Temporal(TemporalType.TIMESTAMP)</code>

Enumerated Types

Previously the java map generated integer constants for enumerated types because enumerated types were not available in Java.

Enumerated types are now mapped to java enums. The enums will use the default `@Enumerated(EnumType.ORDINAL)` mapping.

Binary Associations

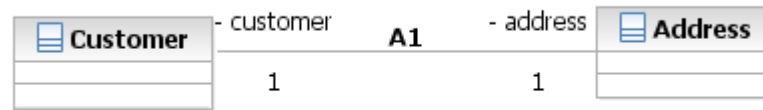
Binary associations will be represented in entity beans by using the `OneToOne`, `OneToMany`, `ManyToOne`, and `ManyToMany` annotations. The action language will be analyzed to determine if the association is bidirectional or unidirectional.

For all the examples assume the classes are in the Purchasing domain.

One to One Unidirectional Association

A one to one unidirectional association creates an attribute to formalize the relationship and set and get methods for the reference on *one* side of the association.

In this example the A1 association is navigated only from Customer to Address in the action language. The Customer entity formalizes the association but the Address class does not.



```
@Entity
```

```
public class Purchasing_Customer implements
java.io.Serializable {
    private Purchasing_Address acrossA1_to_address;

    @OneToOne
    public Purchasing_Address getAcrossA1_to_address() {
        return acrossA1_to_address;
    }
    public void setAcrossA1_to_address(
        Purchasing_Address other_side) {
        acrossA1_to_address = other_side;
    }
}
```

```
@Entity
```

```
Public class Purchasing_Address implements
Java.io.Serializable {
}
```

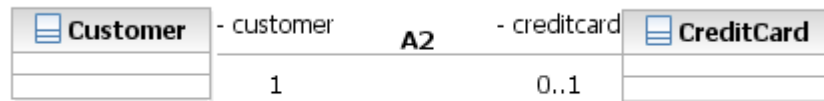
One to One Bidirectional Association

A one to one unidirectional association creates an attribute to formalize the relationship and set and get methods for the reference on *both* sides of the association.

In this example the A2 association is navigated in both directions in the action language. The Customer entity and the CreditCard classes both formalize the association. The mappedBy property specifies that the acrossA2_to_creditcard and acrossA2_to_customer fields formalize opposite sides of the association.

The one to one association could be formalized in either the Customer or the Credit Card class. The transformation rules will pick the class where the EjbJoinColumn marking is specified. If no join column is specified and there is one conditional participant and one unconditional participant, the participant with the unconditional multiplicity will be chosen. For example, the Credit Card always has a Customer, but the Customer does not always have a CreditCard. If none of these rules

apply, i.e. both sides are unconditional or conditional, the first participant will be chosen.



```

@Entity
public class Purchasing_Customer implements
    java.io.Serializable
{
    private Purchasing_CreditCard acrossA2_to_creditcard;

    @OneToOne
    public Purchasing_CreditCard
        getAcrossA2_to_creditcard()
    {
        return acrossA2_to_creditcard;
    }
    public void setAcrossA2_to_creditcard(
        Purchasing_CreditCard other_side)
    {
        acrossA2_to_creditcard = other_side;
    }
}

```

```

@Entity
public class Purchasing_CreditCard implements
    Java.io.Serializable
{
    private Purchasing_Customer acrossA2_to_customer;

    @OneToOne(mappedBy="acrossA2_to_creditcard")
    public Purchasing_Customer getAcrossA2_to_customer()
    {
        return acrossA2_to_customer;
    }
}

```



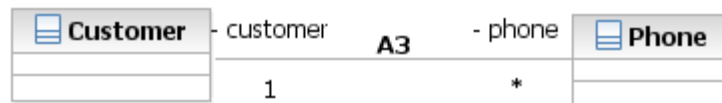
```

    }
    public void setAcrossA2_to_customer(
        Purchasing_Customer other_side)
    {
        acrossA2_to_customer = other_side;
    }
}

```

One to Many Unidirectional Association

If there is a 1 to * association between two classes and the association is only traversed from the 1 side to the * side, the OneToMany annotation is generated.



```

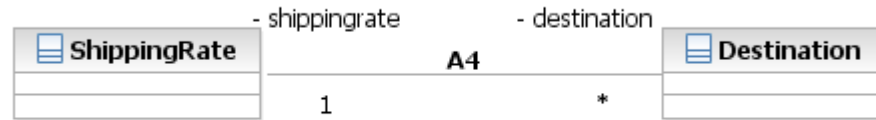
@Entity
public class Purchasing_Customer implements
    java.io.Serializable
{
    private Set<Purchasing_Phone>
        acrossA3_to_phone;

    @OneToMany
    public Set<Purchasing_Phone>
        getAcrossA3_to_phone()
    {
        return acrossA3_to_phone;
    }
    public void setAcrossA3_to_phone(
        Set<Purchasing_CreditCard> other_side)
    {
        acrossA3_to_phone = other_side;
    }
}

```

Many To One Unidirectional Association

If there is a 1 to * association between two classes and the association is only traversed from the * side to the 1 side, the ManyToOne annotation is generated.



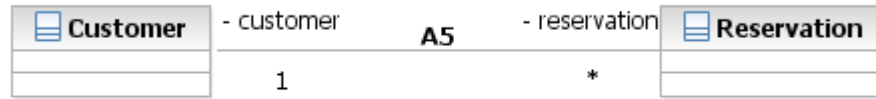
```

@Entity
public class Purchasing_Destination implements
    Java.io.Serializable
{
    private Purchasing_ShippingRate
        acrossA4_to_shippingrate;
    @ManyToOne
    public Purchasing_ShippingRate
        getAcrossA4_to_shippingrate()
    {
        return acrossA4_to_shippingrate;
    }
    public void setAcrossA4_to_shippingrate(
        Purchasing_ShippingRate other_side)
    {
        acrossA4_to_shippingrate = other_side;
    }
}

@Entity
public class Purchasing_ShippingRate implements
    java.io.Serializable
{
}
  
```

One to Many Bidirectional Association

When there is a 1 to * association and it is traversed in both directions in action language, the OneToMany and ManyToOne annotations are inserted. The OneToMany side specifies the mappedBy property.



```

@Entity
public Purchasing_Customer implements java.io.Serializable
{
    private Set<Purchasing_Reservation>
        acrossA5_to_reservation;

    @OneToMany(mappedBy="acrossA5_to_customer")
    public Set<Purchasing_Reservation>
        getAcrossA5_to_reservation()
    {
        return acrossA5_to_customer;
    }

    public void setAcrossA5_to_reservation(
        Set<Purchasing_Reservation> other_side)
    {
        acrossA5_to_reservation = other_side;
    }
}

```

```

@Entity
public Purchasing_Reservation implements
java.io.Serializable
{
    private Purchasing_Customer
        acrossA5_to_customer;

    @ManyToOne
    public Purchasing_Customer getAcrossA5_to_customer()

```

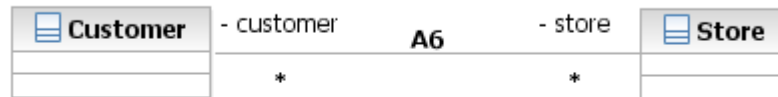
```

    {
        return acrossA5_to_customer;
    }
    public void setAcrossA5_to_customer(
        Purchasing_Customer other_side)
    {
        acrossA5_to_customer = other_side;
    }
}

```

Many to Many Unidirectional Association

When a * to * association is navigated in one direction only, the ManyToMany annotation is generated.



```

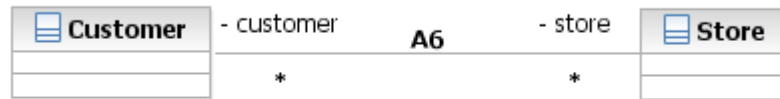
public class Purchasing_Customer implements
    java.io.Serializable
{
    private Set<Purchasing_Store>
        acrossA6_to_store;

    @ManyToMany
    public Set< Purchasing_Store >
        getAcrossA6_to_store()
    {
        return acrossA6_to_store;
    }
    public void setAcrossA6_to_store(
        Set<Purchasing_Store> other_side)
    {
        acrossA6_to_store = other_side;
    }
}

```

Many to Many Bidirectional Association

When a * to * association is navigated in both directions, the ManyToMany annotation is generated in both participating classes. The first participant in the association will define the relationship including any specified JoinTables. The second participant will specify the mappedBy property of the ManyToMany annotation.



```

@Entity
public class Purchasing_Customer implements
    java.io.Serializable
{
    private Set<Purchasing_Store>
        acrossA6_to_store;

    @ManyToMany
    public Set< Purchasing_Store >
        getAcrossA6_to_store()
    {
        return acrossA6_to_store;
    }
    public void setAcrossA6_to_store(
        Set<Purchasing_Store> other_side)
    {
        acrossA6_to_store = other_side;
    }
}

@Entity
public class Purchasing_Store implements
    java.io.Serializable
{
    private Set<Purchasing_Customer>
        acrossA6_to_customer;
  
```

```

@ManyToMany(mappedBy="acrossA6_to_store")
public Set< Purchasing_Customer >
    getAcrossA6_to_customer()
{
    return acrossA6_to_customer;
}
public void setAcrossA6_to_customer(
    Set<Purchasing_Customer> other_side)
{
    acrossA6_to_customer = other_side;
}
}

```

Join Columns

You can specify the column used to join the relationship using the `EjbJoinColumn` marking on the participant. For example suppose in the unidirectional association example we wanted to specify the name of the column used to join the A1 association.

NOTE: Only one participant in the association should specify the `EjbJoinColumn` marking.

Attach the `EjbJoinColumn` to the address participant by adding the following to the `properties.txt` file:

Participant, CustomerRelations.Purchasing.A1.Address.address, EJBJoinColumn, ADDRESS_ID

The Customer entity includes the `JoinColumn` annotation:

```

@Entity
public class Purchasing_Customer implements
    java.io.Serializable {
    private Purchasing_Address acrossA1_to_address;

    @OneToOne
    @JoinColumn(name="ADDRESS_ID")
    public Purchasing_Address getAcrossA1_to_address() {
        return acrossA1_to_address;
    }
}

```

```

        public void setAcrossAl_to_address(
            Purchasing_Address other_side) {
            acrossAl_to_address = other_side;
        }
    }

```

Primary Key Joins

If the two primary keys of the related entity are identical no extra columns are required to implement the association. Specify the `EjbPrimaryKeyJoinColumn` marking to T to add the `@PrimaryKeyJoinColumn` annotation.

```

@Entity
public class Purchasing_Customer implements
    java.io.Serializable {

    private Purchasing_Address acrossAl_to_address;

    @OneToOne
    @PrimaryKeyJoinColumn
    public Purchasing_Address getAcrossAl_to_address() {
        return acrossAl_to_address;
    }

    public void setAcrossAl_to_address(
        Purchasing_Address other_side) {
        acrossAl_to_address = other_side;
    }

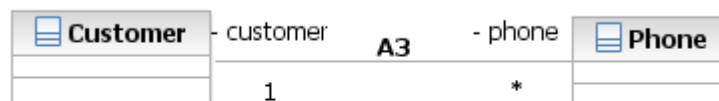
}

```

Join Table

Another way to implement an association is by using a join table, a table containing a column for each of the identifier attributes of the participating classes. To specify a `JoinTable`, mark the association with the name of the join table using the `EjbJoinTable` marking. Mark the name of the join table column for each of the participants by adding the `EjbJoinTableColumn` marking to each of the participants.

For example, we could specify the join table and column names for the one to many unidirectional association A3:



BinaryRel, CustomerRelations.Purchasing.A3, EJBJoinTable, CUSTOMER_PHONE

Participant, CustomerRelations.Purchasing.A3.Customer.customer, EJBJoinTableColumn, CUSTOMER_ID

Participant, CustomerRelations.Purchasing.A3.Phone.phone, EJBJoinTableColumn, PHONE_ID

The JoinTable annotation is inserted under the OneToMany

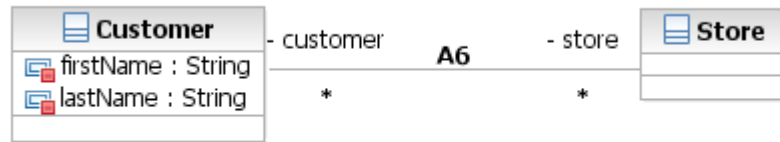
@Entity

```
public class Purchasing_Customer implements
    java.io.Serializable
{
    private Set<Purchasing_Phone>
        acrossA3_to_phone;

    @OneToMany
    @JoinTable(name="CUSTOMER_PHONE"),
        joinColumns={@JoinColumn(name="CUSTOMER_ID")},
        inverseJoinColumns={@JoinColumn(name="PHONE_ID")}
    public Set<Purchasing_Phone>
        getAcrossA3_to_phone()
    {
        return acrossA3_to_phone;
    }
    public void setAcrossA3_to_phone(
        Set<Purchasing_CreditCard> other_side)
    {
        acrossA3_to_phone = other_side;
    }
}
```

Ordered Associations

To order an association, set the SortKey property of the participant to a comma separated list of the names of the sort key followed by the direction of the sort. Use ASC for ascending and DESC for descending order. For example, to sort the list of customers that shop at a store ascending by first name only, set the SortKey property of the Customer participant to "firstName ASC". The OrderBy annotation is entered and a List is used to represent the relationship rather than a Set.



```

@Entity
public class Purchasing_Store implements
    java.io.Serializable
{
    private List<Purchasing_Customer>
        acrossA6_to_customer;

    @ManyToMany(mappedBy="acrossA6_to_store")
    @OrderBy("firstName ASC")
    public List< Purchasing_Customer >
        getAcrossA6_to_customer()
    {
        return acrossA6_to_customer;
    }
    public void setAcrossA6_to_customer(
        List<Purchasing_Customer> other_side)
    {
        acrossA6_to_customer = other_side;
    }
}

```

To sort ascending by last name as the primary key and then first name as a secondary key, set SortKey of the Customer participant to "lastName ASC, firstName DESC".

Fetch Optimization

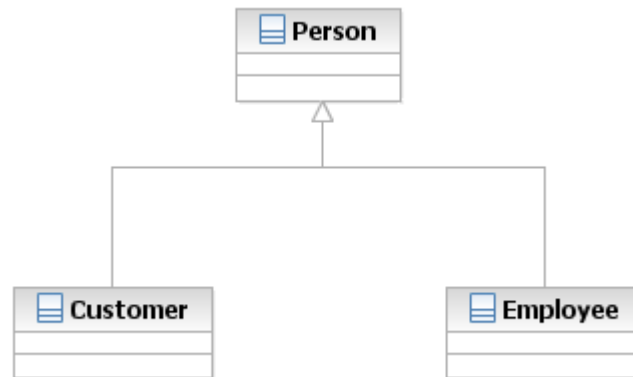
When an entity is retrieved from the database its entities related with multiplicity many are not retrieved by default. If the relationship is traversed after the source entity is retrieved, the application will need to make another query to the database. For associations that are frequently traversed, you can set the EjbFetchType to EAGER to cause the entity and its related classes to be fetched in one database query.

Use this optimization sparingly and consider carefully the entire network of entities. Using this optimization on too many associations could cause slow performance because a large amount of data must be queried each time a parent entity is retrieved.

Inheritance Relationships

The Single Table strategy is used for inheritance hierarchies. The instances of all the classes in the hierarchy are stored in a single table. The table contains columns for the superset of attributes contained in all classes in the hierarchy. A single table containing the superset of all the attributes of the classes in the hierarchy is created. A discriminator attribute is added to determine the subclass of the row.

The `@Inheritance` annotation will be added to the class at the root of the inheritance hierarchy.



```
@Entity
```

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

```
public class Purchasing_Person implements
java.io.Serializable
{
}

```

The `EjbDiscriminatorColumn` specifies the name of the column used to determine the class stored in a row. The `DiscriminatorColumn` annotation is inserted. For example if we set the `EjbDiscriminatorColumn` marking for the `Person` class to `"PERSON_TYPE"` and the `EjbDiscriminatorColumnType` to `INTEGER`, the following code will be generated:

```
@Entity
```

```
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
```

```
@DiscriminatorColumn(name="PERSON_TYPE", discriminatorType  
= DiscriminatorType.INTEGER)
```

```
@DiscriminatorValue("1")
```

```
public class Purchasing_Person implements
java.io.Serializable
{
}
```

For INTEGER and CHAR discriminator types, the templates will automatically assign numbers or characters to each subtype. For String discriminators, the default is to use the class name. The default discriminator value can be overridden by specifying the `EjbDiscriminatorValue` for a class in the hierarchy.

Behavior

Domain Services

The domain interface maps to a session bean. The session bean has a method for each service.

The `EjbPersistenceContextUnitName` marking specifies the name of the persistence unit for the domain. The persistence units are data sources defined in the `persistence.xml` deployment descriptor.

The `EjbPersistenceContextType` specifies whether the persistence context is transaction scoped or extended. Transaction scoped contexts are destroyed after a transaction completes. All the entity beans are detached from the context when the transaction completes. They become plain java objects and subsequent changes with set functions are not written to the database.

Extended persistence contexts are maintained outside of transactions.

The transformation rules create remote and local interfaces that implement the domain interface. The rules also create a domain session bean that implements both the remote and local interfaces. For example, the java templates generate an interface class:

```
public interface <domain_name>_IF
{
    // signatures of services provided by the domain
}
```

EJB defines local and remote versions of the interfaces:

```
@Remote
Public interface <domain_name>Remote
{
}
```

```
@Local
public interface <domain_name>Local
{
}
```

A stateless session bean implements the local and remote interfaces:

```
@Stateless
public class <domain_name>Bean
    implements <domain_name>Remote, <domain_name>Local
{
    // implementations of domain services
}
```

Transactions

The transactional behavior of domain session beans and domain services can be controlled by the `EjbTransactionAttribute` marking. By default, domain session beans and domain services require a transaction. If the caller already has a transaction context, the session bean uses the context. If the caller has no transaction context, a new context is created. The new context extends for the duration of the domain service call.

To set the transaction attribute for all services of the domain session bean, set the `EjbTransactionAttribute` marking for the domain.

To set the transaction attribute for a single service, set the `EjbTransactionAttribute` marking for the domain service.

If a domain service has a different transaction attribute than the domain, the domain service transaction attribute will override the domain marking for that domain service only.

Setting the `EjbTransactionAttribute` causes the `@TransactionAttribute` to be inserted, i.e.

```
@Stateless(name="MyDomainBean")
@TransactionAttribute(TransactionAttributeType.NEVER)
public class MyDomainBean implements MyDomainRemote,
MyDomainLocal
{
}
```

The legal values for `EjbTransactionAttribute` are:

EJBTransactionAttribute Marking Value	Meaning
MANDATORY	The domain services must always be a part of the caller's transaction. The session bean can't start its own transaction.
REQUIRED	The domain service uses the callers transaction. If the caller has no transaction, a new transaction is started.
REQUIRES_NEW	Always start a new transaction even if the caller has one already.
SUPPORTS	Use a transaction if the caller has one. Otherwise, don't use a transaction.
NOT_SUPPORTED	The callers transaction is suspended during the call. The call doesn't use a transaction context.
NEVER	The caller must not have a transaction context. If it does, an exception will be thrown.

See pages 364 – 370 of [BURKE2006] for a discussion transaction attributes.

Class Services

Class services are implemented by the adjunct `<domain_name>_<class_name>` class to support the separation of business logic from data prescribed by EJB.

State Actions

State actions are implemented by the adjunct `<domain_name>_<class_name>` class to support the separation of business logic from data prescribed by EJB.

Initialization Actions

System and domain initialization actions are not supported in EJB. The application should provide a domain service to initialize the system. The domain service session bean can be called from outside to initialize the system.

State Models

Entity Beans with state models will include an extra field to hold the current state of the instance.

Each domain service invocation originating from a session bean has its own task context. The task context contains the instance lists of all non-entity classes as well as the task and event queue. After the domain service invocation completes, the task consumes the events in the queue. When the queue is empty, the task packages up its output parameters and returns them to the client.

Since there are multiple session beans running concurrently, actions, class operations and services must know the current task when generating an event. The Task is stored as a thread local context.

Incident Handles

Entity beans may have attributes of type IncidentHandle. The incident handles and their parameters are stored in tables related to the entity bean. By default, these tables will use the EJB naming conventions.

To override the default table name generated by the incident handle entity bean, set the EjbIncidentHandleTableName marking on the system to the name of the table used to store incident handles.

To override the default table name generated by the incident handle parameter entity bean, set the EjbIncidentHandleParameterTableName marking on the system to the name of the table used to store incident handle parameters.

Timers

Timers are implemented using the EJB Timer Service. The GENERATE AFTER action language statement is supported for entity beans. Periodic incident handles are supported where the incident is a domain service, an event to an entity or a class or instance based operation of an entity bean. In addition a TimeServices realized domain provides services to set a timer for a particular day and time.

The EJB TimerService is injected into each domain Session Bean. The timer service is stored in the PfdContext. When a delayed event is generated or a periodic incident is created, an EJB timer is created. The timer info contains a persistent form of a PfdIncident. The domain Session Bean provides a timeout method to handle timer incidents. The timeout method gets the persistent incident, converts it to a PfdIncident, enqueues it and then processes the event loop :

```
@Stateless
public class PurchasingBean implements PurchasingRemote,
PurchasingLocal
{
    // inject the timer service into the domain session bean
    @Resource javax.ejb.TimerService timerService;

    @PersistenceContext(unitName = "Purchasing")
    private EntityManager entityManager;

    /** The MBSE execution context for the bean. */
    public SysContext pfdContext;

    @PostConstruct
    public void initBean()
    {
```

```

    // store the timer service in the context
    pfdContext = new SysContext(entityManager, timerService);
    pfdContext.register();
    pfdContext.task.registerSystemRouter(new Router());

    // setup the connections and realized implementation interfaces
    SW.registerRealizedClass(new System.SW.SW_Impl());
    pfdContext.unregister();
}

@Timeout
public void processTimerEvents(javax.ejb.Timer timer)
{
    IncidentHandleBean timer_incident = null;
    if (timer.getInfo() instanceof IncidentHandleBean)
    {
        timer_incident = (IncidentHandleBean)timer.getInfo();
    }
    else if (timer.getInfo() instanceof IncidentTimerInfo)
    {
        timer_incident =
((IncidentTimerInfo)timer.getInfo()).getIncident();
    }

    System.out.println("Timer expired");

    if (timer_incident != null)
    {
        // register thread
        pfdContext.register();

        try
        {

            PfdTask.getTask().enqueueIncident(IncidentHandleHelper.toIncidentHandle(timer_incident));

            // spin the event loop until no more processing
            pfdContext.task.processOOA();
        }
        finally
        {
            // unregister thread
            pfdContext.unregister();
        }
    }
}
}

```

When a delayed event is generated to an entity, the EJB timer service creates a single-action timer with the persistent form of the event incident:

```

public void generateAfter_Pulse(int process, PfdTask task,
PfdActiveObject sender, long delay_)
{

```

```

IncidentHandleBean incident =
    new IncidentHandleBean(PfdIncident.EVENT_INCIDENT,
        Sys.DOMNUM_IncidentHandleCases,
        IncidentHandleCases.EVNUM_GenId_Pulse, 0);
incident.setDestinationId(makeDestinationIdString());
incident.setPriority(0);
((SysContext)PfdTask.getContext()).getTimerService().
    createTimer(delay_, incident);
}

```

The templates use the TimeServices domain to manage periodic incidents. The TimeServices interface offers the full capability of EJB timers including single action and interval timers.

The TimeServices domain will provide the following services to set single action and interval timers for a particular date and time. The TimeServices domain supersedes periodic incident services provided by Software Mechansims.

Setting a Timer

The service **TimeServices:CreateSingleActionDurationTimer** creates a single action timer that executes an incident once after a duration of time.

The service **TimeServices:CreateSingleactionDateTimer** creates a single action timer that executes an incident once on the specified date and time.

The service **TimeServices:CreatePeriodicDurationTimer** creates a periodic timer that executes the incident once after the specified delay and periodically at the specified interval.

The service **TimeServices:CreatePeriodicDateTimer** creates a periodic timer that executes the incident once on the specified date and then periodically at the specified interval.

Cancelling a Timer

Each of the services used to set a timer returns a `ts_timer_handle_t`. The `ts_timer_handle_t` can be persisted in an entity and then used later to cancel the timer.

To cancel a particular timer, call **TimeServices:CancelTimer** and provide its `ts_timer_handle_t`.

To cancel all timers, call **TimeServices:CancelAllTimers**.

Setting a Date

To set a date, call the **TimeServices:CreateDate** or **TimeServices:CreateDateTime** service.

Attribute Selection

Get and set accessor functions are generated for each attribute. The implementation of entity beans is contained in two separate classes.

The logic class contains a reference to the entity bean and is a façade for the entity bean. The accessor function for the logic class gets the attribute value from the entity bean. For example:

@Entity

```
public class Purchasing_CustomerBean
{
    private String name;
    String getName() { return name; }
    String setName(String name) { this.name = name; }
}
```

```
Public class Purchasing_Customer
{
    private Purchasing_CustomerBean entityBean;
    public getname() { return entityBean.getName(); }
    public setname(String in_name)
    { entityBean.setName(in_name); }
}
```

Instance Creation

The create method for entity beans creates an instance of the entity bean, fills in its attribute values and then calls the entity manager to persist the entity bean.

Instance Deletion

The deleteObject method for entity beans unplugs the instance from its associations and then calls the entity manager to remove the entity.

Find

For entity beans global finds (FIND CLASS) are implemented as queries in the EJB Query Language (EJB QL). The query is executed by the entity manager. Instances of the logic classes are created for each of the entities returned in the results. The logic class have the same interface as nonentity classes and thus the generated code is the same as for non-entity classes.

For Each

For entity beans global FOR EACH statements are implemented as queries in the EJB Query Language (EJB QL). See the section on Find for more information.

Navigation

Association navigation for entity beans are implemented in the logic class. The logic class façade accesses the entity bean relationship methods. The related entity beans are adapted to instances of the logic class.

Link and Unlink

Links and Unlinks of entity beans are implemented by the entity bean relationship accessors. The logic class calls the get and set methods of the entity bean relationships.

Service Invocation

Service invocations are implemented by calls to Java methods.

SubSuperNavigation

SubSuperNavigations are implemented using java instanceof operator and casts.

4. Deployment

Specifying the Data Source

The persistence.xml file specifies the data source to be used during deployment. Create a data source that references the database for the application. Set the system marking EjbDataSource to the name of the data source. The datasource will be output in persistence.xml as follows:

```
<persistence-unit name="UnitName" transaction-type="JTA">
    ...
    <jta-data-source>data_source_name</jta-data-source>
    ...
</persistence>
```

JNDI Names for Session Beans

The JNDI name for a session bean affects how it is located in a remote client using the initial context. The JNDI name for a session bean by is ejb/<jndi_prefix><domain_name>. The default jndi_prefix is the system name. To change the prefix, specify the EjbJndiPrefix system marking to the desired name.

Table Name Length Checking

Some databases such as ORACLE restrict the maximum length of table names. By default, the table name for an entity bean is the name of the domain followed by an underscore followed by the name of the class. If the generated table name is longer than the maximum, the templates report an error. The default maximum length is 30. To

overridden the default the maximum length, set the system property `EjbMaxTableNameLength`.

To shorten the table name for the entity bean, specify a short(one or two character) `EjbTablePrefix` property on the domain.

5. Design Decisions

This section describes the decisions that were made when designing the mappings from PIM to EJB and the rationale behind them.

Annotations vs. XML Deployment Descriptor

EJB deployment information can be specified by annotated Java code or in a separate XML file called the Deployment Descriptor. XML Deployment Descriptors decouple the Java from the EJB APIs. The generated code would look just like regular java code. The annotated Java code is advantageous because it is easier to read. Using XML Deployment Descriptors would help decouple realized code from EJB. In generated code, we can always change the properties and generate new annotations or plain java so we chose annotations to optimize readability.

Representing Composite Primary Keys

There is more than one way to represent a composite primary key. A separate primary key class can be created with the primary key attributes or a reference to a primary key class can be embedded in containing class using the `EmbeddedId`. A separate class was chosen because it will be easier to generate and will integrate more readily with the existing generated code.

Multitable Mappings

Multitable mappings allow entities to be broken up between multiple tables. Multitable mappings are not supported at this time. They seem mostly useful for integrating with legacy database structures. At this point, we don't anticipate integrating with legacy databases.

Collection Types for Many Associations

A set was selected to be used when storing a many relationship. The Set is most consistent with the semantics of action language.

FIFO and LIFO Instance Ordering

FIFO and LIFO Instance ordering of entity beans is not implemented at this time. In the future it could be added by adding a sequence number or timestamp attribute to the entity and sorting ascending or descending on that attribute.

Cascading

Cascading specifies which persistence operations should be automatically performed on related entities. For example, when an entity is deleted, its associated entities can be deleted as well. Cascading can be performed on persist, fetch, remove, refresh, delete or all of these. Burke & Monson-Haefel recommends the following on pg. 158: "Be aware of how your entities will be used before deciding on the cascade type. If you are unaware of their use, then you should turn off cascading entirely. Remember, cascading is simply a convenient tool for reducing the EntityManager API calls." We will address Cascading later to determine if it is required.

Inheritance Strategy

The Single Table per Class Hierarchy was selected because it is simplest implementation and offers the best performance. There is only one table to administer. On the negative side, the table is not normalized because not all columns are relevant for all classes in the hierarchy. The NOT NULL constraint can't be applied to the columns of this table.

Association Classes

One possible way to represent Association Classes is to use a join table. The Association class is a JoinTable of the related classes. The identifier of the Association class is composed of the identifiers of both related classes. This strategy will not work because of the way action language is specified. An instance of the association class is created and then the instance is used during the link. When the association class is persisted, it does not yet know the values for its primary keys. The JoinTable annotation does not work well with relationships that are not ManyToMany or bidirectional.

To implement association classes, each participant in the association has a relationship to the associative class. The associative class then has a relationship to back to each participant.

6. Markings

System

Marking Name	Default Value	Effect
EjbJarDisplayName	System language id	The display name specified in the ejb-jar.xml deployment descriptor.
EjbApplicationServer	WebLogic	Determines which application server specific deployment descriptors are generated, i.e.

		weblogic-ejb-jar.xml
EjbDataSource	""	Specifies the name of the data source where entity bean data is located. The data source will be included in persistence.xml deployment descriptor in the jta-data-source element.
EjbGenSequenceGenerators	F	If T, Add SequenceGenerators to all generated identifier annotations. Create a sequence table named SQ_<class_table_name>. If F, use the default automatic id strategy for generated identifiers. Generate the file generate_sequences.sql containing the SQL to create the sequences.
EjbIncidentHandleParameterTableName	""	Override the default EJB table name for the table used to store incident handle parameters.
EjbIncidentHandleTableName	""	Override the default EJB table name for the table used to store incident handles.
EjbJndiPrefix	System name	The prefix added before the JNDI name of each domain exposed as a session bean.
EjbMaxTableNameLength	30	The maximum length for table names. If an entity bean's table name contains more than this number of characters, an error message will be reported at transformation time.

Domain

Marking Name	Default Value	Effect
EjbPersistenceContextUnitName	""	The name of the persistence context for the domain. The

		persistence context will be defined in the persistence.xml deployment descriptor. All entity beans contained in the domain will exist in the persistence context.
EjbSessionBean	"F"	If "T", a stateless session bean interface to the domain is generated. If "F", no session bean is generated.
EjbTablePrefix	""	A short prefix for the names of the tables generated for entities in a domain. Used to construct names for the entity beans. Helps stay within table name limits for some databases such as Oracle.
EjbTransactionAttribute	""	If EjbSessionBean marking is set to T for the domain, define the default transaction attribute for services provided by this domain. The transaction attribute defines whether a transaction is required by a domain service and if the domain service can use the transaction context of the caller. See section Transactions on page 20 for legal values for this marking.

Domain Service

Marking Name	Default Value	Effect
EjbTransactionAttribute	""	If EjbSessionBean marking is set to T for the domain, define the transaction attribute this domain service. The transaction attribute defines whether a transaction is required by a domain service and if the domain service can use the transaction context of the caller. See section Transactions on page 20 for legal values for this marking.

Class

Marking Name	Default Value	Effect
EjbEntityBean	F	If T, create an entity bean representing this class. Insert the Entity annotation for the class and generate an Entity bean. If F, create a plain java class.
EjbTable	""	If set, insert the Table annotation @Table(name="<EJBTable>") before the class declaration. If not set, a default name consisting of the EjbTablePrefix for the domain and the name of the class is used.
EjbDiscriminatorColumn	""	Relevant to the root class of an inheritance hierarchy using the single table inheritance mapping. Specifies the name of the column used to determine the type of subclass stored in a row. If not set, use the default column name.
EjbDiscrimintorColumnType	""	Relevant to the root class of an inheritance hierarchy using the single table inheritance mapping. Specifies the type of the column used to determine the type of subclass stored in a row. Should be one of STRING, INTEGER, or CHAR.
EjbDiscriminatorValue	""	Relevant to any class in an inheritance hierarchy. Specifies the value of the discriminator column for this class.

Attribute

Marking Name	Default Value	Effect
EjbColumn	""	If set, insert the Column annotation @Column(name="<EjbColumn>") before get and set methods. If not set, no Column is specified and EJB will use the default database mappings.
EjbIdGenerationStrategy	""	If set on an identifier attribute, insert the @GeneratedValue annotation and set the strategy to the value of this attribute. Supported strategies are AUTO,

		IDENTITY, TABLE and SEQUENCE.
EjbIdGeneratorProperties	""	Relevant for an identifier attribute with EjbIdGenerationStrategy set to TABLE or SEQUENCE. Specifies all properties of the SequenceGenerator or TableGenerator annotation except for the name. The value should be a set of name value pairs, i.e. for a sequence generator specify the name of sequence table as follows: sequenceName="RESOURCE_SEQUENCE"
EjbTransient	F	If T, use the @Transient annotation before the get and set methods for this attribute.

Participant

Marking Name	Default Value	Effect
EjbFetchType	""	If set to EAGER, retrieve this association participant in a single database query when its related participant is retrieved. If set to LAZY, retrieve this participant in a separate query.
EjbJoinColumn	""	If set, specifies the name of the column of the entity table that refers to the related object. The JoinColumn annotation is specified. If not set, the JoinColumn annotation is not added.
EjbPrimaryKeyJoin	""	If set on an identifier attribute, insert the @GeneratedValue annotation and set the strategy to the value of this attribute. Supported strategies are AUTO, IDENTITY, TABLE and SEQUENCE.
EjbJoinTableColumn	""	If set, specifies the name of the column in the join table that refers to the participant object. Specifies the joinColumns and inverseJoinColumns properties of the JoinTable annotation. If set, the EjbJointTable marking must be set for the association. If multiple columns are used for the join, specify the columns in a comma separated list.
SortKey	""	For a participant with many multiplicity specify the sort order of the traversal from the opposite participant. Specify

		<p>a comma separated list of attribute names followed by direction. Use ASC for ascending and DESC for descending. The attributes should be attributes of the participant class. For example to sort by firstName and then lastName, use "firstName ASC, lastName ASC". For lastName only use "firstName ASC".</p> <p>NOTE: The SortKey property for Participants should be set in the properties.txt file. Do not use the SortKey on the Properties view. It is not currently extracted.</p>
--	--	---

Association

Marking Name	Default Value	Effect
EjbJoinTable	""	<p>If set, specifies the name of the JoinTable implementing the association. If set, each participant should specify the EjbJoinTableColumn.</p> <p>If not set, use the EjbTablePrefix for the domain followed by the relationship number.</p>

User Defined Type

Marking Name	Default Value	Effect
EjbBlob	F	<p>If T, insert the @Lob annotation for attributes of this type. An attribute marked with the Lob annotation is stored in the database using a BLOB or CLOB type. These types should be used for very long strings or large bitmaps. For example, ORACLE has restrictions on the maximum number of characters that can be stored in a string. For large string types, a CLOB is required.</p>

7. Annotation to Marking Index

Annotation Name	PIM Element	Marking Name or Stereotype
-----------------	-------------	----------------------------

Entity	Class	EjbEntityBean
Table	Class	EjbTable
Table	Domain	EjbTablePrefix
Table	System	EjbIncidentHandleTableName
Table	System	EjbIncidentHandleParameterTableName
PersistenceContext	Domain	EjbPersistenceContextUnitName
PersistenceContext	Domain	EjbPersistenceContextType
TransactionAttribute	Domain	EjbTransactionAttribute
TransactionAttribute	DomainService	EjbTransactionAttribute
Column	Attribute	EjbColumn
Id	Attribute	Identifier stereotype
Lob	Attribute	EjbBlob
GeneratedValue	Attribute	EjbIdGenerationStrategy
GeneratedValue	System	EjbGenSequenceGenerators
TableGenerator	Attribute	EjbIdGeneratorProperties
SequenceGenerator	Attribute	EjbIdGeneratorProperties
SequenceGenerator	System	EjbGenSequenceGenerators
IdClass	Attribute	Identifier stereotype
Transient	Attribute	EjbTransient
Temporal	DataType	sys_date_t, sys_time_t, and sys_timestamp_t user defined types
Enumerated	DataType	Enumerated data type
JoinColumn	Participant	EjbJoinColumn
PrimaryKeyJoinColumn	Participant	EjbPrimaryKeyJoinColumn
JoinTable	Association	EjbJoinTable
JoinTable	Participant	EjbJoinTableColumn
JoinColumn	Participant	EjbJoinTableColumn
OrderBy	Participant	SortKey
DiscriminatorColumn	Class	EjbDiscriminatorColumn
DiscriminatorColumn	Class	EjbDiscriminatorColumnType
DiscriminatorValue	Class	EjbDiscriminatorValue

8. Model Restrictions

The following modeling elements are not currently supported:

- Persistent IncidentHandles may only carry parameters with basic types. Complex types such as serializables or groups are not currently supported.

9. References

[BURKE2006] Burke, Bill and Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (Fifth Edition)*. O'Reilly, Cambridge, 2006.

10. Appendix

How to Specify Stereotypes for Model Elements

Stereotype for model elements are set in the UML Editor. This section gives instructions on how to set stereotypes in various modeling editors.

Rational Software Modeler

1. Switch to the Modeling perspective.
2. Select the model element to which the stereotype will be applied.
3. In the Properties view, click on the Stereotypes tab.
4. Click on the Apply Stereotypes... button. The Apply Stereotypes dialog is shown.
5. Check the stereotype to apply.
6. Click Ok. The Apply Stereotypes dialog is dismissed.
7. The stereotype name surrounded by << >> now appears before the attribute name in Project Explorer. The icon representing the model element may change as well.