

# Specifying Persistence in Platform Independent Models

Weerasak Withhawaskul, Ralph Johnson  
Software Architecture Group  
Department of Computer Science  
University of Illinois at Urbana-Champaign  
Urbana, IL 61801  
withhawa@uiuc.edu, johnson@cs.uiuc.edu

## Abstract

Object persistence is addressed by many standards. Unfortunately, those standards are often specific to technology, middleware and programming languages. These not only complicate the model but also limit the flexibility in changing from one standard to another. This paper describes a Platform Independent Model (PIM) that supports object persistence and shows how the PIM is transformed into different executable Platform Specific Models (PSMs). We use *Mercator*<sup>1</sup>, a model verification and transformation engine, to show that a persistence-aware model can be transformed into different implementations. Our future goal is to extend the Mercator to support other aspects such as object distribution, messaging, transaction and how to apply them in the same PIM.

## 1. Introduction

Model Driven Architecture [Kle03] is a vision, a set of related specifications and an approach to new software development paradigm. It specifies a system independently of the platform that supports it, specifies platforms, chooses a particular platform for the system, and transforms the system from a platform independent model (PIM) to a platform specific model (PSM) [MDA01]. A PSM refers to a specific programming language, operating system, or DBMS while a PIM does not. Nevertheless, a PIM needs enough information to generate the PSM. The additional information is provided by either an automated tool or software designers or both.

In object persistence, the PSM might store data in XML in a file system or as relation in a relational database while a PIM would be completely independent of the choice of persistence. A mapping technique is a set of rules for translating a PIM to a PSM for a specific technology [Mel02]. For example, there would be separate mapping

techniques for XML and a relational database among others. At a PIM level, software developers *mark* or specify “which” entities need to be saved and “which” entity properties are used for identity. Then, the “how” the entities are implemented will be specified during the transformation step. Developers can then focus on business functionality and defer a persistence method decision and specific implementation choices later. The transformation will *map* the platform independent business model into a more detailed platform specific model using mapping rules. These rules are specified at a metamodel level so that they are applicable to all sets of source models. We specify additional information required by the mapping that is not in the model with an *annotation*. The transformation takes a marked PIM and an annotation from the developer and creates a PSM that can be translated into a target implementation code. The mapping verifies the correctness of the marked PIM and generates a corresponding PSM according to its mapping technique. Figure 1 shows an overview of the mapping process.

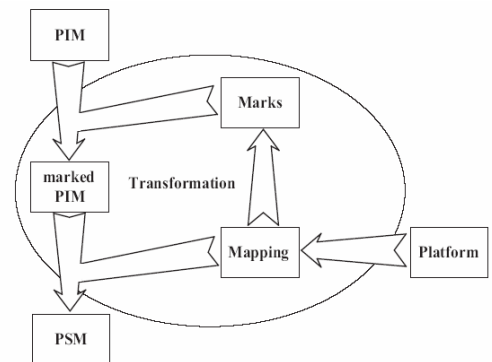


Figure 1 – the PIM-to-PSM mapping process

Persistence enables objects to live over its execution environment. [Eva03] groups objects into three categories. First is an entity that represents concepts in a problem domain. We use entities to model things in the real world such as banking accounts, purchase orders, etc. An entity contains an identity to guarantee uniqueness. Second is a value object that has no conceptual identity. It represents design element that describes what it is, not who it is, for example, colors, numbers etc. Value objects are often used as attributes of entities and are immutable. Third is an object that is neither an entity nor a value object, that is,

<sup>1</sup> The name is from Gerhard Mercator, a Flemish geographer and mathematician who invented the Mercator projection, an isogonic cylindrical projection that mapped a sphere onto a flat plane.

it is unique but has no identity. It cannot exist alone and is usually contained in an entity. For example, a purchase order is an entity that can be identified by purchase number. A purchase order may contain many purchase items. Each item has no identity and without the purchase order, its existence has no meaning. This paper focuses on the persistence mechanism of an entity that may contain other entities, value objects or owned objects. We formally define platform independent persistence metamodel using an UML profile and describes the transformation to different target implementations.

We have developed Mercator, a persistence-aware model verifier and transformation prototype. Mercator persistence mapper uses the UML Class and Deployment diagrams [UML03] to describe object properties and relationships as well as object access and distribution. In a class diagram, classes that represent entity instances will contain information required for persistence. A developer marks classes with a stereotype «persistence» or its derived stereotype to indicate that instances of these classes represent objects that can be persisted. In Deployment diagram, objects in the same node will interact via local interfaces while objects in different nodes will communicate via remote interfaces.

This paper will focus on the persistence aspect of the entity model. The next section discusses the persistence formalism of a platform independent model. Section 3 defines a transformation and annotations for PIM-to-PSM entity mapping in the Mercator. Section 4 shows an example of a vehicle object model and transformations to XML and EJB. Section 5 concludes the paper and discusses the future work.

## 2. Persistence Formalism of a Platform Independent Model

In the Mercator's metamodel, «persistence» is a base stereotype that provides persistence mechanism for a derived stereotype, for example, an «entity». The persistence mechanism stores the current state of an object instance by traversing its object graph into a stream that can be stored into a permanent storage such as a file, database or network connection so that it is not destroyed when its execution environment stops. Once stored, we can lookup, obtain and resume its life cycle when its execution environment resumes. We define the persistence formalism as follows:

*Definition:* In a PIM, we mark a persisted object *o* of a class *A* with a stereotype «persistence». When we apply «persistence» or its derived stereotypes into a PIM, the marked PIM needs to be verified. The only constraint for the persisted object in the marked PIM is that one of the

class attributes must have a stereotype «primaryKey» where primaryKey is of type UML::Foundation::Class.

```
context c : Class
pre:
  c.stereotypes->exists(s : Stereotype | s.name = "Persistence")
  and
  c.attributes->exists(attr : Attribute |
    attr.stereotypes->exists(s : Stereotype |
      s.name = "primaryKey"))
post:
  c.operations->exists(create(primaryKey : String) : void) and
  c.operations->exists(findByPrimaryKey(primaryKey :
    String) : Object) and
  c.operations.exists(store(void) : void) and
  c.operations.exists(remove(void) : void)
```

After the transformation, two class operations, *create()* and *findByPrimaryKey()*, and two instance operations, *store()* and *remove()* are created. These are published persistence operations that can be called. Therefore, at the PIM level, we do not have to worry about how persistence is internally implemented as long as we use these operations. One possible implementation is to have a persistence manager that handles persistence for all objects in the model. The decision such as a persistent method, datastore provider or additional implementation or platform related details will be deferred until we create a corresponding PSM during a mapping process. During the mapping, the developer will choose an entity persistence method and annotate information specific to each PSM. For example, a JDBC PSM requires the name of the DBMS so that a correct JDBC [Java00] class driver is used. This way, PIM will contain a cleaner, more abstract view of the designed system and be easier to understand and maintain. Table 1 shows the stereotypes used in the Mercator's persistence model.

## 3. Mercator's PIM-to-PSM Entity Transformation

Mercator performs three primary tasks. First, it takes an input PIM model as a XML document. Second it verifies the correctness of the marked PIM. For example, stereotypes specific to PSM are not allowed and will generate an error. Third, it performs a transformation by activating a matching mapper corresponding to the marked stereotypes in the input model. The mapper takes the marked model and its corresponding annotation to generate the target PSM. The output result can directly be generated into an executable code. In Mercator, we use Java as an implementation language.

For instance, Mercator persistence transformation engine checks the marked PIM for the «persistence» stereotype or subclasses of this stereotype, i.e. «entity». It then verifies the completeness of the input model according to the

Stereotype	Applies To	Description
«persistence»	Attribute, Class	An attribute or a class which instances that needs to be persisted.
«transient»	Attribute, Class	A default property indicating that an attribute or a class needs no persistence.
«entity»	Class	A class which instances needs entity capability.
«primaryKey»	Attribute	Unique attribute that identifies each object.
«generated»	ModelElement	An indication that class/operations/attribute are mechanically generated.
«get»	Attribute	The attribute contains a public getter method.
«set»	Attribute	The attribute contains a public setter method.
«add»	Attribute	The attribute is a collection that contains an add method.
«remove»	Attribute	The attribute is a collection that contains a remove method.

Table 1 – Stereotypes in the persistence model

constraints of the persistence. For example, if a class with «persistence» does not have an attribute with «primaryKey» and the annotation property allows auto-generation, it will insert a default key attribute *oid::String* and mark it with «generated» stereotype. If the class has a primary key attribute, the mapper will use it as the instance key.

In this paper, we will demonstrate two Mercator persistence mappers, *JavaXMLMapper* and *CMPEJBMapper* and how they are used to transform an object model into an executable model. Other possible implementation platforms for persistence model include JDO [JDO03], Hibernate [Hib03], CORBA [Cor02] and JDBC [Java00]. Figure 2 shows the Mercator mapper hierarchy.

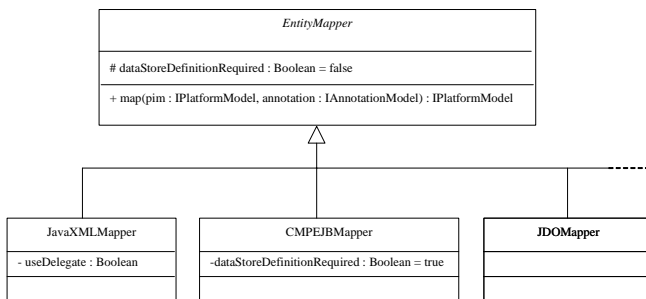


Figure 2 – Partial Mercator mapper hierarchy

The *JavaXMLMapper* is the Mercator’s Java-object-to-XML mapper based on JavaBeans’ *XMLEncoder* introduced in Java 1.4. The *XMLEncoder* API uses JavaBeans’ *construct* and public accessor methods to introspect and build the object graph into an XML output while the *XMLDecoder* restores them back into objects. Persisted objects must follow JavaBeans’ public *construct* and access specification. Since entities do not generally conform to this platform specification, so in order to store entities, we can either transform them into Java beans or create a helper class that acts as a persistence delegate. The decision on which method we should choose depends on the design intention. For example, a JavaBean transformation is simple and no additional helper class is needed. However, if the object is a singleton which its private constructor does not follow JavaBeans specification or if the accessors of object attributes are not made public, a helper class is required. The following steps show how *JavaXMLMapper* transforms the model.

The *CMPEJBMapper* uses the Container Managed Persistence (CMP) 2.0 in Enterprise JavaBeans (EJB) [Sha00]. We use three main features in CMP 2.0, the Container Managed Relationship (CMR) that specifies entity beans and their relationships, the EJBQL that provides a standard query language to obtain EJBs from a datastore, and the local interfaces for objects that reside in the same deployed node.

In the next section, we will show that we can transform the same persistence-marked PIM into different platform specific models by means of applying mapping rules and annotating the model. The followings are steps in the entity model transformation.

1. Create class diagram describing object model.
2. Mark entity classes with «entity» and specify primary key(s).
3. Specify the target PSM.
4. Customize the PSM with annotation
5. Generate an executable code.

#### 4. Case Study

Our example is a Vehicle object model. A *Vehicle* class is a base class that contains basic information about the vehicle. There are many subclasses of *Vehicle*, for instance, *Automobile*, *Truck*, *SUV*, *Bus*, *RaceCar* etc. Each vehicle may be kept in a primary garage. Each garage has capacity and cannot keep more vehicles than its capacity. At the platform specific models, we based our Java and EJB metamodels on those defined in [EDOC02] and [UMLEJB].

## Case Study 1: JavaXMLMapper

### Step 1 – Create class diagram describing object model.

A UML class diagram of the vehicle model is shown in figure 3. Since the figure represents an object model with no regard to any persistence technology, it is an example of an unmarked PIM.

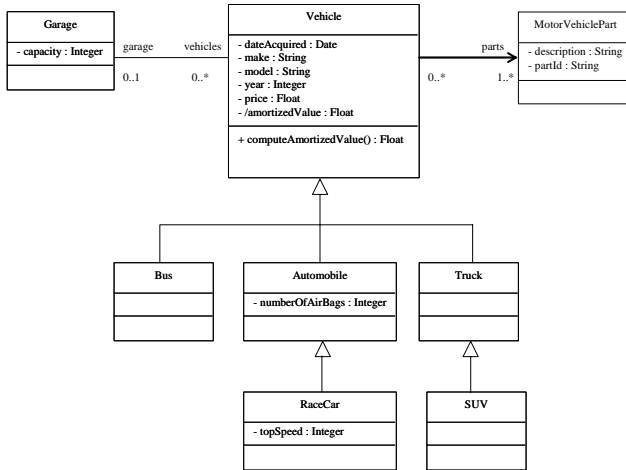


Figure 3 – Unmarked Vehicle class diagram

### Step 2 – Mark entity classes with «entity» and specify primary key(s).

We want to indicate that instances of Vehicle class hierarchy are entities. So we mark them with «entity». Since Vehicle is the root node of the hierarchy, we mark Vehicle with «entity» and all subclasses will inherit it. Then the precondition in section 2.1 is checked against the Vehicle class. First, the Vehicle is a Class model element. Next, the stereotype constraint is checked. Since the Vehicle class does not have an attribute with the stereotype «primaryKey», therefore the mapper inserts a new key attribute *oid::String* and marks it with «primaryKey» and «generated». Every model element that is inserted by a mapper will have the «generated» mark. If in the future, we add a new attribute, *vehicleSerialNo* and specify it with «primaryKey», the *oid* is no longer needed and can be removed. We follow the same step for Garage and MotorVehiclePart. MotorVehiclePart already has a primary key, *partId*, so the default key is not inserted.

Next, the mapper looks into each non-transient class attribute and association to determine whether their types support persistence. UML Foundation data types have persistence support built-in. If the referred types do not have «persistence», instances of those types will not be persisted. It is important to persist all related objects so that the object graph is correctly stored.

Now the mapper will create the public *create(Class primaryKey)* and *findByPrimaryKey(Class primaryKey)* static methods and the public *store()* and *remove()* methods and mark them as «generated». In figure 4, every class in the model has «entity».

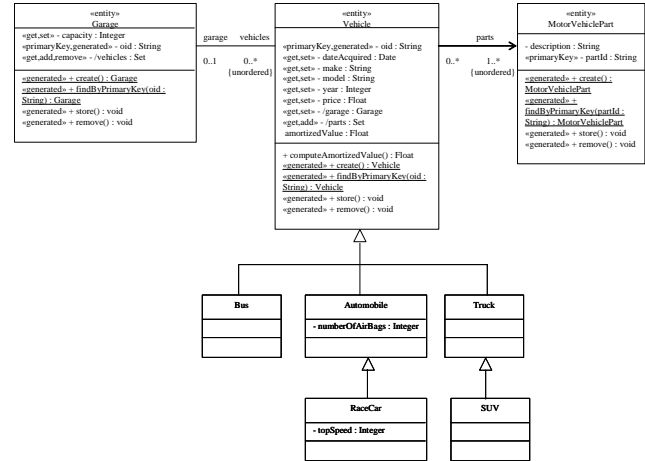


Figure 4 – Marked PIM class diagram with «entity»

We also need a deployment diagram to indicate how instances of each class interact. The deployment diagram determines the scope of object invocation. If an object sends a message to another object in the same node, the receiver interface will be local. On the other hand, if they are not in the same node, the interface of the receiver will be remote. If a message is sent from both local and another node, both interfaces are created. Therefore, the decision to define local and remote interfaces is made in this diagram. If the mapper does not find a deployment diagram, it will create one and put every class instance into the same deployment node. That means all class interfaces will be local. Figure 5 shows the default deployment diagram that contains instances in the same node.

### Step 3 – Specify the target PSM

In the first case, we use JavaXMLMapper to transform the model to a Java PSM. JavaXMLMapper uses Java's *XMLEncoder* to implement entity persistence. *XMLEncoder* is a Java 1.4 API that uses Java serialization and produces XML output of the object graph from *PersistentDelegates*. The requirement is that each object must follow JavaBeans specification so that it will know which property needs to be persisted. The mapper will iterate over each non-transient entity attribute and produce its getter and setter methods so as to expose these attributes as JavaBeans. Custom persistence delegates for specific instances of each class can be developed and put into the annotation model of this JavaXMLMapper.

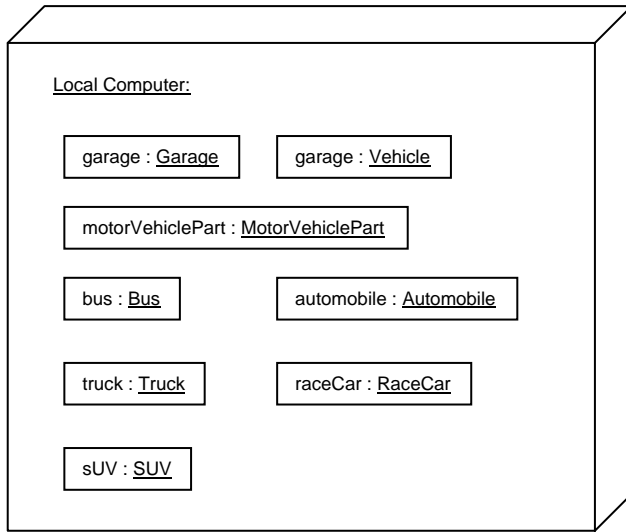


Figure 5 – Deployment diagram

Persistence PIM is a high level abstract model that does not show how a persisted object is stored or how the object is referenced. These decisions will be made by the Mercator mapping engine. There are three considerations on the PIM-to-PSM mapping. First, PIM does not require information about files to store objects but at the PSM level, we need to name a file that stores the persisted object. Second, PIM uses a static method *findByPrimaryKey(String primaryKey)* that takes a unique key as the input parameter to obtain each persisted object while the Java serialization assumes that we can identify the file containing serialized object and read the object from its input stream. Third, the static method, *create(Class primaryKeyClass)* is a standard way to instantiate an entity while Java uses the *new* keyword.

JavaXMLMapper default mapping policy creates a PSM that stores objects into files, each file contains a serialized representation of the instance and the file name contains a unique instance identifier for lookup. The format of the file name is the name of the class followed by ‘.’ and the contents of the *primaryKey* (in this case, *oid* is a string representation of a sequence number, followed by the file extension ‘.xml’. For example, a Vehicle object with *oid* = “1532” will store into a file name ‘Vehicle.1532.xml’. This way, each object will be stored in a unique file and the mapping template for *create()*, *findByPrimaryKey()*, *store()* and *remove()* can be generated.

In summary, the JavaXMLMapper performs the following steps:

1. Copy all classes from PIM to a newly created PSM.
2. If the PIM does not have a deployment diagram, a default local node diagram is created.
3. For each «persistence» class starting from the highest level superclass:

- a. Check the class primary key attribute. If there is no «primaryKey» attribute, create a default *oid::String*.
- b. For each non-transient attribute and association, generate JavaBeans getter and setter.
- c. Apply the mapping template to create method body for *create()*, *findByPrimaryKey()* and *store()*.

After applying the mapping rules, the result of the PIM-to-PSM mapping is shown in figure 6.

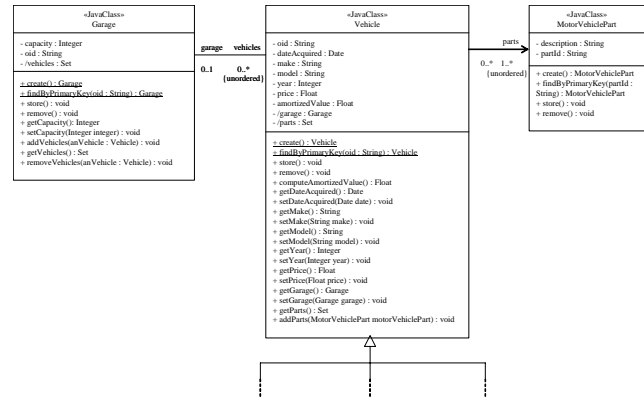


Figure 6 – Java XML PSM of the Vehicle

#### Step 4 – Customize PIM with annotation

JavaXMLMapper checks parameters from its annotation model. So if we want to customize the model transformation, we can put parameters, replace mapping algorithms in this annotation model. For example, we can change the file naming convention by overriding the *mapToName(String className, String primaryKey)*. Or if we want to change the file location, we override the *getBaseDirectory():String* method. We can also extend the persistence model to store the object XML file into a native XML database.

#### Step 5 – Generate an executable code

The result of JavaXMLMapper transformation is a Java PSM that can be translated directly into Java code. The Mercator provides model-based code generation that reads a PSM model and produces Java source files. The generated code are compiled into Java and executed in a Java runtime environment. Our test cases store entities into files, load them back and compare the entities’ structure (properties and associations). A result of a test case that stores a RaceCar object is shown in figure 7.

```

<?xml version="1.0" encoding="UTF-8"?>
<java version="1.4.1_02"
class="java.beans.XMLDecoder">
  <object class="vehicle.RaceCar">
    <void property="garage">
      <object class="vehicle.Garage"/>
    </void>
    <void property="numberOfAirBags">
      <int>2</int>
    </void>
    <void property="topSpeed">
      <int>180</int>
    </void>
  </object>
</java>

```

Figure 7 – Result of persistence file in XML

## Case Study 2: CMPEJBMapper

### Step 1 – Create class diagram describing object model

This step is independent of any PSM so we will follow the same step as the JavaXMLMapper example.

### Step 2 – Mark entity classes with «entity» and specify primary key(s).

As in the JavaXMLMapper example, we mark Vehicle, Garage and MotorVehiclePart with «entity». Vehicle and Garage do not have a primary key attribute, so *oid::String* is generated. The MotorVehiclePart already has a primary key field.

### Step 3 – Specify the target PSM

CMPEJBMapper creates the following UML model elements in the target PSM model.

1. Create an empty class diagram. This diagram will contain a CMP EJB PSM.
2. If the PIM does not have a deployment diagram, create a default local node diagram.
3. For each class *A* that contains «entity»

#### 3.1 Create Bean interface.

- If *A* is depended by Classifiers in the same node, create an interface *LocalA* that extends *javax.ejb.EJBLocalObject*. Add abstract business methods from PIM methods. Move business methods' implementation code into the bean class. Mark it with «EJBLocalInteface».
- If *A* is depended by Classifiers in a different node, create an Interface *A* that extends *javax.ejb.EJBObject*. Add abstract business methods from PIM methods. Each business methods can throw *java.rmi.RemoteException*. Move business methods' implementation code

into the bean class. Mark it with «EJBRemoteInteface».

3.2 Create a bean class, *ABean* that implements *javax.ejb.EntityBean*. This bean contains bean context and *ejb*-prefixed methods from the mapping template. Mark this bean class with «EJBEntity».

3.3 Create a bean key class, *AKey* from the «primaryKey» attribute. Implement *equal()* and *hashCode()* methods. Mark this key class with «EJBPrimaryKey».

3.4 Create Home interface.

- If a *LocalA* is created, add a home interface, *ALocalHome*. Mark it with «EJBHomeInteface».
- If an *A* is created, add *AHome* that extends *javax.ejb.EJBHome*. Mark it with «EJBHomeInteface».
- Add *create()*, *findByPrimaryKey()*, *findAll*, *create()*.
- Tag *create()* with «EJBCreateMethod».
- Tag *findXXX()* with «EJBFinderMtehod».

3.5 Put dependency lines from *AHome* and *ALocalHome* to *ABean*. Mark the lines with «EJBRealize».

4. Create a deployment descriptor container [UMLEJB].

- 4.1 If the container does not exist, create a new container.
- 4.2 For each EJB, add <ejb> entry in the container.
- 4.3 Generate CMR fields from object relationships in the class diagram.

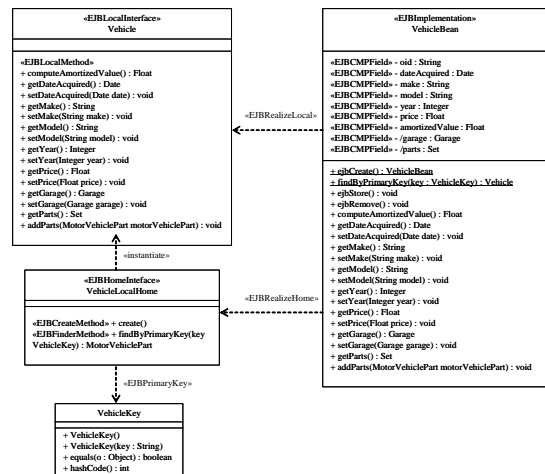


Figure 8 – A partial CMP EJB for Vehicle bean

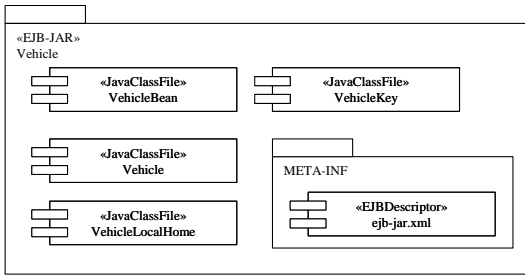


Figure 9 – A partial EJB Implementation model

#### Step 4 – Customize the PSM with annotation

At this step, we will specify EJB implementation specific information such as

1. EJB container vendor
2. Datastore provider. In case of relational database, if data tables do not exist, a Data Definition Language (DDL) is created from a table mapping method. Possible mapping methods include top-down, meet-in-the-middle and bottom-up. If data tables already exist, an object-to-table mapping is required.
3. Vendor specific extensions such as session timeout, bean and data cache settings, local transaction settings, locale invocation selection that is not in the EJB specification.

#### Step 5 – Generate an executable code

The following are steps to generate and optional deploy the result into an executable code.

1. Package all classes into a jar file
2. Generate EJB deployment descriptor, ejb-jar.xml from the deployment descriptor component.
3. Package the jar file, its supported jars and ejb-jar.xml into an enterprise archive (ear) file.
4. Optionally deploy the ear file into an EJB container. Some EJB container supports hot deploy and automatically detect and deploy the new ear. Other container needs manual update or requires a restart.

### 5. Summary and Future Work

This paper contributes two key aspects. First, it defines Platform Independent and Platform Specific Metamodels for object persistence. Second, it provides transformation algorithms between each mapping to target platforms. The Mercator's persistence model is introduced with the stereotype «persistence» and its constraint specification. Platform independent persistence model hides specific implementation techniques from the domain model. Techniques such as object-relational mapping are performed by pluggable mappers with a potential for customization and optimization. We show the result of the persistence PIM transformation into two implementations.

Additional mapping support for CORBA, JDO are being developed and will be reported in the future. We also consider using UML 2.0 metamodels and its action semantics to represent both model structure and behavior and will allow cross target language implementation.

For persistence model, we consider ways to support multiple persistence methods in the same object model. For example, a domain model may contain application properties that should be stored in an XML file while the domain objects should be persisted in a relational database. We are working on the annotation specification that allows multiple persistence methods in the same object model.

Our long term goal is to provide pervasive services to platform independent models by defining specifications and mapping techniques for core platform independent pervasive services. In addition to the persistence metamodel, we have identified object distribution, messaging, transactions, concurrency and workflow. We believe that providing separate profiles for each domain specific aspect will provide flexibility in choosing which aspect to be applied in the model. We anticipate a big challenge in finding ways to applying various aspects into the same model and ensuring model integrity and consistency during aspect weaving and conflict resolution mechanism.

#### Reference

- [Cor02] Common Object Request Broker Architecture Specification, Object Management Group, 2002.
- [EDOC02] UML Profile for Enterprise Distributed Object Computing Specification, Object Management Group, 2002.
- [Eva03] E Evans. Domain-Driven Design, Tackling Complexity in the Heart of Software. Addison-Wesley, 2003.
- [Hib03] Hibernate2 Reference Documentation. <http://hibernate.bluemars.net>.
- [J2EE] Java 2 Platform, Enterprise Edition 1.3, Sun Microsystems.
- [Java00] B Joy, G Steele, J Gosling, G Bracha. The Java Language Specification. Addison-Wesley, 2000.
- [JDO03] Java Data Objects Specification. 2003.
- [Kle03] A Kleppe, J Warmer, W Bast. MDA Explained: The Model Driven Architecture: Practice and Promise. Addison Wesley, 2003.
- [MDA01] Model Driven Architecture, ormsc/2001-07-01.
- [Mel02] S Mellor. Model-Driven Architecture, OOIS-MDSD Workshop 2002.
- [Sha00] Java 2 Platform, Enterprise Edition: Platform and Component Specifications, 2000.

- [UML03] Unified Modeling Language Specification 1.5, Object Management Group, 2003.
- [UMLEJB] J Greenfield, UML Profile for Enterprise Java Beans Public Draft. Rational Software. 2001.