Java/EJB Translation Rules

version 0.1
6/15/00

1. INTRODUCTION

Model Based Software Engineering (MBSE), as presented by Pathfinder Solutions, is a rigorous, complete, unambiguous, high level executable model of a particular problem domain. As such, this model can be translated into other executable implementations in the form of source code.

Model data is stored in an MBSE meta-model database. A set of translation rules in the form of ASCII templates, read and interpreted by Springboard, produces the source code. Springboard is a translation engine that extracts the semantic information contained in your UML analysis models and presents it in textual form via a flexible and simple template notation. See the Springboard Users Guide for more information on what data is accessible through the templates and the format and structure of templates.

(???? pic)

This document is describes the mapping of analysis into a Java/EJB design using Springboard and translation templates. The reader is assumed to have a working knowledge of Model Based Software Engineering (MBSE). The reader is also assumed to be familiar with the Unified Modeling Language (UML™) and EJB. See Enterprise JavaBeans, Second Edition, published by OReilly and associates, for more information on EJB.

It is also assumed the reader is familiar with the terminology, concepts and conventions presented in:

"Model Based Software Engineering – an Overview of Rigorous and Effective Software Development Using UML™" 1998, Pathfinder Solutions, Inc.

The document is organized by the major constructs within the MBSE meta-model. A brief description of the analysis construct is provided, followed by the mapping rules used to generate Java source code to be integrated with an EJB container and server. Where choices in the implementation can be made, properties are defined which guide the templates on how to translate the construct. These properties can be set either within the Model Editor through Pathfinder Extensions, or can be set at translation time using the SetProperty capability in the Springboard Template Language.

2. TERMS and DEFINITIONS

Templates	a set of text files that capture patterns to define the structure, fixed contents, and variable contents (corresponding to OOA model elements) of the target document set, using Syntactic Elements that conform to the notation specified in this document
EJB	
Extraction	the portion of the translation process when Springboard reads the OOA modeling information captured in your CASE database
MBSE Meta-Model	
Parsing	the portion of the translation process when Springboard reads the templates in preparation for production
Production	the portion of the translation process when Springboard executes the templates, populating your target document set with the OOA information acquired during extraction
Property	
Target document	a file or document produced by Springboard as the result of executing an OUTFILE directive
Target document set	the complete set of target documents produces by a single execution of Springboard on a set of templates
Translation	the execution of Springboard performs 3 steps in order: Template Parsing, OOA model Extraction, and Target Document Production

3. Target Document Set

Considering the task of translation at the highest level, the goal is to establish a set of instances of target documents with a set of OOA analysis data according to specified templates, or templates. The templates specify the specific analysis information required, how it’s arranged, and in what context.

The above paragraph helps us identify the major tasks/components of a translation:

- Develop a set of templates to define our target document set
- Employ Springboard to validate the correctness of the new templates
- Apply the proven templates with Springboard to produce the Target Document Set

The target document set consists of a set of Java files, organized in a directory structure that reflects the hierarchical Java package structures. The target document set also includes, where possible, deployment descriptors for the target EJB server.

3.1 Directory Hierarchy

Java organizes classes by packages, and, in the development environment, these package names are reflected in the directory structure. The generated Java files will be placed in a directory structure based on the domain chart, with the system name (Domain Chart Name) as the top level class, and each of the domains as a subdirectory and subpackage of the system.

A Package property on the domain chart will be available to specify the upper parts of the package hierarchy. Most packages reflect the DNS domain name of the company producing the Java software – e.g., com.pathfindersol.java.mechanisms. The corresponding high level directories will be created in the document set.

3.2 Java Files

Generally, each UML class and Domain will result in one Java file that contains the implementation of the class. In some cases, more than one file and Java class is required to support the EJB design.
Java filenames must reflect the top level class they contain.
The content of the files, the Java source code, will conform to de-facto standard format and naming conventions, and will be tailored to any coding standards required by the client.
In addition, a makefile is generated for the package that uses the Sun JDK command line compiler. If a standard IDE is chosen as the Java development environment, it may be possible to generate the project file for the IDE.

3.3 EJB Specific Files

There are some EJB specific files that can be generated based on the UML models and properties attached to them. The XML deployment descriptor for EJB 1.1 is one of them. Depending on the EJB server specific deployment process, other files may be generated to aid and automate the deployment process.

3.4 Mechanisms

A set of mechanisms, in the form of source code, will be provided by Pathfinder Solutions. These mechanisms provide the classes required to execute the MBSE semantics within the Java language.

4. Structural Design

Hardware
Software
Relational DB solution

5. MBSE to Relational Database Translation Rules

Relational databases define the persistent storage of data. The data stored is reflected on the Information Model for each domain.

5.1 Attribute

supertype: none; subtypes: none

An Attribute maps to an column in a class's table. The data type of the attribute is mapped into the data types supported by the database. Attributes with the Identifier property are added to the constraints of the table.

5.2 Object (UML Class)

supertype: none; subtypes: none

Each persistent object, marked by the Persistent property, results is a table in the database. Each table has an additional primary key integer attribute.

5.3 Relationship

supertype: none; subtypes: BinaryRel, SubSuperRel

5.3.1 BinaryRel

supertype: Relationship; subtypes:none

Binary relations are formalized by placing the primary key attribute in the right table to support relational database Normal Form. In some cases, an additional table will have to be created to store the relationship, if it does not already exist in the models.

5.3.2 SubSuperRel

supertype: Relationship; subtypes:none

Map into a 1:1 relationship or copy attributes and relationships down to leaves????

5.4 Current State

supertype: none; subtypes: none

The current state of persistent, active objects is stored in the database as an integer.

5.5 InterDomain Relationships

Multiple domains will be stored in the database. Without interdomain relationships, each domain is an island, unconnected to the others. Specifying relationships between classes in different domains can speed up the access process by optimizing out the Java processing. This may not be applicable to all systems.

6. MBSE to Java/EJB Translation Rules

This section is broken down by the into sections based on the following MBSE constructs, Domain Chart, Information Modeling, State and Service Modeling, and Action Language. While this breakdown overlaps somewhat, it is easiest to consider these as sections.

This document only covers the large grained mappings for MBSE/UML to Java and EJB. Each section describes the mapping in a straight Java design, then the mapping using EJB. See the Springboard Users Guide for the full list of analysis elements accessible in the templates.

In cases where there are alternative translation rules that can apply, properties can be assigned to MBSE analysis constructs to guide the templates in the translation. Properties are name-value pairs associated with analysis elements, where the names are project-specific string constants. Properties can be specified in the model editor, or at translation time in the templates, using the SetProperty command in the Springboard Template Language. In each section, if there are possible alternative implementations, these are discussed, as well as the trade-offs, and property names are defined to guide the translation.

6.1 Domain Chart Constructs

6.1.1 Bridge

supertype: none; subtypes: none; A requirement flow line connecting two Domains on the domain chart.

A bridge defines the requirements level relationship between domains, as well as the service invocations that are made from one to the other. There is no implementation reflection of bridges in the implementation, except for package imports.

6.1.2 Constant

supertype: Expression; subtypes: none; an expression that has a fixed value

System and domain wide constants are mapped to *public static final* member variables of the System of Domain classes.

6.1.3 DataType

supertype: none; subtypes: BaseType, GroupType, GroupIterType, InstanceReferenceType, ServiceHandle, UserDefinedType; The data type for an atomic data item

6.1.3.1 BaseType

supertype: DataType; subtypes: none; A built-in, predefined type

All base data types map to base data types in Java, int, String, double, etc.

Since Java passes everything by value, it is not possible to support output parameters from services with native Java data types. Instead, a set of lightweight carrier classes are defined to wrap the base Java types and allow these classes to carry the output data back to the calling service. This restricts calls to services with output parameters to be standalone calls, not part of a more complex expression like an If condtition. To support output parameters in services used anywhere, these carrier classes would have to replace the the Java data types, which may interfere with EJB integration. ????

6.1.4 UserDefinedType

supertype: DataType; subtypes: UserEnumerate, UserNonEnumerate; A data type defined by the user

6.1.4.1 UserEnumerate

supertype: UserDefinedType; subtypes: none; An enumerate defined by the user
Enumerated data types are not supported in Java. Instead, they are mapped to a series of *public static final int* member variables of either the System or Domain class, depending upon the scope of their definition.

6.1.4.2 UserNonEnumerate

supertype: UserDefinedType; subtypes: none; A non-enumerate data type defined by the user
Since there is no concept of a typedef in Java, all non-enumerated user defined types are implemented as their corresponding Java base types.

6.1.5 Domain

supertype: DataTypeScope; subtypes: none
Domains are mapped into Java packages that contain the classes within the domain. In addition, a domain Java class is created that contains the domain services and the domain data types.
In EJB, the domain class will be mapped to a session bean, stateless or stateful, depending on results of further performance and thruput investigations. A property called EJBSessionBeanType could also be defined if the translation templates needed to support both session bean types.
OReilly EJB, p362 recommends avoiding chained stateful session beans. Since domain invocations will result in chained session beans, the recommendation seems to be stateless sessions.

6.1.6 System

supertype: DataTypeScope; subtypes: none
The system is mapped to a package with the same name as the domain chart. This package contains all of the domain packages and the Sys class, which defines the system wide enumerated types and support for the mechanisms.

6.2 Information Modeling Constructs

6.2.1 Attribute

supertype: none; subtypes: none
An Attribute maps to an class attribute in Java. Currently the attribute is package visible and accessed directly in the implementation.
In EJB entity beans, it also maps to a property or field. Read and Write accessors will be implemented using the set/get<attrName> pattern. Since it is possible that some attributes may not be persistent, an optimization would be an Attribute level boolean Persistent property.
If the IM is the basis for the RDB schema, using the relational database schema translation templates, then the mappings for EJB can be easily generated. If another schema is used, the EJB mapping becomes more complicated, and should be done using the EJB servers deployment capabilities.
If the Attribute is part of an Entity bean and has the property Identifier set to TRUE, then that attribute will become part of the EJB primary key.

6.2.2 Object (UML Class)

supertype: none; subtypes: none
A UML class is mapped to a Java class. All MBSE modeled classes are subtypes of PfdObject, for passive classes, or PfdActiveObject, for classes with state machines. PfdActiveObject also inherits from PfdObject.

A class is persistent if the Persisitent property is set to TRUE. The translation rules will assume that a table has been created via the RDB templates to correspond to this class.

Persistent classes are mapped into EJB enitity beans. Entity beans can be either container managed or bean managed. By default, the translation rules will use container managed persistence, since this is generally easier, and since there is a 1:1 mapping between the classes and the RDB schema. If container managed persistance is required, an EJBEntityBeanType property will be added.

MBSE/UML classes mapped to entity beans will generate 4 classes/interfaces within the domain package:

- Remote interface <classname>
- Primary Key class <classname>PK
- Home interface <classname>Home
- Bean class <classname>Bean

See documentation on EJB Entity beans for defintions and uses of these classes and interfaces.

6.2.3 Relationship

supertype: none; subtypes: BinaryRel, SubSuperRel

6.2.3.1 BinaryRel

supertype: Relationship; subtypes:none

By default, a binary association maps to a Java object reference. This assumes that all instances of all classes within the domain are in memory at the same time.

Since, in an EJB implementation, all instances related to an object may not be in memory at once, there must be a way to traverse an association from an in memory class to related classes in the database. To acheive this, formalizing attributes, already added to the tables through the RDB schema translation, will be added to the EJB classes. See the action language section on Relationships for more information on the use of formalizers to traverse relationships, link, and unlink.

References to transient classes will not have a formalizing attribute.

Associations that are [0|1]..* to [0|1]..* (many to many) do not require an associative object in MBSE. The associative table is created by the RDB templates, but no corresponding Java/EJB class will exist.

6.2.3.2 SubSuperRel

supertype: Relationship; subtypes:none

Subtype or inheritance relationships are mapped to the *extends* keyword in Java. Jave does not support multiple inheritance, but then neither does MBSE/UML.

EJB Entity beans implement the EJB interfaces, and can therefor extend other modeled classes or PfdObject or PfdActiveObject.

Not sure how this works with container managed beans???? Steve to investigate

6.3 State and Service Modeling Constructs

The combination of domain services, state machines and actions, and class services make up the dynamic model of the domain. These constructs model the dynamics of the business logic of the particular domain. In EJB, all business logic is given to the session beans, keeping the entity beans very simple containers of data and relationships.

So, two basic implementation alternatives exist for mapping the state actions and instance base class services: attach them to the entity bean or create a separate stateless session bean that would operate on the entity bean.

Attaching services and states to an entity bean creates the possibility of loopback invocations within the processing. Loopbacks are when instance A calls a service of instance B, which then calls another service of instance A. Loopback actions are not encouraged, as they damage the threading and synchronization model of the EJB container, but settings on the bean deployment can allow them.

Implementing states and instance based services as stateless session beans would avoid loopbacks. Each state or service on an entity instance would go through a different session bean, so, in theory, an entity bean could have multiple session beans trying to operate on it. As long as the session bean kept its action within the context of the larger domain transaction, this would be OK. ???? in the loopback example, Multiple stateless session beans may then be trying to access the same entity bean within the same stack, and may block or deadlock, depending upon the EJB containers detection capability. ???? test this.

6.3.1 Action

supertype: none; subtypes: Service, State; the parent of all analysis elements with action language

6.3.1.1 Service

supertype: Action; subtypes: DomainService, ObjectService

6.3.1.1.1 DomainService

supertype: Service; subtypes: none

For the straight Java design, domain services map to static methods of the domain class. For realized domains, an interface with the same methods and signatures is defined. To realize the interface, this domain interface must be satisfied by a hand-coded class. At startup, an instance of this class is created and registered with the realized domain. The static methods invoked by the analyzed services are then passed to the registered class that implments the domain interface.

For the EJB design, domain services will be mapped to instance based methods of the domain's session bean class. Services that generate events will create a PfdTask instance to manage the event queue and wait until all events are processed and timers expired before returning.

Issue ???? loopback calls to a session bean are not allowed. So, calls to a domain service within cannot refer back to the same session bean instance. However, it can create a new session bean instance and invoke the service from there. Transaction scope and entity bean access (deadlock detection) must be investigated.

6.3.1.1.2 ObjectService

supertype: Service; subtypes: none

In the Java design, object services will map to Java class methods, static methods for class based services.

In EJB, the class based services will be promoted to the domain level, but with only package visibility. Instance based services will remain with the entity bean or with its corresponding session bean.

6.3.1.2 State

supertype: Action; subtypes: none

In the Java design, state actions are implemented as instance based methods invoked by the state machine mechanisms.

In EJB, state actions will remain with the entity bean or with its corresponding session bean.

6.3.2 Event

supertype: none; subtypes: none

Events are transient within a domain service invocation. There are no EJB issues to deal with.

6.3.3 Generate

supertype: EventAccessor; subtypes: none; an invocation of an event generate accessor
Event generation places events on the event queue within the context of a specific Task, shared among all the beans involved in a particular domain service. ???? how to make this happen

6.3.4 State Transition Table

The table the describes the state machine in terms of current state, incoming event, and next state.
This table will be stored statically in the active object itself. In the case of entity beans, the transition table will be stored in the same class as the stae actions themselves.

6.4 Action Language

6.4.1 AttributeSelection

supertype: Expression; subtypes: none; an expression that reads or writes an attribute value
In both the Java and EJB designs, attribute reads and writes map to invocations of the attribute’s get or set method.

6.4.2 Create

supertype: ObjectAccessor; subtypes: none; an invocation of an object create accessor
In the Java design, create statement maps to the creation of a new in- memory instance.
In EJB, the create statement of a class that is an entity bean maps to the ejbCreate invocation that creates the instance in the underlying database.

6.4.3 CreateServiceHandle

supertype: Statement; subtypes: none; an invocation of this built-in service
???? still to be resolved – see the discussion in Mechanisms

6.4.4 Delete

supertype: ObjectAccessor; subtypes: none; an invocation of an object delete accessor
In the Java design, delete statement maps to the removal of references to the in-memory instance, resulting in its availablility for garbage collection.
In EJB, the delete statement of a class that is an entity bean maps to the remove invocation that deletes the instance in the underlying database.

6.4.5 Expression

supertype: none; subtypes: AttributeSelection, BinaryExpression, Constant, EventAccessor, LocalVariable, ObjectAccessor, ParameterVariable, RelationshipAccessor, ServiceInvocation, UnaryExpression; an invocation of a function and/or something that returns/has a value (as an rvalue), or something that can have it's value set (as an lvalue)

6.4.6 Instance Lookups - Find/Foreach

supertype: ObjectAccessor; subtypes: none; an invocation of an object find accessor
A lookup statement in MBSE may have two sources, a relationship navigation or the class instances, and each source Find may be of two types, a find first/last (FIND), find all (FOREACH), and each Find type may include an optional Where clause for further limitations on the returned instances.

In the straight Java design, lookup statements map to searches on the in memory relationship or instance lists. The instances are iteratively compared with the Where clause.

In the EJB design, for entity beans, not all the class instances or related instances may be in memory, so lookup must map to database searches, using the EJB as an interface. Each type of lookup referenced in the AL of the domain will have an interface in the Home interface as well as the bean class.

???? EJB find capabilities may be limited for container managed persistence. Bean managed persistence may be more flexible for mapping our types of finds. Lookups would result in the return of a Collection of Primary Key objects, which the container would translate to instances.

6.4.7 InvokeServiceHandle

supertype: Statement; subtypes: none; a statement that invokes a ServiceHandle

???? still to be resolved – see the discussion in Mechanisms

6.4.8 Link

supertype: RelationshipAccessor; subtypes: none; an invocation of a relationship link accessor

In Java, a link results in bidirectional pointers set in the participating objects (and in an associative object, if necessary).

In EJB, a link does the same thing, since both objects must be in memory to be linked. However, for entity beans, the LINK also sets the formalizing database attribute for the relationship. Links may also result in the creation of associative objects in the database to support many to many relationships.

6.4.9 Navigation

supertype: RelationshipAccessor; subtypes: none; an invocation of a relationship navigation accessor

In Java, relationship navigation, from within a FIND or FOREACH statement, is taken from the pointer list within the class.

In EJB, for entity beans and relationships between entity beans, the navigation is mapped to a find/lookup method in the Home interface and bean class to find the associated instances from the database.

6.4.10 ServiceInvocation

supertype: Expression; subtypes: none; the invocation of an object or domain service

An invocation of a class method of the domain or UML class, in Java.

In EJB, maps to the appropriate invocation of a service, either in a session bean or an entity bean. In some cases, it may need to create a new session bean before invoking the method.

6.4.11 SubSuperNavigation

supertype: Expression; subtypes: none; a "cast" from a supertype to one of its subtypes

Results in a simple downcast in both Java and EJB designs.

6.4.12 Unlink

supertype: RelationshipAccessor; subtypes: none; an invocation of a relationship unlink accessor

The reverse of LINK, in Java, it removes the bidirectional references between the instances.

In EJB, it also removes the formalizing attributes from the databse table. Unlink will also remove any associative objects in the database.

6.5 MBSE Semantics

6.5.1 Timers

A single timer thread will be shared among all the session beans implementing the domain interfaces. The timer thread will accept timer settings and respond to the requesting Task when the timer has expired. This thread should be started outside the EJB environment, but shareable within it.

6.5.2 Instance Lists

An instance list is the set of all instances of a particular class. In this design, the set is only completely visible within the relational database that supports the system. As such, there are really 2 instance lists – the complete set in the database, and the partial set in memory. It is assumed that the EJB container takes care of insuring that an instance (entity bean) is not in memory twice.

Transient class instances are held completely in memory, but should not overlap between tasks or instances of the domain.

6.5.3 Service Handles

Invoking service handles????

EJB does not provide any support for asynchronous communication between two entities. The model is strictly client server, where all processing is synchronous and controlled by the client, invoking the server and blocking until processing is complete and the results returned. Thus, the concept of service handles does not map well to the communication between domains on the client and domains on the server. ???? Steve and Greg to investigate

Between domains on the same side of the boundary, the mapping is straightforward.

6.5.4 Task

The task consists of the MBSE event loop and the interface to timers. Within the context of the EJB design, there could be many tasks running at the same time, coordinated through the EJB container and the underlying relational database.

Each domain service accepts parameters, does processing, and, if it generates an event, creates a task, deposits the event, and spins the event loop. Allocation of events to the proper task. ???? how

Domain service re-entrance. ????

7. MODELING CONVENTIONS AND RESTRICTIONS

Springboard expects your OOA models follow the conventions described in "Pathfinder Solutions OOA/UML Modeling Guide".

Ideally, there are no conventions and restrictions placed on the analysis by the implementation or design technology. Practically, there is a small amount of feedback, especially for a design that is under development and intended to be used before it is complete. The following are conventions and constraints on the model for use with the Java EJB design. This list will change as development progresses.

- Use of output parameters from services in complex expressions.
- Loopback limitation on entity beans and session beans. Make as much inter-class communication as possible asyncronous via events to avoid this.