# pmMDA

poor man Model Driven Architecture

*Marcel Baumann*

*Version 1.0.10*

## Table of Contents

# 1 Introduction

The *pmMDA* application implements a pragmatic approach to realize MDA concepts in commercial and industrial projects. The platform independent model *PIM* is defined with available UML modeling tools[1]. The model is saved to the standard compliant XMI format. A user interface is provided to efficiently edit the PSM attributes. This information can be saved and visualized in the modeling tool. The *pmMDA* environment reads the model and generates platform specific model PSM. The generation of the artifacts is done with pluggable cartridges. This approach is sometimes called light MDA in the literature.

The generated artifacts use services of open source frameworks to execute on a specific hardware.

Our architectural framework is the product of years of experience in the development and deployment of distributed systems. It allows the complexity of such systems to be kept under control and the technical risks to be minimized. On the basis of this knowledge developers can create robust multi-tiered applications much quicker and maintain them at a comparatively low cost. The framework is continuously enhanced to ensure alignment with state of the industry technologies, and current best practices.

# 2 Concepts

The application builds on the following assumptions.

- The input model is a UML/MOF compatible model. The XMI standard is used to exchange information. The system architect can use UML modeling tools such as ArgoUML, Poseidon or IBM Rose to create a representation of her domain.
- The internal model is optimized to support code generation. Attributes and relationships are converted to property and indexed properties. Convenience methods are provided to simplify the development of new code generator cartridges.
- Cartridges uses the velocity template language to generate source artifacts. The context of the scripts provide access to the underlying model.
- Velocity templates can be modified to reflect programming styles or frameworks used in the project. The objects defined in the velocity context and their tagged values and attributes are documented in this document.
- New cartridges can be written to extend the generator. The cartridges are pluggable. Dependencies between cartridges, meaning a cartridge uses tagged values or attributes of another one can be defined.
- Documentation for all mechanisms and architectural decisions are provided. The source code is extensively documented using JavaDoc.

# 3 Interactive Work Environment

The *pmMDA* provides a work environment to manipulate *PIM* models and add the information specific to the used cartridge. The cartridge will emit source codes from

1 Preference is given to open source tools such as ArgoUML. Data exchange standards – XMI – provides interoperability with all major UML tools.

the *PIM* and *PSM* models. The environment enables architects to specify complex domain models and generate a MOF compatible export.

- The domain classes, associations and attributes should be created with a third party UML modeling tool. The *pmMDA* support all tools using XMI as exchange medium.
- A tree list user interface visualizes the UML model and empowers the user to efficiently edit the tagged values specific to the used cartridge. The user could directly manipulate these tagged values in his UML modeling tool but this approach is cumbersome.
- A code view supports the declaration of all instances for an existing reference code or hierarchical reference code type. The view is available as an additional panel for business object instances.
- An attribute view support the edition of business logic associated with attributes. Each attribute can have an initialization formula, computation rules selected through context constraints, validation rules. An attribute can also be triggers for the computation of other attributes. The concepts supported in *pmMDA* are documented in the article "Business Object Attributes" [mb-attributes-2003]. The view is available as an additional panel for business object properties.

## *3.1 Requirements*

### 3.1.1 Tagged Values

- The tagged values can be sorted alphabetically based on the tag name. Because all tags have a prefix they are also sorted by cartridge.
  An ordering per cartridge is available through prefix conventions.
- Missing tagged values defined in a cartridge can initialized with a default value. The default value is a constant and not dependent of other tagged values.
- Mo more needed tagged values used previously in a cartridge can automatically be removed the next time the model is loaded in *pmMDA*.
- A tag value should have a tag definition stating the name, description, type of values and default value. If no default value is set the application should transform values of empty string to null when reading the XMI model. Therefore the default value is used to specify if null belongs to the range of legal values of the tag.
- A propagate function is provided to apply all the tagged values to all elements with the same type and name. This mechanism emulates features of the type definition provided in some development environments. The function can be applied at a given level in the tree.

### 3.1.2 Properties

- Each cartridge can define properties used during artifacts generations. The properties are available to all configured cartridges.
- A mechanism is provided to generate nested definition of properties using a notation similar to the one provided in Ant.

- Missing properties defined for a cartridge can be initialized with a default value.

### 3.1.3 Cartridges

- The configuration of cartridge can be performed without recompiling the application. The capabilities of the cartridge can be defined in the program or with the help of a configuration file. The capabilities currently supported are.
    - The list of cartridges it depends on.
    - The stereotype defined in the cartridge
    - The tag values defined in the cartridge with the type and default value
    - The list of obsolete tag values defined previously in the cartridge and no more needed.
    - The datatypes known in the cartridge and their mapping to target languages handled in the cartridge.

### 3.1.4 Datatypes

Datatypes defines language independent model specific abstractions. They are a build-in concept in the UML landscape.

Cartridge use mapping information to match existing datatypes to types specific to the programming languages they generate.

Each model should be build only with datatypes to guarantee that the model is language independent.

# 4 *pmMDA* Core

## *4.1 Meta Model*

The *pmMDA* defines a meta model used to describe the business domain entities it should generate. The meta model is based on the UML standard and provides abstractions for packages, classes, relations and properties. The designers took care to define a model compatible with UML 2.0 to simplify conversions between the *pmMDA* meta model and the models of the various supported modeling tools. All specific information are stored as tagged values to guarantee transparent data exchange with any UML based model.

Convenience methods are provided to ease the development of custom cartridges.

The model supports features specific to the *pmMDA* application.

- Packages are considered as scoping environments. The hierarchy of sub-packages as defined in UML has no semantic signification. The package construct is used to defined attributes global to all classes declared in it.
- The concept of reference code is important to the family of applications we are developing. A reference code is a set of enumeration values belonging to the same business domain type. For example the currencies as defined in the ISO standard define a classical reference code. Implicit support of regular and hierarchical codes is provided at class level without breaking compatibility with UML.
- Relationships are handled as indexed properties to map easily our models to the Java bean conventions. Modern programming languages such as Java or C++ do not provide support for associations and model them with indexed properties. These fields are implemented in general with the containers classes defined in the language standard library.
  Only aggregation or composition relations are handled as indexed properties because these types are the only ones realizing the semantic of properties. The difference between aggregation and composition is who is the owner of the indexed items. The Java bean standard does not request the framework to finalize owned objects therefore no difference exists in the *pmMDA* model.
- Reference codes and hierarchical reference codes are identified with a corresponding class stereotype. This approach eliminates tedious inheritance or implementation relationship. It also clearly states that enumeration codes are a special kind of entities. This solution is UML compliant.

## *4.2 Extension Mechanisms*

### 4.2.1 Templates

Templates are the mechanism used to generate source artifacts of a PIM model using a cartridge. The templates are stored in a folder accessible to the users. The content of the template can be modified for example to reflect the coding guidelines of a project or add additional artifacts.

The architect modifying the templates should have a working knowledge of the

underlying template language[2].

### 4.2.2 Stereotypes

An item in a UML element can have at most one stereotype. The *pmMDA* architecture supports the definition of stereotypes. Stereotypes can be added to various UML element such as classes or packages.

The developers of cartridges should use new stereotypes sparingly. The major drawback of stereotype is that at most one can be defined per entity.

### 4.2.3 Tagged Values

Tagged values are the standard extension mechanism of the UML models. The *pmMDA* architecture supports the definition of new tagged values.

The developers of cartridges should document all new tagged values in details. The UML environment does not support rich semantic relations between tagged values and over elements of the models. This restriction is one major reason for the existence of the *pmMDA* application.

### 4.2.4 Datatypes

Datatypes are a standard extension mechanism to define new types in UML models. Types are defined as an extension to a model and are platform independent. . They do not belong to a package or have language specific aspects therefore different platforms can be targeted without modifying the model.

Datatypes should only be used to specify the type of simple or of an indexed property. It is allowed to use datatypes in tagged values.

The code generators can provide their mapping rules from user defined datatypes to their representation in the target programming language. The mapping rules are defined in a XML file for each cartridge.

Default mechanisms are provided to translate a datatype into a type definition of the target language using the mapping definitions. The translation result is stored in the associated cartridge attribute.

### 4.2.5 Model Dependencies

The cartridges extend UML models with stereotypes and tagged values. The model is a representation of domain models with a strong orientation toward the conventions adopted in the Java beans framework. Therefore convenience methods are provided in the MDA model to directly manipulate Java beans related attributes. These methods are also the Java bean specialization of the MOF/UML model.

Tagged values are used to insure compatibility to MOF models as created in UML modeling tools.

### 4.2.6 Properties

Properties are configuration parameters for a cartridge. The user can define custom

2 The actual template language for the delivered cartridges is Velocity.

values in the application properties configuration file. Each cartridge defines the set of properties it provides for configuration. The cartridge can define default value for properties not overwritten by the user.

Naming conventions avoid collision of property names in the configuration file.

### 4.2.7 Attributes

Attributes are values computed in each cartridge. Each cartridge defines the set of attributes it provides to its code generation templates and is responsible to compute them before the template is executed.

Naming conventions avoid collision of attribute names in the MDA model.

Attributes are strings or null values.

## *4.3 Persistence*

The persistence layer uses open source frameworks implementing meta model standards defined in the object management group OMG. The *pmMDA* supports XMI 1.0 /UML 1.3 and XMI 1.1 / UML 1.4, implicit support for XMI 1.2 is also provided. The open source library *Novosoft UML*[3] and *MDR* – Meta Data Repository – are used to implement these standards.

Domain models can be defined with UML modeling tools such as ArgoUML and Poseidon. The development team can also use commercial tools such as Rational Rose and export their model to a XMI format.

The persistence layer can read all these formats and update the files with the information added during a pmMDA session.

## *4.4 Engine*

The MDA engine executes the configured cartridges against UML compatible model. The model implementation defined in *pmMDA* provides various convenience methods to simplify the implementation of custom cartridges.

## *4.5 Cartridges*

Custom cartridges implements the cartridge interface. The package provides mechanisms to publish the required tagged values to the model editor.

The developer is free to implement their own code generation mechanisms. The application has extensive support for Velocity based code generators.

The cartridge framework provides the following mechanisms.

- A cartridge has a name, a prefix, and a class implementing it. The prefix is used in the cartridge to qualified all the tagged values and attributes it defines. The prefix is also used to generate the menu names in the generator framework.
- It has a list of cartridges it depends on. The list is used to load all required cartridges and to process all their attributes before calling the generation step of the

3 The Novosoft open source project is dead because the company behind it stopped development and no documentation is available. Projects using the NSUML library are migrating to MDR library, which is better documented and JMI compliant.

cartridge.

### 4.5.1 Initialization

When a new item is created or reseted a set of tagged values are added to it. Similarly older models can be loaded and new cartridge definitions will trigger the creation of tags on some of the items. Cartridges could be interested to define reasonable initial values to the tags based on context information or properties of the item. Separation of concerns is a design goal and the MDA model is not aware of the cartridge initialization.

Each cartridge can register a initialization servant to insure reasonable values in all tags defined in the cartridge. The servant uses the regular property change mechanisms provided in the Java environment. The following conditions are guaranteed.

- The servant is called for each added tag.
- All tag values are initialized to their default values as defined the MDA factory. The initialization is done upon creation of the item or upon loading from a persistence representation. The servant is called only upon completion of the initialization of the added tag.
- Tags defined in the cartridges required from a cartridge are initialized before the cartridge servant is called.

The designer of a cartridge should carefully evaluate the best approach for each tagged value. In general either a default value should provided or more complex initialization logic defined in a servant. In other words the default value of a tagged value can be considered as` a trivial variant of initialization logic[4].

The initialization algorithm is implemented as follow.

1. The element is created. Attributes of the element can be set. No tags are defined.
2. The element is initialized using the MDA factory. All defined tags are created for all registered cartridges. All obsolete tags are removed from the element for all registered cartridges[5].
    1. The added tag is initialized with the default value before the servants are called.
    2. The addition of a tag triggers a change event in the involved element. The cartridge factory is a registered listener for these events. The registration as listener in a new element is performed when the element is initialized.

The element supports the registration of change listeners. The factory manages the list of interested listeners for all changes.

### 4.5.2 Validation

A cartridge can define validation rules verifying the consistency and completeness of

4 Initialization logic should no try to overwrite an initialization value. No information is available to distinguish the case where the user has chosen a value equals to the initialization value. The decision should build on the fact that a tagged value has null as value.

5 No tags are added to system classes. These classes are standard classes provided in the target environment and never generated.

the model for cartridge's purposes. Separation of concerns is a design goal and the MDA model is not aware of the cartridge validation.

A validation rule has the following characteristics.

- The level of the rule is either error, warning, or informative.
- The kind of the rule specifies if it must be applied each time a property of the class is changed or as needed in the application to insure the consistency of the model.
- The rule is applied on a class and its properties. The verification code can access objects known from instances of the class on which the rule is applied.
- The rule returns an optional message with a level and a text helping the user to identify the reason of the problem.

The framework provides helper classes to register rules in the system. The registration process define

- The object owning the rules. The cartridge defining the rules is used as owner.
- The class on which the rules should be applied.

# 5 Architecture

The core engine of the *pmMDA* provides extension mechanisms to plug-in new cartridges in the code generator and add capabilities to the meta model. The selected solution must respect the following assumptions.

- All meta model extensions must be MOF and UML compliant. Available extension mechanisms are stereotypes and tagged values. A major restriction of stereotype is at most one stereotype can be defined for an entity.
- Extensions such as reference codes are implemented as convenience methods in the model but are stored as tagged values.

## *5.1 XMI 1.2 Mapping*

The standard XMI 1.2 and UML 1.4 Java implementation is the MDR library of the "Net Beans" project. The preferred model format of our application is XMI models created with this library. Open source UML tools such as "ArgoUML" or commercial alternative ones such as "Poseidon" use this library.

The format translator must provide answer to the following aspects

- Read a XMI UML model and transform it to the internal core model
- Update the tag values of the model to reflect the expectations of all loaded cartridges. The update is at least performed when the model is saved. The description of the items is also updated.
- Save the modified tag values, either added, deleted or modified ones to the MDR model
- Provide a factory interface to implement more complex operations on the model and propagate the changes to the MDR model
    - Add, delete and rename a property

- Add, delete, move and rename a classifier either a class or a datatype
- Add, delete and rename a package

The factory approach is used to abstract the operations from the underlying model. Therefore other libraries, for example NSUML, Rational Rose petal, or Eclipse ECORE can also be supported. The factory approach also avoid full synchronization of the internal and the persistent model without contextual information.

## *5.2 Standard Packages*

The described models reference standard packages provided through the Java development kits, external libraries or commercial frameworks. The application should avoid adding tagged values on classes belonging to these packages.

# 6 Cartridge Factory

The cartridge factory manages all registered cartridges.

## *6.1 Properties*

The cartridge factory provides the following options to control the application. The options are defined as properties.

- *pmmda-should-generate:* The property indicates if the model should be validated before generating artifacts. The default value is true.

# 7 Cartridges Overview

## *7.1 Purpose*

The pmMDA is delivered with a set of standard cartridges developed and used to generate mission critical systems.

Architects are free to develop their own cartridges and artifacts templates. The application provides a pluggable architecture. New cartridges are registered through a property files. No compilation of the core or new deployment is necessary.

## *7.2 Cartridge Architecture*

A cartridge should provides a set of constructs to smoothly integrate with over ones.

- Cartridge implementation
  - The cartridge must be written in Java and implement the cartridge interface. The interface defines a set of responsibilities the cartridge should provide.
    - The name of the cartridge
    - The set of cartridges it depends on.
    - The list of tags it defines and the list of obsolete tags defined in previous versions of the cartridge and no more used.
  - A set of Velocity templates. The set can be empty.
    - The architecture recommends to store the templates in a directory having the name of the cartridge prefix. This rule is a recommendation only.
  - A set of properties used to configure the cartridge.
    - These properties should be defined in the application property file.
    - The cartridge provides logic to handle situations were not all properties were defined in the property file. Default values should be implicit and the users have the right to remove them from their application property file.
  - A set of validation rules used to validate a model before the artifacts generation task is called.

Cartridges should not buffer Velocity templates. The Velocity framework performs this task better.

## 7.3 Standard Cartridges

The cartridges provide solutions for the following design areas.

- The BOG cartridge creates the business object model and the lightweight representation of business objects. Relationships 1-0..1, 1..N, and N..M are supported. Together with the business object graph framework the application performs CRUD operations on graphs of business objects.
- The OJB cartridge realizes object relation mapping with OJB. All configuration artifacts for the Apache project OJB are created. The schema of the database is also generated as a set of DDL instructions.
- The CTO cartridge realizes the middleware layer to transfer business object graphs and lightweight representation to the clients using CORBA. All transformations rules are generated.
- The OEX cartridge creates a standard explorer user interface displaying and manipulating business object graphs. Standard views and forms are generated.
- The EJB cartridge creates the infrastructure to create CORBA or EJB session based server application. The generated servers scale up to thousand users.

Rules must exist where the generated artifacts are created to streamline the packaging and deployment of the applications.

- Common
  - Business object model and lightweight representation. The class source files of the model are stored in packages corresponding to the packages defined in the MDA model. The lightweight representations are in the same package or
  - Data transfer objects – e.g. CORBA IDL structures – All IDL files are stored in an *IDL* directory. The module name is defined in the template and is usually the company name and a submodule *transport*.
  - The transformers used to transform the business object model into data transfer objects and vice versa. The class source files are stored in a sub-package of the MDA package of the model being transformed.
  - The common part of the used frameworks realizing complete sets of functions.
- Server
  - The CORBA server implementing the business object graph CRUD services.
- Client
  - The CORBA wrapper implementing the connection to the CORBA server providing the business object graph CRUD services.

An example of such a structure is as follow.

- CORBA Artifacts
  - CORBA IDL
    - /gendir/idl *contains the IDL structures definition generated with CTO cartridge and the IDL server definition generated with EJB cartridge.*

    - /gendir/java *contains the Java mapping to CORBA IDL created through the Java IDL compiler. The compiler has options to separate the client files from the server files. Therefore this distinction is delegated to the Ant file and not handled in the pmMDA.*
  - OJB
    - /gendir/ojb *contains the XML configuration files and the DDL create jobs.*
- Java
  - net.pmmda.dnm.users *contains the business objects of the data model.*
    - net.pmmda.dnm.users.client
    - net.pmmda.dnm.users.lw *contains the lightweight business objects of the data model. As an alternative these classes can be in the same package as the business objects.*
    - net.pmmda.dnm.users.transform *contains the transformers converting business object graphs or their lightweight representation into data transfer objects based on CORBA.*
    - net.pmmda.dnm.users.server
    - net.pmmda.dnm.users.ui *contains the user interface classes for the explorer based user interface.*

# 8 User Interface

An explorer based user interface is provided to view and edit domain models. The client is used to efficiently input tagged values. The model with its classes and properties is often defined with a UML modeling tool.

## *8.1 Dialogs*

The dialogs for MDA elements follow the same structural rules. The current implementation uses the form layout manager from the “JGoodies” project. The display and edition of tagged values in done through a common class.

## *8.2 Preferences*

The application provides a preference dialog to tailor the behavior of the application.

- The number of last opened projects available through the file menu is configurable.
- For each defined cartridge the user can decide if the cartridge is loaded or is active. The dependencies between cartridges are visualized.

# 9 Velocity Templates

## *9.1 Tips and Tricks*

- Use a velocity plugin for eclipse to simplify the creation and modification of scripts.
- Complex computations should be programmed in the cartridge. Velocity is not a programming language but a simplistic template language. Future development such as support of scripting languages, e.g. Beanshell could remove this restriction.
- Velocity does not update a variable if expression returns null and it is impossible to affect explicitly null to a variable or to unset it. To avoid logic error set the variable first to false before evaluation an expression, which can return null.

# 10 Further Readings

The article *Business Object Graphs: A pragmatic approach to MDA* describes a framework to exchange graphs of value data objects between clients and server.

The experience report *Lessons Learned in Insurance Project: Successes, Pitfalls and Enhancements* documents how the framework and MDA approaches were used in a mission critical insurance application successfully deployed.

# 11 Business Object Cartridge

The business object cartridge generates the source code for the business object layer of an application. The generated business object classes follow the Java bean conventions. The cartridge must have access to all business objects classes defined in a package. This assumption is reasonable because the classes should anyway be defined in the UML model.

Convenience extensions are provided for custom constructors, reference codes, and the visitor pattern. The reference code framework support hierarchical delegation of reference codes to other managers. The visitor pattern assumes that all node types to visit are defined in the same package. Please read the user guide of the cartridge for more details.

The generated source code is compliant with the business object graph framework. Business objects structures can be manipulated through this framework. The code is also compliant with the explorer framework. Business object structure can be visualized with this framework.

The cartridge knows how to handle reference codes. They are handled as a reference to a external object. Clients of the business objects should only manipulate codes and never identifiers of code values. This approach considers the reference codes as business domain enumeration types.

Power users are free to modify or extend the velocity templates to tailor the generated source codes to the needs of their environment.

## *11.1 Dependencies*

No dependencies to other cartridges exist.

## *11.2 Stereotypes*

The following stereotypes are defined and used in the business object cartridge.

| ***Stereotype*** | ***Definition*** |
|---|---|
| Business object | This class stereotype flags the application specific class as a business object. |
| Reference code | This class stereotype flags the application specific class as a reference code. |
| Service | This class stereotype flags the application specific class as a service defining a facade to the application function. |
| Interface | The package stereotype flags the packages containing classes visible to the clients of the application. |

These stereotypes are defined to allow a clean modeling of the business domain. The hierarchical reference code is not a stereotype because it is too similar to the reference code.

The interface stereotype as public interface flag embeds multiple meanings.

- Classes referenced either directly or through transitive closure in a service specification must belong to interface packages or be system classes.
- Only business classes member of an interface package are made available in the standard CRUD service.

## 11.3 Tagged Values

The following tagged values are defined and used in the business object cartridge. The first group of tags defines Java bean aspects. These semantic meaning of these aspects is described in the Java bean standard. The second group provides support for indexed and ordered properties.

The UML standard tags used in the cartridge are documented in the next table.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| persistence | FALSCH | boolean | Property (simple, indexed), class | Flag indicating if the element is persistent or not. |
| transient | FALSCH | boolean | Property (simple, indexed) | Flag indicating if the element is transient or not. |
| volatile | FALSCH | boolean | Property (simple, indexed) | Flag indicating if the element is volatile or not. |

A dependence rule exists between the class and its properties. The class is persistent if at least one of its properties is persistent.

The table below documents the cartridge specific tag definition.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| bo-generate | FALSCH | boolean | package | Flag indicating if the package should be processed or not. |
| bo-generate | FALSCH | Boolean | class | Flag indicating if class without MDA stereotypes should be processed or not. |
| bo-root | null | string | package | If defined the logical root directory where the Java artifacts are generated[6]. |
| bo-is-root | FALSCH | boolean | class | Flag indicating if the class is a root for the bog framework. |

6 The logical root directory is a variable mapped to an existing directory in the properties file. If the variable is not found a warning is issued and the default root directory is used. If not defined the default root directory specified in the properties file is used.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| bo-has-cycles | FALSE | boolean | class | Flag indicating if the graph below the root can contain cycles or is a tree[7]. |
| bo-is-hierarchical[8] | FALSCH | boolean | class | Flag indicating if a reference code is hierarchical or not. |
| bo-mode | Read-write | Enumeration | property (simple, indexed) | Specifies if the property is read, write, or read write. |
| bo-change-event | FALSCH | boolean | property (simple, indexed) | Flag indicating if the property is changeable. |
| bo-veto-event | FALSCH | boolean | property (simple, indexed) | Flag indicating if the property is veto able. |
| bo-navigable[9] | | string | property (simple, indexed) | Defines the property used to navigate to the owner. |
| bo-lightweight | FALSCH | boolean | Property (simple, indexed) | Flag indicating if the property is in the lightweight class. |
| bo-container-class | Array List | string | indexed property | The qualified name of the container classifier (class, datatype) used to store indexed properties. |
| bo-interface-class[10] | List | string | indexed property | The qualified name of the interface classifier (class, datatype) used to store indexed properties. |
| bo-is-ordered[11] | FALSCH | boolean | indexed property | The container is ordered. |
| bo-key-class | | string | indexed property | If the property is ordered, the tag defines the classifier (class, datatype) of the key. |

The enumeration for the business object mode has the following values: READ, WRITE, READ_WRITE. The value cannot be null.

The class name for the container class should implement the list interface defined in the utility package.

7 The cycle flag should only be set for classes being data object and with root flag set.
8 The tagged value is only relevant if the class has a reference code stereotype.
9 The property must be read and write to allow the definition of the link to the owning object. The type of the property must be compliant with the class of the owning objects. If the property is lightweight one – meaning the property is navigable and lightweight and the property used to implement the navigation is also lightweight - the lightweight property is always navigable. The visitors for data object and lightweight data object tree do not traverse the back link of a navigable property.
10 The interface class should be an interface of the container class.
11 The type of the container must implement the map interface if the tagged value is true.

If the property is an indexed one, the type of the property defines the items stored in the container of the property. The following container types are supported to realize indexed properties. If the indexed property is ordered when the tag *bo-key-class* is defined otherwise it is not necessary.

- List, Collection or Array can be used as container type. The storage container can be an array list, a vector, a linked list or an array. The index in the collection is the integer primitive type.
- Map or dictionary used as an ordered container type. The storage container can be a hash map, a hash table or a tree hash map. The index in the map is the configured tagged value.

## 11.4 Attributes

Attributes are computed during runtime in the cartridge and are only relevant to the cartridge. The attributes are available to the code generation templates to simplify the adequate code generation.

| *Attribute* | *Type* | *Target* | *Definition* |
|---|---|---|---|
| bo-mo-package | string | package | The qualified name of the package containing the business model type classes. |
| bo-if-package | string | package | The qualified name of the package containing the business model type interfaces. |
| bo-lw-package | string | package | The qualified name of the package containing the business model type lightweight classes. |
| bo-mo-class | string | class | The name of the business object type class. |
| bo-if-class | string | class | The name of the business object type interface. |
| bo-lw-class | string | class | The name of the business object type lightweight. |
| bo-generate | boolean | class | The class should be generated. |
| bo-vetoable | boolean | class | Flag defining if the class has vetoable properties. |
| bo-changeable | boolean | class | Flag defining if the class has changeable properties. |
| bo-has-interface | boolean | class | Flag indicating if the class has an interface describing its contract or not. |
| bo-lightweight | boolean | package | Flag indicating if the package contains at least one lightweight class. |
| bo-lightweight | boolean | class | Flag indicating if the class has a lightweight representation[12]. |

12 A class is lightweight if it contains at least one lightweight attribute or its ancestor is a lightweight class.

| *Attribute* | *Type* | *Target* | *Definition* |
|---|---|---|---|
| bo-used-for-navigation | boolean | Property, indexed property | Flag indicating if the property is used for backward navigation as part of a relation with a back navigation |
| bo-field-name | string | property | The name of the field name. |
| bo-container-class | string | Indexed property | The component name of the container class used to store indexed properties. If the associated tag is a datatype the attribute value is the mapping defined in the configuration. |
| bo-interface-class[13] | string | Indexed property | The component name of the interface class used to store indexed properties. If the associated tag is a datatype the attribute value is the mapping defined in the configuration. |
| bo-key-class | string | Indexed property | The component name of the key class for ordered properties. |
| bo-datatype[14] | string | datatype | The primitive type or class associated with a datatype for the Java language. |

A class has vetoable or changeable properties if at least one of its children has such properties. The helper classes to support the associated event propagation are always declared in the root class of the inheritance tree.

## 11.5 Properties

The cartridge provides the following options to control the generated code. The options are defined as properties.

- *bo-cartridge.language:* The target programming language used to generate the data object classes. The current supported values are Java-1.4, Java-1.5, C++-9x. The default value is Java-1.4.
- *bo-cartridge.common-root-folder:* The root folder defines the folder, where all Java classes common to server and client are generated.
- *bo-cartridge.server-root-folder:* The root folder defines the folder, where all Java classes of the server are generated.
- *bo-cartridge.client-root-folder:* The root folder defines the folder, where all Java classes of the client are generated.
- *bo-cartridge.model-package-extension:* The extension to the package name, where all business object classes are generated. If not defined the package name is used.
- *bo-cartridge.lightweight-package-extension:* The extension to the package name, where all lightweight business object classes are generated. If not defined the

13 The interface class should be an interface of the container class.

14 The datatype is converted to a class or primitive representation based on the cartridge mapping configuration. It is the responsibility of the cartridge and associated template to differentiate between primitive types and classes if necessary.

package name is used.

- *bo-cartridge.interface-should-be-generated:* If the value is set interfaces are generated otherwise no interfaces are generated.
- *bo-cartridge.interface-package-extension:* The extension to the package name, where all interface business object classes are generated. If not defined the package name is used.
- *bo-cartridge.business-object-class-prefix:* The prefix appended to the name of all generated business object classes.
- *bo-cartridge.business-object-class-postfix:* The postfix appended to the name of all generated business object classes.
- *bo-cartridge.lightweight-object-class-prefix:* The prefix appended to the name of all generated lightweight business object classes.
- *bo-cartridge.lightweight-object-class-postfix:* The postfix appended to the name of all generated lightweight business object classes.
- *bo-cartridge.interface-object-class-prefix:* The prefix appended to the name of all generated business object interfaces.
- *bo-cartridge.interface-object-class-postfix:* The postfix appended to the name of all generated business object interfaces.
- *bo-cartridge.class-field-prefix:* The prefix appended to the name of all generated fields of business object classes.
- *bo-cartridge.class-field-postfix:* The postfix appended to the name of all generated fields of business object classes.

## *11.6 Validation Rules*

- The qualified name of a package must be unique in the context of the model.
- The name of a class must be unique in the context of the package.
- The name of a property or an indexed property must be unique in the context of the class.
- Each package should have a description.
- Each class should have a description.
- Each property should have a description.

## *11.7 Design*

### 11.7.1 Rules

1. A class of the MDA model is generated if it does not belong to standard or bought packages. A standard package is member of the name spaces delivered with the Java environment or is recognized through the tagged value "*bo-generate*". The class must have one of the expected stereotype to be eligible.
2. The business object interface is emitted for all classes with the associated

stereotype. If the class has no ancestor when the methods of the interface are emitted. If the class has an ancestor it is assumed the parent class implements the business object interface.

3. The previous rule requires a special treatment of subclasses with changeable or vetoable properties. If a class has such a property it must propagate the information through the inheritance hierarchy. Therefore it is guaranteed that the root class instantiates the needed helper classes to propagate the events. Naturally only the root class needs to instantiate the classes[15].
4. Data types are never generated because they are primitive types part of the target programming language.

The cartridge support multiple language targets. Language specific features are all defined in the Velocity templates. The language selection is performed through application properties.

### 11.7.2 Java Beans

The cartridge provides mechanisms to support software development using source configuration management systems. A new file is only generated if it contains difference against the existing one.

The cartridge supports all functions defined in the Java bean standard. The generation is used for business object and lightweight classes.

- Support for the business object stereotype. All classifiers and visibility for the class and fields are supported.
- Simple and indexed properties with the *BeanUtils* extensions to support the collection package. The getter and setter for indexed properties are extended to support ordered indexed properties.
- Changeable and vetoable properties. Changeable and vetoable features are defined for each property. These features are implemented with the help of support classes provided in the J2SE bean package. The behavior respects the bean conventions.
- Default constructors with initialization of fields with primitive types or the associated wrappers.
- Specialized constructor to initialize fields with parameters values.
- A deep copy method is provided.

The copy method distinguishes between indexed properties and ordered properties. Care was taken to generated legible and indented source code.

### 11.7.3 Datatype Support

The cartridge provides sophisticated mechanisms for datatypes. Datatypes are used to create models independent of the target programming languages. These datatypes must be mapped to their target primitive or class type in the target language. The cartridge must provide translations for

15 This rule increases the complexity of the code generator but minimizes the resource consumption during runtime.

- All simple and indexed properties which classifiers are datatypes. The cartridge mapping files is used to translate the datatype to its corresponding type.
- All cartridge specific tag values containing type definitions should also be translated. This approach provides programming language independence at the level of cartridge specific tags. Each tag is mapped to the corresponding attribute containing the translated datatype. The names of the related tag and of attribute are the same.

The mapping of datatypes to their associated classifiers is a function with two parameters. The first parameter is the target language and the second is datatype name. The implementation assumes that the mapping does not change during the generation of artifacts. Updates on mappings without restarting the application is supported in the application framework.

### 11.7.4 Code Generation Tuning

There are several possibilities to fine-tune the appearance of the generated Java source code. Among them are the creation of accessor methods for properties, the type of the container for indexed properties.

- *Accessor methods*: The accessor methods for attributes are generated automatically. If the attribute has a multiplicity of 1..1 or 0..1, a setter and a getter are created. For attributes with a finite multiplicity, an array is generated, and the accessor methods include setter and a getter with an index parameter, an add and a remove. For an unbounded multiplicity, a collection is generated and the appropriate add and remove methods.
- *Modifying templates:* More advanced hand tailoring of the generated code is possible if you modify the code generation templates.
- *Selective reverse engineering:* Generated methods and fields can be marked with a JavaDoc tag to hinder their addition to the model when performing reverse engineering.

### 11.7.5 Reference Codes

The generated Java beans have extensive support for reference codes and hierarchical reference codes.

The cartridge provides a user interface to define the values of reference codes and define programming constants for the value unique codes.

### 11.7.6 Visitor Pattern

The visitor pattern is transparently provided in all generated Java beans. The visitor pattern is used to traverse a arbitrary tree or graph of Java bean instances. Convenience classes are provided to implement simple visitors efficiently with the help of the functor pattern.

The implemented visitor is a depth first algorithm. It provides a property indicating the current depth in the traversal. These two characteristics are the blocks to define various sophisticated visitors and functors collecting state information when traversing a graph of business objects or a graph of lightweight business objects.

The generated pattern is the variant being embedded in the business object classes. Other variants could be generated by modifying the code generation templates.

## 11.8 History

New users of the framework requested a major extension of the data object concepts. Instead of supporting trees of data objects they now request also graphs of data objects. Graphs contains cycles and must be handled appropriately. The visitors were extended to handle correctly graphs. A new tag was added to state that a set of data object instances are composed as trees or graphs.

Properties have the transient flag indicating if they are persistent or transmitted over the wire. Aggregations should be extended to support the same flag. Therefore a class is transient if all its properties and aggregations are transient.

The UML standard version 1.5 defines a standard tag *persistence*. The tag can be applied to associations, attributes and classifiers – class, interface, data type -. Therefore the following decisions are implemented.

- Persistence is modeled with the tag persistence. The type of the tag is boolean.
- The value of the tag should be coherent with the value of the transient constraint.
- Cartridge specific tags modeling persistence are obsolete and removed. The DOG cartridge is responsible to define the tag and default value.

# 12 Persistence Cartridge

The persistence cartridge generates the configuration files for the OJB Jakarta persistence layer. This layer is an object to relational mapping library solving the impedance mismatch problem.

Convenience extensions are provided for the creation of the database with DDL statements. The reference integrity rules for the database are also generated. Foreign keys used to link to business objects or reference codes are protected through corresponding integrity rules. Indexes on primary and foreign keys are generated to increase retrieval performance.

## *12.1 Dependencies*

The persistence cartridge OJB requests the business object cartridge BO.

## *12.2 Stereotypes*

No new stereotypes are defined in the OJB persistence cartridge.

## *12.3 Tagged Values*

The following tagged values are defined and used in the persistence cartridge.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| orm-schema | null | String | package | Name of the database schema. |
| orm-table-name | null | String | class | Name of the table containing instances of the class. |
| orm-column-name | null | String | Property, indexed property | Name of the column containing the property values[16]. |
| orm-column-type | null | String | property | JDBC type of the property |
| orm-column-size | null | Decimal | property | Size of the JDBC type if relevant[17]. |
| orm-is-primary | FALSCH | boolean | property | Flag indicating if the property is part of the primary key. |
| orm-can-be-null[18] | FALSCH | boolean | property | Flag indicating if the property can have null values. |
| orm-order-by | null | Enumeration | indexed property | Flag indicating if result sets should be ordered with this property. The value can be null. |

The schema name is the identifier of the logical database in which the tables are created. The schema is used as prefix to the table name for all SQL statements. The

16 For an indexed property the column contains the foreign key to the referenced entity.
17 If two values are required the first value is defined before the decimal point and the second value after.
18 If the property is part of the primary key, it cannot be null.

point is automatically generated.

The enumeration for the ordered by has the following values: ASC, DESC. The value can be null.

### 12.3.1 Persistence

A property is persistent if the standard UML persistence tag is set and if the transient flag is set to false. A class is persistent if at least one property is persistent. A persistent property cannot be volatile or transient.

A directed relation is persistent if its end class is persistent. It is the responsibility of the code template to generate the foreign key columns and define the associated access visibility for OJB framework. If the relation is mandatory the foreign key column can be defined as not null otherwise null values are legal.

### 12.3.2 Column Type

If the property has a tag “orm-column-type” value it is used. If no value is specified the mapping rules defined in the cartridge XML configuration is used. The XML configuration can provide rules for the mapping of datatypes to various target languages such as JDBC or SQL.

## *12.4 Attributes*

Attributes are computed during runtime in the cartridge. The attributes are available to the code generation templates to simplify the adequate code generation.

| *Attribute* | *Type* | *Target* | *Definition* |
|---|---|---|---|
| orm-table-name | String | class | The name of the table containing instances of the class. |
| orm-column-name | String | property | The name of the column containing values of the property. |
| orm-type-sql | String | classifier | The attribute for the classifier representation for the SQL language |
| orm-type-jdbc | String | classifier | The attribute for the classifier representation for the JDBC language |

The type of the column is not an attribute. The reason is that each database has its own conventions and extensions. Therefore only the specific template knows which SQL type is the optimal one.

## *12.5 Properties*

The cartridge provides the following options to control the generated code. The options are defined as properties.

- *orm-cartridge.root-folder:* The root folder defines the folder, where all OJB configuration files are generated.

- *orm-cartridge.database:* The name of the database type in which the objects will be saved. The following database are supported: PostgresSQL, MySQL, DB2, Cloudscape, Torque. The torque type is used to generate a Torque persistence XML file. This file can be processed with TORQUE to generate various database definition files.
- *orm-cartridge.orm:* The name of the object relational database mapper tool used to connect the object oriented programming language with the relational database.
- *orm-cartridge.table-prefix:* The prefix of all table names created in the database.
- *orm-cartridge.table-postfix:* The prefix of all table names created in the database.
- *orm-cartridge.column-prefix:* The prefix of all column names created in the database.
- *orm-cartridge.column-postfix:* The prefix of all column names created in the database.
- *orm-cartridge.generate-fk-index*: Flag indicating if indexes are generated for all foreign keys defined in the schema. The default value is false.
- *orm-cartridge.generate-constraints*: Flag indicating if constraints are generated for the database. The default is false.
- *orm-cartridge.inheritance* defines how inheritance structures are mapped to the database structure. The values are *join* – all classes are stored in the same table -, *extend* – each class has a table with only the attributes of the class -, *expand* – each class has a table with all its attributes and the ones of its ancestors-. The default value is *expand*.
- *orm-cartridge.ddl*: The property is extended with the database name it should defined. The value is the template file used to generate the data definition of the database. The legal values for database are the ones defined in the database property.
- *orm-cartridge.ddl-integrity*: The property is extended with the database name it should defined. The value is the template file used to generate the data definition of all integrity rules of the database. The legal values for database are the ones defined in the database property.
- *orm-cartridge.store-reference-code-mode*: The property defines how reference codes should be stored in the database. The legal values are regular, views, single-table.

## 12.6 Validation Rules

The validation rules are only applied to persistent beans and properties.

- The bean must have a table name.
- The property must have a JDBC type if the type of the property is not a datatype. If the type is a datatype the cartridge assumes that the datatype mapping rule in the cartridge property contains the expected information therefore no check is performed.
- If the type is String it must have a length, otherwise the length should be zero.

## *12.7 Design*

### 12.7.1 OJB Configuration

The persistence cartridge cartridge writes the XML repository file in the format expected from the open source tool OJB. This file can be read from the persistence layer without editing it before.

The cartridge has only one complex algorithm. It must extract from the business object model and the available tagged values all relationships between objects and map them to OJB constructs. The following rules are applied.

- *1-1 Relation*: The relation is defined as a forward relation. The foreign keys to connect the tables are declared in the owning table. This mechanism support multiple relationships to the same table. The insertion order is bottom-up, first the owned entities, second the owning entities defined for 1-0..N relations.
- *1-0..1 Relation*: The relation is defined as a forward relation. The foreign keys to connect the tables are declared in the owning table. This mechanism support multiple relationships to the same table. The insertion order is bottom-up, first the owned entities, second the owning entities.
- *1-0..N Relation*: The relation is defined as a backward relation. The foreign keys to connect the tables are declared in the owned table. The foreign key columns are optional. This mechanism support multiple relationships to the same table. The insertion order is top-down, first the owning entities, second the owned entities.
- *N..M Relation*: The relation is defined through a correlation table and two 1 to N relations. The rules to create the columns containing foreign keys are derived from the above ones.
- *Composition:* Composition relationships are mapped to an integrity rule requesting a cascaded delete.
- *Aggregation*: The application is responsible to delete business objects no more referenced in aggregations if necessary.

The list of all classes having a simple or an indexed property to a class can be retrieved from the model class. The name of the foreign key column is the table name of the owning class concatenated with the name of the property.

The code generation needs more concrete mapping rules.

- 1-0..1 property called *attribute* from an *owner* class.
  - The name of the attribute in the table is defined in a tagged value.
  - The type of the attribute in the table is defined in a tagged value.
  - The mandatory flag of the attribute is defined in a tagged value.
- 1-0..1 and 1-1 relation called *relation* from an *owner* class to an *owned* class.
  - The name of the relation in the table is the name of the property in upper case.
  - The property in java is called *relation* with a type *owned* class. The property is defined in the *owner* class.

- The OJB repository contains an anonymous field descriptor *relation-handle* of type *Id-Type* and a reference descriptor called *reference* pointing to *reference Handle.* Both descriptors are defined in the *owner* class.
- The database definition contains a field *REFERENCE_FK* of type ID-TYPE. The field can be null if the relation is optional otherwise not.

- 1-0..N relation called *relation* from an *owner* class to an *owned* class.
  - The name of the relation in the table of the owned is the name of the owner table and the name of the relation.
  - The indexed property in Java is called *relation* with a type *owned* class. The property is defined in the *owner* class.
  - The OJB repository contains an anonymous field descriptor *owner-relation-handle* of type *Id-Type* defined in the *owned* class. A reverse collection descriptor *relation* of type *owned* is defined in the *owner* class.
  - The database definition contains a field *OWNER_RELATION_FK* of type ID_TYPE. The field can be null if the relation is optional otherwise not.

### 12.7.2 Database Definition

The database definition is a mapping between the model to the relational world. Each primitive property is mapped to one column in the table. Type transformations are based on JDBC conventions. The primary key is always the identifier declared for all persistent business object class. Property referencing other business object classes are transformed into foreign key references to the referenced instances.

The foreign keys are generated as defined in the chapter how aggregations are mapped to database foreign relations.

Referential integrity rules are generated for all relations to insure consistency at database level. Restriction for deletion is created for 1 to 1 and 1 to N relationships.

The following mapping conventions are available.

| | ***DB2*** | ***SQL-Server*** | ***Oracle*** | ***MySQL*** | ***PostgreSQL*** |
|---|---|---|---|---|---|
| boolean | | Bit | Byte | Boolean | Boolean |
| integer | | Int | Number | Int | Int |
| float | | Float, real | Number | Float | Numeric |
| currency | | Money | N/A | N/A | money |
| String(fixed) | | Char | Char | Char | char |
| String(variable) | | Varchar | Varchar, varchar2 | Varchar | Varchar |
| Binary object | | Varbinary, image | Long Raw | Blob, Text | Binary, Varbinary |

## 12.8 History

The initial version of the persistence cartridge used the Apache OJB project as O/R mapping layer. Users requested support of the hibernate O/R mapping layer because this layer has a huge market penetration. Due to the fact that hibernate is also the reference used to define the forthcoming standard EJB 3.0 the decision to support hibernate was quite natural. An initial analysis shows that the following areas must be solved.

- The extend inheritance mode was added to better support hibernate and support the better stability of this feature in OJB.
- The concept of schema needed to be extended. Not only is the schema used as prefix of table names but also to distinguish the mapping files of a set of classes to a set of tables. The cartridge manages one mapping file per schema name. Such a schema can be used in more than one package.
- Mapping of one to one and one to many relationships. To reflect the programming language concepts only directed relationships are supported.
- Mapping of single property types to their equivalent representation in the mapping tools.
- Mechanisms to declare additional structures such as indexes, identifier sequences, relationships in the underlying database and integrity rules.

# 13 CORBA Cartridge

The CORBA cartridges generates the IDL definition files for all business object classes transmitted between the client and the server applications. Null values are handled for older CORBA versions.

Convenience extensions are provided to transform graphs of business objects into their CORBA representation and vice-versa. The CORBA cartridge needs the business object cartridge. The CORBA cartridge role is to generate transfer object for all business objects and their lightweight representation, which should be sent over the wire.

The cartridge generates one IDL declaration file for each Java class being transmittable. All transmittable attributes and aggregations are generated as part of this file. If an attribute contains another business object the include declaration is also generated. The following attributes are transmitted.

- All properties and indexed properties of a business object not being transient are transmitted.
- All properties and indexed properties of a lightweight business object not being transient are transmitted.
- All structures are generated as value types to streamline the transmission. Support for null values for primitive types are provided.

This approach respect the semantic of the transient keyword in the Java language and guarantees interoperability with other client server middleware such as J2EE.

## *13.1 Dependencies*

The CORBA cartridge CTO requests the business object cartridge BO.

## *13.2 Stereotypes*

No new stereotypes are defined in the CORBA cartridge.

## *13.3 Tagged Values*

The following tagged values are defined and used in the persistence cartridge.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| cto-package-idl | | String | Package | Name of the IDL package name. |

All attributes of the business objects not being transient are added to the transfer objects.

## *13.4 Attributes*

Attributes are computed during runtime in the cartridge and are only relevant to the cartridge. The attributes are available to the code generation templates to simplify the adequate code generation.

| *Attribute* | *Type* | *Target* | *Definition* |
|---|---|---|---|
| cto-bo2dto-package | String | package | The qualified name of the package containing the transformers from business objects to transfer objects are generated. |
| cto-lw2dto-package | String | package | The qualified name of the package containing the transformer from lightweight objects to transfer objects are generated. |
| cto-bo2to-class | String | class | The name of the translator class for business objects. |
| cto-lw2to-class | String | class | The name of the translator class for lightweight objects. |
| cto-mo-class | String | class | The name of the business object IDL structure, and the name of the file containing it. |
| cto-lw-class | String | class | The name of the lightweight object IDL structure, and the name of the file containing it. |
| cto-idl-type | String | Property, indexed property | The IDL type compatible with the UML model type. |

The algorithm to create the name of the translator class is coded in the cartridge. Until now no requirements were identified requesting a more flexible approach.

## 13.5 Properties

The cartridge provides the following options to control the generated code. The options are defined as properties.

- *cto-cartridge.idl-root-folder:* The IDL root folder defines the folder, where all data transfer object definitions as CORBA IDL files are generated.
- *cto-cartridge.java-root-folder*: The root folder defines the folder, where all Java transformer definition are generated.
- *cto-cartridge.idl-prefix:* The prefix of all table names created in the database.
- *cto-cartridge.idl-postfix:* The prefix of all table names created in the database

## 13.6 Validation Rules

The validation rules are only applied to beans an properties being transfered over CORBA.

- The property must have an IDL type.
- The bean must have an IDL structure and file name.
- If the bean is lightweight it must have an lightweight IDL structure and file name.

## *13.7 Design*

The cartridge generates the IDL declaration of the types to transfer and the transformation methods to convert a data object graph into an IDL structure and vice-versa.

A major design decision is that foreign keys are not transferred and therefore not part of the CORBA structures. The transformation methods uses the bean setter and getter to update the Java objects attributes. Therefore the foreign keys are set through the logic embedded in the setter methods.

### 13.7.1 IDL Definitions

The mapping of data object graphs defined in Java to CORBA structure is straight forward because both languages are object-oriented. To simplify the structure of IDL description the following conventions are used.

- CORBA strings are dynamic and directly map to Java strings
- Forward references of business objects are mapped to a sequence of the CORBA representation of the referenced type. This approach elegantly solves the problems with optional references.
- Backward references of business objects are mapped to a sequence of the CORBA representation of the referenced type. This approach elegantly solves the problems with optional references.

### 13.7.2 Transformer Classes

The transformers are responsible to translate a graph of business objects or of lightweight objects into their CORBA representation. The following principles are used.

- Business object or lightweight object graphs are traversed with the help of generated traversal algorithms. The same mechanism is used for the reverse transformation.
- The transformer provides factory methods to create the CORBA transfer object associated with the business model. Similar methods are provided to create the associated business model objects based on the transfer object.
- The cartridge provides the conversion rules for types which are not business objects. Support is provided to handle null references of Java types. CORBA structures are provided for major types and their null values. Each structure contains a field for the value to transfer and a flag indicating if the value is defined. If no value is defined the structure contains a zero value for the type.

# 14 Enterprise Java Bean Cartridge

The enterprise Java bean cartridge provides all J2EE artifacts.

The server interface for the business object graph is generated for the CORBA legacy interface and for the J2EE session bean interface. A server interface is generated for each package containing business objects. The interface is either a CORBA server or a stateless session bean.

- Each root object type declared in the business object graph cartridge has the expected services defined in the framework
  - Retrieve a graph of business objects where the root is of the defined type. The parameters are an identifier and a constraint string. Both are optional.
  - Store a graph of business objects where the root is of the defined type. The parameter is the root of the business object graph.
  - Remove a graph of business objects where the root is of the defined type. The parameter is the identifier of the root object.
  - Retrieve a graph of lightweight objects where the root is of the defined type. The parameters are an identifier and a constraint string. Both are optional.
- Services are provided for the reference code manager to transfer a reference code type to the client.
  - Checks if a set of reference code types have newer versions available. Each request defines the class name of the reference code and its timestamp. The answer specifies if a new version exists or not.
  - Retrieve a set of reference code types, either hierarchical ones or regular ones. Each element contains the class name of the reference code, its timestamp, a flag indicating if it is hierarchical or not and the set of values.

The server implementation retrieves the subsystems it depends on through the locator pattern.

A server is only generated if at least one root object with the data object stereotype is declared in the package.

## *14.1 Dependencies*

The CORBA cartridge EJB requests the business object cartridge BO and the CORBA transfer object cartridge CTO.

## *14.2 Stereotypes*

To be written.

## *14.3 Tagged Values*

The following tagged values are defined and used in the persistence cartridge.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| ejb-server-name | | String | Package | Name of the server providing the interface for the framework. |

The server provides the services requested in the business object graph framework and the associated reference code manager.

## 14.4 Attributes

Attributes are computed during runtime in the cartridge and are only relevant to the cartridge. The attributes are available to the code generation templates to simplify the adequate code generation.

| *Attribute* | *Type* | *Target* | *Definition* |
|---|---|---|---|
| ejb-server-package | String | package | The qualified name of the package containing the server specific classes. |
| ejb-client-package | String | package | The qualified name of the package containing the client specific classes. |

## 14.5 Properties

The cartridge provides the following options to control the generated code. The options are defined as properties.

- *ejb-cartridge.generate-corba:* If the value is set CORBA server interfaces and client and server side implementation are generated for legacy applications and other programming languages, otherwise no CORBA artifacts are generated.
- *ejb-cartridge.server-corba-root-folder:* The root folder defines the folder, where all Java classes of the CORBA server are generated. If the key is undefined the value is *bo-cartridge.server-root-folder* is used.
- *ejb-cartridge.client-corba-root-folder:* The root folder defines the folder, where all Java classes of the CORBA client are generated. If the key is undefined the value is *bo-cartridge.client-root-folder* is used.
- *ejb-cartridge.generate-bean:* If the value is set session beans and deployment descriptors are generated.
- *ejb-cartridge.server-j2ee-root-folder:* The root folder defines the folder, where all Java classes of the J2EE server are generated. If the key is undefined the value is *bo-cartridge.server-root-folder* is used.
- *ejb-cartridge.client-j2ee-root-folder:* The root folder defines the folder, where all Java classes of the J2EE client are generated. If the key is undefined the value is *bo-cartridge.client-root-folder* is used.
- *ejb-cartridge.common-j2ee-root-folder:* The root folder defines the folder, where all Java classes of the J2EE client are generated. If the key is undefined the value is *bo-cartridge.common-root-folder* is used.

- *ejb-cartridge.server-package-extension*: The extension to the package name, where all client related classes are generated. If not defined the package name is used.
- *ejb-cartridge.client-package-extension*: The extension to the package name, where all server related classes are generated. If not defined the package name is used.
- *ejb-cartridge.server-locator-classname:* The qualified name of the locator class in the application where the session bean is deployed. The locator is lazy instantiated when a session bean instantiated in the application.

## *14.6 Validation Rules*

- To be written

# 15 Object Explorer

The object explorer provides a simple navigator user interface with a left pane displaying a tree of all lightweight business objects and a right pane displaying a list of the corresponding business objects. The attributes shown in the list can be parametrized. The explorer user interface uses a sophisticated declarative framework. This framework empowers developers to create such interfaces with minimum effort.

Actions are provided to display a view of all simple properties of a business object and edit it, add a new business object, insert one before another one and remove and existing one.

The user interface is configured through a template generated description of the properties and relations between business objects. The explorer framework uses this description to build the user interface and to control its behavior. The configuration defines the following characteristics.

- The business object types displayed in the user interface with their interface, lightweight and regular class implementation.
- The relations between business objects. Relations to classes which are not business objects are never visualized in the navigator. The user interface is build so that no declaration order of the interface descriptors is requesting.
- The fields displayed in the table model on the right panel in the explorer framework.
- The icon selector of the displayed items in the navigator.
- The view used to show and edit the properties of the business object instance.

## *15.1 Dependencies*

The object explorer cartridge OEX requests the business object cartridge BO. The following information are in particular used.

- The rich client user interface classes are generated only if the flag “should be generated” of the business object cartridge is true.
- The root directory where the rich client classes are generated is the one defined in the business object cartridge.

The type factory mapping the lightweight nodes to the heavy classes is only generated if at least one business object has the lightweight flag set in the package.

## *15.2 Stereotypes*

No new stereotypes are defined in the OEX object explorer cartridge.

## *15.3 Tagged Values*

The following tagged values are defined and used in the persistence cartridge.

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| oex-view-class | null | String | class | The class of the property view to display the business object. |

| *Tag* | *Default* | *Type* | *Target* | *Definition* |
|---|---|---|---|---|
| oex-visible | FALSCH | boolean | property, indexed property | Flag indicating if the property is visible in the table view. |
| oex-is-name | FALSCH | boolean | property | Flag indicating if property is used for the name of the object in the navigator. |

The flag defines if the simple property is visible in the table view or in the navigator view. If the type of the property is a business object it is displayed in the navigator view, otherwise it is displayed in the table view. All simple properties are always accessible in the dialog view used to edit the business object.

The flag also defines if the indexed property is visible in the navigator view.

## 15.4 Attributes

Attributes are computed during runtime in the cartridge and are only relevant to the cartridge. The attributes are available to the code generation templates to simplify the adequate code generation.

| *Attribute* | *Type* | *Target* | *Definition* |
|---|---|---|---|
| oex-rich-client-package | String | package | The qualified name of the package containing the user interface classes for the rich client. |

## 15.5 Properties

The cartridge provides the following options to control the generated code. The options are defined as properties.

- *oex-cartridge.rich-client-package-extension:* The extension to the package name, where all rich client related classes are generated. If not defined the package name is used.
- *oex-cartridge.rich-client-folder:* The root folder defines the folder, where all Java classes of the rich client are generated. If the key is undefined the value is *bo-cartridge.client-root-folder* is used.

## 15.6 Validation Rules

- To be written

# 16 Service Cartridge

The service cartridge generates services for an application. The features defined in the service oriented architecture approach are provided through the cartridge.

The cartridge uses the web services standard to implement services. These services are accessible from various programming languages. The services and associated parameters are described using WSDL syntax.

## *16.1 Dependencies*

No dependencies exist to other cartridges.

## *16.2 Stereotypes*

The following stereotypes are defined and used in the business object cartridge.

| *Stereotype* | *Definition* |
|---|---|
| Service | This class stereotype flags the application specific class as a service. |

## *16.3 Tagged Values*

## *16.4 Properties*

## *16.5 Design*

The WSDL service description is used to generate Java, C++ and C compliant client or server interfaces. The Java interfaces are generated using the Apache open source project Axis. The C and C++ interfaces are generated using the gSOAP interfaces.

# 17 Reference Code Cartridge

Reference code types are supported in the data object graph cartridge. The designer should have support to defines the values of a reference code type. The values can be stored in a database, a serialized file or as program constants.

The constant definition is stored in a XML file. This file can be edited with an XML editor or with the provided editor.

## *17.1 Dependencies*

# 18 Open Points

The following aspects are still open.

- The project needs a tool for defining and tracking requirements. Which one should we use?
- The project needs a defect tracking system. A possible solution is to register the project in an open source factory and uses their defect tracking system.
- Is it possible to merge our *pmMDA* tool with other open source MDA tools?

# 19 Glossary

## 19.1 Terms

| | |
|---|---|
| Component Name | The name of the component without its package information. |
| Qualified Name | The name of the component with its package information. |

## 19.2 Abbreviations

| | |
|---|---|
| IDL | *I*nterface *D*escription *L*anguage |
| ISO | *I*nternational *S*tandard *O*rganization |
| JDK | *J*ava *D*evelopment *K*it |
| JRE | *J*ava *R*untime *E*ngine |
| MDA | *M*odel *D*riven *A*rchitecture |
| MOF | *M*eta-*O*bject *F*acility |
| OMG | *O*bject *M*anagement *G*roup |
| PIM | *P*latform *I*ndependent *M*odel |
| PSM | *P*latform *S*pecific *M*odel |
| UML | *U*nified *M*odeling *L*anguage |
| USDP | *U*nified *S*oftware *D*evelopment *P*rocess |

# 20 References

The framework could only be realized with the help of a set of wonderful open source projects. We are grateful to the Apache foundation and the Jakarta project for their powerful applications and libraries.

| | |
|---|---|
| ArgoUML | UML modeling tool<br>http://argouml.tigris.org |
| BeanUtils | manipulation library for Java beans<br>http://jakarta.apache.org/commons/beanutils |
| DocCheck | quality insurance for detailed design documentation<br>http://java.sun.com |
| Eclipse | Integrated development environment<br>http://www.eclipse.org |
| JDK 1.4 | Java development kit 1.4<br>http://java.sun.com |
| OJB | Object Java Bridge: O/R mapping tool<br>http://db.apache.org/ojb |
| Poseidon | UML modeling tool<br>http://www.gentleware.com |
| Velocity | code generation template engine |

http://jakarta.apache.org/velocity

# Bibliography

mb-attributes-2003: Marcel Baumann, Business Object Derived Attributes, 2003