TUHH
Technische Universität Hamburg-Harburg

Generation of EJB3 Artifacts in a Modeling Platform

Master Thesis

Submitted by:

Xinhua Gu

Informatik-Ingenieurwesen

Matriculation Number: 15248

Supervised by:

Prof. Dr. Ralf Möller (STS)

Prof. Dr. Rolf-Rainer Grigat (TI1)

M.Sc. Miguel Garcia (STS)

Hamburg, Germany

26th May 2006

Acknowledgment

I would like to thank Prof. Dr. Ralf Möller for giving me the opportunity to work on this master thesis topic and supervising it. I also thank Prof. Dr. Rolf-Rainer Grigat for accepting being my co-supervisor.

Furthermore, I would like to thank M. Sc. Miguel Garcia, for his valuable and continuous feedback and help in enhancing the scope of this work.

Declaration

Hereby, I declare that this master thesis, with the subject “Generation of EJB3 Artifacts in a Modeling Platform”, has been prepared by me. All literal and content related quotations from other sources are clearly pointed out, and no other sources or aids than the declared ones have been used.

Xinhua Gu

Hamburg, Germany
28th May. 2006

Table of contents

Abstract

EJB3 as a part of the new generation Java 5 Enterprise Edition has lots of improved features. These features make an EJB3 persistence entity more object oriented and model driven friendly. Despite the fact that EJB3 development has been greatly simplified, a significant amount of repetitive work remains that burdens the developer during database mapping or code pattern design. In this work, code generation of EJB3 artifacts based on MDA is discussed. Based on analyses of catalogs of database mappings and of their associated code patterns, a generic recipe for code generation is derived.

Chapter 1. Introduction

In this chapter the motivation and the goals of our work are described. Additionally, a short overview of the chapters in this thesis is given.

1.1. Motivation

Compared with the previous version of Enterprise Java Beans, the upcoming generation of Java EE (EJB3) has several great improvements, perhaps the most outstanding of which is the new standard mechanism of EJB persistence, specified by JSR220. Such mechanism provides a transparent persistence layer based on radical reforms. These new features not only simplify the development process, while enhancing the software reusability at the same time, but also make EJB3 more object oriented and model driven friendly. For example, an EJB3 entity can be written in POJO and there are not anymore enforced callback methods. Furthermore, through elaborate mapping annotations supported by the underlying EJB3 ORM engine, classes and relationships in a UML model can be directly mapped into database schema. For all that, the assignment of annotations and the code patterns for database manipulation still mean a lot of work for the programmer. For example, a programmer could repeatedly apply a design pattern or a mapping strategy to maintain the assurance that items will be kept in sequence not only at runtime but also in the database for an indexed collection (e.g. a sequence or an ordered set for an association end). Such a development approach is error prone and results in low productivity. In order to overcome it, we come up with the idea to bring MDA concept to EJB3 persistence development, i.e. generate EJB3 codes of persistence layer in a model driven manner.

In the realm of J2EE application, the concept "MDA" isn't a new word. There are already some tools on the market for old the EJB version (EJB2.x). Although these MDA generated J2EE codes cannot replace manual codes totally, but through MDA code generation, the development process is hugely improved and the development cost is reduced enormously. But one disadvantage of these tools is that the programmer must know J2EE design pattern very well so that he can tune the generated pattern codes to the architect's wish. There is a sentence from a J2EE MDA report [Middleware03] to describe the disadvantages of these J2EE MDA tools: *"It makes brain surgeons better brain surgeons, but it won't make janitors into brain surgeons."* Similarly, we also reviewed the tools for transparent persistence mechanism such as OpenAccessJDO for JDO. Using this tool, the programmer is assumed to be very experienced in mapping object to database with ORM mechanism e.g. for each relationship in the model, an explicit mapping must be manually set. Therefore, to some extent, such tools are more like the tools for configuration.

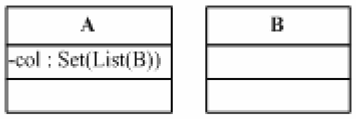
In our viewpoint, when MDA concept is used in EJB3 persistence code generation, codes generated from MDA tool should offer a higher level, more transparent API for programmers, i.e. the programmer need not be forced to be familiar with the underlying ORM techniques. Ideally, when manipulating POJOs at runtime, the programmer should be totally unaware of the interaction with database.

1.2. Objective

The objective of this thesis is to generate EJB3 artifacts for persistence layer on the Octopus modeling platform. The EJB3 artifacts to be generated can be placed in two categories: (1) Mapping annotation and (2) Codes for database interaction.

The position and content of a mapping annotation in codes determines the physical schema in database for a model-level class. Therefore, all databases mapping information are stored through the assignment of Java 5 annotations. According to these assigned annotations, the underlying EJB3 ORM engine triggers appropriate EJB QL for each operation on objects at runtime. Through such background EJB QL query, when the state of an object is changed at runtime, the database will reflect this change transparently i.e. make the change persist. This behaviour is also known as "object query". To make things concrete, let us take a look at following example:

Suppose that we have two classes, A and B. A class contains a collection type (Set) field "col", in which the element type is again a collection (List).



We consider first how to map this model into database. A second question is how the code looks like, as e.g. when adding a list of "B" instances into the collection "col". Rather than solving this isolated problem, the challenge is rather finding general mapping strategies for similar cases and design a code pattern for the mapping strategies.

It should be noticed that our UML model (a class diagram) is subject to the limitation of Octopus descriptive capability i.e. we cannot take an arbitrary Class diagram as our imported UML model. As delivered "out-of-the-box", the code generated by Octopus is compatible with Java 1.4 but not with Java 1.5 which is the precondition for using EJB3. In this thesis, we will discuss the extension of code generation in Octopous. We will however not extend Octopus UML to cover for example all the innovations of UML 2.0.

1.3. Overview

In the next chapter (Thinking in Database mapping) we will discuss and analyze the database mapping for some possible constructs in a class diagram. During the discussion we will give a range of mapping annotation for generation and list all encountered problems at the end of chapter. In chapter 3 (Thinking in Code pattern), we will perform an analysis integrated with Octopus platform. After the anatomy of Octopus-generated code patterns, all potential problems and challenges will be discussed. In the Chapter 4 (Concrete Problems discussion), along with analysis process, we will attempt to give solutions to the problems gathered from previous chapter. Details of all solutions for generation of EJB3 artifacts from discussion will be summarized as a complete generic recipe in Chapter 5. In Chapter 6 an implemention of this recipe will be briefly introduced. In the last chapter (Conclusion), the complete transformation recipe will be reviewed as well as some suggestions and outlook for future work will also be given.

Chapter 2. Thinking in Database Mapping

In JSR-220 Persistence, source code annotations (so called “metadata”) have replaced the XML configuration files of both JDO (Java Data Objects) and Hibernate ORM engines. The purpose remains the same: to convey to the ORM engine the mapping between Java classes and the RDBMS schema. These annotations not only determine the structure of such schema, but to a great degree the performance of the applications interacting with the database. In view of these concerns, we need to come up with an automatic procedure (informally, a “recipe”) for assigning JSR220 Persistence annotations to the Java classes generated from an UML class model.

In this chapter we review the constructs of Octopus UML Class Diagrams and discuss for each of them candidate ORM mappings. In order to choose among alternative mappings, we’ll anticipate both potential problems and their solutions. The corresponding database schema layout will be presented, in order to ease the exposition. The chapter closes with a summary of the annotations selected for generation. This chapter assumes a working familiarity with the JSR220 Persistence specification [JSR220-Persistence].

2.1. Classification of mapping annotations

In [Annotations], all metadata annotations are classified into two categories: logical and physical mapping annotations.

> *The logical mapping annotations (allowing you to describe the object model, the class associations, etc.) and the physical mapping annotations (describing the physical schema, tables, columns, indexes, etc) [Annotation, Section 2.2]*

This classification is equivocal because some annotations for association still affect the structure of database schema, e.g. by creating a foreign key in a table. All along we are concerned only with those annotations that influence the back-end database schema. Therefore, we will focus on Section 8.1 “Entity” and section 9.1 “Annotations for ORM” of [JSR220-Persistence]. All these annotations were defined following a principle of “configuration by exception,” which is supposed to reduce the amount of explicit information that needs to be given to those where a deviation from the usual case occurs.

2.2. Classes and Interface

In an EJB3 persistence model, there are only two different types of object, i.e. an *entity* and an *embeddable component*[1] (which are annotated with @Entity or @Embeddable respectively.) The difference between an entity and embeddable component is that an entity may have a mapped table (and therefore, primary key) while an embeddable component is materialized as fields in row (thus explaining the term “embedded”). An embeddable component must attach itself to a table mapped by an entity. As can be seen, we won’t use the word “entity” in any general meaning (specifically not in the sense of the Entity-Relationship Model) but with the specific meaning given by the JSR-220 Persistence specification.

2.2.1. Entity mapping

Not all classifiers declared in a UML class model can be directly mapped as EJB3 entities. First of all, let’s review the requirements for an entity mapping:

> *Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.([JSR220-Persistence] Section 2.1)*

From this, we can see that an interface cannot be mapped as entity, only a class type can. For UML classes we’ll simply generate “@Entity” above the appropriate Java class header. The class name (which will be used by default by the ORM engine as table name) presents however a potential problem. If there are two classes with the same name in different packages, then a “table name collision” will occur in the database. To prevent this “collision” the annotation “@Table” can be used to specify a unique table name.

In “@Entity” annotation, there is an option element “AccessType” which designates the manner for the entity manager to manipulate the instance variables of this entity. There are two choices for this option “FIELD” of “PROPERTY”. (see Section 2.1.1of [JSR220-Persistence]) Here, we prefer to use “FIELD”. The reasons are:

1. From the point of view of model driven generation, the accessor methods for an instance variable are not supposed to include business logic. Therefore, an access type of “PROPERTY” presents no advantages compared to “FIELD”.
2. Since in the Octopus code pattern a getter method for a collection type instance variable will return an unmodifiable collection, this will prevent any update if “PROPERTY” is chosen.

[1] Enumerated type is treated as an embeddable component in EJB3 ORM mechanism.

2.2.2. Embeddable component mapping

An Embeddable component provides for a larger granularity of the database so that lots of unnecessary "join query" (which are considered as "performance killer") can be avoided. Normally, an embeddable component is used in a "Has-A" relationship, e.g. Class A has an attribute of Class B instance. In this case, Class B can be mapped without problems as an embeddable component (more information in Section 9.1.32 and 9.1.33 of [JSR220-Persistence]).

Regarding the "@Embeddable" annotation, the following points should be noticed:

1. Suppose that we have two Classes A and B. Class A as embedded class and Class B as embedding class. If A and B both have a String type attribute "name". Then, the two attributes will be mapped to different columns in the same table with same name! To avoid this overlap, the optional element "OverrideColumn" in "@Embeddable" can be used. Alternatively, an "@Column(name=xxx)" can be generated above each overlapping attribute. A unique column name can be specified through "name" option element.
2. For an enumerated class type (JDK1.5) no annotation is necessary. By default, ordinal values will be used.

2.3. Inheritance and polymorphism

The JSR220 specification offers three strategies for inheritance mapping. If an "@Inheritance" annotation is not assigned, the "Single table per class" strategy will be used. The other two strategies have specific drawbacks. Normally the "Table per class" strategy is not advocated especially regarding polymorphic queries or association. Moreover, this strategy does not allow using "AUTO" and "IDENTITY" for identity generation ([Annotations] section 2.2.4.1). For "Joined subclass", separate tables for subclasses increase the granularity of the database layer, with a consequently reduced performance. Therfore, we prefer to use the default mapping. For more information about inheritance strategies, please read [Annotations] section 2.2.4 or [King05] section 3.6.

If we follow the "Single per class" strategy, all we can do is to map subclasses as an Entity. By default, a discriminator column named "TYPE" will be created in the table to distinguish each subclass. We still need to notice the following points:

1. "AccessType" of a subclass will be inherited from super class (or root entity of the entity hierarchy.). It is unnecessary to define it again in subclass.
2. In "Single table per class" strategy, although each subclass is annotated with "@Entity", it is actually embedded into its super class table i.e. that is not necessary to assign a primary key for subclass. It is very important to know this, especially when we assign primary key for each entity in model driven manner.
3. Because interfaces cannot be mapped at all (neither as entity nor as embeddable). The inheritance strategies cannot be applied to "interface inheritance".

2.4. Attributes and operations

From an UML class diagram, we can distinguish three cases that are relevant from the viewpoint of persistence: “Inlined attributes”, “Attributes by relationship” and “Derived attributes”. “Inlined attributes” and “Derived attribute” are directly defined in Class. “Attributes by relationship” are actually the reference attributes which are generated as a result from UML associations (such fields materialize the association ends) ([Pitman05] section2.2)

For a “Derived attribute”, there is a “@Transient” annotation. This annotation simply prevents the attribute from being persisted in the database. If an attribute is an “Inlined attribute”, then things get a bit more complex. The following problems are possible for the mapping of an “Inlined attribute”:

1. We cannot determine the character of the attribute, e.g. whether it is mutable (which can be annotated with “@Temporal”) or whether it serves as a version field for an optimistic lock mechanism (which can be annotated with “@Version”) etc.
2. Furthermore, from the model definition, we can not determine the database constrains for the column mapped by this attribute, i.e. uniqueness, nullability, insertability, updatability, length of column, precision etc.
3. We also do not know which attribute or attributes should be mapped as primary key. (It is also possible that a class might contain no attribute declarations)
4. Unfortunately, the JSR220 specification does not offer corresponding annotations for supporting indexed collection type (which would be handy in connection with association ends marked as {ordered}). The developer should find the way to handle it himself. A generic recipe for database mapping of attributes should handle not only the case of an “Inlined attribute” whose type is an indexed collection, but also the case of nested collections.

A straightforward solution to the “primary key” problem is to create a “surrogate key” for the entity ([King05] Section 1.2.3). This additional attribute could be of type long and annotated with “@Id”. We can set the option element “generate” to “GeneratorType.AUTO” which makes the entity manager decide the appropriate identity generation strategy for underlying database. It should be notice that the name of this additional attribute must be unique in the entity or a naming collision will occur.

As a simple approach, we do not place any annotation for primitive type attributes and we the let entity manager perform “configuration of exception”. The default database constrains for the mapped column will be:

unique	nullable	insertable	updatable	length	scale	precision
false	true	true	true	225	0	0

For single object type or collection type “Inlined attribute”, we will discuss them in the following chapters.

UML model defined operations do not participate in our database mapping. Business logic is coded in them and has only meaning at the code level. It does not make sense to assign mapping annotations to them.

2.5. Relationships between UML classifiers

2.5.1. Dependency and Generalization

There are only three representations for a dependency relationship in model:

- a class type is used as a local variable type in other class
- a class type is used as parameter type in other class, or
- the static method of a class type is invoked by other class.

Consequently, dependency is a very weak relationship and it will be ignored by ORM.

Generalization is a "IS-A" relationship between two types. In previous section, we have discussed it. It should be noticed that in our case, generalization between interface and class (multiple inheritance) or interface and interface (interface inheritance) cannot be mapped into database schema.

2.5.2. Association

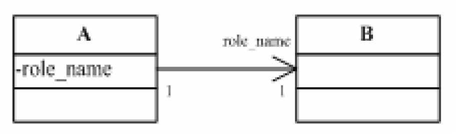
Association is most common kind of relationship in UML modeling. Mapping a UML association into a relationship at the database level is also a main task for an ORM engine. The JSR220 specification gives us four annotations for mapping associations: "@ManyToMany", "@ManyToOne", "@OneToMany and "@OneToOne". Using these annotations, the developer can map unidirectional or bidirectional "one to many" "one to one" or "many to many" associations. However, there are some other possible associations in a UML model that not covered, e.g. association with association class, or self association. Mapping these associations is still the responsibility of the developer. In our model driven approach, the developer is to be relieved from this task.

In the following subsection, we will discuss how to use the four ORM mappings for annotations to handle the different varieties of UML-level associations. All along we'll consider the resulting database schema for each mapping.

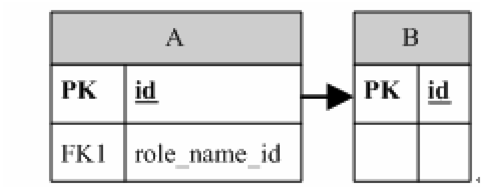
2.5.2.1. Unidirectional association mapping

➢ *Unidirectional One to one*

Scenario: There is a "one to one" association between A and B (from A to B). In Class A @OneToOne is added on the reference field that refers to B. ("role_name" indicates role name of B in this association)

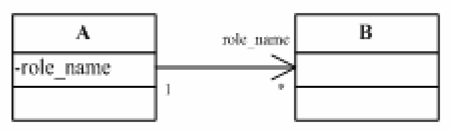


In table A, a unique constrain will be added on foreign key “role_name_id” to table B. (“role_name” indicates the role name of B in this association)

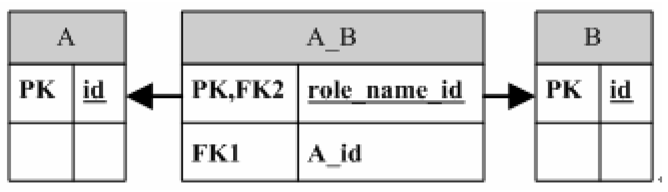


➢ *Unidirectional One to many*

Scenario: There is a “one to many” association between A and B. (From A to B) In Class A @OneToMany is added on the reference field which refers to B. (“role_name” indicates the role name of B in this association).

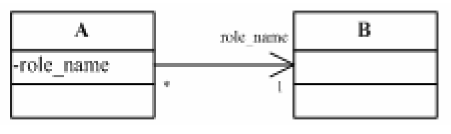


Notice that, @OneToMany annotation used in a unidirectional association does not support direct foreign key mapping. Furthermore, a unique constrain will be placed on the foreign key to table B.

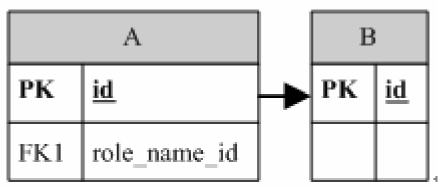


➢ *Unidirectional Many to one*

Scenario: There is a “many to one” association between A and B. (From A to B) In Class A @ManyToOne is added on reference field which refers to B.



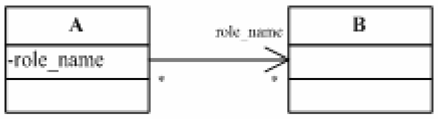
The mapped database schema:



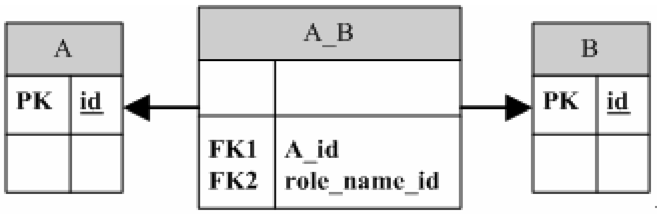
Notice that, foreign key “role_name_id” to table B does not have unique constrain. This is the only difference from the database schema of unidirectional “one to one”. In unidirectional “one to one”, foreign key to table B must be unique, that is not required in unidirectional “Many to one”.

➢ *Unidirectional Many to many*

Scenario: There is a “many to many” association between A and B. (From A to B) In Class A @ManyToMany is added on the reference field which refer to B.



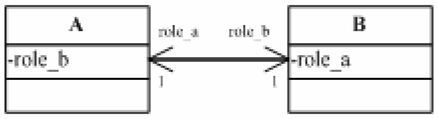
The database schema is similar to the one which is mapped by unidirectional “one to many”. The only difference is, there is no unique constrain on foreign key “role_name_id” in join table “A_B”.



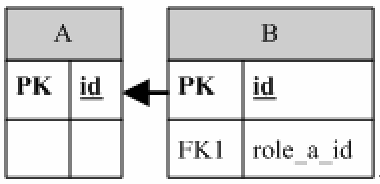
2.5.2.2. Bidirectional association mapping

➢ *Bidirectional One to one*

Scenario: There is a “one to one” association between A and B. If A is owner side, B is inverse side, in Class A @OneToOne is added on the reference field which refers to B and in Class B @OneToOne with option element “MappedBy” is added on the reference field which refers A. (“role_a” is the role name of A, “role_b” is the role name of B)

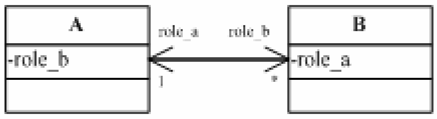


The schema is the same as unidirectional “one to one”. If B is owner side and A is inverse side, then Table B will contain the foreign key “role_a_id” to table A, and a unique constrain will be putted on it.

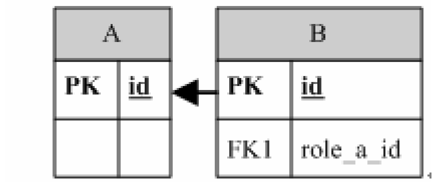


➢ *Bidirectional One to many / Many to one*

Scenario: There is an “one to many” association between A and B.(A is one side) A is assigned as inverse side, B is owner side, in Class B @ManyToOne is added on the reference field which refers to B and in Class B @OneToMany with option element “MappedBy” is added on the reference field which refers to A.



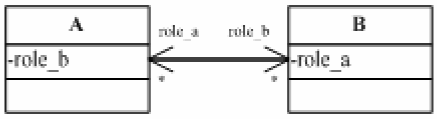
The schema is the same as unidirectional “many to one”.



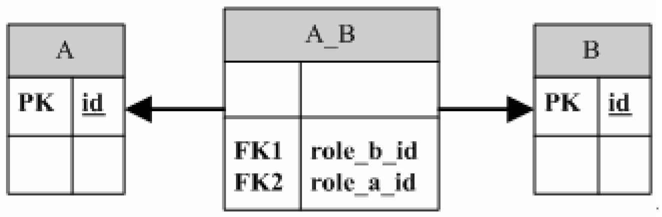
In @ManyToOne option element “mappedBy” is not defined, that means in bidirectional “one to many” or “many to one” association, “many” side must be owner side.

- *Bidirectional Many to many*

Scenario: There is a “many to many” association between A and B. If A is owner side, B is inverse side, in Class A @ManyToMany is added on the reference field which refers to B and in Class B @ManyToMany with option element “MappedBy” is added on the reference field which refers to A. (“role_a” indicates role name of A and “role_b” indicates role name of B)

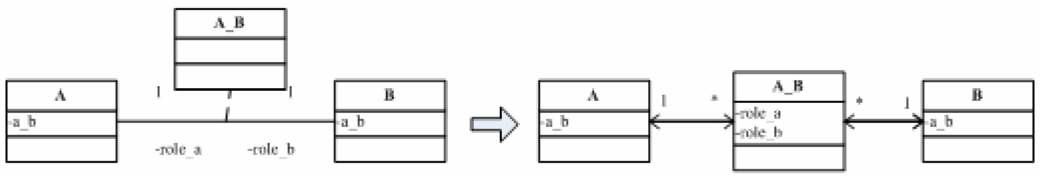


There is no any unique constrain on both foreign keys.

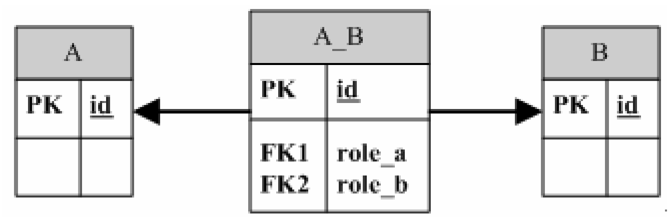


- *Bidirectional Many to many with association class*

Scenario: A_B is an association class between A and B. The association between A and B can split into two bidirectional associations. In case of “many to many”, they are bidirectional “one to many” between A and A_B (A is one side) and bidirectional “one to many” between B and A_B (B is one side).

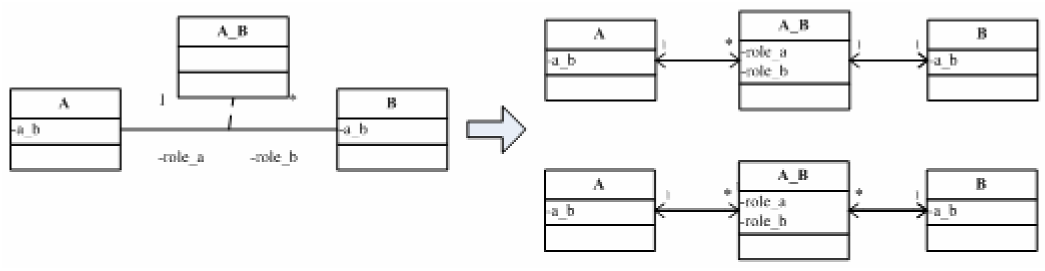


Because in bidirectional “one to many” or “many to one” association, “many” side must be owner side, so, all foreign keys will be hold in association class. Then we have following figure. (“role_a” indicates role name of A and “role_b” indicates role name of B)

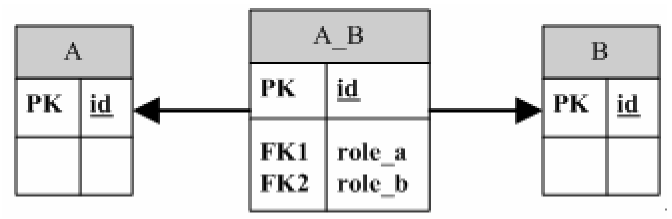


➢ *Bidirectional One to many with association class*

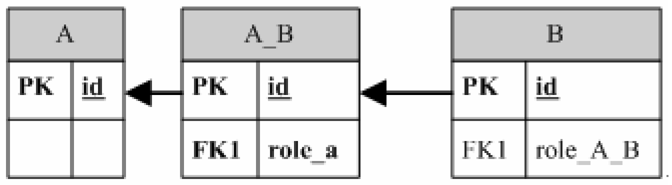
In case of one to many association between A and B (A is one side), the relationship between A_B and B should be bidirectional “one to one” or “one to many”. (B is one side).



If “one to one” is used, that means each B instance can only be related to one A instance, i.e. B side as a many side is an unique collection type. If B side is a non unique collection type, then bidirectional “one to many” between B and A_B should be used. In this case, the mapping is same as bidirectional Many to Many with association class.



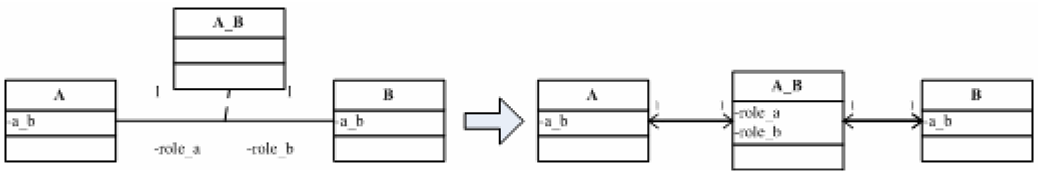
In fact, if the association between B and A_B is one to one, the database schema will have two possible forms. That is because in bidirectional “one to one”, arbitrary side can be owner side. Suppose that if we assign B as the own side, then B table will hold the foreign key. The data schema will be:



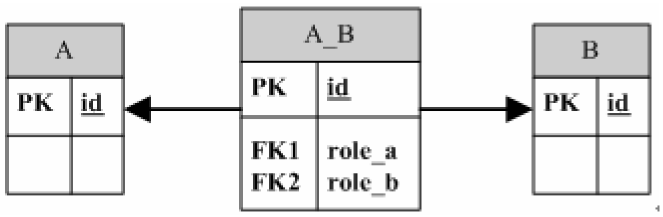
Here, “role_A_B” indicates role name of A_B in association between B and A_B. Clearly, when B is inverse side, association class A_B will hold the both foreign keys again.

➢ *Bidirectional One to one with association class*

In this case, the association can be split into two bidirectional “one to one” associations.



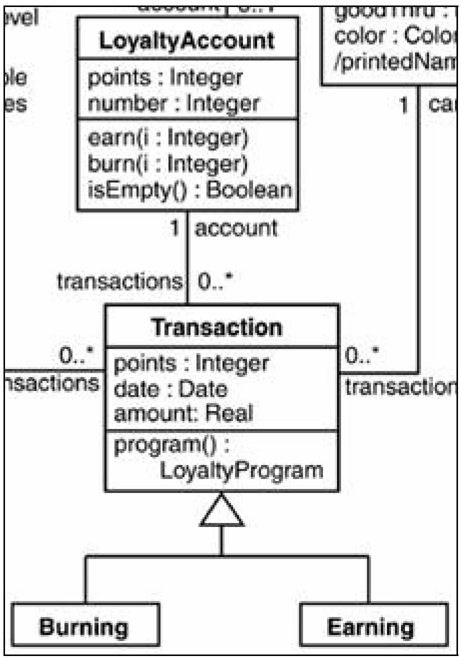
As a convention, we assign the both association end as inverse side, so that the association class can hold the both foreign keys.



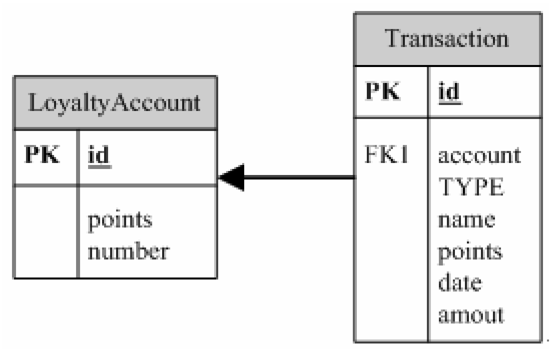
2.5.2.3. Polymorphic association

A polymorphic association is an association that may refer to instances of a subclass of the class that was explicitly specified in the mapping metadata.([King05] section 6.4.1)

Scenario: In RandL project, "Transaction" class owns two sub classes "Burning" and "Earning".



If we use default inheritance mapping strategy "per single table" i.e. "Burning" and "Earning" Class are only annotated with @Entity. All "Inlined attributes" in both Classes will be mapped as columns in table Transaction. Suppose both "Burning" and "Earning" classes have a field "name". For the relationship between "LoyaltyAccount" and "Transaction" we can use normal bidirectional "one to many" association mapping. Here, if "LoyaltyAccount" Class is set to be the owner side. Then, we will get following database schema:

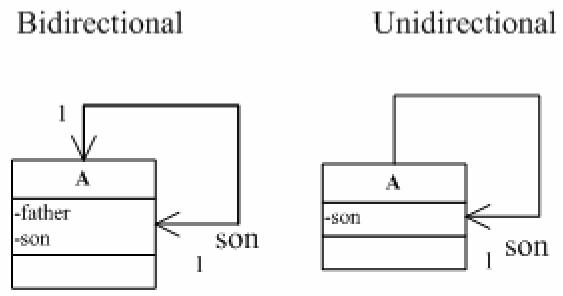


(Column “TYPE” is the discriminator column).

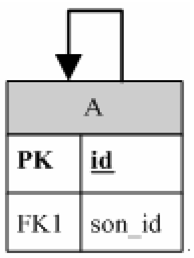
2.5.2.4. Self-association

➢ *One to one self association*

Scenario: Class A has a bidirectional “one to one” association to itself. A has two role names; one is “father” the other is “son”. Field “father” and “son” declared in Class A are corresponding reference fields for the two roles. In this association the owner side and inverse side still need to be assigned. Suppose that we assign “son” side as inverse side, i.e. to put @OneToOne annotation with option element “mappedBy” on the “son” field and the “father” field will be only annotated with @OneToOne.



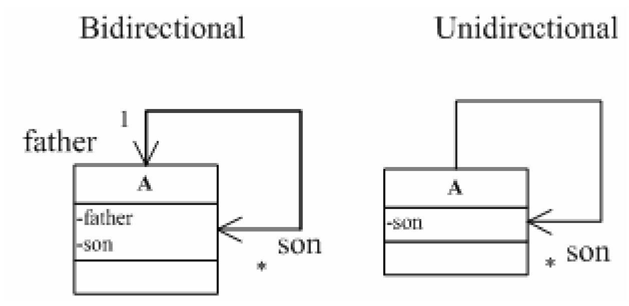
Then we get following database schema:



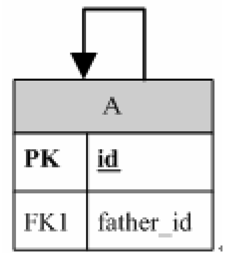
If the association is unidirectional (from father to son), the unnavigable side “father” is implicit owner side. Class A will contain one reference field “son” which is annotated @OneToOne. The database schema will be the same as above figure.

➢ *One to many self association*

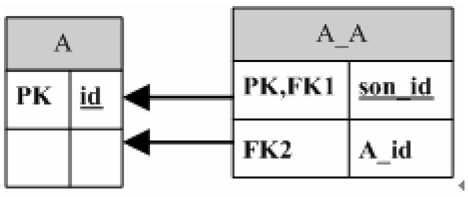
Scenario: Class A has a bidirectional “one to one” association to itself. A has two role names; one is “father” the other is “son”. Also, field “father” and “son” in Class A are corresponding reference fields for the two roles.



Because in bidirectional one to many, many side must be owner side, “son” will contain a foreign key to “father”. Thus, the database schema will be:



When the association is unidirectional, i.e. from “father” to “son”, “father” will be the owner side. In database schema a join table will be created:



2.5.2.5.Multiplicity constrains

EJB3 ORM engine does not take care of the precise multiplicities value of an association end, i.e. it does not distinguish [0..7] and [0..*]. At annotation level, we can only define an association either “many” side or “one” side. But, we can use option element of association mapping annotation to give a furthest support for multiplicity constrains. Suppose that there is a “one to many” association between A and B. B has multiplicity [1..*]. That means, the association end must also exist, if the association exists, i.e. association is not optional. Therefore, we can set option element “optional” in association mapping annotation to give such constrain:

Observe the multiplicity of a navigable association end,

- If the lower bound of the multiplicity is “0”, then add “optional=false” in association mapping annotation of the opposite side.
- If the lower bound is a number “>= 1”, then add “optional=true” in association mapping annotation of the opposite side. (Absence of “optional” option element in annotation, the association will be also explained as an optional.)

2.5.2.6. Summary of association mapping annotation

Owner side and inverse side:

- In unidirectional association, unnavigable side must be owner side. There is no inverse side.
- "mappedBy" option element only appears in inverse side. It indicates the foreign key in the owner side table.
- In bidirectional "one to many" association, owner side must be a "many" side. That is because in "@ManyToOne" annotation there is no "mappedBy" to set.
- In bidirectional "one to one" or "many to many" association, arbitrary side can be owner side.

Non-supported association mapping

- An indexed association end with upper-bound > 1 (with or without association class)
- A non-unique association end with upper-bound > 1 (in case of unidirectional one to many)

Position of forgein Key:

(FK=forgein key)		owner side table	Join table
One to many	Uni.		FKs for both sides
	Bi.	FK points to one side	
One to one	Uni.	FK points to navigable side	
	Bi.	FK points to inverse side	
Many to many	Uni.		FKs for both sides
	Bi.		FKs for both sides

2.6. Aggregation and composition

Both aggregation and composition are stronger version of association. (Composition is the strongest). An Aggregation indicates a "…own a …" relationship and the difference form normal association is very subtle. This subtle semantic difference cannot be mapped in database. A Composition is a stronger association. In this relationship, one side must be an "owner" and the other side is/are "part". The life cycle of "part" depends on the life cycle of "owner", in other words, if "owner" disappears, "part" cannot exist all by itself. In order to map this semantic constrain into database, we should use option element "cascade" in association mapping annotation, i.e. set "cascade=CascadeType.DELETE".

2.7. Conclusion for model driven database mapping

2.7.1. Main problems for database mapping

1. *Mapping decision for "@Entity" and "@Embeddable"*

The problem is we must decide which class type is mapped as an Entity and which is mapped as an embeddable component. A simplest approach is to map all the class type in model as entity. But in consideration of performance, it is not a good approach.

Suppose we prescribe that all class types which is used as an "Inlined Attribute" type will be mapped as embeddable component, e.g. "Date" class in RanL Project, there are still some problems in special situations. For example, when "Date" class has a self association, because it is mapped as embeddable component; it will not own a database identity and the association will not be mapped.

The possible solution for this problem is to give particular conditions for entity or embeddable component mapping.

2. *Potential problem of association class mapping*

In our mapping strategy for association class mapping, association class will hold the both foreign keys of the association ends. This schema will bring problem when we perform a remove operation. For example, A_B is an association class between A and B. In A_B table, there is a record links to two records in A table and B table respectively (through foreign key), If we remove the association from A side, only a "update" EJB3QL statement is executed on A_B table, not "delete" statement. This result in that the record in A_B table actually will not removed. Only the both foreign keys will be wiped off. This problem will make the A_B table increased unending.

The possible solution for this problem is to map the both foreign keys as a composite primary key of the association class table. (Please refer to the database schema of unidirectional "one to many"). Unfortunately, it is impossible to make such mapping in current entity manager (e.g. Hibernate entity manager). In practice, by using embeddable class (as a primary key class) and "@EmbeddedID", we mapped the both foreign keys as composite primary key successfully, but precondition is the association between A and A_B or B and A_B must be unidirectional. In case of bidirectional, we failed to map. For more information please consult the Hibernate Entity manager community.

In next chapter we will find that this solution will collide with code pattern of Octopus.

3. *Problem of various collection type mapping*

An association end could be one of the four collection types i.e. Sequence, Bag, Set or Ordered Set. By database mapping, EJB3 ORM engine does not take care this information of collection type.

- For the case that the association end is indexed collection (Sequence or Ordered Set)

Each element in indexed collection will be stored in order. Although there is an annotation "@OrderedBy" specified in JSR220 document, through this annotation, collection is only sorted at runtime, the order of collection is actually not persist in database. Handling this index information and making it persist in database is again the developer's work. The challenge for us is to give a generic strategy for supporting such indexed association (with association class or without association class) in view of model driven.

- For the case that the association end is unique collection (Set or Ordered Set)

Suppose that there is a bidirectional "one to many" between A and B and B is the many side. In B table, there will be foreign key points to A table. If B is a Set or Ordered Set, the value of foreign key in each B record cannot be duplicated. In order to make this constrain we have two approaches available. One is to put "@UniqueConstraint" annotation to specify the unique column in the table. Another simpler approach is based on codes, i.e. before saving a B instance into database, check if same instance is already in the collection at runtime.

- For the case that the association end is non-unique collection (Bag and Sequence)

We will have problem when association is unidirectional "one to many". This is so because the mapped database enforces a unique collection type on "many" side.

2.7.2. Selected annotations for generation

According to the preceding discussion, the following metadata annotations are supposed to be generated:

➢ @Entity	➢ @ManyToOne
➢ @Embeddable	➢ @OneToOne
➢ @Table	➢ @OneToMany
➢ @Id	➢ @ManyToMany
➢ @Transient	➢ @OrderedBy

2.8. Summary

To achieve a model driven database mapping, a generic mapping strategy for each possible construct in a UML class diagram must be given so that appropriate mapping annotations can be automatically assigned in the right place in accordance with these strategies. In this chapter, we attempted to give a mapping strategy for each construct in a Class diagram and listed all encountered problems and difficulties by mapping. As a conclusion, all annotations which will be generated in model driven database mapping are confirmed.

Chapter 3. Thinking in Code Pattern

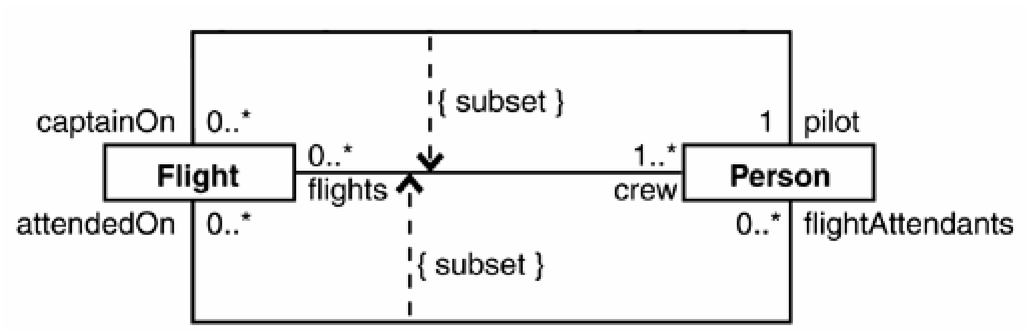
We do not only want EJB3 artefacts generated with mapping information but also want the generated code to provide for manipulating database data in an object-oriented manner. The interaction behaviour on database depends on the logic presented in the generated code. Thus, in this chapter, we will first analyze the Octopus code pattern in depth. Afterwards, we will expose and discuss some differences between EJB3 artefacts and Octopus generated code style. Finally, a new code generation approach will be introduced.

3.1. Limitations of Octopus UML

Octopus UML has its own textual syntax. The capability of its expression rather limited. It cannot be used to represent the whole UML2.0 syntax. Since an imported UML model (Class diagram) will be firstly conversed into Octopus UML, the limitation will also restrict the presentation of imported UML model. In the following text we enumerate some of the limitations relevant for EJB3 generation.

➢ *Absence of UML notation*

There is no expression in Octopus UML syntax for some UML notations, especially, for the constraint notation in generalization or multiple associations. These constraints could be performed by using OCL, but at this point OCL is not part of the translation to EJB3. An example of constraints notation which is used in two associations is shown in following figure:



From [Kleppe03]

➢ *Special enumerated type*

Octopus UML has symbol <enum> to define an enumerated type. In <enum> definition, only "string" can be set as enumeration value and no operations may be defined. This in contrast to the more complex Enumeration supported by Java 1.5 or above.

➢ *Limitation on type for attributes*

The type of an attribute may be a primitive type, a user-defined classifier (class, interface, enumeration) or a collection type (Sequence, Bag, OrderedSet, Set). Other than the String type, OCL supports only three primitive types, i.e. "int", "float" and "boolean". The four OCL primitive types together with the String type are known as "Basic types" (see section 7.2 in [Kleppe03])

➢ *Some other limitations*

Class, attribute and operation cannot be defined as "final"; meanwhile, attribute and operation can not be defined as "static" etc. In our case, these should cause no trouble because, "final" class , attribute or operation will be ignored by EJB3 ORM engine and a static operation has no impact on database schema.

> *The entity class must not be final. No methods or persistent instance variables of the entity class may be final ([JSR220-Persistence] section 2.1)*

3.2. Octopus Code Pattern

For convenience, we follow some naming conventions next:

1. An "attribute defined field" in Java code corresponds to "Inlined attribute" in UML. (Because, this field is generated by <attribute> symbol in Octopus UML file)
2. An "association defined field" in code corresponds to "Attribute by relationship". (Because, this field is generated by <association> or <associationclass> symbol in Octopus UML file.)

Since the body of each Java file (class, interface and enumeration) consists of three parts, "field", "method" and "the code body in method", we will also analyze the code pattern for each of them.

3.2.1. Generation of fields

3.2.1.1. Attribute defined field

If field is not of an object type, the generated type will be assigned as per the following table:

in Octopus UML file	generated type
Real	float
String	String
Integer	int
Boolean	boolean

The type of attribute defined field could be a collection type which must be one of OCL supported collection types, i.e. "Bag", "Set", "Sequence" or "Ordered Set". The collection could be nested, e.g. Set(Bag(Sequence)). The generated collection type for this field will only depend on the type of root collection (in preceding example, "Set" is the root collection).

in Octopus UML file (root collection)	generated type
Set	java.util.Set
Bag, Sequence or Ordered Set	java.util.List

The attribute defined field name is the name of attribute defined in model. (with "f_" as prefix)

3.2.1.2. Association defined field

Association defined field will be created according to particulars of the association.

➢ *In case of no association class:*

If the opposite side is not navigable, field will not be created.
If the opposite side is navigable and is a "many" side, a collection type association defined field will be created. (The type of the element in this collection will be the object type of opposite side)
If the opposite side is navigable and is a "one" side, a non-collection type association defined field will be created. (The type will be the object type of opposite side)

The role name of the opposite side will be set as association defined field name (with "f_" prefix). If the role name is not defined for opposite side (in Octopus UML <noName> symbol is set), the class name of opposite side will be treated as role name.

➢ *In case of association class:*

If there is an association class, then both association ends must be navigable. (This condition will be checked in first step of Octopus code generation process). If the opposite side is "many" side, a collection type association defined field will be created. (The type of the element in this collection will be the association class type) If this side is "one" side and the opposite side is "one" side too, a non collection type association defined field will be created. (The type will be the association class type) The association class name will be set as association defined field name. (with "f_" prefix)

3.2.2. Generation of methods

3.2.2.1. Methods for attribute defined field

For each attribute defined field, Octopus will generate its accessor methods i.e. setter and getter.

Section 1	get__()
	set__(element_type)

(Here, "__" stands for attribute defined field name i.e. role name of opposite side; "element_type" is field type)

If attribute defined field is a collection type, Octopus will generate additional methods for collection manipulation.

Section 2	addTo__(element_type)
	addTo__(collection_type)
	removeFrom__(element_type)
	removeFrom__(collection_type)
	removeAllFrom__()

(Here, "element_type" is still field type; "collection_type" is either java.util.List or java.util.Set)

It should be noticed that in the original Octopus distribution no methods are generated for indexed collection type, e.g. get__At(), addTo__At() etc.

3.2.2.2. Methods for association defined field (without association class)

If the opposite side is navigable, the methods in "Section 1" (for association defined field) will be generated.

If the opposite side is "many" side, the methods in "Section 2" will be added.

If this side is navigable too, i.e. association is bidirectional, following methods will be added:

Section 3	z_internalAddTo__()
	z_internalRemoveFrom__()

(Here, "__" is association defined field name)
For the explanation of the both "inner methods" the reader is referred to [Kleppe05].

3.2.2.3.Methods for attribute defined field (with association class)

Similarly, all methods in “Section 1” for the association defined field will be created, in addition, two more methods (getter and setter for opposite side object) will be also created despite that there is no relevant field declared. The only reason for this is that the developer should not be conscious of the existence of association class. When he operates the association from one side, he does only know the role name of the opposite side. Therefore, “get__() and set__(element_type)” (“__” stands for role name of the opposite side) is a dedicated API for this purpose.

Section 1	get__()
	set__(element_type)
	get_&&_()
	set_&&_(element_type)

(“__” stands for role name of the opposite side; “_&&_” stands for the name of association defined field i.e. class name of association class.) In this case, role name of opposite side (“__”) and association defined field name (“_&&_”) is not same.

If the opposite side is “many” side, all methods in “Section 2” will be still added. Notice that “__” stands for role name of opposite side, not the name of association defined field.
Since an association with association class defined in Octopus UML must be bidirectional, “inner methods” in “Section 3” still need to be added. But there is a small difference from preceding one:

Section 3	z_internalAddTo_&&_() z_internalRemoveFrom_&&_()

(Here, “_&&_” stands for association defined field name i.e. association class name.)

3.2.2.4. Summary of Methods generation in Octopus Code Pattern

If we combine the three previous Sections, then the generation process can be described in a systematic way.

Section 1	Section 2	Section 3

get__()
set__()
	addTo__(element)
	addTo__(collection)
	removeFrom__(element)
	removeFrom__(collection)
	removeAllFrom__()
		z_internalAddTo__()
		z_internalRemoveFrom__()

get_&&_() set_&&_()		z_internalAddTo_&&_() z_internalRemoveFrom_&&_()

For an attribute defined field:
(“__” stands for then name of attribute defined field)
Section 1 will be generated, if field is a collection type, Section2 will be added.

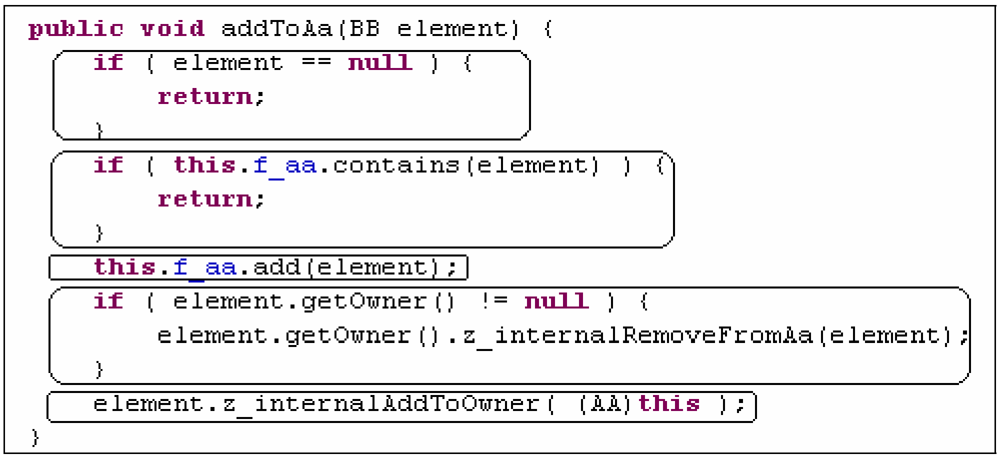
For an association defined field:
(“__” stands for role name of opposite side;”_&&_” stands for the name of reference field which refers to association class)

If opposite side is navigable, Section 1 will be generated. In case of association class, "get_&&_()" and "set_&&_()" will be added.
If opposite side is navigable and is "many" side, Section 2 will be generated.
If association is bidirectional, Section 3 is generated. In case of association class, "z_internalAddTo__()" and "z_internalRemoveFrom__()" will be replaced by "z_internalAddTo_&&_()" and "z_internalRemoveFrom_&&_()".

3.2.3. Generation of method body

3.2.3.1. Code fragment based analysis

In order to simplify the code analysis process, we partition the code body of each method into several code fragments. Each code fragment is independent of other fragments and encapsulates a snippet of logic. For example, for an addTo__() code body :



```
public void addToAa(BB element) {
    if ( element == null ) {
        return;
    }
    if ( this.f_aa.contains(element) ) {
        return;
    }
    this.f_aa.add(element);
    if ( element.getOwner() != null ) {
        element.getOwner().z_internalRemoveFromAa(element);
    }
    element.z_internalAddToOwner( (AA)this );
}
```

We can split the code body into five parts (code fragments):
1. check if parameter is null (check null parameter)
2. check if collection already contains this element (check duplication)
3. add element into collection (add)
4. clean the old relationship of element, if it has one (clean relationship from opposite side)
5. build a new relationship for element (build relationship from opposite side)

For each code fragment, we have given a fixed name, these named code fragment will appear repeatedly in different methods. Under different situation, the combination of these fragments in a method is also different. Moreover, a code fragment is an abstract of a snippet of logic, it could have different implementations. For example, "check null parameter" code fragment may have following forms:

`if ( element == null ) {` `return;}`	`if ( element == null ) {` `return null;}`

Actually they do the same thing. Therefore, we consider only the logic specific to the code.

3.2.3.2. Association sensitive methods

Under different association conditions, some generated methods will be constructed from different combination of code fragments. We call these methods as “association sensitive methods”. These methods are: “set__()” method in “Section 1” , “addTo__(element)” and “removeFrom__(element)” methods in “Secion 2”. (In case of association class, only “addTo__(element)” is sensitive for different under different association conditions). In followed text, we will discuss the code fragment detail for the three methods in case of without association class. Due to the complexity of code pattern for association class case, the involved methods will be reviewed in next section solely.

To make things concrete, we suppose that there is an association between A class and B class. B side (opposite side) has a role name “role_b” and A side has a role name “role_a”. The association sensitive methods in A class (this side) will be reviewed here.

➢ *set__(element_type)*

(Here, we assume that B is “one” side, i.e. “element_type” is class type of B. If B is “many” side, the combination of code fragments will be the same, only the implementation of particular code fragment might be different.)

1	`if ( this.role_b != null ) {`
	`    this.role_b.z_internalRemoveFromRole_a( (A)this );}`

2	`this.role_b = element;`

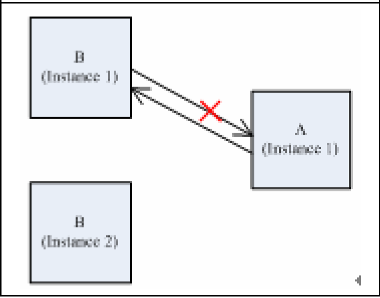
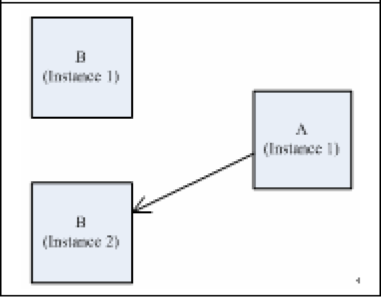
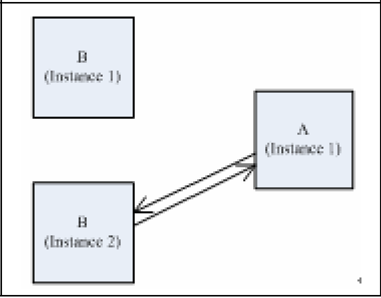
3	`if ( element != null ) {`
	`    element.setRole_a( (A)this );}`

1. clean relationship from opposite side (generated iff the association is bidirectional and this side is “one” side)
2. set
3. build relationship from opposite side (generated iff the association is bidirectional)

The first code fragment “Clean relationship from opposite side” is used to guarantee the “Agreement” principle from “ABACUS” rules (see “Associations are ‘Marriages’ in [Kleppe05]). In our scenario, before a new association is built between A and B, A must clean the old relationship to other B instance because A can only contain one relationship to B (A is “one” side). If it has one, it should first notify its related B instance to destroy (clean) the link to it. Since the “clean” action is actually performed by B and invoked by A, we use the term “from”.

The second code Fragment in “Set” builds the link from A to B. This behaviour also destroyed the link from A to its old related B instance. Afterwards, last code fragment “build relationship from opposite” will create a link from new instance B to A so that the bidirectional association is completed. The following table shows the process:

(Instance 2 of B will be set in instance 1 of A)

Clean relationship	Set	Build relationship
		

➢ *addTo__(element_type)*

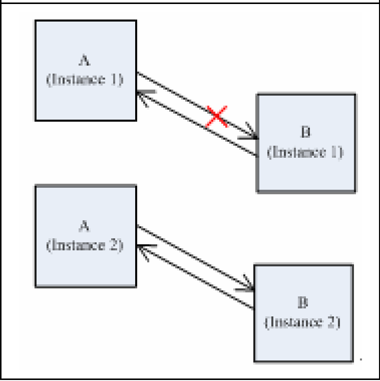
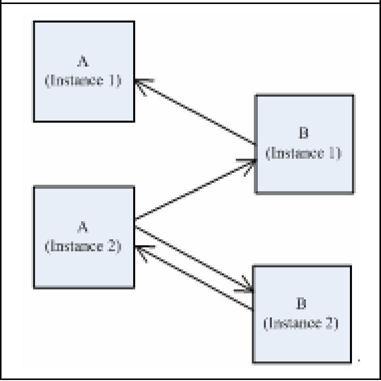
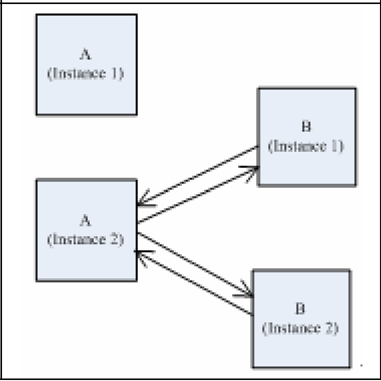
(Suppose that B side is "many" side, then "element_type" indicates B class type.)

```
1  if ( element == null ) {return;}
2  if ( this.role_b.contains(element) ) {return;}
3  if ( element.getRole_a() != null ) {
       element.getRole_a().z_internalRemoveFromRole_b(element);}
4  this.role_b.add(element);
5  element.z_internalAddToRole_a( (A)this );
```

1. Check null parameter
2. Check duplication (be generated iff the opposite side is unique collection type)
3. Clean relationship from opposite side (generated iff the association is bidirectional and this side is "one" side)
4. Add
5. Build relationship from opposite side (generated iff the association is bidirectional）

The following table shows the process:

(Instance 1 of B will be added to instance 2 of A)

Clean relationship	Add	Build relationship
		

➢ *removeFrom__(element_type)*

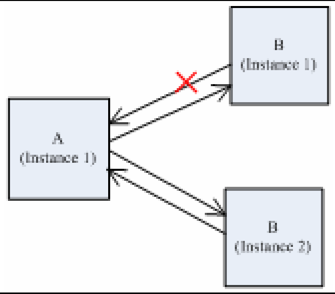
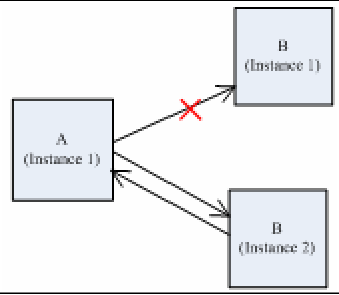
(Suppose that B side is "many" side, then "element_type" indicates B class type.)

1	`if ( element == null ) {return;}`
2	`if(!this.role_b.contains(element)) return;`
3	`element.z_internalRemoveFromRole_a( (A)this );`
4	`this.role_b.remove(element);`

1. Check null parameter
2. Check existence
3. Clean relationship from opposite side (generated iff the association is bidirectional)
4. Remove

The following table shows the process:

(Instance 1 of B will be removed from instance 1 of A)

Clean relationship	Remove
	

Methods for association class case

A given scenario for this section is: there is an association between A and B, A_B is their association class.

1. Relationship related methods in association class

In relationship class case, the association must be bidirectional, furthermore, the relationship between A and A_B and the relationship between B and A_B should be constructed or destroyed at the same time. In an association class, two methods perform this: constructor method and clean() method.

➢ Constructor

1	`if ( a == null && b == null ) return;`

2	`a.z_internalAddToA_B(this);`
	`b.z_internalAddToA_B(this);`

3	`this.role_a = a;`
	`this.role_b = b;`

1. Check null parameter
2. Build relationship from opposite side
3. Build relationship from this side

During the instantiation of the association class A_B, the both relationships to A and B is also constructed.

➢ clean()

1
```
role_a.z_internalRemoveFromA_B(this);
role_b.z_internalRemoveFromA_B(this);
```

2
```
role_b = null;
role_a = null;
```

1. Clean relationship from opposite side
2. Clean relationship from this side

2.Association sensitive methods in association end (e.g. in A class)

➢ addTotRole_b(B par)

(Suppose that B side is “many” side.)

1
```
boolean isPresent = false;
Iterator it = a_B.iterator();
while ( it.hasNext() && !isPresent ) {
     A_B elem = (A_B) it.next();
     if ( elem.getRole_b() == par ) {
          isPresent = true;}}
if ( isPresent ) return;
```

2
```
A_B asscls = new A_B(this, par);
```

3
```
a_B.add(asscls);
```

1. Check the duplication (generated iff the other association end corresponds to a indexed collection type)
2. Build relationship from opposite side
3. Build relationship from this side (this part is not necessary!)

➢ *removeFromRole_b(B par)*

(This method is not association sensitive, because no code fragment is generated under conditions. We put this method here to show that an unnecessary part of code is generated in standard Octopus code pattern)

1
```
A_B foundElem = null;
Iterator it = a_B.iterator();
while ( it.hasNext() ) {
     A_B elem = (A_B) it.next();
     if ( elem.getRole_b() == par ) {
          foundElem = elem;}}
if(foundElem ==null) return;
```

2
```
((A_B)foundElem).clean();
```

3
```
this.f_a_B.remove(foundElem);
```

1. Check existence
2. Build relationship from opposite side
3. Build relationship from this side (this part is not necessary!)

3.3. Cases where Octopus-generated code patterns are inadequate for EJB3

3.3.1. Naming convention for field

All fields declared in octopus generated codes are "private", moreover, no matter association defined field or attribute defined field, field name will be added by "f_" prefix. This design follows the concept of "information encapsulation", which is one of the fundamental tenets of software design (see Item 12 in [Bloch01]). But in our case, this design will bring trouble if developer use EJB3QL query, i.e. QL query will fail when the developer uses identifiers as they appear in the UML-model (e.g. field names without the "f_" prefix). Another possible trouble is, when EJB3 entity is "detached" (outside an entity manager instance) and used as backend Bean of JSF, the disagreement between field name and property methods name (accessor methods name) will result in failure.

3.3.2. Database Identity, object identity and object equality

The essential distinction for these three concepts is: (Section 3.4.1 [King05])

- *Object identity—Objects are identical if they occupy the same memory location in the JVM. This can be checked by using the == operator.*
- *Object equality—Objects are equal if they have the same value, as defined by the equals(Object o) method. Classes that don't explicitly override this method inherit the implementation defined by java.lang.Object, which compares object identity.*
- *Database identity—Objects stored in a relational database are identical if they represent the same row or, equivalently, share the same table and primary key value.*

In an ORM environment, it frequently happens that two objects may have different Object identity but same Database identity. Thus, each comparison of objects should be based on Object equality. To achieve this equals() and hashCode() must be overridden.

In Octopus generated codes, there is a method "getIdString()" used for building default identifier of the object:

1. If object contains a String type field, value of the first String type field will be set as identifier
2. If there is no String type field, but integer type field, value of the first integer type field will be set as identifier
3. If object does not contains any String type or integer type field, "no ID found" will be set as identifier.

This method is cannot be used when comparing objects because the generated identifier is not guaranteed to be unique.

3.3.3. Generic

In our database mapping strategy for cases like “one to many” or “many to many”, the collection is defined using generics to specify the element type, so that we do not need to use “targetEntity” option element in association mapping annotation. But, the ASTs of code generated by Octopus follow the metamodel of J2SE 1.4, with no provisions for “generic type”. It should be noticed that “generic” gives only compiler level protection, it can be mixed with old non-generic codes (if we do not offer a complete octopus extension for “generic” support). But, due to this type of mixture, a potential JVM level exception could arise at runtime.

3.3.4. Enumerated type

A J2SE 1.5 enumerated type is inherently supported by JSR220 specification, i.e. without mapping annotation it will be embedded into owner entity table directly. The Octopus generated enumeration class from <enum> symbol is actually a normal class which implements the type safe enum pattern (see Item 21 [Bloch01]) There will be two main problems by mapping this type of enumeration class into database:
1. In the type safe enum pattern, each constant must be “public static final”, that will prevent all constants (fields) being persisted in the database.
2. Since no public constructor is provided by this pattern, enumeration class can not be mapped as entity.

Consequently, it is necessary to change the code generation to output Java 5 enumerated types.

3.3.5. Collection type attribute defined field

An attribute-defined field can be type with one of the OCL supported collection types, in particular with a nested collection. That increases the complexity and difficulty for database mapping and manipulation codes design.

The quickest solution to mapping collection type fields consists in serializing (or annotating as @Lob i.e. Binary Large Object) the whole collection. However, in that case the attribute cannot participate in WHERE clauses in an EJB QL expression, as per the specification:

Note that state-fields that are mapped in serialized form or as lobs may not be portably used in conditional expressions (The implementation is not expected to perform such query operations involving such fields in memory rather than in the database.)([JSR220-Persistence] section 4.6)

3.4. Conclusion for Octopus code pattern analysis

3.4.1. Main problems and challenges

- Problem with association class

In previous chapter, for solving the removing problem by association class, we suggested to map both foreign keys to association ends. After analyzing the Octopus code pattern, we found this solution as underoptimal. The reason is, in association class constructor and clean() method, both foreign keys values are handled, this behavior will be not approved if they are primary keys at same time.

The application must not change the value of primary key. This includes not changing the value of a mutable type that is primary key or element of a composite primary key. (section 2.1.4 [JSR220-Persistence])

- Problem of rewriting equals() and hashCode()

In order to perform comparisons based on Object equality, one or more than one field values should be chosen for constructing a unique object identifier. Making decision is obviously troublesome in our model driven case. One possible approach is to use "id" field, but it still has a problem: Before an Entity being persistent, its primary key could be null or "0", i.e. the value of "id" field is not guaranteed to be unique all the time.

- Absence of support for various collection types

Our challenge is to give a mapping strategy for all OCL supported collection type (which could be nested), along with codes for each manipulation.

3.4.2. Possibility of fragments based code generation

In fragments based code generation, codes will be assembled from beforehand stored code fragments according to a rule table. Compares with Octopus code generation mechanism (template based), this code generation approach has the following advantages:

1. Avoid redundant repetition in code generation and give an intuitive overview of the logic based on rule table.
2. Transformation to other OO languages which have similar object relational mapping character will be easy.

A practical example for code generation is presented in Appendix B.

3.5. Summary

In this chapter, we discussed the limitation of Octopus UML. Since all imported UML model should be expressed in OctopusUML, these limitations will limit those UML models that can be processed by our transformation to EJB3. By following an analysis based on “code fragments”, we have reviewed the Octopus-generated code patterns, especially the code pattern for association generation. The knowledge of Octopus code patterns will be the foundation for the future discussion of the problems which are exposed in the next chapter.

Chapter 4. Discussion of Concrete Problems

In the previous chapter we exposed some problems. Among them, support for “Index association” and “Collection type/non collection type attribute defined field” appears to be difficult problems in our work. In this chapter, we will discuss them more closely. During the discussion, we will evaluation a candidate solution.

4.1. Problem of Collection type attribute defined field

Let’s consider a Java field that corresponds to a collection typed attribute in a UML class. The quickest way to ORM-map it consists in serializing it, by annotating it as @Lob i.e. as a Binary Large Object, thus serializing all items in the whole collection. However, in that case the attribute cannot participate in WHERE clauses in an EJB QL expression, as per the specification:

> *Note that state-fields that are mapped in serialized form or as lobs may not be portably used in conditional expressions (The implementation is not expected to perform such query operations involving such fields in memory rather than in the database.) (Section 4.6 [JSR220-Persistence])*

From a model definition, an attribute is likely to be a collection of entity type. (Normally, this is not a favourable model design, but we cannot avoid this possibility.) If the field is saved as Lob, the query cannot be used over the relationship any more. Moreover, an entity type has its own table to store its instances, if an already persisted entity instance is serialized into a Lob again, then an inconsistent problem will occur.

For convenience, we prescribe the following naming convention:

- Element type of a collection: the type of element in a collection, it could be a collection type again.
- Ultimate element type: if collection is not nested, element type is equal to ultimate element type. If collection is nested, ultimate element type is the type of the element of the innermost collection, e.g. the ultimate element type of the collection “Sequence(Bag(Set(String)))” is “String”. An ultimate element type is never a collection type.
- Root collection type: if collection is nested, root collection is the outermost collection. E.g. in “Sequence(Bag(Set(String)))”, “Sequence” is the root collection type.
- Leaf collection type: if collection is nested, leaf collection is the innermost collection. E.g. “Sequence(Bag(Set(String)))”, “Set” is the leaf collection type.
- Nested-level: indicates the depth of a nested collection. E.g. in “Sequence(Bag(Set(String)))” has nested-level 3. At the nested level 2, the collection type is “Set”.

4.1.1. In case of non-nested collection type

Element type of non-nested collection could be one of the following possibilities:

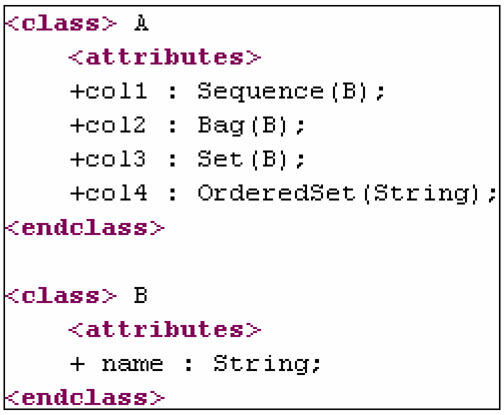
1. One of the OCL supported basic types, i.e. int, float, boolean and String
2. Instance of a classifier, i.e. a type defined through <class>, <associationclass> or <datatype>. It could be an entity or an embeddable component.
3. Instance of an enumerated type defined by <enum> symbol.

4.1.1.1.Thinking in database mapping

Attribute defined field is not like association defined field from <associations> which contains all the information around the defined association. From a non nested collection type attributes field, we only know the association should be “one to many” and the type of the opposite side of this association is the type of element in this collection. Thus, placing a @OneToMany annotation on the field declaration to build a database schema for unidirectional “one to many” is all we can do.

In any association mapping, both association ends should be Entity and each owns a table schema in database. But if the element is an Enumeration type or basic type which are supposed to be embeddable component and will not have a database table. One direct solution to this problem is to create an entity class to wrap the values of Enumeration or basic type instance. We call this wrapper class as “item class” (IC). One the other hand, a class defined from <class>, <associationclass> or <datatype> could be embeddable component or an entity. If the instance of this class is used as element of the collection, this class must be mapped as an entity.

Now let us consider the collection type of the field. Suppose that the collection type is indexed i.e. a sequence or an ordered set. Then the question should be where to save the index information of the collection. The most straightforward answer is to add an index attribute in the entity class, for example an additional field named “sequence”. From this approach an index column will be mapped into database table. For an element whose type is basic, enumeration or embeddable component, we can put this additional field in its IC, because it will not have a table to store it. But if the element type is an entity, the additional field will change the model definition and from the view of a model designer this change will not be acceptable. For example, a model designer may give such UML definition:

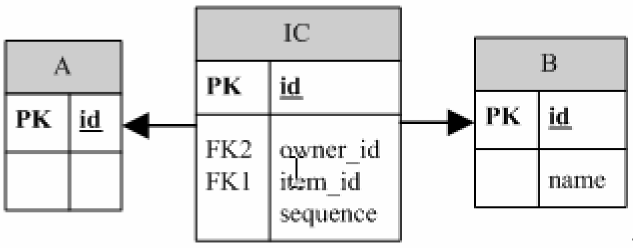


```
<class> A
    <attributes>
    +col1 : Sequence(B);
    +col2 : Bag(B);
    +col3 : Set(B);
    +col4 : OrderedSet(String);
<endclass>

<class> B
    <attributes>
    + name : String;
<endclass>
```

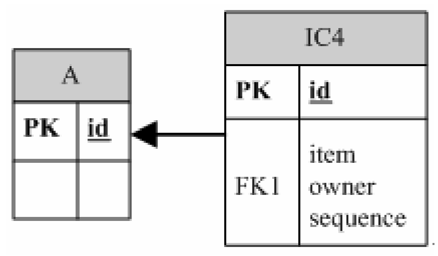
Although there are three collection type around B, in the database there is only one table for B. Designer has no responsibility to add a „sequence" field in class B definition just only for one field declaration „col1 : Sequence(B);". From this consideration, the index information should be saved out of table B. One possible solution to this problem is to generate an IC and put an „item" field in it. The IC works no longer as a wrapper, but as a class which builds the relationship to table B. Moreover, IC will contain index information for table B, if leaf collection type is indexed. In this approach the "one to many" association between A and B from col1 field declaration will be actually split into two associations, i.e. "one to many" between A and IC and "many to one" between IC and B.

Here we create many to one between IC and B because col1 field is a sequence i.e. B can be duplicated in collection. If it is ordered set, then the association between IC and B is definitely "one to one". The difference is, in unidirectional "one to one" foreign key has a unique constraint, i.e. the foreign key value in IC table cannot be duplicated. In other words, same instance of B cannot appear twice in collection of IC.

Another thing should be noticed is that unidirectional "one to many" between A and IC will result a join table to be created in database schema. This join table is totally unnecessary. In order to wipe off join table from database, we can add a field „owner" which points back to A and let the "one to many" be a bidirectional association. According to this mapping design for field col1, we have the following database schema:



Similarly, for col2 and col3 mapping, two IC s will be generated, for example IC2 and IC3. IC2 and IC3 will not contain „sequence" field, because both leave collection is not indexed. As for col4, we still use bidirectional "one to many" instead of unidirectional one to many to avoid unnecessary join table. Then the database schema after mapping will be:



Here, “item” is defined as a String type field in IC4 entity, its corresponding column „item“ in IC4 table should be physical database type CHAR. If the element type of col4 collection is an Enumeration, IC4 will contain an enumeration type „item“ and the corresponding column „item“ in table will be database physical type int. (by default ordinal of enumeration value will be saved.).

4.1.1.2.Thinking in codes pattern

So far we have discussed the mapping of a collection type attribute defined field in non nested collection case. Now is the time to think about the operation codes for this mapping approach. One thing is clear that the developer in business layer is not supposed to know the existence of IC and the database schema. He might operate the collection field like a normal collection, for example adding or removing an element. The problem is, all collection operations only react on collection instance in memory. Therefore, an overriding of these collection operations is necessary for us. But how to override them and where to put the override operation codes will be main obstacle to process. At first, overriden operations of a collection type means a new class which will extend this collection type and override necessary operations in its body. In our case, a collection type could be ArrayList or HashSet. Second, this new class should not be mapped into database table. That is because this class is supposed to only provide functionality extension. From these considerations, the solution is:

- One class will be generated for overriding corresponding operations.
- It should be mapped as embeddable component and extend ArrayList or HashSet according to the collection type of attribute defined field. We called this new class as „embeddable collection class“(ECC).

In ECC we put a collection type field which builds the bidirectional “one to many” association between ECC and IC. Because ECC is embeddable, the association mapping actually effects on A table and IC table. Furthermore, the element type of this collection is IC and IC contains the index information i.e. „sequence” field. Thus, if the collection instance of this field can hold the same order of IC table through column sequence, it will bring lots of convenience. To achieve this, we can use @OrderBy annotation for this field. When a collection of IC records are retrieved from IC table, @OrderBy sorts the collection and assure that the index number of each IC object in List will be the same as its „sequence“ value. Following is a mapping example:

```
@OneToMany(cascade=CascadeType.ALL,mappedBy="owner")
@OrderBy("sequence")
private List<IC> myItems = new ArrayList<IC>();
```

One obvious problem of this approach is the concurrence problem. Generally, a concurrence will occurs when two different ECC instance exit in memory and at same time both add operations are invoked to add a new IC in List. Because in add operation „sequence" of IC will be assigned, then the two new added IC record in IC table may have same value in „sequence" column. But in EJB3 environment, this type of concurrence appears under some given conditions. Normally, when a entity manager repeatedly retrieve a record in one transaction, only one entity instance of this record will be built in memory, meanwhile, its state in memory and its state in database will be synchronized by its owner entity manager. But if this entity instance is out of an entity manager, for example, it is transferred to presentation layer, and then its state in memory will not guaranteed to be synchronous with its state in database. This instance becomes a so-called detached object and may hold a state different from the database state. An entity manager can use „merge" operation to bring the actual state of a detached object back to the database. In this case, a "sequence" concurrence will occurs. In order to avoid „sequence" concurrence problem, developers of EJB3 system had best do not let an entity instance become in detached status.

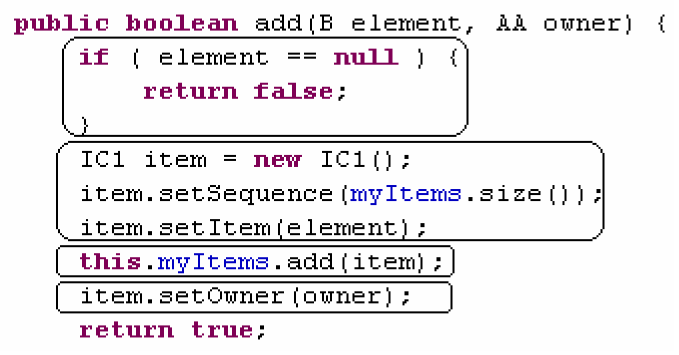
➢ Created methods in ECC

Now, let us observe the field "col1" in preceding example. Suppose that we want to add a "B" instance in to "col1". ECC1 is the embeddable component for "col1", the adding action is actually performed in it. Because the relationship between IC and ECC1 is a bidirectional, after the relationship from ECC to IC is set, an inverse relationship from IC to ECC must be also set. To achieve this, we just need to set "owner" field in IC class. But one problem is, ECC1 is an embeddable class which shares the database identity (primary key) of class A (table A) and thus can not be set in "owner". To solve this problem, we must transfer the owner instance of ECC1 to the IC class. Consequently, three new methods will be created in ECC1:

- add(element_type, owner_class_type),
- insertAt(index,element_type, owner_class_type)
- setAt(index, element_type, owner_class_type).

The owner instance (A class) will be transferred as parameter. Following table shows a code example for adding a new element in col1:

In ECC



```
public boolean add(B element, AA owner) {
    if ( element == null ) {
        return false;
    }
    IC1 item = new IC1();
    item.setSequence(myItems.size());
    item.setItem(element);
    this.myItems.add(item);
    item.setOwner(owner);
    return true;
```

Because this add method is no longer an override method, the body of add operation in owner class i.e. A class must be modified accordingly.

In A class
`public void addToCol1(B element) {` `this.f_col1.add(element,this);` `}`

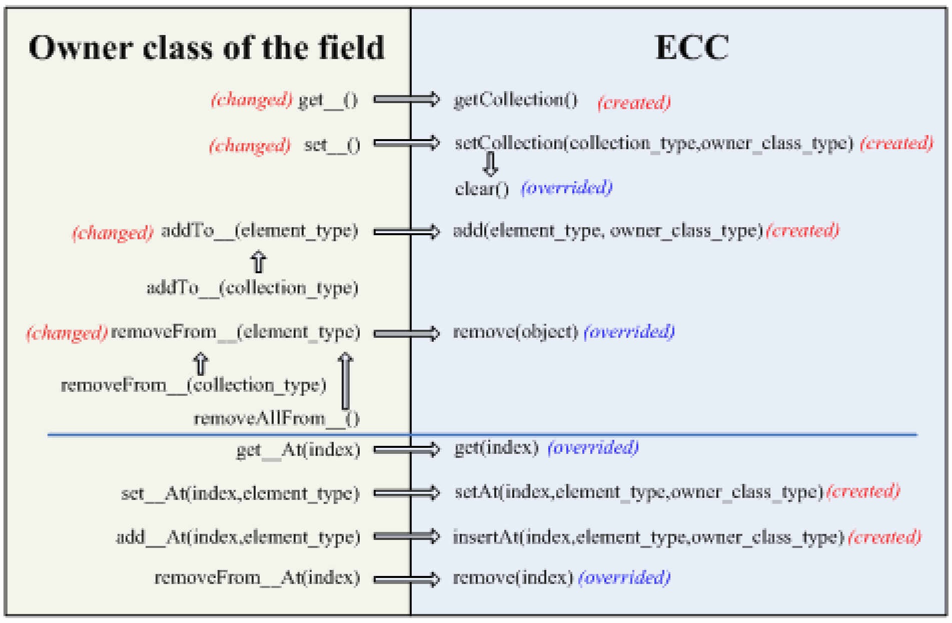
In add operation in ECC1, a new generated IC instance will handle the relationship between IC and B. After the owner instance is set into this IC “owner” field, the relationship between ECC1 and IC is completed.

Except the above motioned methods, there are two more methods should be created:

- getCollection()
- setCollection(collection_type)

The reason is straightforward. Suppose that, when getCol1() is invoked, the collection is supposed to be retrieved from database instead from memory. Furthermore, getCol1() should return B collection not IC collection from ECC. Similary, setCol1 should actually set collection of IC in ECC not B collection. Thus, we let getter and setter of col1 in A class invoke get and set operations in ECC which will handle the transformation between the collection of IC element and the collection of B element.

As a result, following diagram shows all possible methods in ECC and Owner class of the attribute defined field. Here, arrows indicate „invoke“. Some methods which need to be modified in owner class i.e. A class are marked with “(changed)”.



Owner class of the field
ECC
(changed) get__()
getCollection() (created)
(changed) set__()
setCollection(collection_type,owner_class_type) (created)
clear() (overrided)
(changed) addTo__(element_type)
add(element_type, owner_class_type) (created)
addTo__(collection_type)
(changed) removeFrom__(element_type)
remove(object) (overrided)
removeFrom__(collection_type)
removeAllFrom__()
get__At(index)
get(index) (overrided)
set__At(index,element_type)
setAt(index,element_type,owner_class_type) (created)
add__At(index,element_type)
insertAt(index,element_type,owner_class_type) (created)
removeFrom__At(index)
remove(index) (overrided)

One problem of this approach is, due to new created methods in ECC class, we can not use its super class or interface i.e. ArrayList/List or HashSet/Set to declare the field type in owner class, e.g. „private List f_col1= new ECC()“ or „private ArrayList f_col1= new ECC()“. The only doable declaration will be:

```
private ECC f_col1 = new ECC();
```

4.1.1.3. Conclusion

For a collection type attribute defined field (non nested collection case), two additional classes will be created i.e. ECC and IC. ECC will be annotated as an embeddable class and IC will be annotated as an entity. A bidirectional “one to many” will be build between ECC and IC. If the element type is an entity, we build a unidirectional “many to one” for Bag or Sequence case, a unidirectional “one to one” for Set or Ordered Set. If the element type is not an entity, IC table will be the table to store the element value.

For an indexed collection, an additional field “sequence” will be generated in IC class. Accordingly, in ECC class following operation will be added:

- Override methods get(index) and remove(index)
- New method insertAt(index,element_type, owner_class_type)
- New method setAt(index, element_type, owner_class_type).

In owner class, besides that we change the field declared type to “ECC”, there are still some methods need to be modified, they are:

- Accessor methods for the field
- addTo__(element_type)
- removeFrom__(elemnent_type)

A problem for this ECC,IC strategy is, when element type is an entity, IC will actually server like an “association class” (it is not real association class, because of a unidirectional relationship on one side). Then, we have the same problem as the one by association class, i.e. when we remove an element from the collection field, its correlative IC record will not be removed from IC table. Do not like association class, the solution “mapping both foreign keys as composite primary keys” make sense here, because the foreign keys values will not be changed through our codes.

4.1.2. In case of nested collection type

For a nested collection type, two things will be concerned:

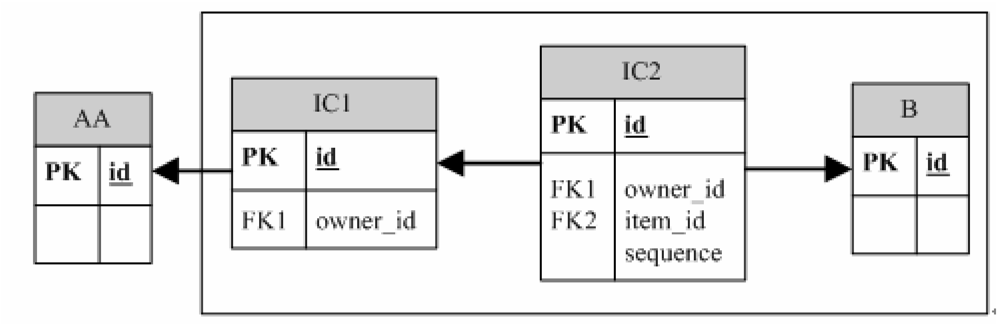
1. nested level of the nested collection
2. the collection type at each nested level

4.1.2.1. Thinking in database layer

At first let us take a look at this scenario:

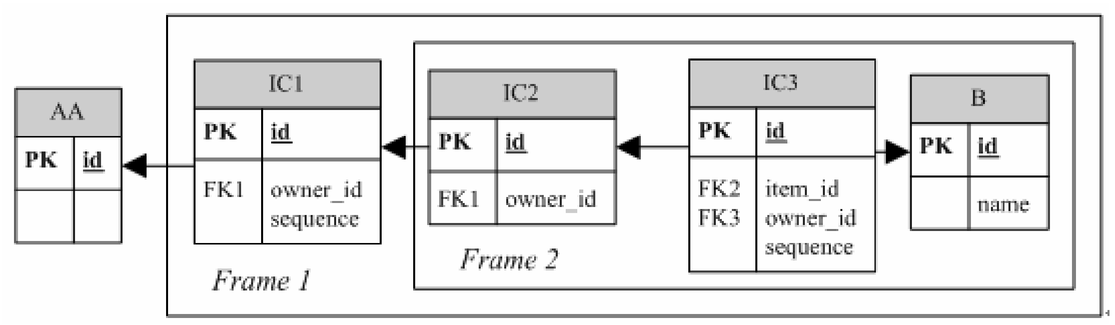
```
<class> AA
    <attributes>
    + col1 : Bag(Sequence(B));
    + col2 : OrderedSet(Bag(Sequence(B)));
<endclass>
```

Here field „col1“has a “one to many” association to collection object Sequence(B). If we treat IC as one item in this Bag collection then IC has a “one to many” association to class B again. A possible database schema will be like following diagram:



Here, IC1 contains only one foreign key to AA table and has a “one to many” relationship between AA and IC2. Obviously, this part of mapping is for Bag(...). IC2 table maintains the information of end element type. The sequence column in IC2 table stores the index information for Sequence. As for IC2, IC1 serves as „owner class“. To achieve this mapping, we still need two ECCs for AA and IC1 separately.

As we have seen, repeatedly using ECC,IC strategy can handle collection with arbitrary nested level. For example, in order to map “col2” of preceding example, ECC,IC strategy will be repeated for three times. The database schema will be:



Here, Frame2 corresponds to Bag(Sequence(B)) and Frame1 corresponds to OrderedSet(...).

4.1.2.2. Thinking in codes layer

In IC class, field “item” is created to wrap the ultimate element value, but for a nested collection, except the last IC class, the “item” field will be embedded with correlative ECC.

It would be best if we do not change anything of the codes of ECC and IC. But unfortunately, when we remove some thing from the collection type field, the old codes will trigger a problem.

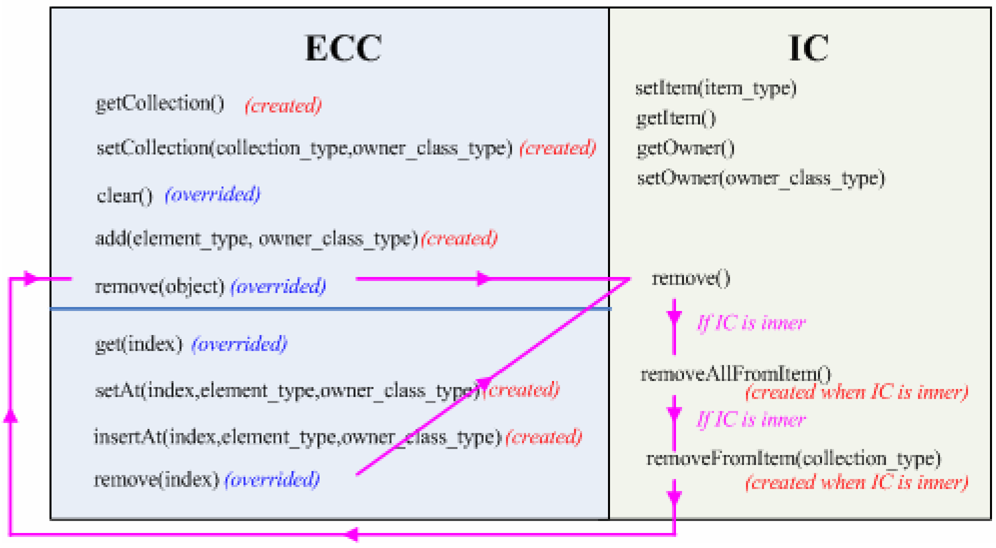
In ECC class, remove operation is used to delete a record in database. But in a nested collection case, remove operation will remove a collection object not a single “basic type”, entity, datatype or enumeration type. Here remove a collection means remove each element in the collection iteratively. This behaviour is totally different from „remove single element“.

To make things concrete, we observe the field “col2” in preceding example. Suppose that for this field following classes are created:

- ECC1 and IC1 for ordered set
- ECC2 and IC2 for bag
- ECC3 and IC3 for sequence

The action of remove operation in ECC1 and ECC2 should be different from the remove action in ECC3. If we put a remove operation in each IC, then the remove operation in ECC will invoke its IC’s remove method and no matter removed object is single instance or a collection type the codes in ECC will be the same. Now each IC’s remove method will perform the actual remove action on database.

Obviously, in IC1 and IC2, remove operation will delect collection and the remove action in IC3 will delete the relationship to a B record in database. Thus, by code generation we should distinguish „inner IC“ such as IC1, IC2 and „the end IC“ such as IC3.


ECC
getCollection() (created)
setCollection(collection_type,owner_class_type) (created)
clear() (overrided)
add(element_type, owner_class_type) (created)
remove(object) (overrided)
get(index) (overrided)
setAt(index,element_type,owner_class_type) (created)
insertAt(index,element_type,owner_class_type) (created)
remove(index) (overrided)
IC
setItem(item_type)
getItem()
getOwner()
setOwner(owner_class_type)
remove()
If IC is inner
removeAllFromItem()
(created when IC is inner)
If IC is inner
removeFromItem(collection_type)
(created when IC is inner)

Suppose that we remove an element from col2. First, remove() of ECC1 will be invoked, it will invoke the IC1 remove() afterwards. Because IC1 is an „inner IC“, it will call removeAllFromItem() and removeFromItem(collection_type) to remove a Bag collection. RemoveFromItem(collection_type) will remove each element from the Bag in iterative manner and each remove action calls remove() in ECC2. Again, ECC2 invokes remove() in next „inner IC“ IC2. The remove() in IC2 is responsible to remove each element in the Sequence collection. The remove action for each element is actually taken by ECC3’s remove method. Then ECC3 invokes IC3 remove(), because the element in Sequence collection is B class type and the collection reaches the end , thus,IC3 is „the end IC“. The whole remove action will stop at remove() in IC3.

In following table, we show the different between remove() in “inner IC” and remove() in “the end IC”.

In „inner IC“	In „the end IC“
`public void remove() { this.sequence=-1; this.owner= null; removeAllFromItem(); }`	`public void remove() { this.sequence=-1; this.owner= null; }`

4.2. Problem of non collection type attribute defined field

When attribute defined field is not collection type， the field type cold be

- One of the four “basic type” i.e. float, int, boolean or String
- an enumerated type or
- a class instance defined by <class> <associationclass> or <datatype>

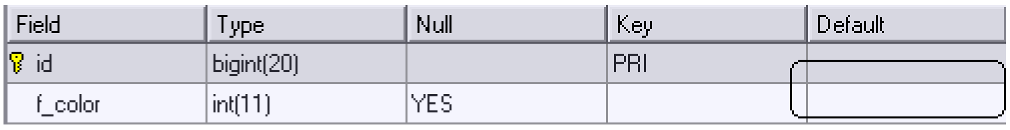
The four OCL supported “basic type” can be directly mapped into a column. Therefore, here we will only discuss the other two possibilities.

4.2.1. Field type is Enumeration type

Tiger style Enumeration is also supported by field type mapping in JSR220 specification. But codes from octopus standard enumeration generation are not supported. That is already discussed in previous chapter. Thus, we need transfer the old enum codes to a now one. Before beginning our discussion, let us first take a look at following scenario:

```
<package> pack
<class> aa
    <attributes>
        +color:Color;
<endclass>
<enumeration> Color
    <values>
        silver;
        gold;
<endenumeration>
<endpackage>
```

Here, Color is the Enum Class as a field which is used by Class aa. If Enum Class is written in Tiger style, it will be simply embedded into aa Table in database without any annotation. But even so, when we observe the table schema, we notice that there is no default value for column f_color,. Unluckily, @Column annotation defined in JSR220 does not provide the possibility for a default setup too.



Field	Type	Null	Key	Default
id	bigint(20)		PRI	
f_color	int(11)	YES		

In other hand, according to octopus code pattern, a default value of an enumeration should be assigned for the field.

```
private Color f_color = Color.lookup(0);
```

The benefits from this code is, when aa class is initiated and persistent in database, the default value of Enum Color will be also saved in table. From this point view, we need to create a lookup method to adapt this approach. Then the possible Enum Class in Tiger style will be:

```
public enum Color {
    DEFAULT,
    silver,
    gold;
    static public Color lookup(int index) {
        if(index == DEFAULT.ordinal())
            return DEFAULT;
        else if(index == silver.ordinal())
            return silver;
        else if(index == gold.ordinal())
            return gold;
        return DEFAULT;
    }
```

Furthermore, a tiger style Enumeration class as an embedded class will not has its own table, i.e. the comparison between two enumeration classes will never happen. Thus, there is no need to overwrite hashCode() and equals() for it.

4.2.2. Field type is a single object instance

In previous chapter “Thinking in database mapping”, we have motioned a decision problem for a single instance type attribute defined field. Concerning the performance, we should use @Embeddable whenever possible. But, in some special cases, a class type must be an entity:

1) Class type is defined as an association end
2) Class type is „ultimate element type“ of an attribute defined collection type field
3) Class type contains attribute defined collection type field
4) Class type has an self association
5) Class type is super class of an entity and locate at the top level of the hierarchy

Here is the explanation for each situation listed above:
1. The first point is straightforward, if the class is not an entity, association mapping can not be performed.
2. From the ECC, IC strategy discussion, we have realized that a class type (not an enum) should have its own table in database and must not be wrapped by IC class, or, a database inconsistent problem will occur.
3. In this case, as an “owner class”, class type must be an entity and own a table in database, or there is no place to embed with ECC.
4. Severing as the both association ends, the class type must be an entity which has a table in database.
5. As a super class at top level, it must provide a table for embedding its child class. This class type can be abstract.

According to the listed situation, we can draw the following conclusion:
Suppose owner class A has an attribute defined field „f“ which is a class type B. If B fulfils any one of the 5 listed situations, field „f“ will be mapped through @OneToOne i.e. an unidirectional one to one is build between A and B. If B does not encounter any situation in list, B will be mapped as embeddable component.

4.3. Problem of indexed collection used in an association

Association defined field is something different from attribute defined field. The field is generate according to association definition and the association information is more comprehensive, follow these information database mapping will be straightforward with association mapping annotation. Thus, our analytic point locates in codes layer only i.e. find out the way to change the code pattern from Octopus to support indexed collection type association end. Because the octopus code pattern is different for association class case and the case of association without association class, therefore, we will discuss them separately.

4.3.1. Association end is an Indexed Collection (without association class)

In <association> definition, collection type of an association end is specified through symbols <ordered> and <notUnique>.

Without <ordered> without <notUnique>	Set
With <ordered> without<notUnique>	Ordered Set
With <notUnique> without<ordered>	Bag
With both <ordered> and <notUnique>	Sequence

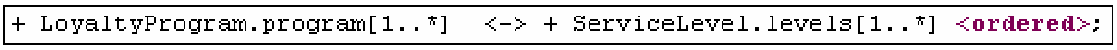
It should be notice that, in Octopus UML, <ordered> must be putted before <notUnique>. If applicable, <composite> and <aggregate> symbols must be placed at the last.

When an association is an indexed collection type i.e. sequence or ordered set, it must be decorated with <ordered> at least in <association> definition.

4.3.1.1. Make Index persist

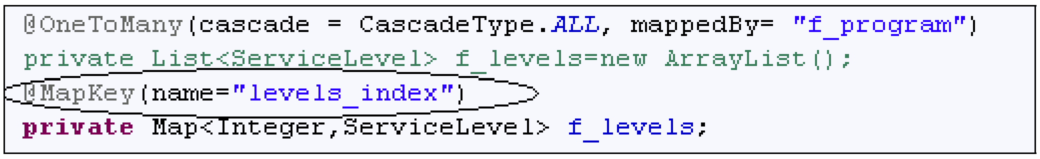
Java.util.Map as a collection interface is supported in JSR220. We can easily think of using map key to save the index. Now let us discuss this approach.

In example project RandL, we have following association:



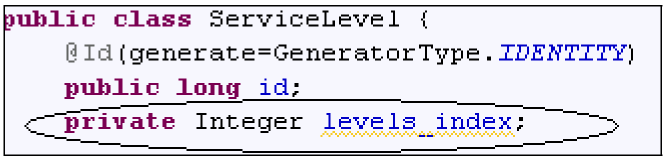
```
+ LoyaltyProgram.program[1..*]  <-> + ServiceLevel.levels[1..*] <ordered>;
```

Suppose we defined the role name levels in LoyaltyProgram as a Map.:



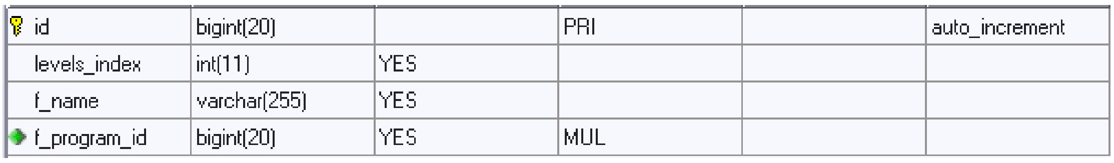
```
@OneToMany(cascade = CascadeType.ALL, mappedBy= "f_program")
private List<ServiceLevel> f_levels=new ArrayList();
@MapKey(name="levels_index")
private Map<Integer,ServiceLevel> f_levels;
```

@MapKey indicates which field in ServiceLevel is treated as the key for the Map. Here the key is levels_index which should be created in ServiceLevel class.



```
public class ServiceLevel {
    @Id(generate=GeneratorType.IDENTITY)
    public long id;
    private Integer levels_index;
```

Now, from above setup we will get following database schema for ServiceLevel table:



id	bigint(20)		PRI		auto_increment
levels_index	int(11)	YES			
f_name	varchar(255)	YES			
f_program_id	bigint(20)	YES	MUL		

We notice that if we do not use Map and just let levels_index as an additional persist field defined in ServiceLevel class.The result database schema is the same as above. Then, the mapping codes will be cleaner than which uses @MapKey:

```
@OneToMany(cascade = CascadeType.ALL, mappedBy= "f_program")
private List<ServiceLevel> f_levels=new ArrayList();
```

As conclusion, we only need to create a new field named "index" in the class of an indexed collection association end. In another side of association no additional mapping action need to be performed.

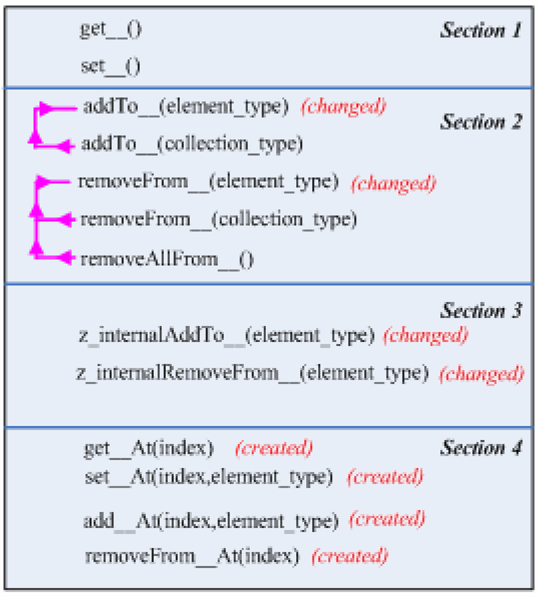
4.3.1.2. Involved Operations

First, let us take a look at the following uml definition:

```
<associations>
    + AA.a_role1[0..*] -> + BB.b_role1[1..*]<ordered>;
```

According to the conclusion from above section, an „index" field will be added in BB class. For each time by adding a new element in b_role1 collection in AA class, the value „index" should be also persistent. Similarly, when one element is remove from b_role1 collection, each „index" value of rest element in this collection must be also adjusted.

Thus, by handling index information of other side, three operations will be involved. They are set operation, add operation and remove operation. Now, let us take a look the following diagram and discuss for each section.


get__()
set__()
Section 1
addTo__(element_type) (changed)
addTo__(collection_type)
removeFrom__(element_type) (changed)
removeFrom__(collection_type)
removeAllFrom__()
Section 2
Section 3
z_internalAddTo__(element_type) (changed)
z_internalRemoveFrom__(element_type) (changed)
get__At(index) (created)
set__At(index,element_type) (created)
add__At(index,element_type) (created)
removeFrom__At(index) (created)
Section 4

In above diagram arrow indicates „invoke"and "__ "stands for association defined field.

Section 4:
In chapter "Thinking in code pattern" we have discovered that for each association field, Octopus generates three sections of methods at most. Each section of methods will be generated under special condition. Here, in the above diagram, we expanded it with a new section of methods i.e. "Section 4". It contains all methods for manipulation of indexed collection.

	Condition for generation
Section 1	If association defined field dose exist
Section 2	If association defined field is collection type
Section 3	If association is bidirectional
Section 4	If association defined field is indexed collection type

Here, we give the concrete description for each method:

- get__At(index): return an element at specified position
- set__At(index,element type): replace an element at specified position with a new one
- add__At(index,element type): insert a new element at specified position
- removeFrom__At(index): remove an element from specified position

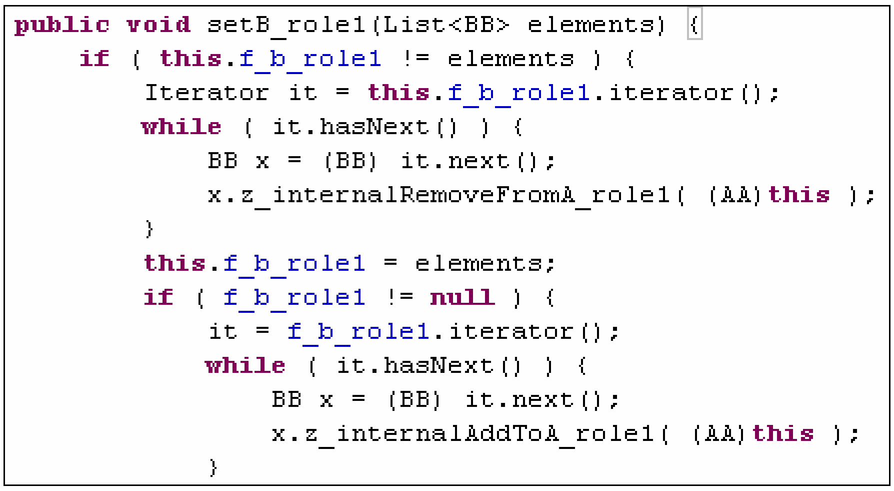
In following table we show the mechanism of code generation for these four methods.
In first column we listed all involved code fragments, and in the first row are the conditions for generation. Here, "uni" indicates that the opposite is unique (in UML {unique}) association. By contraries, "non-uni" means not. "one" indicates that this side has multiplicity "one" and "many" means this side has multiplicity "many". Moreover, "Y" indicates code will be generated, "N" indicates code will not be generated)

		Unidirectional		Bidirectional			
		Uni.	no_uni	Uni.		No_uni	
				one	many	one	many
get__At(index)	Check index bound	Y	Y	Y	Y	Y	Y
	return	Y	Y	Y	Y	Y	Y
set__At(index, element)	Check null Parameter	Y	Y	Y	Y	Y	Y
	Check index bound	Y	Y	Y	Y	Y	Y
	Check Duplication	Y	N	Y	Y	N	N
	Handle index information	Y	Y	Y	Y	Y	Y
	Clean relationship from opposite side	N	N	Y	N	Y	N
	Set	Y	Y	Y	Y	Y	Y
	Build relationship from opposite side	N	N	Y	Y	Y	Y
addTo__At(index, element)	Check null Parameter	Y	Y	Y	Y	Y	Y
	Check index bound	Y	Y	Y	Y	Y	Y
	Check Duplication	Y	N	Y	Y	N	N
	Handle Index information	Y	Y	Y	Y	Y	Y
	Clean relationship from opposite side	N	N	Y	N	Y	N
	Add	Y	Y	Y	Y	Y	Y
	Build relationship from opposite side	N	N	Y	Y	Y	Y
removeFrom __At(	Check index bound	Y	Y	Y	Y	Y	Y
	Handle Index information	Y	Y	Y	Y	Y	Y

index)	Clean relationship from opposite side	N	N	Y	Y	Y	Y
	Remove	Y	Y	Y	Y	Y	Y

Conclusion: all above listed methods should be created if association defined field is an indexed collection type. The code body for these methods will be constructed from different code fragments under different conditions.

Section1:
As an example let us first take look at the codes body of setB_role1() generated by standard octopus code pattern.



```
public void setB_role1(List<BB> elements) {
    if ( this.f_b_role1 != elements ) {
        Iterator it = this.f_b_role1.iterator();
        while ( it.hasNext() ) {
            BB x = (BB) it.next();
            x.z_internalRemoveFromA_role1( (AA)this );
        }
        this.f_b_role1 = elements;
        if ( f_b_role1 != null ) {
            it = f_b_role1.iterator();
            while ( it.hasNext() ) {
                BB x = (BB) it.next();
                x.z_internalAddToA_role1( (AA)this );
            }
```

In above codes, we see setB_role1() invokes inner add methods to add each element in collection. Because the parameter elements is a "List" type , its "iterator" will go over the elements in this collection in proper sequence.

Returns an iterator over the elements in this collection. There are no guarantees concerning the order in which the elements are returned (unless this collection is an instance of some class that provides a guarantee).(java.util.Collection.iterator()[J2SE1.5])

Thus, if „z_internalAddToA_role1()" method can save the index information for each element, then the collection will be persist in database also in correct order without any change on set__(). We do not need to worry the collection type of parameter could be non indexed collection. In fact when b_role1 is an indexed collection type i.e. sequence or orderedSet. The generated collection type for parameter must be a "List".

Conclusion: no method in Section 1 will be changed.

Setction2:
In this section, main add and remove operation will be performed. Although there are two add methods and three remove methods, the real work is actually done by addTo_(elememtType) and removeFrom__(element_type).i.e. other methods in this section invoke the two methods.

Conclusion: addTo_(elememtType) and removeFrom__(element_type) need to be modified with additional index handling behaviour.

Setion3:
In this section, there are only two methods „inner add“ and „inner remove“. Both methods need to be modified to handle index information.

4.3.2. Indexed Collection in association with association class

In Octopus UML, an association (with association class) must be bidirectional. According to the analysis result from chapter “Thinking in code pattern”, we have following table for code generation in case of an association with association class.

Section 1	Section 2	Section 3

get__()
set__()
get_&&_()
set_&&_()
	addTo__(element)
	addTo__(collection)
	removeFrom__(element)
	removeFrom__(collection)
	removeAllFrom__()
		z_internalAddTo_&&_()
		z_internalRemoveFrom_&&_()

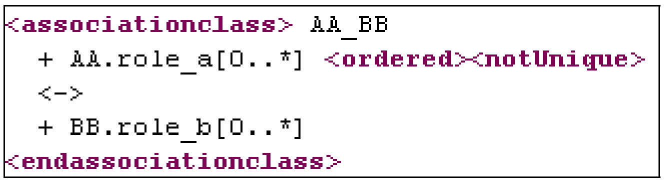
Here, “__” is the role name of opposite side, “_&&_” is the association class name.

If opposite side of the association is an indexed collection type, we still need to add one section methods to generation i.e. “Section 4” which has been discussed in last section.

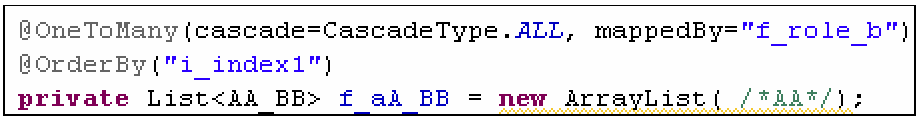
4.3.2.1. Make index information persist

In no association class case, index can be saved through additional “index” field in association end class. We can still follow this field approach, but the question is where to create it, i.e. in association end or in association class.

Consider that we have followed uml definition:


```
<associationclass> AA_BB
  + AA.role_a[0..*] <ordered><notUnique>
  <->
  + BB.role_b[0..*]
<endassociationclass>
```

AA side is a sequence and BB side is a Set. From chapter “Thinking in code pattern” we realize that the role name in this association will be declared as field in association class. In each association end class, association class type will be a field type. In fact, octopus still generate getter and setter of role name in one association end class despite that this role name is not declared in class. In above example, BB class will contain getRole_a() and setRole_a() methods. But role_b dose not exist as a field in AA class. The problem is the “index” field is supposed to be referenced by @OrderBy, however BB class does not have such field with AA type. Therefore, the only place for the “index” field is in association class AA_BB. Then, in BB class @OrderBy can be added on the field which is association class type.

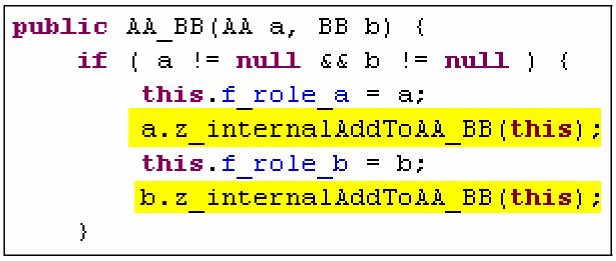


```
@OneToMany(cascade=CascadeType.ALL, mappedBy="f_role_b")
@OrderBy("i_index1")
private List<AA_BB> f_aA_BB = new ArrayList( /*AA*/);
```

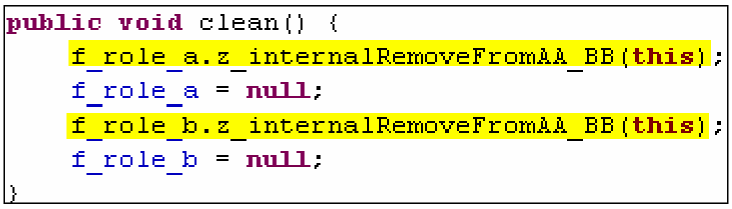
In above codes, “i_index1” field is declared in AA_BB class to hold the index information for role name “role_a”. In case that both association ends are indexed collection type, two “index” fields will be added in association class.

4.3.2.2. Involved operations

In association class there are only two association related operation. One is constructor of association class which built the relationship on both side. Another is clean() which destroy the relationship for the both side. These two methods all invoke the inner methods of each association end. See below:



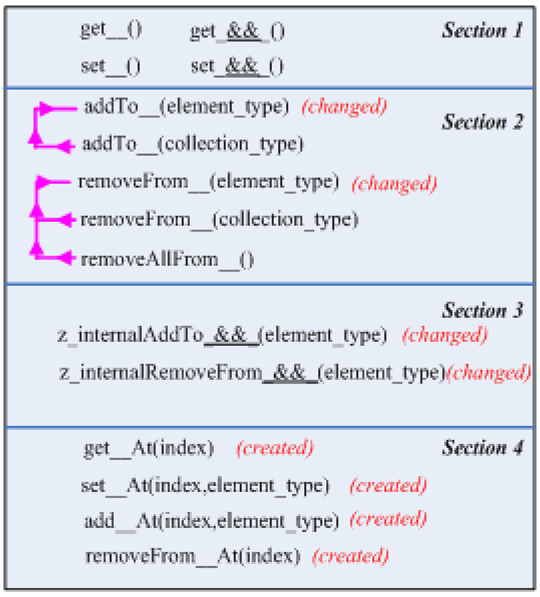
```
public AA_BB(AA a, BB b) {
    if ( a != null && b != null ) {
        this.f_role_a = a;
        a.z_internalAddToAA_BB(this);
        this.f_role_b = b;
        b.z_internalAddToAA_BB(this);
    }
```



```
public void clean() {
    f_role_a.z_internalRemoveFromAA_BB(this);
    f_role_a = null;
    f_role_b.z_internalRemoveFromAA_BB(this);
    f_role_b = null;
}
```

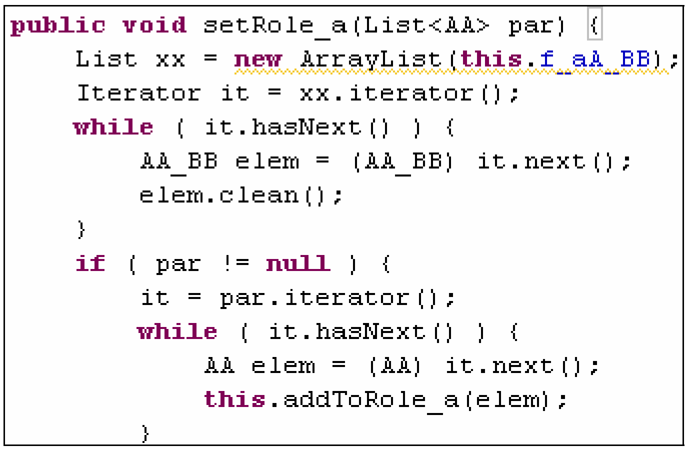
From above codes we know that if “inner methods” of an association end are already modified to handle index information, then we do no need to change the two methods in association class any more.

All the same, we begin with the “Section diagram” and find out which methods should be changed or created.


get__()
get_&&_()
Section 1
set__()
set_&&_()
addTo__(element_type) (changed)
Section 2
addTo__(collection_type)
removeFrom__(element_type) (changed)
removeFrom__(collection_type)
removeAllFrom__()
Section 3
z_internalAddTo_&&_(element_type) (changed)
z_internalRemoveFrom_&&_(element_type)(changed)
get__At(index) (created)
Section 4
set__At(index,element_type) (created)
add__At(index,element_type) (created)
removeFrom__At(index) (created)

Section1:

In this section, we do not take care of get_&&_() and set_&&_(), because a developer will not be aware of the existence of them. Here, we still observe the codes body of set__() in BB class



```
public void setRole_a(List<AA> par) {
    List xx = new ArrayList(this.f_aA_BB);
    Iterator it = xx.iterator();
    while ( it.hasNext() ) {
        AA_BB elem = (AA_BB) it.next();
        elem.clean();
    }
    if ( par != null ) {
        it = par.iterator();
        while ( it.hasNext() ) {
            AA elem = (AA) it.next();
            this.addToRole_a(elem);
        }
```

The set__() will invoke addTo__(element_type). Thus if addTo__(element_type) is changed to handle index information then this method do not need to be changed.

Conclusion: no method in Section 1 will be changed.

Section2:

The two methods addTo__(element_type) and removeFrom__(element_type) are still the core methods to perform the add and remove action. Here, “changing” the both methods does not mean to add „index handling“ codes, but implies to be changed for supporting unique and non unique collection type. (In standard Octopus code pattern, this is neglected) The reason why we do not need to add „index handling” codes in these two methods is: When association between AA and BB is build, a new AA_BB will be initialized. In constructor codes above we see, during initialization, the inner add methods of both side are invoked. In the same way, when association is destroyed, clean() method in AA_BB will be called, which will invoke inner remove method of both side. Therefore, if the „index handling“ codes is already added in the „inner methods“ of indexed collection side.

There is no need to add it again in add and remove methods.

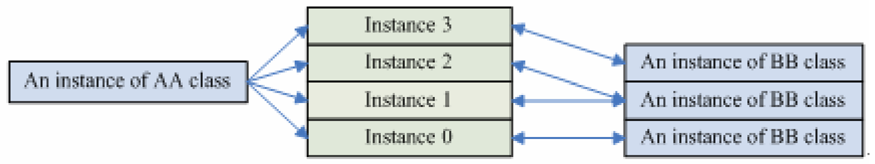
Section3:

In section3 we must notice that „_&&_“indicates association defined field i.e. association class name. In preceding example it is the field „aA_BB“, so the name for the inner methods are: „z_interneralAddToAA_BB()“ and „z_internalRemoveFromAA_BB()“.Codes for “handling index information” will be inserted into these two methods.

Section 4:

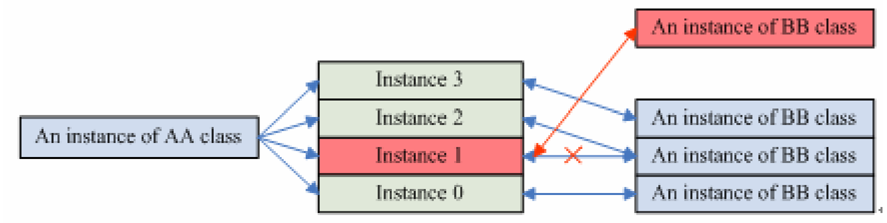
Similar to “association without association class” case, here, the four index collection related methods will be created. But one thing should be noticed that in association class case, all operation do not effect on the collection of association end but actually the collection of association class. This make inner logic of the methods more complex, especially, in set__At().

Suppose that we have a AA instance which holds a collection of association class instance:



When a new instance of BB class is going to set in position “1”of the collection, we need two steps to achieve it:

1. Destroy the old relationship between “instance 1” and “an instance of BB class”
2. And then build a new relationship to the new added instance of BB.



In following table we show the mechanism of code generation for these four methods.

In first column we listed all involved code fragments, and in the first row are the conditions for generation. Here, “uni” indicates that the opposite is unique (in UML {unique}) association. By contraries, “non-uni” means not. “one” indicates that this side has multiplicity “one” and “many” means this side has multiplicity “many”. Moreover, “Y” indicates code will be generated, “N” indicates code will not be generated)

		Unidirectional		Bidirectional			
		Uni.	no_uni	Uni.		No_uni	
				one	many	one	many
get__At(index)	Check index bound	Y	Y	Y	Y	Y	Y
	return	Y	Y	Y	Y	Y	Y
set__At(index, element)	Check null Parameter	Y	Y	Y	Y	Y	Y
	Check index bound	Y	Y	Y	Y	Y	Y
	Check Duplication	Y	N	Y	Y	N	N
	Clean relationship from opposite side	N	N	Y	N	Y	N
	Set	Y	Y	Y	Y	Y	Y
addTo__At(index, element)	Check null Parameter	Y	Y	Y	Y	Y	Y
	Check index bound	Y	Y	Y	Y	Y	Y
	Check Duplication	Y	N	Y	Y	N	N
	Clean relationship from opposite side	N	N	Y	N	Y	N
	Add	Y	Y	Y	Y	Y	Y
removeFrom __At(index)	Check index bound	Y	Y	Y	Y	Y	Y
	Remove	Y	Y	Y	Y	Y	Y

4.4. Summary

Along with the analysis for “indexed collection type attribute defined field”, we educed ECC,IC strategy as a possible solution. In this strategy, mapped database schema will be extended to handle different type of collection. Moreover, we can use this strategy repeatedly for a nested collection case. In the process of discussing "support of indexed collection in an association", we by analysing the original code pattern, noticed that, some codes need to be modified, and also we settled the logic constitution to support the indexed collection.

Chapter 5. Recipe for MDA driven EJB3 persistence artifacts

In chapter 3 and chapter 4, several potential problems were discussed under the categories of *Database Mapping* and *Code Pattern*. In particular, two difficult problems (“Index association” and “Collection type attribute defined field”) were analyzed in detail in Chapter 5. Building upon those explanations, this chapter brings together in a concise manner all previous solutions and strategies. Along with some considerations about Model, a generic recipe for the generation of MDA driven EJB3 persistence artifacts is also offered in full here.

5.1. Extension of Octopus Java Model

The main purpose of extending the Octopus Java model (OJpackage Meta Model) is to make Octopus capable of representing Java 5 annotations and enumerations. Since OJpackage itself is a simplified Java model based on Java 1.4, the extension will focus only on the required functionality and not in fully completing the Meta Model.

5.1.1. Extension for Annotation (OJAnnotation)

The possible Target types of Metadata Annotations for ORM mapping are TYPE, FIELD and METHOD, as specified in JSR220. Consequently, at the metamodel level, an Annotation should also relate to these Target types only. In the OJpackage Meta Model, the closest common super type to TYPE, FIELD and METHOD is OJVisibleElement. Since each Target type can be decorated with more than one Annotations type, the relationship between OJVisibleElement is “one to many”. Moreover, our Annotation shouldn’t be applicable to subtypes of OJElement other than those for TYPE, FIELD and METHOD.

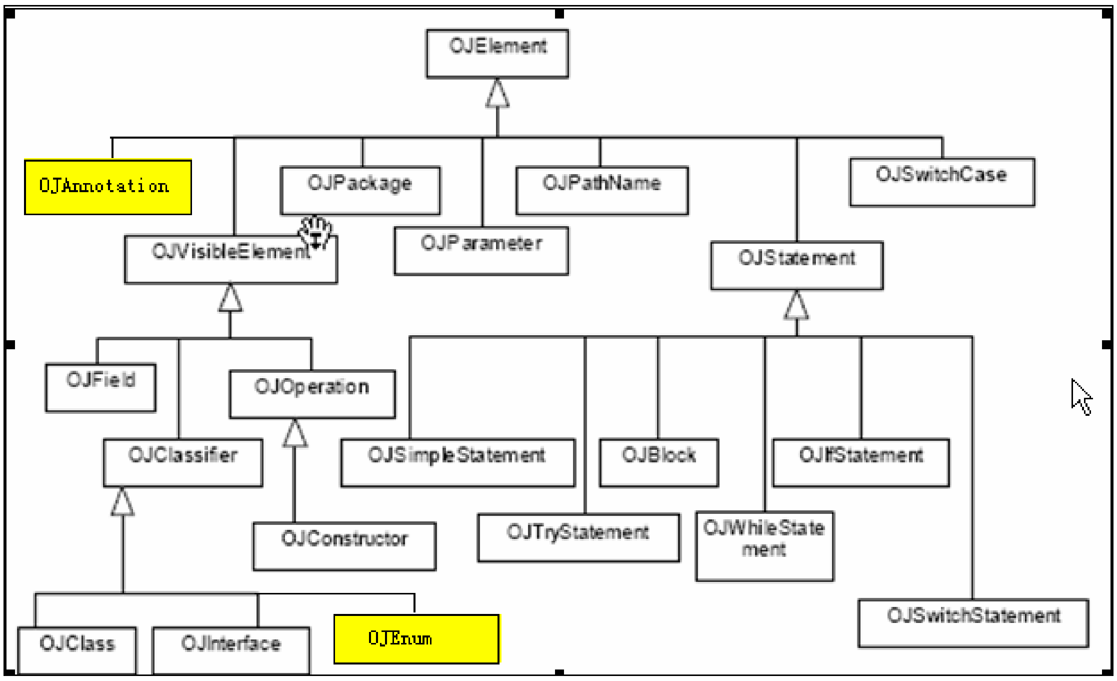
The contents between parentheses of a mapping Annotation may vary in structure. These contents may include several optional elements and even Annotations again. As a matter of convenience, we declare the content to be simply of string type. It will be directly appended to the name of the Annotation when serialized in code, e.g. “@”+annotation_name+”(“+content+”)”. Consequently, in our new Model Element for Annotation a.k.a OJAnnotation there will be only two attributes, the name of the Annotation and the content string.

Octopus is capable to determine the Type Path for "import" part. For Annotation, Octopus can easily conclude the Type Path for each mapping Annotation through the Annotation name, e.g. the Type Path for "@Entity" is "javax.persistence.Entity". As a disadvantage, imports required because of showing up in the "contents" of an annotation (an uninterpretable string) won't be able to be derived automatically with this design.

5.1.2. Extension for Enumeration (OJEnum)

Because all enumerated types are classes, its Model Element (OJEnum) can straightforwardly extend OJClassifier. Although the Enumeration type is a generalization of Class, we still recommend placing OJEnum at the same level as OJClass taking into account the code generation mechanism in Octopus: the generation process for Interface, Class and Enum are performed separately. In other words, Octopus uses a dedicated code creator in the shape of class EnumCreator for generating the code for enumerated types.

The OJPackage Meta Model after extension is shown in the following figure:



Extended OJPackage Meta Model

5.2. Qualifications for imported EJB3 persistence Model

Our recipe for EJB3 persistence artifacts does not aim at arbitrary UML Class Diagram as an EJB3 persistence Model. Some preconditions imposed on an imported Class Diagram must be met before code generation because of the limited descriptive capabilities of OctopusUML. Additionally, some limitations of the EJB3 Persistence standard as of JSR220 also stand in the way of directly mapping some UML class diagram constructs. A best effort has been made at anticipating the restrictions on the possible UML models for EJB3 Persistence, however more practical case studies would improve our confidence on the completeness of such restrictions. The restrictions identified so far follow.

A UML Class Model must fulfill all the listed qualifications in order to be mapped to an EJB3 persistence Model:

- ➢ should not contain interface or template definition
- ➢ should not contain constraints in associations or in Polymorph.
- ➢ Class can not be final
- ➢ Enumeration can not be defined as nested Class
- ➢ Primitive type must be one of string, int, float(real) or boolean type.
- ➢ Each declared Class, Enum or Datatype cannot be used outside its owner package.
- ➢ No UML Class should contain inner class.
- ➢ Stereotype and attribute properties are not supported
- ➢ Association qualifier is not supported (only Role name will be considered)

5.3. Naming Algorithm for „Name collision“ problem

Unless special provisions are made, name collisions might occur very often during code generation. For example, if both association ends of an association class are {ordered} then two additional “index” fields will be created in the association class. The name of both fields could collide with each other or with an existent field in association class. The following algorithm (Naming Algorithm) provides a way to generate unique name strings.

- ➢ i=1:
- ➢ name =„Name“
- ➢ Check if name already exist
- ➢ If true, change name to „Name“+i (e.g. „Name1“) and i++, goto step 3)
- ➢ if false, end.

At the beginning, an index “i” is initiated with value 1 and a variable “name” is evaluated by a string “Name”. The value of “name” will be compared with other string names that could be field names or class names etc. If comparison returns a positive result, variable “name” and index “i” will be combined to make a new value for variable “name”. Afterwards, index will be increased. The check step will be repeated until no more collision appears.

5.4. Strategy for Class Mapping

5.4.1. Recipe of Mapping Annotation

Situation 1: When class is defined by <class> or <datatype>
(Annotation in left column will be added when any of the conditions listed in right column is met.)

Annotation	Conditions
@Entity(access=AccessType.FIELD)	[1] If class is defined by <abstract><class> [2] If class is not a sub class but is the super class of other classes. [3] If class is defined as an association end [4] If class has a self association [5] If class contains attribute defined collection type field [6] If class is „end element type" of an attribute defined collection type field
@Entity	[7] If class is a sub class
@Embeddable	[8] If class does not meet any one of above 1-7 conditions and it is used as an inlined attribute type(not collection) in another class

Situation 2: When class is defined by<Enum>
(No mapping annotation needs be added)

Annotation	Conditions
	[9] If class is enum

Situation 3: When class is defined by <associationclass>
(Association class must be an Entity)

Annotation	Conditions
@Entity(access=AccessType.FIELD)	[10] if class is association class

5.4.2. Recipe of generated Code patterns

Situation 1: When class is annotated with @Entity(access=AccessType.FIELD)

- Put “implements Serializable” after class declaration
- Leave out the Octopus-generated prefix “f_” on each field name in class
- Add a long type field “id” for primary key in class
- If “name collision” for field “id” occurs, apply the “Naming Algorithm”
- Put Annotation @Id(generate=GeneratorType.IDENTITY) on the new field “id”.
- If class is an association class, create a no-arg constructor in it.
- If class is an association class, create setter methods for both reference fields
- If another class with same name exist in other package, put Table(name=“table-name“) after @Entity(access=AccessType.FIELD) to give a specified name of mapped table. “Table-name” is generated by “naming algorithm”.

Situation 2: When class is annotated with @Embeddable

- If two or more fields with a same name are declared in different embedded classes, each field with the same name will be annotated with @Column(name=”column-name”). “Column-name” is generated by “Naming algorithm”.

Situation 3: When class is an enumerated Type

- Create an enum Class with “Public” visibility
- If there is no field defined with name “Default” (case insensitive), a new field named „DEFAULT“ will be created in enum Class as the first value of this enum.
- Create a static method named “lookup” with “Public” visibility in order to return a value of this enum based on given ordinal.

5.5. Strategy for Association mapping (without association class)

Scenario: When an Association exists between A and B, A and B may be equal (self association)

5.5.1. Recipe for Mapping Annotation

Mapping annotation will be placed above association defined field on the relevant association end. The process of annotation assignment can be split into five steps. (All “Field” presented in table means association defined field i.e. reference field)

Step 1: Determining owner side and inverse side of association

When association is bidirectional (elements in the first row are conditions for determining)

	One to one	One to many	Many to one	Many to many
Owner side	Arbitrary side	Many side	Many side	Arbitrary side

When association is unidirectional (elements in the first row are conditions for determining)

	One to one	One to many	Many to one	Many to many
Owner side	Unnavigable side	Unnavigable side	Unnavigable side	Unnavigable side

Step 2: Determining Mapping Annotations for association defined field

When association is bidirectional

(Elements in the first column are conditions for determining)

(Elements in the first row are annotation targets)

(„xxx“ stands for the name of association defined field in owner side)

	Field in Owner side	Field in Inverse side
One to one	@OneToOne	@OneToOne(mappedBy=xxx)
One to many/Many to one	@ManyToOne	@OneToMany(mappedBy=xxx)
Many to many	@ManyToMany	@ManyToMany(mappedBy=xxx)

When association is unidirectional

(Elements in the first row are conditions for determining)

(Element in the first column is annotation target)

	One to one	One to many	Many to one	Many to many
Field in owner side	@OneToOne	@OneToMany	@ManyToOne	@ManyToMany

Step 3: Determining Mapping Annotations for indexed association defined field

Irrespective of whether the association is unidirectional or bidirectional:

(Elements in the first row are conditions for determining)

(Element in the first column is annotation target)

	The opposite side is indexed many	otherwise
Field in association end (this side)	@OrderBy(xxx)	--

„xxx“ stands for index field name of the opposite side.

Step 4: Determining „CascadeType“ option element for association annotation

Irrespective of whether the association is unidirectional or bidirectional:

(Elements in the first row are conditions for determining)

(Element in the first column is option element owner (annotation))

	When this side has multiplicity one and get <composite> decorated	otherwise
Association Annotation for the field in association end (this side)	CascadeType.persist CascadeType.remove	CascadeType.persist

Step5: Determining „optional“ option element for association annotation
When association is bidirectional
(Elements in the first row are conditions for determining)
(Element in the first column is option element owner (annotation))

	Lower cardinality of the opposite side >= 1	Otherwise
Association Annotation for the field in association end (this side)	optional=false	--

When association is unidirectional
(Elements in the first row are conditions for determining)
(Element in the first column is option element owner (annotation))

	Lower cardinality of navigable side >=1	Otherwise
Association Annotation for the field in unnavigable side	option=false	--

5.5.2. Recipe for Code Pattern

Since the original Octopus codes pattern for non-indexed association is already adapted to our requirements, we only give the code pattern recipe for the indexed (in UML, {ordered}) association case.

Suppose we have an association between A and B, B is navigable and its collection type is indexed.
The following modifications will be carried out on class B:

- Create an index field “index” on Class B
- If “name collision” for field “index” occurs, “naming algorithm” will be performed to generate another unique field name.
- Create normal getter and setter methods for field “index”

The following modifications will be carried out on class A:
(“__” stands for the name of association defined field which refers to B)

- Modify addTo__(B element)
- Modify removeFrom__()
- Create get__At(int index)
- Create set__At(B element, int index)
- Create addTo__At(B element, int index)
- Create removeFrom__At(int index)

Keep on, if association is bidirectional
- Modify z_internalAddTo__(B element)
- Modify z_internalRemoveFrom__(Be element)

Now let us take look at the modification detail for each method: (Suppose that "__" is "Role_b", the name of index field for Class B is "i_index1"

addTo__(B element)

1	`if ( element == null )return;`
2	`if(this.role_b.contains(element)) return;`

3	`if ( element.getRole_a() != null ) {`
	`    element.getRole_a().z_internalRemoveFromRole_b(element);}`

4	`this.role_b.add(element);`
5	`element.i_index1 = this.role_b.size()-1;`
6	`element.z_internalAddToRole_a((A)this);`

1. Check null parameter
2. Check duplication (generated iff the opposite side is unique (in UML {unique}) association)
3. Clean relationship from opposite side (generated iff the multiplicity of this side is one and association is bidirectional)
4. Add
5. Handle Index information (be generated iff the opposite side is indexed collection type)
6. Build relationship from opposite side (be generated iff the association is bidirectional)

removeFrom__(B element)

1	`if ( element == null )return;`
2	`if(!this.role_b.contains(element) ) {return;}`

3	`Iterator ii =role_b.subList(index+1,role_b.size()).iterator();`
	`while ( ii.hasNext() ) {B item = (B) ii.next(); item.i_index1--;}`
	`beRemoved.i_index1=-1;`

4	`element.z_internalRemoveFromRole_a((A)this);`
5	`role_b.remove(element);`

1. Check null parameter
2. Check existence
3. Handle index information (be generated iff the opposite side is indexed collection type)
4. Clean relationship from opposite side (be generated iff the association is bidirectional)
5. Remove

get__At(int index)

1	`if ( index >= role_b.size() || index < 0 ) return null;`
2	`return role_b.get(index);`

1. Check index bound
2. Return

set__At(B element, int index)

1	`if ( element == null )return;`
2	`if ( index >= role_b.size() | index < 0) return;`

3	`if ( role_b.contains(element) && role_b.indexOf(element) != index ) {`
	`    removeFromRole_b(element);}`

4	`element.i_index1 = index;`
	`getRole_bAt(index).i_index1=-1;`

5	`if ( element.getRole_a() != null ) {`
	`    element.getRole_a().z_internalRemoveFromRole_b(element);}`

6	`role_B.set(index, element);`
7	`element.z_internalAddToRole_a((A)this);`

1. Check null parameter
2. Check index bound
3. Check duplication (be generated iff the opposite side is unique (in UML {unique}) association)
4. Handle index information (be generated iff the opposite side is indexed collection type)
5. Clean relationship from opposite side (be generated iff the multiplicity of this side is one and the association is bidirectional)
6. Set
7. Build relationship from opposite side (be generated iff the association is bidirectional)

addTo__At()(B element , int index)

1	`if ( element == null )return;`
2	`if ( index >= role_b.size() | index < 0) return;`

3	`if ( role_b.contains(element) && role_b.indexOf(element) != index ) {`
	`    removeFromRole_b(element);}`

4	`Iterator ii =role_b.subList(index,role_b.size()).iterator();`
	`while ( ii.hasNext() ) {`
	`    B item = (B) ii.next();`
	`    item.i_index1++;}`
	`element.i_index1 = index;`

5	`if ( element.getRole_a() != null ) {`
	`    element.getRole_a().z_internalRemoveFromRole_b(element);}`

6	`role_b.add(index,element);`
7	`element.z_internalAddToRole_a((A)this);`

1. Check null parameter
2. Check index bound
3. Check duplication (be generated iff the opposite side is unique (in UML {unique}) association)
4. Handle index information (be generated iff the opposite side is indexed collection type)
5. Clean relationship from opposite side (be generated iff the multiplicity of this side is one and the association is bidirectional)
6. Add
7. Build relationship from opposite side (be generated iff the association is bidirectional)

removeFrom__At(int index)

1
```
if ( index >= role_b.size() | index < 0) return;
```
2
```
if(role_b.get(index) != null){
Iterator ii =role_b.subList(index+1,role_b.size()).iterator();
while ( ii.hasNext() ) {B item = (B) ii.next(); item.i_index1--;}
role_b.get(index).i_index1=-1;}
```
3
```
if(role_b.get(index) != null){
    role_b.get(index).z_internalRemoveFromRole_a((A)this);}
```
4
```
role_b.remove(index);
```

1. Check index bound
2. Handle index information (be generated iff the opposite side is indexed collection type)
3. Clean relationship from opposite side (be generated iff the association is bidirectional)
4. Remove

z_internalAddTo__(B element)

1
```
this.role_b.add(element);
```
2
```
element.i_index1 = this.role_b.size()-1;
```

1. Add
2. Handle index information (be generated iff the opposite side is indexed collection type)

z_internalRemoveFrom__()

1
```
int index = this.role_b.indexOf(element);
Iterator ii =role_b.subList(index+1,role_b.size()).iterator();
while ( ii.hasNext() ) {
    B item = (B) ii.next(); item.i_index1--;}
element.i_index1=-1;
```
2
```
role_b.remove(index);
```

1. Handle index information (be generated iff the opposite side is indexed collection type)
2. Remove

5.6. Strategy for Association Mapping (with association class)

Scenario: When A and B are two association ends, A_B is their association class.

5.6.1. Recipe for Mapping Annotation

Mapping annotations will be placed on association defined fields on each end and association class. The process of annotation assignment can be split into five steps. (All “Field” present in table means association defined field i.e. reference field)

Step 1: Determining owner side and inverse side

In case of „one to one“ association, we create two bidirectional „one to one“ between each association end and the association class. In both associations the association class will be owner side and both ends are inverse.

In case of „many to many“ association, we create two bidirectional „one to many“ between each association end and the association class. (Both association ends are assigned as “one” side in new associations.) The association class is still the owner and both ends are inverse.

In case of “one to many” association, we create two bidirectional associations:
(assume that A is “one” side and B is “many” side)
1. A “one to many” association between A and A_B, A is assigned as “one” side in the new association, and A_B is the owner.
2. A “one to one” association between B and A_B, A_B is owner.

Step 2: Determining mapping annotation for association defined field in each end

When association is „one to one“ or „many to many“
(Elements in the first column are conditions for determining)
(Elements in the first row are annotation targets)
(„xxx“ stands for the name of association defined field in association side)

	Field in each associaion end	Field in associaion class
One to one	@OneToOne(mappedBy=”xxx”)	@OneToOne
Many to many	@OneToMany(mappedBy=”xxx”)	@ManyToOne

When association is „one to many“/“many to one“
(Element in the first row is condition for determining)
(Elements in the first column are annotation targets)
(„xxx“ stands for the name of association defined field in association side)

	One to Many or Many to One
Field in association end with “one” multiplicity	@OneToMany(mappedBy=”xxx”)
Field in association end with “many” multiplicity	@OneToOne(mappedBy=”xxx”)

(Element in the first column are conditions for determining)
(Elements in the first row is annotation target)

	Field in association class
If “many side” is a Set or Ordered Set	@OneToOne
If “many side” is a Bag or Sequence	@ManyToOne

Step 3: Derterming mapping annotation for indexed association defiend field
(Elements in the first row are conditions for determining)
(Element in the first column is annotation target)
(„xxx“ stands for index field name in association class)

	the opposite side is not indexed many	the opposite side is indexed many
Field in association end	--	@OrderBy(xxx)

Step 4: Determining „CascadeType“ option element for association annotation
Irrespective of whether the association is unidirectional or bidirectional:
(Elements in the first row are conditions for determining)
(Element in the first column is the option element owner (annotation))

	When this side has multiplicity one and get <composite> decorated	otherwise
Association Annotation for the field in association end (this side)	CascadeType.persist CascadeType.remove	CascadeType.persist

Step 5: Determining „optional“ option element for association annotation
(Elements in the first row are conditions for determining)
(Element in the first column is the option element owner (annotation))

	Lower cardinality of opposite side >= 1	Otherwise
Association Annotation for the field in association end	optional=false	--

5.6.2. Recipe for Code Pattern

Like previous section about code pattern for association without association class, here only the code pattern recipe for indexed (in UML {ordered}) association case will be reviewed.

Suppose we have an association between A and B, A_B is the association class. The collection type of B is indexed.

The following modifications will be carried out on class A_ B:

- Create a index field “index” on Class A_B
- If “name collision” for field “index” occurs, “naming algorithm” will be performed to generate another unique field name.
- Create normal getter and setter methods for field “index”

The following modifications will be carried out on class A:

(“__” stands for the name of association defined field which refers to B)

(“_&&_” stands for the name of association defined field which refers to A_B, this name is equal to the class name of association class i.e. “A_B”)

- Modify addTo__(B element)
- Modify removeFrom__()
- Create get__At(int index)
- Create set__At(B element, int index)
- Create addTo__At(B element, int index)
- Create removeFrom__At(int index)
- Modify z_internalAddTo_&&_(B element)
- Modify z_internalRemoveFrom_&&_(Be element)

Now let us take look at the modification detail for each method: (Suppose that “__” is “Role_b”, the name of index field for Class B is “i_index1”

addTo__(B element)

```
1  if ( element == null )return;
2  boolean isPresent = false;
   Iterator it = a_B.iterator();
   while ( it.hasNext() & !isPresent ) {
       A_B elem = (A_B)it.next();
       if ( elem.getRole_b().equals(element)){isPresent = true;}}
   if ( !isPresent ) return;
3  if ( element.getA_B() != null ) {((A_B)element.getA_B()).clean();}
4  new A_B(this,element);
```

1. Check null parameter
2. Check duplication (be generated iff the opposite side is unique (in UML {unique}) association)
3. Clean relationship from opposite side (be generated iff the multiplicity of this side is one and the association is bidirectional)
4. Add

removeFrom__(B element)

```
1  if ( element == null )return;
2  A_B foundElem = null;
   Iterator it = a_B.iterator();
   while ( it.hasNext() ) {
       A_B elem = (A_B)it.next();
       if ( elem.getRole_b().equals(element)) {foundElem = elem;}}
   if ( foundElem != null ) return;
3  ((A_B)foundElem).clean();
```

1. Check null parameter
2. Check existence
3. Remove

get__At(int index)

1	`if ( index >= role_b.size() || index < 0 ) return null;`
2	`return role_b.get(index);`

1. Check index bound
2. Return

set__At(B element, int index)

1	`if ( element == null )return;`
2	`if ( index >= role_b.size() | index < 0) return;`

3	`boolean isPresent = false;`
	`A_B where = null;Iterator it = a_B.iterator();`
	`while ( it.hasNext() && !isPresent ) {`
	`    A_B elem = (A_B) it.next();`
	`    if ( elem.getRole_b().equals(element)) {`
	`        isPresent = true;where = elem;}}`
	`if ( isPresent & a_B.indexOf(where) != index ) {`
	`    removeFromRole_b(element);}`

4	`if ( element.getA_B() != null ) {((A_B)element.getA_B()).clean();}`

5	`A_B slot = a_B.get(index);`
	`slot.getRole_b().z_internalRemoveFromA_B(slot);`
	`slot.setRole_b(element);`
	`element.z_internalAddToA_B(slot);`

1. Check null parameter
2. Check index bound
3. Check duplication (be generated iff the opposite side is unique (in UML {unique}) association)
4. Clean relationship from opposite side (be generated iff the multiplicity of this side is one and the association is bidirectional)
5. Set

addTo__At()(B element , int index)

1	`if ( element == null )return;`
2	`if ( index >= role_b.size() | index < 0) return;`

3	`boolean isPresent = false;`
	`A_B where = null;Iterator it = a_B.iterator();`
	`while ( it.hasNext() && !isPresent ) {`
	`    A_B elem = (A_B) it.next();`
	`    if ( elem.getRole_b().equals(element)) {`
	`        isPresent = true;where = elem;}}`
	`if ( isPresent & a_B.indexOf(where) != index ) {`
	`    removeFromRole_b(element);}`

4	`if ( element.getA_B() != null ) {((A_B)element.getA_B()).clean();}`

5	`A_B asscls =new A_B(this,element);`
	`a_B.add(index,asscls);`
	`Iterator ii = a_B.subList(index+1,a_B.size()).iterator();`
	`while ( ii.hasNext() ) {`
	`    A_B item = (A_B) ii.next();`
	`    item.i_index1++;}`
	`asscls.i_index1 = index;`

1. Check null parameter
2. Check index bound
3. Check duplication (be generated iff the opposite side is unique (in UML {unique}) association)
4. Clean relationship from opposite side (be generated iff the multiplicity of this side is one and the association is bidirectional)
6. Add

removeFrom__At(int index)

1	`if ( index >= role_b.size() | index < 0) return;`

2	`A_B beRemoved =  a_B.get(index);`
	`beRemoved.clean();`

1. Check index bound
2. Remove

z_internalAddTo_&&_(B element)

1	`this.a_B.add(element);`
2	`element.i_index1 = this.a_B.size()-1;`

1. Add
2. Handle index information (be generated iff the opposite side is indexed collection type)

z_internalRemoveFrom_&&_()

1	`int index = this.a_B.indexOf(element);`
	`Iterator ii =a_B.subList(index+1,a_B.size()).iterator();`
	`while ( ii.hasNext() ) {`
	`    A_B item = (A_B) ii.next();`
	`    item.i_index1--;}`
	`element.i_index1=-1;`

2	`f_role_b.remove(index);`

1. Handle index information (be generated iff the opposite side is indexed collection type)
2. Remove

5.7. Strategy for Attribute Mapping (Collection type)

Scenario: When class A has an attribute defined field named "attr" and its type is one of the four OCL supported collection types.

5.7.1. Creation process in ECC, IC Strategy

In order to perform the strategy, some classes will be created. The creation process consists of three steps:

Step1: Creation of ECC, IC pair:
Situation 1: If "attr" is a non-nested collection:

- Create two classes named "ECC" and "IC" i.e. one pair of ECC and IC: (ECC, IC)
- If the name "ECC" or "IC" causes a "naming collision", "Naming algorithm" is performed to generate an unique name
- We mark the ECC with "Start ECC" and IC with "End IC"

Situation 2: If "attr" is a nested collection with nested level *n* :
(E.g. a 2-level nested: "Sequence(Bag(OrderedSet()))")

- Create (n+1) pairs of ECC and IC: (ECC, IC)(ECC1, IC1)….(ECCn, ICn)
- The name of each ECC and IC is generated through "Naming algorithm".
- We mark the ECC of first pair with "Start ECC" and the IC of last pair with "End IC". The other ECCs and ICs in pairs are all marked with "Inner ECC" or "Inner IC"

Each pair of ECC and IC corresponds to a collection type according to the collection definition. For example, if there is a nested collection "Sequence(Bag(OrderedSet()))", (ECC,IC) corresponds to Sequence, (ECC1,IC1) for Bag and (ECC2,IC2) for OrderedSet.

Step2: Creation of fields in ECC and IC
Fields on ECC:
("IC_name" stands for the name of IC in the same pair)

Name	Visibility	Type	initialization	Generation Condition
myItems	private	List<IC_class>	new ArrayList<IC_class>()	true

Fields on IC:
(If "Start ECC" is located in the same pair, "Owner_class" stands for class A, if not, it stands for the name of IC in the previous pair)
(If IC is not an "End IC", "Item_class" stands for the name of the ECC in the next pair. If it is, "Item_class" stands for the ultimate element type in the collection, i.e. OCL supported basic type, enumerated type or annotated class type)

Name	Visibility	Type	initialization	Generation Condition
id	public	long	none	true
owner	private	Owner_class	none	true
item	private	Item_class	new Item_class()	true
sequence	public	int	-1	if the next pair(relative to the pair which this IC located) corresponds to an indexed collection type

Step3: Creation of methods in ECC and IC
(If collection type is "Set", "collection_type" stands for "java.util.Set", otherwise, "java.util.List")

Methods will be created in ECC:
(If ECC is a "Start ECC", "Owner_class" stands for class A, if not, it stands for the name of IC in the previous pair)
("element_type" stands for the type of the field "myItem")

- getCollection()
- setCollection(collection_type coll,owner_class_type owner)
- clear()
- add(element_type elem,owner_class_type owner)
- remove(object)

Keep on, if the pair of ECC corresponds to an indexed collection type

- get(int index)
- setAt(int index,element_type elem)
- insertAt(int index, element_type elm,owner_class_type owner)
- remove(int index)

Methods will be created in IC:

- getter setter methods for "owner" and "item"
- remove()

Keep on, if IC is an "Inner IC"

- removeFromItem(collection_type coll)
- removeAllFromItem()

5.7.2. Recipe for Annotation Mapping

The targets for the mapping annotations are: (No mapping annotation needed to put on field "attr")

- The correlative ECC and IC classes for field "attr".
- fields in ECC or IC

Annotation on ECC and IC:

ECC	@Embeddable
IC	@Entity(access=AccessType.FIELD)

Annotation on field in ECC:

(Elements in the first column are conditions)

(Element in the first row is the annotation taget)

	myItem
If the pair in which this ECC located corresponds to indexed collection	@OneToMany(cascade=CascadeType.ALL,mappedBy="owner") @OrderBy("sequence")
Otherwise	@OneToMany(cascade=CascadeType.ALL,mappedBy="owner")

Annotation on fields in IC:

id	@Id(generate=GeneratorType.IDENTITY)
owner	@ManyToOne(cascade=CascadeType.ALL)
sequence	--

When IC is an “Inner IC” or IC is an “End IC” but ultimate element type is an OCL supported basic type, enumerated type or a “@Embeddable” annotated class type:
No annotation assigned for “item” field.

When IC is an “End IC” and the ultimate element type is a “@Entity” annotated class type:
(Elements in the first column are conditions)
(Element in the first row is the annotation target)

	item
The last ECC,IC pair corresponds to a Set or Ordered Set	@OneToOne
Otherwise	@ManyToOne

5.7.3. Recipe for Code Pattern

The following modifications will be carried out on class A:

- Change the type declaration of „attr“ to its „Start ECC“ Class type
- Modify the getter and setter methods of field „atr“
- Modify addTo__(element type)
- Modify removeFrom__(element type)

If “attr” field is an indexed collection type

- Create get__At(int index)
- Create set__At(element type, index)
- Create addTo__At(element type, index)
- Create removeFrom__At(element type ,index)

(Here, „__“ is „Attr“, “element type” is the type of element defined in collection type “attr”) get__()

```
return attr.getCollection();
```

1. Return

set__(element_type element)

```
return attr.getCollection();
```

addTo__(element_type element)

```
this.attr.add(element,this);
```

removeFrom__(element_type element)

```
attr.remove(element);
```

get__At(int index)

```
return attr.get(index);
```

set__At(element_type element, int index)

```
this.attr.setAt(index,element,this);
```

add__At(element_type element, int index)

```
this.attr.insertAt(index,element,this);
```

removeFrom__At(element_type element, int index)

```
this.attr.removeAt(index);
```

The codes body for the methods on ECC which are listed in creation process step3:

getCollection()

```
ArrayList results = new ArrayList();
Iterator it = myItems.iterator();
while ( it.hasNext() ) {
    results.add(((IC1)it.next()).getItem());}
return results;
```

setCollection(collection_ type coll, owner_class_ type owner)

```
if ( getCollection().equals(coll) ) return;
clear();
Iterator ii = coll.iterator();
while ( ii.hasNext() ) {
    add((java.util.List)ii.next(),owner);}
```

clear()

```
1  Iterator ii = new ArrayList(myItems).iterator();
   while ( ii.hasNext() ) {
        IC1 item = (IC1)ii.next();
        myItems.remove(item);
        item.remove();}}
```

add(element_type element, owner_class_type owner)

```
1  if ( element == null )  return;
---
2  IC1 item = new IC1();
   item.setItem(element);
   this.myItems.add(item);
---
3  item.setSequence(myItems.size()-1);
---
4  item.setOwner(owner);
```

1. Check null parameter
2. Add
3. Handle index information (be generated iff this ECC,IC pair corresponds to an indexed collection)
4. Build relationship from opposite side

get(int index)

```
1  return (List) getCollection().get(index);
```

remove(object element)

```
1  if ( element == null ) {return false;}
---
2  IC2 item = new IC2();Iterator ii = myItems.iterator();
   boolean out_range= true;
   while ( ii.hasNext() & out_range ) {
        item = (IC2)ii.next();
        if ( item.getItem().equals(element) )out_range=false;
   }if ( out range ) {return false;}
---
3  int index= myItems.indexOf(item);
   Iterator it = myItems.subList(index+1,myItems.size()).iterator();
   while ( it.hasNext() ) {
        IC1 elem = (IC1)it.next();
        elem.setSequence(elem.getSequence()-1);}
---
4  this.myItems.remove(index); item.remove();
---
5  return true;
```

1. Check null parameter
2. Check existence
3. Handle index information (be generated iff this ECC,IC pair corresponds to an indexed collection)
4. Remove
5. Return

setAt(int index, element_type element, owner_class_type owner)

1	`if ( element == null )  return;`
2	`if ( index < 0 | index >= myItems.size() ) return;`

3	`IC1 item = new IC1();item.setItem(element);`
	`item.setSequence(index);myItems.set(index,item);`
	`item.setOwner(owner);`

1. Check null parameter
2. Check index bound
3. Set

insertAt(int index, element_type element, owner_class_type owner)

1	`if ( element == null )  return;`
2	`if ( index < 0 | index >= myItems.size() ) return;`

3	`Iterator ii = myItems.subList(index,myItems.size()).iterator();`
	`while ( ii.hasNext() ) {`
	`     IC1 item = (IC1) ii.next();`
	`     item.setSequence(item.getSequence()+1);}`

4	`IC1 item = new IC1();item.setItem(element);`
	`item.setSequence(index);`
	`myItems.add(index,item);`
	`item.setOwner(owner);`

1. Check null parameter
2. Check index bound
3. Handle index information (be generated iff this ECC,IC pair corresponds to an indexed collection)
4. Add

remove(int index)

1	`if ( index < 0 | index >= myItems.size() ) return;`

2	`Iterator it = myItems.subList(index+1,myItems.size()).iterator();`
	`while ( it.hasNext() ) {`
	`     IC1 elem = (IC1)it.next();`
	`     elem.setSequence(elem.getSequence()-1);}`

3	`IC1 item = myItems.get(index);`
	`myItems.remove(index);`
	`item.remove();`

1. Check index bound
2. Handle index information (be generated iff this ECC,IC pair corresponds to an indexed collection)
3. Remove

The codes body for the methods on IC which are listed in creation process step3:

getOwner()

```
return this.owner;
```

setOwner(Owner_class_type owner)

```
this.owner= owner;
```

getItem()

```
return item.getCollection();
```

setItem()

```
item.setCollection(element,this);
```

remove()

```
1 this.owner= null;
2 this.sequence=-1;
3 removeAllFromItem();
```

1. Remove
2. Remove (be generated iff this ECC,IC pair corresponds to an indexed collection)
3. Remove (be generated iff this IC is an "Inner IC")

removeFromItem(element_type element)

```
item.remove(element);
```

removeAllFromItem()

```
Iterator it = new HashSet(getItem()).iterator();
while ( it.hasNext() ) {
    Object item = it.next();
    if ( item instanceof List ) {
        removeFromItem((List)item);}}
```

5.8. Summary

In this chapter, based on the results of discussions from previous chapters, the particular solutions and strategies to each potential problem especially the two main problems "Index association" and "Collection type attribute defined field" were covered in great detail. At the same time, with the aim of realizing these solutions, we also provided some integrated measures for improving Octopus. Through a systematic implementation process, a generic recipe for the generation of MDA driven EJB3 persistence artifacts is presented.

Chapter 6. Introduction to OctopusEE Beta

OctopusEE (Octopus Enterprise Edition) Beta is the Implementation of the MDA-driven generation of EJB3 persistence artifacts based on the generic recipe in Chapter 5.

6.1. Requirements

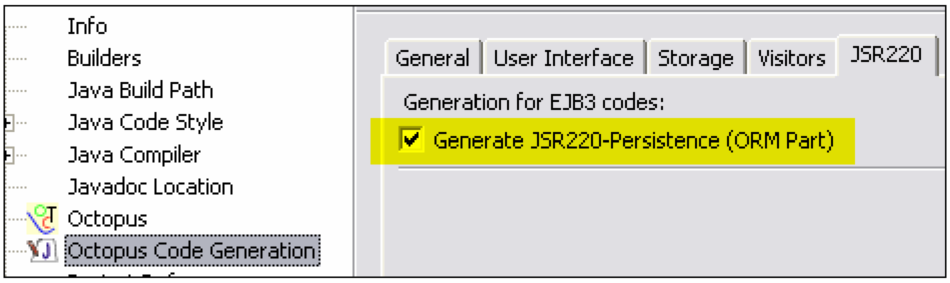
EJB3 runs on Java 1.5 VM or above, thus the installation of Java 1.5 is an essential requirement. Besides this, an Entity Manager needs to be set for providing the EJB3 persistence environment (also referred to as “persistence engine” or “ORM engine”). We recommend using “Hibernate Entity Manager” which is founded on “Hibernate Core” and “Hibernate Annotation”.

6.2. Setup in Eclipse

6.2.1. Configuration in Property page

In the “Properties” page of your “octopus project”, make sure that the “JDK Compiler compliance level” is set to “5.0”

In order to let OctopusEE generate EJB3 artifacts, you need to turn the option for EJB3 generation on. This switch can be found under “Properties”→“octopus code generation”→”JSR220”



Generation for EJB3 codes:

[✓] Generate JSR220-Persistence (ORM Part)

6.2.2. Configuration of build path:

The following .jar files are needed in the build path of your “octopus project”:

- *hibernate-entitymanager.jar* from root directory of Hibernate Entity Manager package.
- *ejb3-persistence.jar* (the core library for EJB3 persistence) and
- *hibernate-annotation.jar* to be found in the lib directory of the Hibernate Entity Manager installation
- *hibernate3.jar* from root directory of Hibernate Core package. Add the whole lib directory in build path. We also need *hibernate-tool.jar* from Hibernate Tool package for generation of DDL file. In order to get *hibernate-tool.jar* working, some additional jars are required, you can take these information from “chapter 4 Ant tools” of the Hibernate Tool document.

6.3. Generated files

The original Octopus distribution will generate an “utilities” package in addition to the package defined in the .uml file. OctopusEE will output a new “DDLGenerator.java” in “utilities” package. This file is used for generating the database schema by means of DDL. It can be configured to let the DDL be executed directly by the DBMS during code generation.

A log4j configuration file “log4j.properties” is created under “src” directory. From default configuration, log information will be displayed in console.

Furthermore, OctopusEE will generate two more packages. One is “META-INF” and the other is “test”. The “META-INF” folder contains the XML configuration files for the project. They are “hibernate.cfg.xml” which is used for DDL generation and “persistence.xml” which contains the ORM mapping information to be used by the Entity Manager at runtime. In the “test” package, a simple JUnit file is created.

6.4. Testing

6.4.1. Creating database schema

First, a database named “test” should exist in MySql beforehand. If you want to use another DBMS (Oracle, Mssql etc.) or specify a different databsase name, you can set appropriate property value in “hibernate.cfg.xml”. More information about hibernate configuration please consult the book *Hibernate in Action* or the online documentation.

Run “DDLGenerator.java” as a standalone Java application. By default, the generated DDL will be saved in file “schema.ddl”. You can configure the generation behavior through changing some parameter in “DDLGenerator.java” methods.

```
schemaExport.setDelimiter(";")
             .setOutputFile("schema.ddl")
             .create(false,true);
```

The first parameter of the create() method is a switch for “console display”. If true, all generated DDL will displayed on console. The second boolean parameter controls whether the DDL will be executed on the DBMS (when set to “true”, all tables defined in DDL will be created in database.

6.4.2. Writing JUnit tests

You can put the test codes in test() methods in “test.java”. In this file, two entity manager instances are created as local variable “em” and “em2”. You can handle the entities in different entity manager instances (which in principle might correspond to two different DB connections), e.g. you can save an entity in “em” and later retrieve it from “em2”.

The Entity Manager will trigger corresponding EJB3QL for each “object query”. If you want to observe the background EJB3QL expression, simply change the “hibernate.show_sql” property in “persistence.xml” to “true”.

```
<property name="hibernate.show_sql" value="true"/>
```

6.5. Summary

In this chapter, we introduced OctopusEE Beta and explained how to use it. OctopusEE Beta provides a way to check the correctness of our transformation recipe. It will be improved in the future as improvements are made to those transformations.

Chapter 7. Conclusion

After discussion in chapters 2 through 4, a complete procedure for generation of EJB3 artifacts on Octopus platform is given in chapter 5. But there are still two problems that the recipe does not address: “potential problem by association class mapping” mentioned in section 2.7.1 and “Problem of rewriting equals() and hashCode()” in section 3.4.1. In the recipe we still apply the mapping strategy of section 2.5.2.2 for an association class mapping. In order to solve this problem, we can map both foreign keys as a composite primary key; accordingly, the code pattern used in Octopus in association class must be redesigned. (see section 3.4.1). For problem of “equals() and hashCode()” a possible solution is based on an extension of Octopus UML so that model designer can define a “business key” (section 4.1.6 [King05]) for an entity through constraint notations.

In future work, this recipe can be improved in the following ways:
1. ECC class can be written in a template so that the created classs amount can be reduced. It makes sense by mapping a nested collection with a very high nested level.
2. Redesign the database mapping strategy for an association class in order to solve the problem metioned in section 2.7.1.
3. Extend the descriptive capacity of Octopus UML so that the generic recipe can be expanded accordingly.

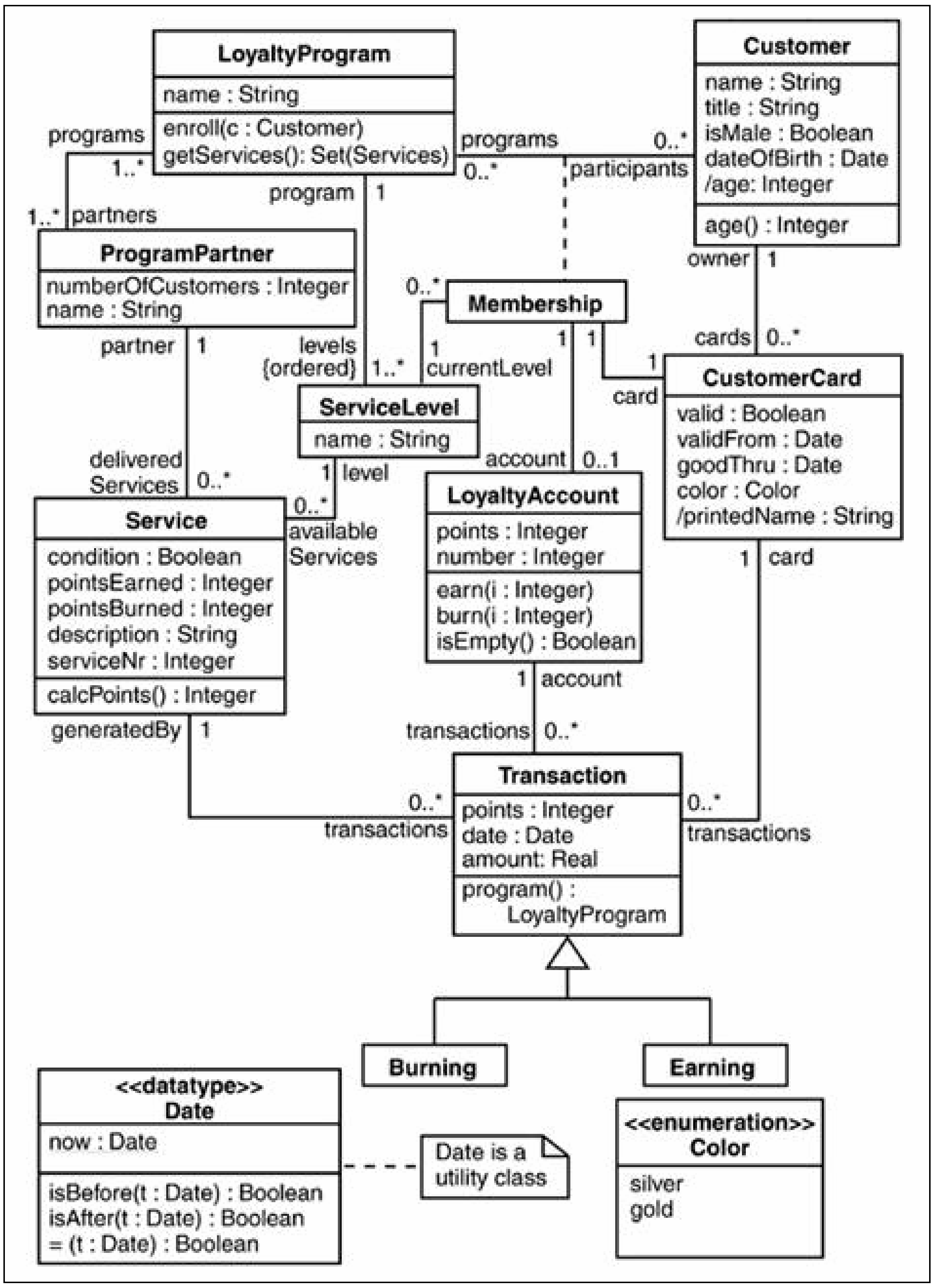
As a suggestion, we encourage to use fragments based code generation instead of Octopus inherent template-based generation. (see section 3.4.2) As an experiment, a code generation process based on code fragments is shown in Appendix B.

For a complex application system based on EJB technology, the states of a component (persist state, runtime state or client related state etc) in middleware must be managed along with the specific business logic of the system. In a model based application, OCL is widely used to specify runtime checks of these states so that a UML model for a component can be used in black–box testing, providing a more abstract view of middleware in which developer can neglect the complexity of distribution and concurrency. ([Wolff01]). Consequently, OCL invariants could be used in the future to check the persistent state of entities so that the database integrity can be ensured. To archieve this, in future work, we can transform OCL expression to static EJB QL expression i.e. JSR220 “namedQuery” for entities. We can also extend the OCL invariants check as callback methods of EJB so that application transaction can rollback once the integrity and consistency of the underlying database is broken.

Actually, some efforts are already underway to apply OCL to business components, based on the previous version of EJB (2.x), for example the usage of OCL in the „PleXX” framework described in the Diplomarbeit [Mitrik04]. Because EJB3 persistence is a radical improvement, it is planned as future work the transformation from OCL to EJB3QL.

Appendix A

Class Diagram of Project Royal and Loyal



Appendix B

An example of fragments-based code generation

Methods for association defined field:

Section 1	Section 2	Section 3	Section 4
get__()	addTo__(element)	z_internalAddTo__()	get__At()
set__()	addTo__(collection)	z_internalRemoveFrom__()	set__At()
get_&&_()	removeFrom__(element)	z_internalAddTo_&&_()	addTo__At()
set_&&_()	removeFrom__(collection)	z_internalRemoveFrom_&&_()	removeFrom__At()
	removeAllFrom__()		

„_&&_“ indicats the name of field which refers to association class (String value equals to Type name of association Class)

„__“ : reference field named by role name of the opposite side (String value equals to Role name of opposite side)

Role name: the role name of an association end. The first letter must be uppercase.

Name of field: The first letter must be lowercase.

Note: In Section3, if association owns an association class,methods z_internalAddTo_&&_() and z_internalRemoveFrom_&&_() will replace z_internalAddTo__() and z_internalRemoveFrom__().

Generation Condition: When all the conditions in list are fullfilled, the method will be generated

Necessary informations for generation: the informations which can affect the code detail.

Generation for Name: the generated name of this method

Generation for body: Codes body generation consists of serval steps. They will be performed in order.

B.1. Fragments based generation for Section 1

get__()

Generation Condition:

1. the opposite side is navigable

Necessary informations for generation:

[A]. Role name of opposite side
[B]. Name of reference field
[C]. Type name of reference field
[D]. if reference field is collection type

Generation for Name: `public [C] get[A](){}`

Generation for Body:

1

Fragment name	Generation condition	Codes snippet
Parameter check	[D] is true	Index=1

2.

Fragment name	Generation condition	Codes snippet
Return	[D] is true	Index=2, [1]=[B]
	[D] is false	Index=1,[1]=[B]

set__()

Generation Condition:

1. the opposite side is navigable

Necessary informations for generation:

[A]. Role name of opposite side
[B]. Name of reference field
[C]. Type name of reference field
[D]. Type name of the opposite side
[E]. Role name of this side
[F]. Type name of this side
[G]. If association is unidirectional

[H]. If multiplicity of this side is many
[I]. If multiplicity of the opposite side is many
[J]. If association has association class
[K]. Type name of assocation class
[L]. Name of the field which refers to association class

Name: `public void set[A]([C] element){}`

Body:

1.

Fragment name	Generation Condition	Codes snipped
Duplication check	[G] is false & [J] is false	Index=2,[1]=[B]

2.

Fragment name	Generation Condition	Codes snipped
Clean relationship from opposite side	[G] is false & [H] is false & [I] is false & [J] is false	Index=2,[1]=[B],[2]=[E],[3]=[F]
	[G] is false & [H] is false & [I] is true & [J] is false	Index=1.[1]=[B],[2]=[D,[3]=[E],[4]=[F]
	[J] is true & [I] is false	Index=6,[1]=[L],[2]=[K]
	[J] is true & [I] is true	Index=7,[1]=[L],[2]=[K]

3

Fragment name	Generation Condition	Codes snipped
Set	[J] is false	Index=1,[1]=[B]
	[J] is true & [I] is false	Index=2,[1]=[B],[2]=[K]
	[J] is true & [I] is true	Index=3,[1]=[D],[2]=[A]

.4.

Fragment name	Generation Condition	Codes snipped
Build relationship from opposite site	[J] is false & [G] is flase & [I] is false	Index=1,[1]=[E],[2]=[F]
	[J is false & [G] is flase & [I] is true	Index=2,[1]=[D],[2]=[E],[3]=[F]

set_&&_()

Generation Condition:

1. association with association class

Necessary informations for generation:

[A] Name of the field which refers to association class
[B] Type name of association class

Name: `public void set[B]([B] element){}`

Body:

1.

Fragment name	Generation condition	Codes snippet
Set	true	Index=1,[1]=[A]

get_&&_()

Generation Condition:

Association with association class

Necessary informations for generation:

[A]Name of the field which refers to association class
[B]Type name of association class

Name: `public [B] get[B](){}`

Body:

1.

Fragment name	Generation condition	Codes snippet
Return	ture	Index=1,[1]=[A]

B.2. Fragment based generation for Section2

addTo__(element)

Generation Condition:

1. the opposite side is navigable and 2. the opposite side is many

Necessary informations for generation:

[A]. Role name of opposite side	[H]. If multiplicity of this side is many
[B]. Name of reference field	[I]. If multiplicity of the opposite side is many
[C]. Type name of reference field	[J]. If opposite side is indexed collection
[D]. Type name of the opposite side	[K]. If opposite side is unique collection
[E]. Role name of this side	[L]. If association has association class
[F]. Type name of this side	[M]. Type name of assocation class
[G]. If association is unidrectional	[N]. Name of the field which refers to association class
	[O]. Index field name of opposite side

Generation for Name: `public void addTo[A]([D] element){}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Parameter null check	[G] is false	Index=1

2.

Fragment name	Generation condition	Codes snippet
Dupplication check	[K] is true & [L] is false	Index=3, [1]=[B]
	[K] is true & [L] is true	index=4,[1]=[B],[2]=[D],[3]=[A]

3

Fragment name	Generation condition	Codes snippet
Clean relationship from opposite side	[G] is false & [H] is false & [L] is false	Index=3, [1]=[B],[2]=[E]
	[G] is false & [H] is false & [L] is true	Index=4,[1]=[M]

4.

Fragment name	Generation condition	Codes snippet
Add	[L] is false	Index=1, [1]=[B]
	[L] is true	Index=2,[1]=[M]

5

Fragment name	Generation condition	Codes snippet
Index handling	[L] is false & [J] is true	Index=1, [1]=[O],[2]=[B]

6.

Fragment name	Generation condition	Codes snippet
Build relationship from opposite side	[G] is false & [L] is false	Index=3, [1]=[E],[2]=[F]

addTo__(collection)

Generation Condition:

1. The opposite side is navigable and 2. the opposite side is many

Necessary informations for generation:

[A]. Role name of opposite side
[B]. Type name of the opposite side
[C]. Type name of parameter

Generation for Name: `public void addTo[A]([C] newElems){}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Add	true	Index=3, [1]=[B],[2]=[A],

removeFrom__(element)

Generation Condition:

1. the opposite side is navigable and 2. the opposite side is many

Necessary informations for generation:

[A]. Role name of opposite side
[B]. Name of reference field
[C]. Type name of reference field
[D]. Type name of the opposite side
[E]. Role name of this side
[F]. Type name of this side
[G]. If association is unidrectional
[H]. If multiplicity of this side is many
[I]. If multiplicity of the opposite side is many
[J]. If opposite side is indexed collection
[K]. If opposite side is unique collection
[L]. If association has association class
[M]. Type name of assocation class
[N]. Name of the field which refers to association class
[O]. Index field name of opposite side

Generation for Name: `public void removeFrom[A]([D] element){}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Parameter null check	[G] is false	Index=2

2.

Fragment name	Generation condition	Codes snippet
Existence check	[G] is false & [L] is false	Index=1,[1]=[B]
	[G] is false & [L] is true	Index=2,[1]=[N],[2]=[M],[3]=[E]

3

Fragment name	Generation condition	Codes snippet
Index handling	[L] is false & [J] is true	Index=2, [1]=[B],[2]=[D],[3]=[O]

4.

Fragment name	Generation condition	Codes snippet
Removement	[L] is false	Index=1, [1]=[B]
	[L] is true	index=4,[1]=[M],[2]=[N]

5

Fragment name	Generation condition	Codes snippet
Clean relationship from opposite side	[G] is false & [L] is false	Index=5, [1]=[E],[2]=[F]
	[G] is false & [L] is true	Index=4,[1]=[M]

removeFrom__(collection)

Generation Condition:

1. the opposite side is navigable and 2. the opposite side is many

Necessary informations for generation:

[A]. Role name of opposite side
[B]. Type name of the opposite side
[C]. Type name of parameter

Generation for Name: `public void removeFrom[A]([C] oldElems){}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Removement	true	Index=3, [1]=[B],[2]=[A],

removeAllFrom__()

Generation Condition:

1. the opposite side is navigable and 2. the opposite side is many

Necessary informations for generation:

[A]. Role name of opposite side

Generation for Name: `public void removeAllFrom[A](){}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Removement	true	Index=4, [1]=[A]

B.3. Fragments based generation for Section 3

z_internalAddTo__()

Generation Condition:

1. association is bidirectional 2. association without associaion class

Necessary informations for generation:

[A]. Role name of opposite side [B]. Name of reference field [C]. Type name of reference field [D]. Type name of the opposite side [E]. Role name of this side [F]. Type name of this side [G]. If association is unidrectional	[H]. If multiplicity of this side is many [I]. If multiplicity of the opposite side is many [J]. If opposite side is indexed collection [K]. If opposite side is unique collection [L]. If association has association class [M]. Type name of assocation class [N]. Name of the field which refers to association class [O]. Index field name of opposite side

Generation for Name: `public void z_internalAddTo[A]([D] element) {}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Set	[I] is false	Index=1,[1]=[B]

2.

Fragment name	Generation condition	Codes snippet
Add	[I] is true	Index=1, [1]=[B]

3.

Fragment name	Generation condition	Codes snippet
Index handling	[I] is true & [J] is true	Index=1, [1]=[O],[2]=[B]

z_internalRemoveFrom__()

Generation Condition:

1. association is bidirectional and 2. association without association class

Necessary informations for generation:

[A]. Role name of opposite side [B]. Name of reference field [C]. Type name of reference field [D]. Type name of the opposite side [E]. Role name of this side [F]. Type name of this side [G]. If association is unidrectional	[H]. If multiplicity of this side is many [I]. If multiplicity of the opposite side is many [J]. If opposite side is indexed collection [K]. If opposite side is unique collection [L]. If association has association class [M]. Type name of assocation class [N]. Name of the field which refers to association class [O]. Index field name of opposite side

Generation for Name: `public void z_internalRemoveFrom[A]([D] element) {}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Index handling	[I] is true & [J] is true & [H] is ture	Index=2, [1]=[B],[2]=[D],[3]=[O]

2.

Fragment name	Generation condition	Codes snippet
Removment	[H] is false	Index=5, [1]=[B]
	[H] is true	Index=1,[1]=[B]

z_internalAddTo_&&_()

Generation Condition:

1. association is bidirectional and 2. association with association class

Necessary informations for generation:

[A]. Type name of assocation class
[B]. Index field name of opposite side
[C]. Opposite side is many
[D]. Opposite side is indexed collection
[E]. Index field name of opposite side

Generation for Name: `public void z_internalAddTo[A]([A] element) {}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Set	[C] is false	Index=1,[1]=[B]

2.

Fragment name	Generation condition	Codes snippet
Add	[C] is true	Index=1, [1]=[B]

3.

Fragment name	Generation condition	Codes snippet
Index handling	[C] is true & [D] is true	Index=1, [1]=[E],[2]=[B]

z_internalRemoveFrom_&&_()

Generation Condition:

1. Association is bidirectional and 2. association with association class

Necessary informations for generation:

[A]. Type name of assocation class
[B]. Index field name of opposite side
[C]. Opposite side is many
[D]. Opposite side is indexed collection
[E]. Index field name of opposite side
[F]. This side is many
[G]. Name of reference field
[H]. Type name of opposite side

Generation for Name: `public void z_internalRemoveFrom[A]([A] element) {}`

Generation for Body:

1.

Fragment name	Generation condition	Codes snippet
Index handling	[C] is true & [D] is true & [F] is true	Index=2, [1]=[G],[2]=[H],[3]=[E]

2.

Fragment name	Generation condition	Codes snippet
Removment	[C] is false & [F] is false	Index=5, [1]=[B]
	!([C] is false & [F] is false)	Index=1,[1]=[B]

B.4. Code fragments

Snippet for „return“ (index, [1])

1	Return_Single_Object	`return [1];`
2	Return_Collection_Object	`return Collections.unmodifiableSet([1]);`
3	Return Single Obj. In collection based on index	`return [1].get(index);`

Snippet for "parmeter null check" (index,[1])

1	Check_Null_return_null	`if ( element == null ) {return null;}`
2	Check_null_return	`if ( element == null ) {return;}`
3	null check based on collection index	`if ( [1].get(index)== null ) {return;}`

Snippet for "existence check" (index,[1],[2],[3])

1	Check_existence_return_null	`if (!this.[1].contains(element)) return;`

2	in case of association class	`[2] foundElem = null;`
		`Iterator it = [1].iterator();`
		`while ( it.hasNext() ) {`
		`    [2] elem = ([2]) it.next();`
		`    if ( elem.get[3]() == element ) {`
		`        foundElem = elem;}}`
		`if (foundElem== null) return;`

Snippet for "dupplication check" (indec,[1])

1	Duppl. Check return null	`if (this.[1]==element) return null;`
2	Duppl. Check return	`if (this.[1]==element) return;`

3.	Duppl. Check for collection	`if ( this.[1].contains(element) ) {`
		`    return;}`

4	Duppl. Check for association class	`boolean isPresent = false;`
		`Iterator it = [1].iterator();`
		`while ( it.hasNext() && !isPresent ) {`
		`    [2] elem = ([2]) it.next();`
		`    if ( elem.get[3]() == element ) {`
		`        isPresent = true;}}`
		`if(isPresent) return;`

Snippet for "index bound check" (index,[1])

1	bound check return null	`if ( index >= [1].size() || index < 0 ) {`
		`    return null;}`

2	bound check return	`if ( index >= [1].size() || index < 0 ) {`
		`    return;}`

Snippet for „Add“ (index,[1],[2])

1	Add	`this.[1].add(element);`
2	Add for association class	`[1] asscls = new [1](this,par);`

3	code body for addTo__(collection)	`Iterator it = newElems.iterator();`
		`while ( (it.hasNext()) ) {`
		`    Object item = it.next();`
		`    if ( item instanceof [1] ) {`
		`        addTo[2](([1])item);}}`

4	add with index	`[1].add(index,element);`

Snippet for “Index handling” (index,[1])

1	evaluate last index of collection	`element.[1] = this.[2].size()-1;`

2	adjusting for removment	`int index = this.[1].indexOf(element);`
		`Iterator ii =[1].subList(index+1,[1].size()).iterator();`
		`while ( ii.hasNext() ) {`
		`    [2] item = ([2]) ii.next();`
		`    item.[3]--;}`
		`element.[3]=-1;`

3	adjust index for addtoAt	`Iterator ii =[1].subList(index,[1].size()).iterator();`
		`while ( ii.hasNext() ) {`
		`    [2] item = ([2]) ii.next();`
		`    item.[3]++;}`
		`element.[3] = index;`

4	replace	`element.[1] = index;`
		`get[2](index).[1]=-1;`

5	only know index	`if([1].get(index) != null){`
		`Iterator ii =[1].subList(index+1,[1].size()).iterator();`
		`while ( ii.hasNext() ) {[2] item = ([2]) ii.next(); item.[3]--;}`
		`[1].get(index).[3]=-1;}`

Sninppet for „Set“ (index,[1],[2])

1	assignment	`this.[1] = element;`

2	Assignment for association class in case of „to One“	`if ( element != null ) {`
		`    this.[1] = new [2](this, element);`
		`} else {this.[1] = null;}`

3	Assignment for association class in case of „to many“	`if ( element != null ) {`
		`    it = element.iterator();`
		`    while ( it.hasNext() ) {`
		`        [1] elem = ([1]) it.next();`
		`        this.addTo[2](elem);}}`

4	Assignment for indexed collection	`[1].set(index,element);`

Sninppet for“ Removement”   (index,[1],[2])

1	remove from collection	`this.[1].remove(element);`

2	in case of association class	`(([1])foundElem).clean();`
		`this.[2].remove(foundElem);`

3	codes body for removeFrom__()	`Iterator it = oldElems.iterator();`
		`while ( (it.hasNext()) ) {`
		`    Object item = it.next();`
		`    if ( item instanceof [1] ) {`
		`        this.removeFrom[2](([1])item);}}`

4	codes body for removeFromAll()	`removeFrom[1](get[1]());`
5	remove in inner remover	`this.[1] = null;`
6	remove according index	`[1].remove(index);`

Sninppet for “build relationship from opposite side” (index,[1],[2],[3])

1	Assignment for opposite side which is „one“	`if ( element != null ) {`
		`    element.set[1]( ([2])this );}`

2	Assignment for opposite side which is „many“	`if ( element != null ) {`
		`    it = element.iterator();`
		`    while ( it.hasNext() ) {`
		`        [1] x = ([1]) it.next();`
		`        x.z_internalAddTo[2]( ([3])this );}}`

3	Assignement for an existent opposite end	`element.z_internalAddTo[1]( ([2])this );`

Snippet for “Clean relationhship from opposite side”   (index, [1],[2],[3],[4])

1	Clean the exsitent association of this side	`Iterator it = this.[1].iterator();`
		`while ( it.hasNext() ) {`
		`    [2] x = ([2]) it.next();`
		`    x.z_internalRemoveFrom[3]( ([4])this );}`

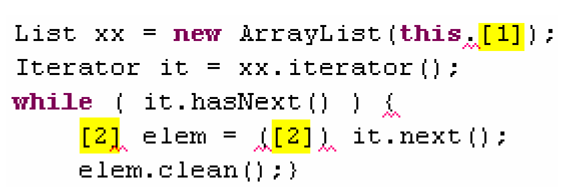
2	in case of „toOne“	`if ( this.[1] != null ) {`
		`    this.[1].z_internalRemoveFrom[2]( ([3]this );}`

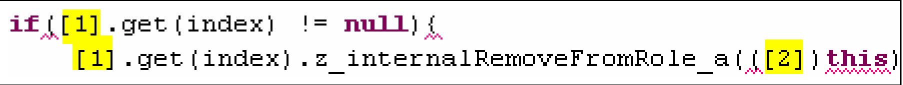
3	Clean the exsitent association of opposite end	`if ( element.get[1]() != null ) {`
		`    element.get[1]().z internalRemoveFrom[2](element);}`

4	in case of association class	`if ( element.get[1]() != null ) {`
		`    (([1])element.get[1]()).clean();}`

5	clean existent opposite end	`element.z_internalRemoveFrom[1]( ([2])this );`

6	Clean both associations in case of „to One“	`if ( this.[1] != null ) {`
		`    (([2])this.[1]).clean();}`

7	Clean the assocation between each end	
		`List xx = new ArrayList(this.[1]);`
		`Iterator it = xx.iterator();`
		`while ( it.hasNext() ) {`
		`     [2] elem = ([2]) it.next();`
		`     elem.clean();}`

8	only know the index	
		`if([1].get(index) != null){`
		`    [1].get(index).z_internalRemoveFromRole_a(([2])this)`

Appendix C

Generated configuration files in OctopusEE

Log4j.properties

```
log4j.rootCategory=INFO, A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=%d{MM-dd@HH:mm:ss} %-5p
(%13F:%L) %3x - %m%n
```

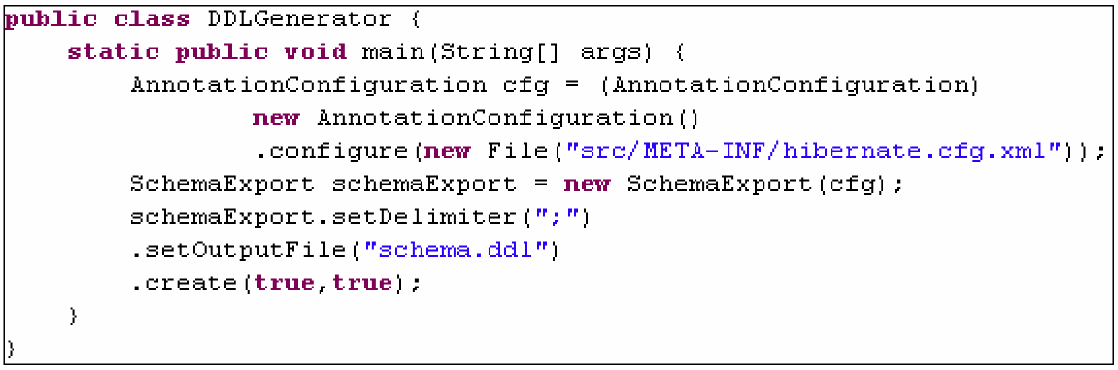
/META-INF/hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory name="context">
        <property name="hibernate.connection.driver_class">
        com.mysql.jdbc.Driver
        </property>
        <property name="hibernate.dialect">
        org.hibernate.dialect.MySQLDialect
        </property>
        <property name="hibernate.connection.url">
        jdbc:mysql://localhost:3306/test
        </property>
        <mapping class ="… "/>
        …
    </session-factory>
</hibernate-configuration>
```

/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-manager>
    <name>context</name>
    <class>…</class>
    ....
    <properties>
        <property name="hibernate.connection.url"
        value="jdbc:mysql://localhost:3306/test"/>

        <property name="hibernate.dialect"
        value="org.hibernate.dialect.MySQLDialect"/>

        <property name="hibernate.connection.driver_class"
        value="com.mysql.jdbc.Driver"/>

        <property name="hibernate.connection.password"
        value=""/>

        <property name="hibernate.connection.username"
        value="Xinhua"/>

        <property name="hibernate.show_sql" value="false"/>
    </properties>
</entity-manager>
```

/utilities/DDLGenerator.java



```
public class DDLGenerator {
    static public void main(String[] args) {
        AnnotationConfiguration cfg = (AnnotationConfiguration)
                new AnnotationConfiguration()
                .configure(new File("src/META-INF/hibernate.cfg.xml"));
        SchemaExport schemaExport = new SchemaExport(cfg);
        schemaExport.setDelimiter(";")
        .setOutputFile("schema.ddl")
        .create(true,true);
    }
}
```

References

[Annotations] Hibernate Annotations Reference Guide Version 3.1 Beta 8
URL: http://www.hibernate.org/hib_docs/annotations/reference/en/html

[Entitymanager] Hibernate EntityManager User Guide Version 3.1 Beta 6
URL: http://jcp.org/en/jsr/detail?id=220

[Dev] Octopus developer document version 2.0.0
URL: http://www.klasse.nl/octopus/octopus-developer-pack.zip

[JSR220-API] JSR-220 Enterprise JavaBeans version 3.0 Simplified API Proposed Final Draft
URL: http://jcp.org/en/jsr/detail?id=220

[JSR220-Contracts] JSR-220 Enterprise JavaBeans 3.0 Core Contracts and Requirement Proposed Final Draft: URL: http://jcp.org/en/jsr/detail?id=220

[JSR220-Persistence] JSR-220 Enterprise JavaBeans version 3.0 Persistence API Proposed Final Draft: URL: http://jcp.org/en/jsr/detail?id=220

[JSR220-Site] JSR220 Specification Site:
URL: http://jcp.org/en/jsr/detail?id=220

[J2SE1.5] J2SE 1.5 API online document
URL: http://java.sun.com/j2se/1.5.0/docs/api/

[Kleppe05] Jos Warmer, Anneke Kleppe; Wed yourself to UML with the power of Association; June 2005. URL: http://www.devx.com/enterprise/Article/28528

[Middleware03] The Middleware Company; Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach, Productivity Analysis; June 2003;
URL: http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf

[Octopus] Octopus Official Site:
URL: http://www.klasse.nl/octopus/index.html

[UML05] UML 2.0 Specification : Superstructure; April, 2005
URL: http://www.omg.org/cgi-bin/doc?formal/05-07-04

[Wolff01] Achim D. Brucker, Burkhart Wolff; Testing Distributed Component Based Systems Using UML/OCL; July 2001
URL: http://www.brucker.ch/bibliography/download/2001/info2001.pdf

[Bates06] Kathy Sierra, Bert Bates; Sun Certified Programmer for Java 5; McGraw-Hill; 2006

[Bloch01] Joshua Bloch, Effective java programming language guide, Addison Wesley; June 2001

[Johnson03] Rod Johnson; Expert One-on-One J2EE Design and Development; Wrow Press; 2003

[Johnson04] Rod Johnson, Juergen Hoeller; Expert One-on-One J2EE Development without EJB; Wiley Publishing, Inc.; 2004;

[King05] Christian Bauer, Gavin King; Hibernate in Action; Manning; 2005

[Kleppe03]Jos Warmer, Anneke Kleppe; The Object Constraint Language Second Edition;Addison Wesley; August 2003

[Malks03] Deepak Alur, John Crupi, Dan Malks; Core J2EE Pattern: Best Practices and Design Strategies, Second Edition; Prentice Hall PTR; June 10, 2003

[McLaughlin04] David Flanagan, Brett McLaughlin; Java 1.5 Tiger: A Developer's Notebook; O’Reilly; June 2004.

[Mitrik04] Dieter A Mitrik; Evaluierung und Implementierung der Object Constraint Language zur Prüfung von Geschäftsregeln im Rahmen des Business Object Frameworks *ple*XX; Januar 2004.

[Pitman05] Dan Pilone, Neil Pitman; UML 2.0 in a nutshell; O’Railly; June 2005

[Tate04] Justin Gehtland, Bruce A, Tate; Better Faster Lighter Java; O’Reilly; Junne 2004