Master Thesis

Trinh, Tien Trung

Migration from legacy Persistence API to JPA (Java Persistence API) based Implementation

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Trinh, Tien Trung

Migration from legacy Persistence API to JPA (Java Persistence API) based Implementation

Master thesis based on the examination and study regulations for the Master of Engineering degree programme
Information Engineering
at the Department of Information and Electrical Engineering
of the Faculty of Engineering and Computer Science
of the University of Applied Sciences Hamburg

Supervising examiner: Prof. Dr. Hans-Jürgen Hotop
Second examiner: Prof. Dr. Thomas Schmidt

Day of delivery: November 04th 2008

Trinh, Tien Trung

Title of the Master Thesis

Migration from the legacy Persistence API to JPA (Java Persistence API) based Implementation.

Keywords

Persistence, annotation, object-relational mapping, relational database, entity, version control, compatibility.

Abstract

The legacy persistence API (Application Programming Interface) is implemented using JDO (Java Data Objects) persistence interface with the underlying persistent store based on LDAP-compliant directory server. This thesis work, whose main purpose is to improve the system performance, carried out the migration of the persistent store to the base of relational database server by utilizing JPA (Java Persistence API) combined with the open source persistence framework Hibernate. The subsequent migration of the legacy persistence API to JPA-based implementation was also carried out. As the software productivity requirement, the JPA-based persistence API is obligated to not only fulfil the system compatibility but also adopt the essential functionalities from the legacy persistence API.

Trinh, Tien Trung

Thema der Masterarbeit

Migration von Legacy Persistenz API auf JPA (Java Persistence API) basierte Implementierung.

Stichworte

Persistenz, Annotation, objekt-relationales Mapping, relationale Datenbanken, Entität, Versions Kontrolle, Kompatibilität.

Kurzzusammenfassung

Die bisherige persistente API (Application Programming Interface) benutzt als Schnittstelle JDO (Java Data Objects), wobei das Speichern der Daten mithilfe eines verzeichnisorientierten LDAP Servers erfolgt. Diese Arbeit beschäftigt sich mit der Leistungsverbesserung der Anwendungen. Hierzu wird die persistente Komponente des Systems durch eine relationale Datenbank ersetzt. Diese neue Komponente wird auf Basis von JPA (Java Persistence API) zusammen mit den Open Source Framework Hibernate entwickelt. Die JPA basierte Implementierung soll kompatibel zum existierende System sein und die wesentlichen Funktionalitäten der persistenten API enthalten.

Contents

1. INTRODUCTION 6
1.1. Motivation 6
1.2. Purpose 7
1.3. Organization of the thesis 7
2. BASICS 8
2.1. Topgallant® Suite 8
2.1.1. Topgallant®Infrastructure 8
2.1.2. Legacy persistence solution 9
2.2. Java Persistence API (JPA) 10
2.2.1. Elements 10
2.2.2. JPA main functionalities 12
2.2.3. Standard JPA annotations 16
2.2.4. JPA callback 18
2.3. Java Architecture for XML Binding (JAXB) 19
2.4. Java DB (Apache Derby) 20
3. ANALYSIS 22
3.1. Problem 22
3.2. Requirements 22
3.3. Solutions 23
3.3.1. Persistence-supporting systems 23
3.3.2. Relational database vs. object-oriented database 28
3.3.3. Chosen solution 30
4. DESIGN, REALIZATION AND IMPLEMENTATION 31
4.1. Migration to RDBMS 31
4.1.1. Introduction and requirement analysis 31
4.1.2. Solution 31
4.1.2.1. Analysis of code generator 33
4.1.2.2. Analysis of the Java-based data model 36
4.1.2.3. Code generator modification 42
4.1.3. Result and verification 51
4.1.4. Conclusion 56
4.2. Compatible persistence API 58
4.2.1. Introduction and requirement analysis 58
4.2.2. Current working mechanism 58
4.2.3. Solution and implementation 60
4.2.4. Verification and conclusion 62
4.3. Essential functionalities 63
4.3.1. Version control 63
4.3.1.1. Introduction and requirement analysis 63
4.3.1.2. Solution and implementation 66
4.3.1.3. Verification 76
4.3.1.4. Conclusion 77

4.3.2. Management of secondary object storage/retrieval ... 78
4.3.2.1. Introduction and requirement analysis ... 78
4.3.2.2. Solution ... 80
4.3.2.3. Verification ... 85
4.3.2.4. Conclusion ... 86
4.4. JPA SHORTCOMING ... 87
4.4.1. Detaching ... 87
4.4.2. Handling of special data types ... 88
4.4.3. Orphaned entity deletion ... 88

5. EVALUATION ... 90

5.1. SYSTEM PERFORMANCE MEASUREMENT ... 90
5.2. COMPARISON AND CONCLUSION ... 92

6. SUMMARY AND CONCLUSION ... 95

6.1. SUMMARY ... 95
6.2. FURTHER WORK ... 96
6.3. CONCLUSION ... 97

REFERENCES ... 98

FIGURES ... 102

ABBREVIATIONS ... 104

APPENDIX ... 105

IMPLEMENTATION EXPLANATION ... 105

1. Introduction

1.1. Motivation

For the software applications which frequently need to access persistent data for various persistence operations (store/retrieve/modify/…), the application performance depends significantly on the response from the underlying persistent store. This is especially true for the cases in which large amount of data is processed. As for an example, one can figure out the scenario in which an application process at a certain point of time needs to request for a vast volume of data and for further processing, the entire application process must wait for the arrived data. It is clear that the shorter the delay time due to data delivery is, the faster the application process becomes and thus the better the application performance can be achieved.

Optimization of the system performance is always required and investigated especially in the business application development. Because of the fact that the system performance relies on many factors such as data processing speed, memory occupation and so on, the optimization can target at increasing the data processing speed or minimizing the memory occupation or even both. From the above scenario, it is obvious to see that the data processing speed is mainly influenced by the persistence solution which provides client transactions, ensures data integrity and so on.

The persistence solution primarily includes the persistence API implementation (or persistence layer) and the persistent store. The persistence API can be implemented with various persistence technologies regarding the handling of persistent objects such as Java Data Objects (JDO) API [6] ("a standard interface-based Java model abstraction of persistence"), entity beans in the Enterprise JavaBeans (EJB) [35] ("EJB technology is the server-side component architecture for Java Platform, Enterprise Edition") and more recently Java Persistence API (JPA) [14] (as part of JSR-220 (EJB 3.0)). The evolution of persistence technologies is more and more advanced with the main goal of simplifying the persistence API implementation and improving the system performance.

In addition to the number of persistence technologies, there is also a variety of kinds of persistent store based on LDAP-compliant directory server [36] or object database management system (ODBMS) [16] or XML base management system [37] or relational database management system (RDBMS) [38]. The important role of the underlying persistent store is especially expressed in the following example. For interactive applications which very often perform the object storing (with vast volume), one can easily see that the speed of storing to LDAP-compliant repository is significantly slow compared to the speed of storing to relational database. This is because LDAP-compliant directory server is highly optimized especially for retrieving than storing. Therefore, it is often preferred to have a persistent store based on LDAP-compliant directory server for the applications which mainly need to retrieve data (with vast volume). In contrast, for the interactive applications which very often perform the object storing (with vast volume); the underlying persistent store should be based on RDBMS.

It can be said that, the choice between LDAP-compliant directory server and RDBMS depends on the application task and user requirement. Because of the fact that the user requirements are often expanded, some legacy persistent store might have to be migrated from the base of LDAP-compliant directory server to the base of RDBMS or vice versa in order to fulfill the expanded user requirements.

1.2. Purpose

With the support of the promising persistence technologies and kinds of persistent store (as mentioned above), this thesis work carried out the design and implementation of an innovative persistence solution as the alternative for the legacy one in order to improve the system performance with respect to the expanded requirements. Particularly, the innovative persistence solution utilizes JPA (version 1.0) combined with the open source persistence framework Hibernate [42] and operates based on the RDBMS (particularly, Java DB). In the scope of this work, the system performance only focuses on the speed of data storage and retrieval.

Furthermore, via the result of this thesis work, some important questions are to be answered. The first question is whether only standard JPA mapping annotations can handle the mapping of a highly complex enterprise reference model to the relational database schema or not. The second question is that with the alternate persistence solution based on JPA and RDBMS, whether the system performance can really be improved or not.

1.3. Organization of the thesis

Chapter 2 (**Basics**) introduces and describes briefly about the important aspects and features of the Topgallant® Suite and the open source frameworks used throughout the design, realization and implementation. Chapter 3 (**Analysis**) begins with the analysis of the addressed problem followed by the specification of the requirements. The last part in this chapter specifies the solution chosen from the potential solutions.

Chapter 4 (**Design, realization and implementation**) first describes how to carry out the migration of the persistent store based on LDAP-compliant directory server to RDBMS and then presents the design, realization and implementation of the compatible persistence API as well as the essential functionalities (including version control and management of secondary object storage/retrieval). Furthermore, at the end of the chapter, several misbehaviors or shortcomings of JPA encountered during the implementation are also specified.

In chapter 5 (**Evaluation**), the innovative persistence solution (based on JPA and RDBMS) is evaluated for the system performance characteristics (in terms of object storage/retrieval speed) in comparison with the legacy persistence solution.

The last chapter 6 (**Summary and conclusion**) begins with the summary of the works carried out, followed by the specification of the further works and finally gives some important conclusions as well as answers the questions specified in the part **Purpose** of the chapter 1 (**Introduction**).

2. Basics

Mainly based on reference from books and available sources, the purpose of this chapter is to introduce and describe **briefly** about the important aspects and features of the Topgallant® Suite and the open source frameworks used throughout the design, realization and implementation.

2.1. Topgallant® Suite

Topgallant® is the trademark of Atlantec Enterprise Solutions GmbH, an international company providing software development and IT consulting services for the marine industry.

2.1.1. Topgallant®Infrastructure

Topgallant®Infrastructure (depicted in **figure 2.1.1**) consists of the interoperable components each of which performs specific task. Particularly, data model from different sources is normalized and mapped by *Topgallant Adapters* [1] in order to provide standardized access to information. There are two types of adapters: *Publisher Adapters* and *Subscriber Adapters*.

The *Enterprise Reference Model* "is a catalogue of specific requirements of the marine industry. It identifies and describes all relevant types of industry business objects, such as organization, facilities, processes, and product domains".

One of the most important components is the *Information Server* [2] that is an "enterprise data directory server which identifies, retrieves, and manages virtually any data reference in an enterprise. It can provide and manage information about data in departments, projects, and enterprises, enabling users to search for data and identify its origin and status. Topgallant Information Server not only operates within a single enterprise but can also support a distributed project environment, which may include numerous companies, consultants, or agencies. It provides the required security and authorization mechanisms to ensure complete control of information ownership and access".

Another component is the *Application Server* [3] providing web-based access to the Topgallant® services.

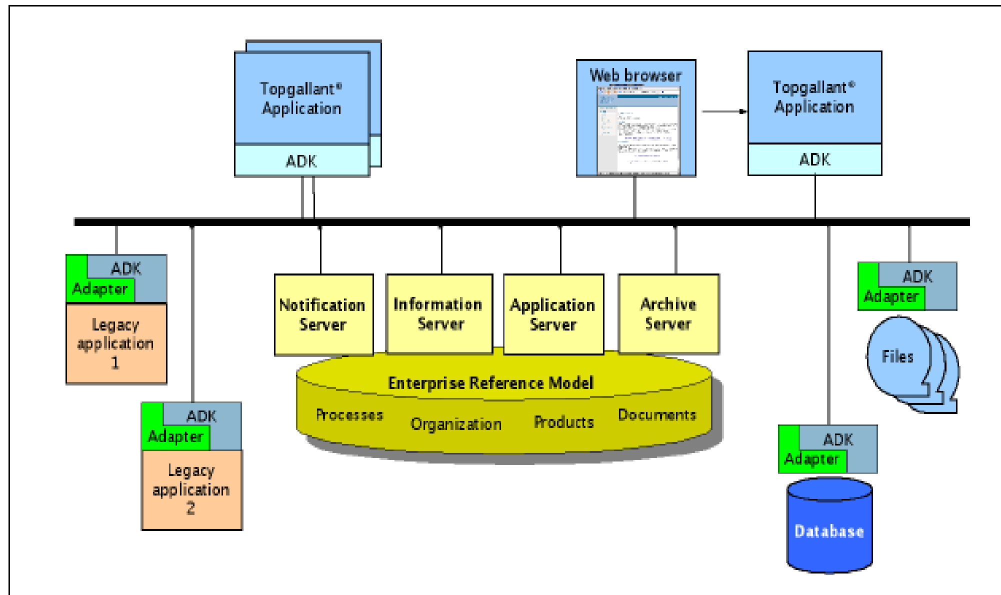

Figure 2.1. 1. Topgallant®Infrastructure. Source [4]

2.1.2. Legacy persistence solution

Most of the Topgallant® applications communicate with the information management component (particularly the *Information Server*) based on the data model called ERM (Enterprise Reference Model). The ERM data model is originally defined in the EXPRESS data description language which is a standard language for exchange of product model. The EXPRESS-based data model is made available to the Java applications via a code generation process which creates bean-style data access interfaces together with bean implementation classes.

The *Information Server*, which is based on LDAP-compliant directory server, provides:

- directory services and object retrieval as well as object storage handling
- "adapter client APIs for different operation modes"
- "mechanisms to create applications supporting a domain specific information model particularly Shipbuilding Product and Process Information (SPP)"

In the Topgallant® ERM data model, objects are classified into first-class category and secondary-class category. The first-class category includes the inter-connected information objects containing [5]: unique identification, type(s), origin and location, creator, timestamp/version, parental relationships and dependencies (to other objects through reference). The secondary-class objects are the ones that do not belong to the first-class category

The *Information Server* acts as the persistent store in which only first-class objects are stored directly. The stored first-class objects act as the containers of the associated secondary-class objects.

To sum up, the legacy persistent store, which is the *Information Server*, is based on LDAP-compliant directory server with LDBM [39] or BDB [40] (Berkeley DB) backend database. The legacy persistence API implementation is based on Java Data Objects (JDO) API [6], a standard persistence interface, which can be used to store Java objects into persistent store.

According to the Atlantec specification, the main characteristics [7] of the current implementation were:

1. "Client-side platform independence through Java
2. Bean-style data access interface
3. Well-documented network protocol with good tool support
4. Good performance for hierarchical data structures
5. Application controllable clustering of secondary objects with first-class objects
6. Direct support of an XML representation of the data
7. Handling of large amounts of data (hundreds of thousands to millions of first-class objects, many millions of secondary objects)
8. Versioned object storage
9. Good query performance on large data sets using query language
10. Large binary objects and document attachment archiving support"

2.2. Java Persistence API (JPA)

JPA [18], which was developed as part of JSR-220 (EJB 3.0), is a "POJO[1] persistence API for Object-Relational Mapping (ORM)[2] to manage relational data in enterprise beans, web components and application clients. It contains a full object-relational mapping specification supporting the use of Java language metadata annotations and/or XML descriptors to define the mapping between Java objects and a relational database".

JPA (defined in the package *javax.persistence*) is a set of interfaces and thus requires some particular implementation. There are a number of JPA implementors providing their specific ORM frameworks from open source (such as Hibernate [42], JPOX [43], OpenJPA [44], TopLink Essentials [45] and so on) to commercial (such as CocoBase JPA [46], SAP JPA [47]).

The main basics of JPA are systematically summarized as followings.

2.2.1. Elements

<u>ORM metadata</u> [24]:

- Specify the mapping between classes and tables, properties and columns, associations and foreign keys, Java types and SQL types and so on.
- *XML-based metadata*: mapping documents are written in and with XML.

[1] Plain Old Java Object: Java objects require no special treatment to be stored. POJO classes do not implement any framework-specific interfaces [41]

[2] ORM "is the automated (and transparent) persistence of Java objects to the tables in a relational database, using metadata that describes the mapping between the objects and the database" [24]

- *Annotation-based metadata (or shortly annotation)*:
    - Annotations are only available in JDK 5.0 or above.
    - Annotations are prefixed by the @ symbol and can take properties (in brackets after the name, comma-separated).
    - Must import "javax.persistence.XXX" where XXX is the annotation name of a JPA annotation.
- Example:

```
@Entity
@PrimaryKeyJoinColumn(name="Plate_SteelPart_ID")
@Table(name="plate")
public class PlateBean extends SteelPartBean
        implements Plate
{
    private Double  surfaceArea;

    @OneToOne(cascade = CascadeType.PERSIST)
    @JoinColumn( name = "mouldedSurface_ID", nullable = false )
    private InformationObjectBean  mouldedSurface;

    …
}
```

Key interfaces (defined in the *javax.persistence* package) [24]:

- *EntityManagerFactory*: an entity manager factory provides entity manager instances which are configured to connect to the same database.
- *EntityManager*: the API used to access a database in a particular unit of work. When *EntityManager* is used without an EJB container, transactions and bootstrapping must be handled by the application code [23]. Two main kinds of *EntityManager*:
    - Container-managed entity manager
    - Application-managed entity manager
- *Query*: interface used to control query execution.
- *EntityTransaction*: the interface used to control resource transactions on resource-local entity managers.

Terminologies [24]:

- Persistence context:
    - A cache of persistent entity instances.
    - No persistence context propagation is defined in JPA, if the application handles the *EntityManager* on its own in J2SE.
- Persistence manager: provides the following services:
    - Basic CRUD (Create, Retrieve, Update, Delete) operations.
    - Query execution.
    - Transaction control
    - Persistence context management.
- Persistence unit:
    - XML-based configuration file of the *EntityManagerFactory*.

  - Possible to specify multiple persistence units in a single configuration file.
  - Each persistence unit, which is identified by unique name, **mainly** includes:
    - Internal specification (e.g. by the server side) such as *Persistence provider* (e.g. Hibernate, JPOX …), *Specific task* (e.g. creating/validating database schema or connecting to database), *Database driver.*
    - Specification which will later be over-written (e.g. by the client side) such as database connection parameters (url, user name, password).

  - Can be further configured with an arbitrary number of properties which are vendor-specific.
- Object Relational Mapping (ORM):
  - ORM [15] "is a programming technique for converting data between incompatible type systems in relational databases and object-oriented programming languages".
  - Object-relational impedance mismatch [26]:
    - A **mismatch** exists between procedural Java and declarative SQL.
    - Objects can have one-to-many and many-to-many associations with other objects. Unfortunately, relational schema normalization does not allow a column to have multiple values.
    - Relational schemas do not support inheritance.
    - Object models do not support transaction semantics.
  - Fortunately, most ORM frameworks help solve those complexities.

Entity states [25]: an entity instance is in one of the following states:

- *New (or transient)*: an entity is new if it has just been instantiated using the new operator, and it is not associated with a persistence context. It has no persistent representation in the database and no identifier value has been assigned.
- *Managed (persistent)*: a managed entity instance is an instance with a persistent identity that is currently associated with a persistence context.
- *Detached*: if the entity instance (with a persistent identity) is no longer associated with a persistence context, usually because the persistence context was closed or the instance was evicted from the context.
- *Removed*: a removed entity instance is an instance with a persistent identity, associated with a persistence context, but scheduled for removal from the database.

Java Persistence Query Language (JPQL) [13]: is used to make queries against entities stored in a relational database. Queries of JPQL are similar to standard SQL queries in syntax, but operate against entity objects rather than directly with database tables. Therefore, the burden of writing complicated SQL commands is really released. The key interface used to control query execution is the *Query* interface.

2.2.2. JPA main functionalities

Some of the main functionalities of JPA are for example connecting to database (**figure 2.2.1**); storing object into database (**figure 2.2.2**) and creating/executing query (**figure 2.2.3**) are illustrated. Particularly, the database establishment and connection is handled by the *EntityManagerFactory* which provides *EntityManager* instances configured to connect to the same database. After that,

the *EntityManager* instance is used as database access handle which can for example store some object into database. One should note that, all resource transactions on the *EntityManager* must be controlled by the *EntityTransaction*. Furthermore, the query execution is controlled by the *Query* instance obtained from the *EntityManager* instance.

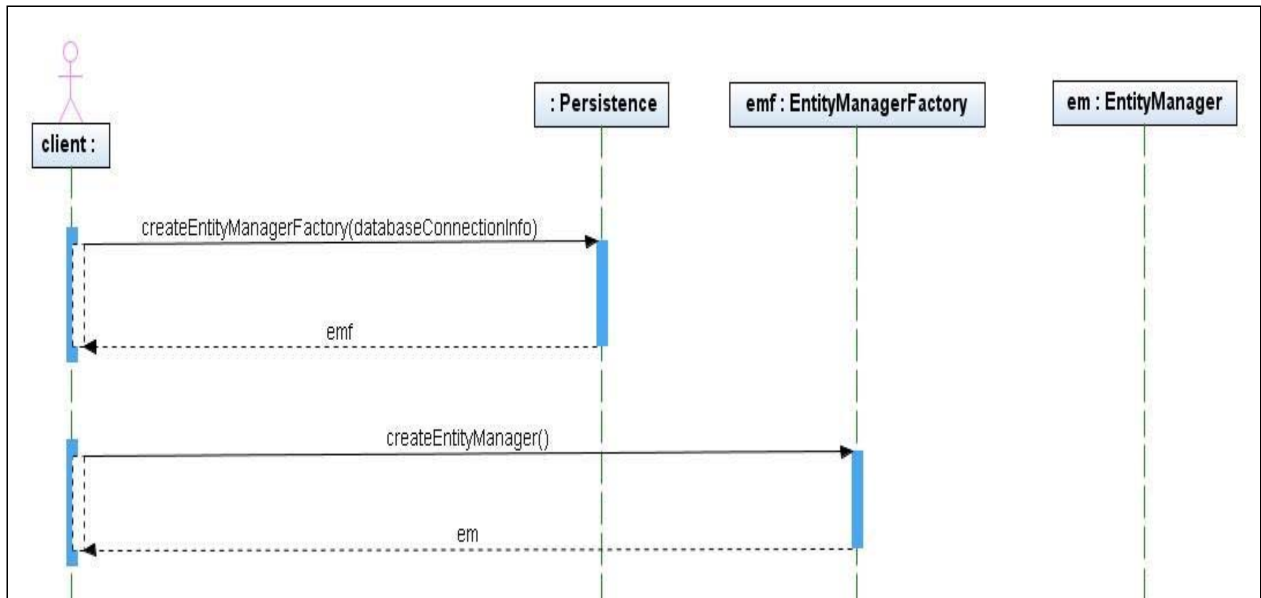

Figure 2.2. 1. Sequence diagram for connecting to database

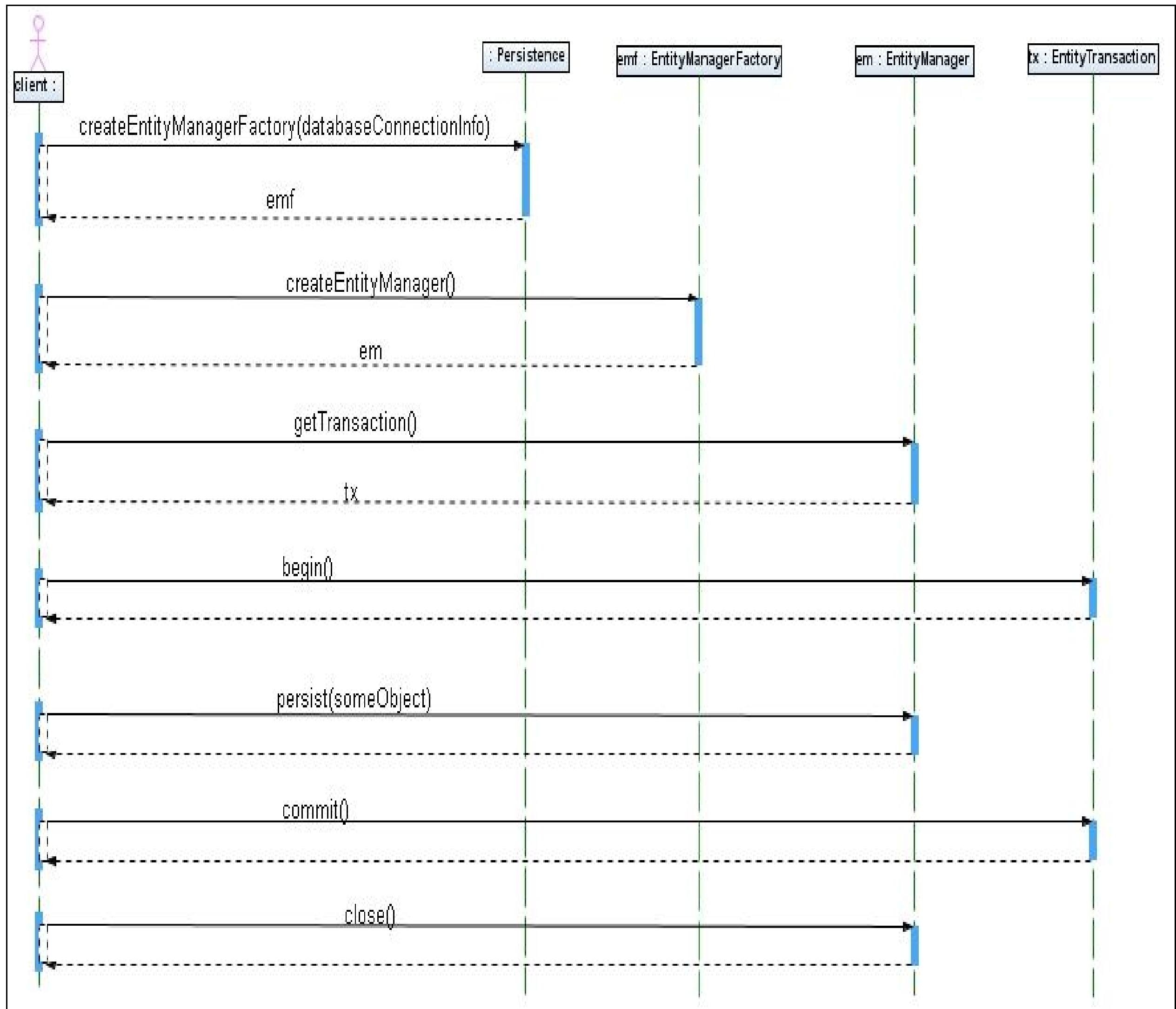

Figure 2.2. 2. Sequence diagram for storing object into database

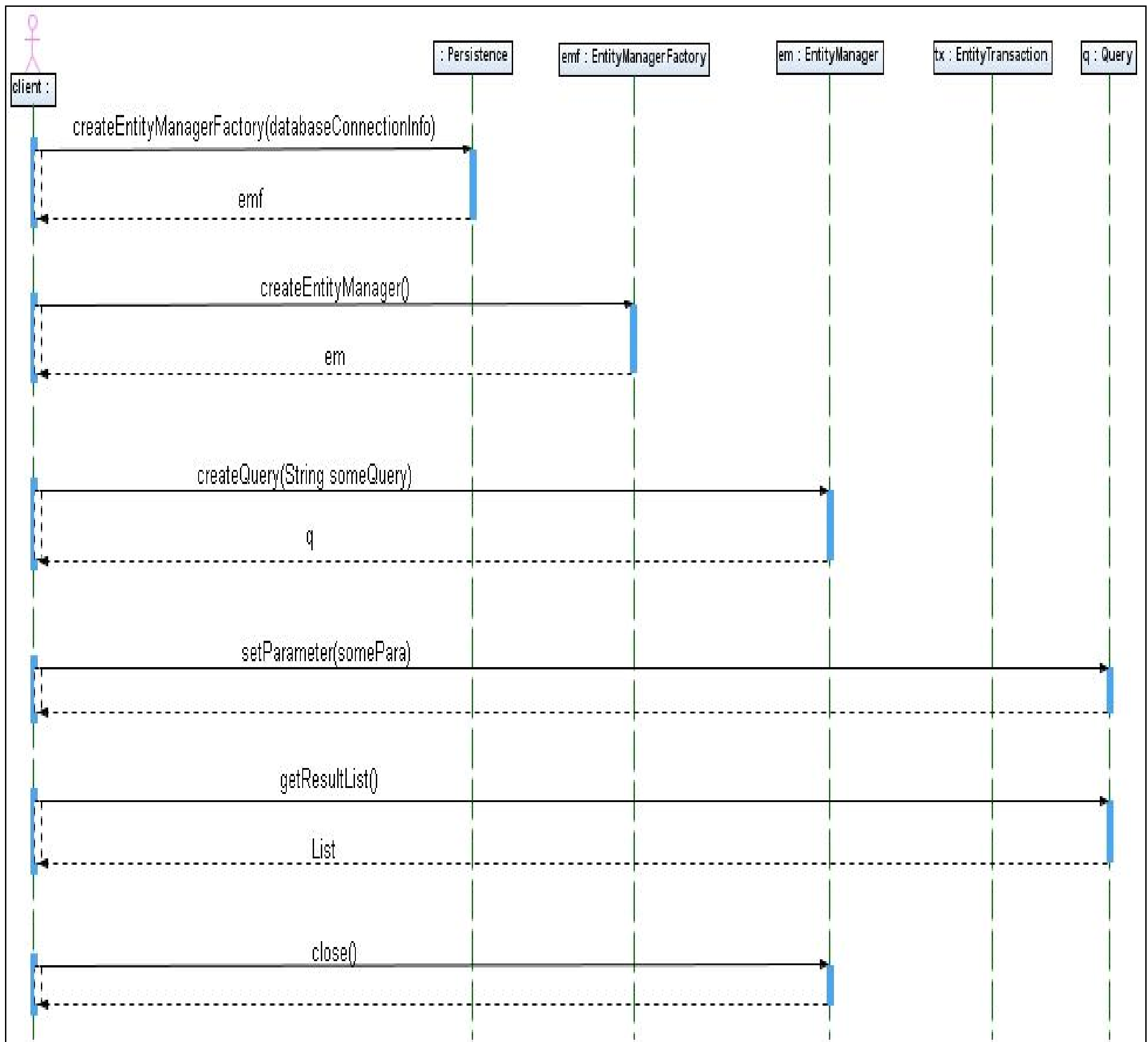

Figure 2.2. 3. Sequence diagram for creating and executing query

2.2.3. Standard JPA annotations

This part presents **briefly** how to use standard JPA annotations for object-relational mapping. In fact, JPA annotations are used to define how to map Java classes[3] to relational database tables. An annotation [27] is a simple, expressive means of decorating Java source code with metadata that is compiled into the corresponding Java class files for interpretation at runtime by the ORM engine to manage JPA behavior.

A Java class is mostly specified with the annotation *@Entity* to become a persistent entity which will be mapped to some table in the database. Annotations can be specified either right above the class attributes (preferred) or right above the class getter method. If a Java class is specified with the annotation, which is not the *@Entity*, that class will not be mapped to a single table. The **figure 2.1.4** shows the one-way mapping from annotated Java class to database table.

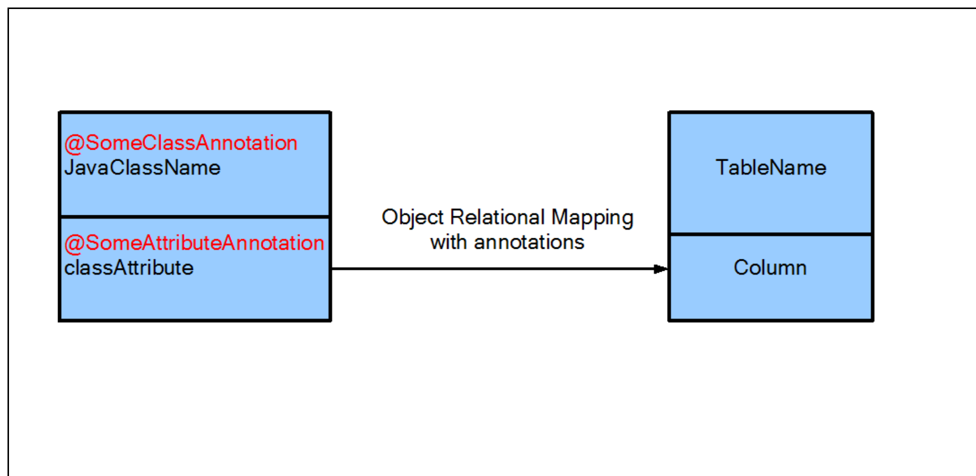

Figure 2.2. 4. ORM with annotations

JPA annotations can be classified into the following **categories** [27]:

- Entity:
  - *@Entity*
  - Usage: to designate a Java class as a JPA entity to be eligible for JPA services

- Database Schema Attributes:
  - *@Table, @Column*, …
  - Usage: to override default behavior and fine-tune the relationship between object model and data model

- Identity:
  - *@Id, @GeneratedValue*, …
  - Usage: to fine-tune how database maintains the identity of the entities

[3] ORM requires the Java class to be a POJO class.

- Direct Mappings:
  - *@Basic, @Enumerated, …*
  - Usage: to fine-tune how database implements the basic mappings for most Java primitive types, wrappers of the primitive types, and enums.

- Relationship Mappings:
  - *@OneToOne, @ManyToOne, @OneToMany, @ManyToMany*, …
  - Usage: to specify the type and characteristics of entity relationships to fine-tune how database implements the relationships.

- Composition:
  - *@Embeddable, @Embedded, @AttributeOverride*, …
  - Usage: to specify objects that are embedded and to override how they are mapped in the owning entity's table.

- Inheritance:
  - *@Inheritance, @DiscriminatorColumn, @DiscriminatorValue*, …
  - Usage: if the entity class inherits some or all persistent fields from one or more super classes.

- Locking:
  - *@Version*
  - Usage: to enable JPA-managed optimistic locking.

- Lifecycle Callback Events:
  - *@PrePersist, @PostPersist, @PreRemove, ...*
  - Usage: to associate methods with JPA lifecycle events if one needs to invoke custom logic at any point during the entity lifecycle.

- Entity Manager:
  - *@PersistenceUnit, @PersistenceContext, ...*
  - Usage: to declare or inject an entity manager or entity manager factory.

- Queries:
  - *@NamedQuery, @ColumnResult, @SqlResultSetMapping, ...*
  - Usage: to pre-define queries and manage their result sets.

In addition to the standard JPA annotations, each JPA vendor with its specific ORM implementation also offers an extended set of annotations for handling advanced mapping situations flexibly.

2.2.4. JPA callback

JPA callback mechanism is enabled via a set of lifecycle event annotations used to define the callback methods executed before or after some lifecycle event. The lifecycle events can be persisting, loading, removing, updating of the correspondent persistent entity.

Callback methods can be defined directly in the entity class or indirectly in separate listener class which then can be associated with the entity class. It is important to note that if the parent entity class is associated with some listener class, all of its sub-classes are also associated with that listener class.

List of lifecycle event annotations specified right above the callback methods:

```
@PrePersist,  @PostPersist,@PreRemove,@PostRemove,  @PreUpdate,  @PostUpdate,
@PostLoad.
```

Listener class is a normal Java class implementing only callback methods marked with the above lifecycle event annotations. One listener class or several listener classes can be associated with the entity class:

```
@Entity
@EntityListeners({EventListener1.class, EventListener2.class})
public class SomeEntityClass
```

If the entity class is associated with some listener class and the entity class itself also implements its own callback methods, callback methods of the listener class will be executed first and then come the callback methods of the entity class.

Comparison between callback method signatures in listener class and entity class:

- In listener class:

```
@SomeLifeCycleAnnotation
// Callback methods implemented in a listener class must
// return void and take one argument.
// The passed argument "Object entity" is the instance of
// the correspondent entity class associated with this
// listener class.
public void callbackMethod(Object entity)
{

}
```

- In entity class:

```
@SomeLifeCycleAnnotation
// Callback methods implemented in an entity class must
// return void and take no argument.
public void callbackMethod()
{

}
```

2.3. Java Architecture for XML Binding (JAXB)

JAXB [28] provides an API and tool that allow automatic mapping between Java classes and XML representations. JAXB provides two main features (illustrated in the **figure 2.3.1**): the ability to marshal Java objects into XML and the inverse, i.e. to un-marshal XML back into Java objects.

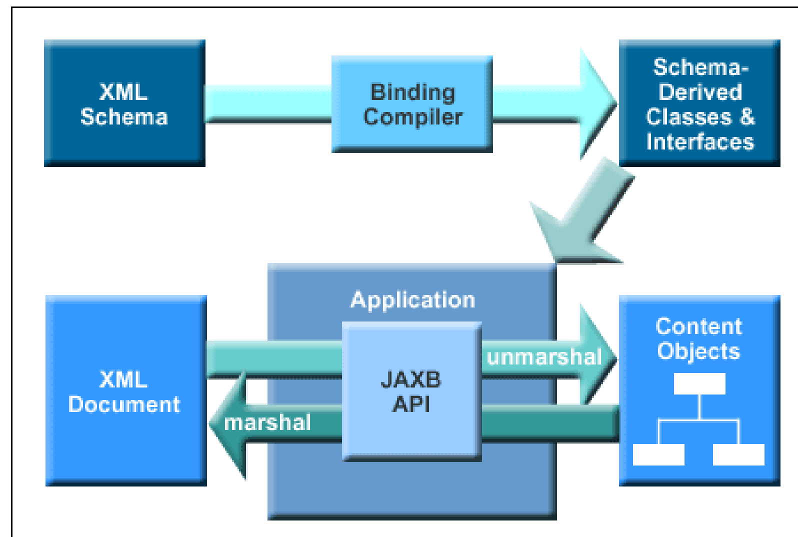

Figure 2.3. 1. JAXB main features. Source [29]

JAXB [32] consists of two parts. First, JAXB contains a **binding compiler** that reads an XML schema and produces the equivalent Java object model (schema-derived classes and interfaces). This generated object model captures the structure of XML better than general-purpose APIs like DOM (Document Object Model) or SAX (Simple API for XML), making it a lot easier to manipulate XML content.

The second part is the **JAXB API** (defined in the *javax.xml.bind* package), through which applications communicate with generated code. This API hides provider-specific implementation code from applications and also provides a uniform way to do basic operations, such as marshalling or un-marshalling.

The main advantage of JAXB is that JAXB [28] allows storing and retrieving data in memory in any XML format, without the need to implement a specific set of XML loading and saving routines for the program's class structure. Therefore, JAXB can make it **easier** to access XML documents from Java-based applications.

In addition, JAXB is designed to meet the following goals [33]:

- **Customizable**: JAXB provides a standard way to customize the binding of existing schema components to Java representations. Sophisticated applications sometimes require fine control over the structure and content of schema-derived classes, both for their own purposes and for keeping pace with schema evolution.

- **Portable**: it is possible write applications implementing JAXB in such a way that the JAXB components can be replaced without having to make significant changes to the rest of the source code.
- **Support validation on demand**: while working with a content tree corresponding to an XML document, it is often necessary to validate the tree against the constraints in the source schema. It should be possible to do this at any time, without the user having to first marshal the tree into XML.
- **Provide clean "round-tripping"**: transforming a Java content tree to XML content and back to Java content again should result in equivalent Java content trees before and after the transformation.

However, from the **figure 2.3.1** above, it is clear to see one disadvantage with JAXB binding is that the generated Java classes are coupled to the schema, which makes the JAXB binding approach harder for writing generic code [34]. Change in schema requires new generation of classes; this may require adaptation of application logic.

2.4. Java DB (Apache Derby)

Java DB (or shortly Derby) [30], as a lightweight database management system written completely in Java, is Sun's supported distribution of Apache Derby (from version 10.2) and not only bundled in Sun JDK 6 but also widely supported (e.g. by NetBeans 6).

Apache Derby [31] is the core technology of Java DB. Derby's database engine is a fully functioning relational embedded database engine. JDBC and SQL are the main programming APIs. Another core component that supplements the Java DB is the Derby network server. The network server extends the reach of the Derby database engine by providing traditional client server functionality.

Derby has a number of advantages [30]:

- Pure Java:
    - Write Once Run Anywhere
    - Single binary runs independently of operating systems.
    - Database on-disk format is platform independent.
- Complete relational database engine: fully transactional, secure, easy-to-use, standards-based (SQL92/99/2003, JDBC API, and Java EE)
- Java DB supports two running modes:
    - Embeddable Database (with driver org.apache.derby.jdbc.EmbeddedDriver):
        - Database only accessible from a single JVM
        - May have multiple applications per JVM (e.g. application server)
        - Easy to use, fast, zero administration
        - Can be embedded in client applications such as desktop, browser, mobile
    - Client/Server Database (with driver *org.apache.derby.jdbc.ClientDriver*):
        - Multi-threaded
        - Row-level locking
        - Secure
- Easy to use, zero maintenance, mature and robust

- Small footprint (2MB)
- Comparable performance to MySQL and PostgreSQL.

However, the main disadvantage of Derby is that it can not run as server under the embedded mode. In fact, when an application accesses the Derby database using the embedded Derby JDBC driver, the Derby engine does not run in a separate process, and there are no separate database processes to start up and shut down. Instead, the Derby database engine runs inside the same JVM as the application; the Derby actually becomes part of the application just like any other jar file that the application uses. Therefore, multiple applications (each of them accesses the Derby database using embedded Derby JDBC driver) can not access the same database.

3. Analysis

This chapter begins with the analysis of the addressed problem followed by the specification of the requirements. The last part in this chapter specifies the solution chosen from the potential solutions.

3.1. Problem

The legacy persistence API[4], which is implemented using JDO persistence interface and operates based on the LDAP-compliant directory server, was measured for the application performance characteristics in terms of object storage/retrieval speed with vast volume of data. The measurement results showed that the object writing speed to LDAP-compliant repository does not satisfy the expanded requirements especially when many users perform object storing (with vast volume) at the same time. This is because LDAP servers are highly optimized for reading operations over writing operations.

Previously, most of Topgallant® applications are not required to perform storing of large amount of objects very frequently. They mainly retrieve large volume of data from LDAP-compliant repository and perform storing of little amount of data. As the user requirements are expanded, some Topgallant® interactive applications are asked to store an increasing large amount of data as fast as possible.

It is planned to improve the application performance in terms of object storage/retrieval speed. Particularly, the performance improvement is first experimented with the migration of the current persistent store, which is mainly based on the LDAP-compliant directory server, to the relational database server with the hope that the object storage/retrieval speed can be enhanced on the base of RDBMS. Because the Topgallant® interactive applications are often required to store/retrieve large amount of data (hundreds of thousands to millions of first-class objects, many millions of secondary objects), the consumed duration of storing/retrieving is one of the main factors which influence the application performance.

3.2. Requirements

To make it possible to migrate the persistent store from LDAP-compliant directory server component to relational database server component, the relational database schema, which is compatible to the Java classes created by the code generation process from the ERM data model, is absolutely vital.

As the software productivity requirement for the alternative persistence solution based on the relational database server, the legacy persistence API implementation has to be replaced with a comparable persistence API implemented in the standard JPA compliant way. Furthermore, the

[4] Described in **part 2.1.2**

JPA-based persistence API implementation must be compatible to the working mechanism of the legacy applications and must also adopt the essential functionalities from the legacy persistence API. The two most essential functionalities are version control and management of secondary object storage/retrieval.

The compatibility to the legacy applications is expressed via the mechanisms of:

- Object instantiation (not simply with the ***new*** operator)
- Created objects management
- Persistence and Query service

It is necessary to note that, the comparable persistence API must be implemented in JPA-compliant way so that it is possible to switch between various JPA implementors in order to possibly achieve the highest performance. In addition, the persistence API to be implemented should try to simplify the current mechanisms of object instantiation and created objects management.

Last but not least, the necessary changes to the working mechanism of the existing applications, which are due to the migration to the new base of RDBMS, must be minimal.

3.3. Solutions

According to the requirements specified above, there are several different possibilities which potentially can be used to solve the problem. They are mainly around different kinds of database and the various persistence-supporting systems.

The paper of *A Comparative Study of Persistence Mechanisms for the Java™ Platform* [PM04] has specified the available systems providing persistence mechanisms (for managing persistent data) which are compared based on a set of criteria such as orthogonality, persistence independence, reusability, performance, scalability and so on.

The various persistence-supporting systems and kinds of database are briefly analyzed as followings in order to justify the chosen solution.

3.3.1. Persistence-supporting systems

The popular systems (providing persistence mechanisms) to be analyzed are Java Object Serialization (JOS), JavaBeans Persistence (JBP), Orthogonal Persistence (OPJ), Java Database Connectivity (JDBC), Java Data Objects (JDO), Enterprise JavaBeans (EJB) and Java Persistence API (JPA). The following analysis is mainly the extraction from the specified references.

- Java Object Serialization (JOS) [8]
    - <u>Mechanism</u>: Java's built-in mechanism for manipulating object as byte stream. Serialization is the process that encodes the object as well as other associated objects into a byte stream. The serialization process encodes enough information about the object type within the byte stream, and thus enables the original object

to be reconstructed by the deserialization process. Only objects of the classes implementing the *java.io.Serializable* interface can be serialized / deserialized.

- Application: JOS can be used to implement lightweight persistence of Java objects to file or database as a BLOB (Binary Large Object). It can also be used for communication via sockets or Remote Method Invocation (RMI).

- Disadvantage:
  - Significant overhead with large-sized objects.
  - Serialization does not offer any transaction control mechanisms
  - A serialized network of interconnected objects can only be accessed as a whole.

- JavaBeans Persistence (JBP) [PM04]
  - Mechanism: a JavaBean is required to expose its public state via "getter" and "setter" methods, thus it suffices to record the values of these properties, as returned by the "getter" methods, in the persistent state, confident that an equivalent object can be created later by calling the associated "setter" methods. The mechanism is based on serialization/deserialization.

  - Advantage: compared to JOS which takes an implementation-oriented approach to persistence, JBP takes an interface-oriented approach.

  - Disadvantage: classes must follow JavaBeans conventions.

- Orthogonal Persistence (OPJ) [9]
  - Mechanism: provides persistence for the full computational model specified by the Java Language Specification (JLS). Persistence is defined as the ability for the computation state to survive in stable storage, across multiple executions of a Java Virtual Machine and in the face of system and application failure.

  - Application: maintain the illusion of a continuously executing Java program, in the face of planned and unplanned system shutdowns.

  - Disadvantage:
    - All objects are treated uniformly regardless of type or longevity.
    - Lack of a high-level view of the persistent store.
    - Inefficiency

- Java Database Connectivity (JDBC) [10]
  - Overview: JDBC API is the industry standard for database-independent connectivity between the Java applications and a wide range of databases. The JDBC API provides a call-level API for SQL-based database access. JDBC API makes it possible to do three things:
    - Establish a connection with a database or access any tabular data source.
    - Send SQL statements.
    - Process the results.

- Advantage:
  - Leverage existing enterprise data
  - Simplified enterprise development
  - Zero configuration for network computers

- Disadvantage:
  - Object-relational mismatch: developer has to write code to map an object model's data representation to a relational data model and its corresponding database schema.
  - No transparent persistence support: automatic mapping of Java objects with database tables and vice versa is not supported.
  - Support only native Structured Query Language (SQL): developer has to find out the efficient way to access database.
  - Database dependent code
  - Maintenance cost
  - Caching is maintained by hand-coding
  - No automatic versioning and time stamping support
  - Can not be scaled easily

- Java Data Objects (JDO) [11]
  - Overview: JDO API provides a standard approach for achieving object persistence in Java technology by using a combination of XML metadata and byte code enhancement to ease the development complexity and overhead. JDO does not define the type of data store: possible to use the same interface to persist Java technology objects to a relational database, an object database, XML, or any data store.

  - Advantage:
    - Portability
    - Transparent database access
    - High performance
    - Integration with EJB

- Enterprise JavaBeans (EJB) [12]
  - Overview: EJB specification defines architecture for the development and deployment of transactional, distributed object applications-based, server-side software components. Organizations can build their own components or purchase components from third-party vendors. These server-side components, called enterprise beans, are distributed objects that are hosted in Enterprise JavaBean containers and provide remote services for clients distributed throughout the network. The entity beans in EJB are "intended to correspond to persistent data, typically a row in a relational database table and have strong availability guarantees in the face of system failures".

  - Benefits to the application developer:
    - Simplicity
    - Application portability
    - Component reusability

  - Ability to build complex applications
  - Separation of business logic from presentation logic
  - Deployment in many operating environments
  - Distributed deployment
  - Application interoperability
  - Integration with non-Java systems
  - Educational resources and development tools

- Benefits to Customers:
  - Choice of the server
  - Facilitation of application management
  - Integration with a customer's existing applications and data
  - Application security

- Disadvantages:
  - Large, complicated specification
  - Increased development time
  - Added complexity compared to straight Java classes
  - Potential to produce a more complex and costly solution than is necessary
  - Continual specification revisions

- Java Persistence API (JPA)[5]
  - Advantage:
    - The most advantage of JPA is vendor neutral because JPA is a generic ORM specification utilizing other ORM technologies available in the industry. Therefore, if the persistence layer (including persistence API implementation and persistent data model) is implemented using only standard JPA annotations and APIs, it is possible to switch to alternative JPA implementor without any modification of the persistence layer in order to experiment the system performance. However, this also results in the disadvantage that is the extended set of annotations and interfaces implemented by the JPA vendor can not be utilized. In fact, each JPA vendor not only implements the standard JPA specification but also provides their own powerful persistence API implementations.

    - The other advantage is that JPA is the stand-alone specification which can be used not only within Java SE environments but also within Java EE. Therefore, JPA is a lightweight and standard persistence API.

  - Disadvantage: JPA requires Java 5 or higher because it relies on the new Java language features such as annotations and generics. Furthermore, there are several interesting omissions from the JPA API that need to be explored such as [21]:
    - Batching support: no explicit control over the batch size and cascading save/delete.
    - Caching: no explicit control over when to use caching, how to invalidate cached data and how this works in a clustered environment.

[5] Further detail about JPA is described in **part 2.2**

- Entity Listeners: the JPA Entity Listeners give developers the ability to enhance the Insert Update and Delete behavior. However, they do not give developers the ability to replace the Insert Update Delete behavior.
- No explicit access to the underlying *java.sql.Connection*.
- No support for creating a transaction with an explicit transaction isolation level.

From the brief analysis above, the persistence-supporting systems of Java Object Serialization (JOS), JavaBeans Persistence (JBP) and Orthogonal Persistence (OPJ) are not suitable due to the significant disadvantages as specified. Therefore, the potentially-appropriate systems are now among Java Database Connectivity (JDBC), Java Data Objects (JDO), Enterprise JavaBeans (EJB) and Java Persistence API (JPA).

As mentioned in the **part 2.1.2**, the legacy persistence API implementation is already based on JDO which results in the object storage speed not as expected. Thus, the choice of systems is now limited to only three candidates that are JDBC, EJB and JPA. It is clear to see that JDBC exposes many disadvantages (as specified above) in which the most disadvantage is the object-relational mismatch.

The candidate EJB seems to be the promising persistence-supporting system. Nevertheless, because EJB is a large and complicated specification which potentially can lead to a more complex and costly solution, the bulky EJB specification is not selected. Instead, the standard light-weight **JPA** (which was defined as part of the EJB 3.0 specifications) is finally employed for experiment as the alternative persistence mechanism despite several disadvantages of it. This decision is made mainly based on the promising advantages of JPA. In addition, JPA is designated to incorporate the ORM benefits from various JPA implementors with which JPA might be coupled.

Indeed, the significant advantage of ORM is the transparent persistence (primary goal of any ORM solution) [19] which is the ability to directly manipulate data stored in a relational database using an object-oriented programming language. With transparent persistence, the manipulation and traversal of persistent objects is performed directly by the object programming language in the same manner as in-memory, non-persistent objects. This is achieved through the use of intelligent caching.

One can easily see several benefits of transparent persistence:

- Easier for developers: ORM frameworks [15] handle the complexity of translating objects to forms which can be stored in the database, and which can later be retrieved easily, while preserving the properties of the objects and their relationships.
- The need for SQL expertise is reduced: [13] database queries of ORM framework resemble SQL queries in syntax, but operate against entity objects rather than directly with database tables.

However, a transparent persistence API [20] hides the persistence completely compared to the non-transparent persistence API which offers a lot of control to the user of the API.

So far, it is clear why JPA is chosen as the alternative persistence mechanism. Moreover, it is important to note that JPA defines only **interfaces** for interacting with the persistence provider

and for mapping entities to database. Therefore, JPA must be used together with some JPA implementor. Among the various JPA implementors[6] including open source and commercial, Hibernate as an open source persistence framework is first selected due to its popularity and industry standard.

The following part focuses on analyzing relational database and object-oriented database.

3.3.2. Relational database vs. object-oriented database

From Sun ™ specification of JPA [14], it is important to note that JPA is a POJO[7] persistence API for object/relational mapping. It contains a full object/relational mapping specification supporting the use of Java language metadata annotations and/or XML descriptors to define the mapping between Java objects and a relational database (RDB). This means that JPA is not designated for using with object-oriented database (OODB)[8]. Therefore, it is not relevant to examine whether the migration of the current persistent store to OODB (instead of RDB) should be experimented or not because the intention is to make use of the JPA.

However, because of the fact that the ultimate goal is to improve the object storage/retrieval speed, it is worthy making a brief comparison between RDB and OODB under the context of data management in object-oriented programming. This is due to the worst case in which the application performance can not be enhanced by using RDB, and then one can think of using OODB as an alternative for further experiment.

Under the context of data management in object-oriented programming:

- RDB [15]
  - Overview: the most common type of database. RDB uses a series of tables to organize data. Data in different tables are associated through the use of declarative constraints, rather than explicit pointers or links. The same data that can be stored in a single object value would likely need to be stored across several of these tables.

  - Disadvantage:
    - *Object-Relational impedance mismatch*[9] between objects and data stored in RDB presents a number of challenges like performance, application maintainability and flexibility.
    - *ORM systems*: even though ORM systems (e.g. Hibernate) help solve Object-Relational mismatch problem, the translation layer of ORM systems can be slow and inefficient (notably in terms of the SQL it writes),

[6] Introduced at beginning of **part 2.2**
[7] Plain Old Java Object: Java objects require no special treatment to be stored. POJO classes do not implement any framework-specific interfaces [41]
[8] (From: http://www.jpox.org/docs/jpa/index.html) JPA is tightly coupled to RDBMS data stores and thus currently no use of other type of data store (such as XML, OODBMS, etc).
[9] Further detail is described in the JPA part of chapter 2

resulting in programs that are slower and use more memory than code written "by hand.".

  - Advantage: capability of query service (SQL queries)

- OODB [16]
  - Overview: information is represented in the form of objects as used in object-oriented programming.
  - Advantage: eliminate the need for converting data to and from its SQL form, as the data would be stored in its original object representation.
  - Disadvantage: lose the capability to create SQL queries even though most commercial object-oriented databases are able to process SQL queries to a limited extent. Therefore, object-SQL mapping system is additionally required.

One can conclude that advantage of RDB is the disadvantage of OODB and vice versa.

It is now clear why RDB is to be used. The subsequent question is which relational database management system (RDBMS) should be selected. Indeed, there are various RDBMS ranging from open source with mostly fully-featured RDBMS available (such as MySQL[10] and PostgreSQL[11]) to commercial (such as Oracle database[12], IBM's DB2[13] and Microsoft's SQL Server[14]). Most of the RDBMS are platform-dependent.

Cost is always a main factor in selection of RDBMS and because this is a research project which aims to make use of available open sources, the Java DB [17] (Sun's supported distribution of the open source Apache Derby) is selected due to the fact that Java DB is pure Java and thus platform-independent (for both software and stored data). More detail about Java DB is described in **part 2.4**.

---

[10] http://www.mysql.com/
[11] http://www.postgresql.org/
[12] http://www.oracle.com/database/index.html
[13] http://www-01.ibm.com/software/data/db2/
[14] http://www.microsoft.com/sqlserver/2008/en/us/default.aspx

3.3.3. Chosen solution

This part summarizes briefly about the solution (including the persistence mechanism and kind of database) chosen from the above analysis. Particularly, the migration of the persistent store from LDAP-compliant directory server to relational database server (particularly Java DB with the running mode of client/server database) is carried out by utilizing JPA (version 1.0) combined with Hibernate as an open source JPA implementor providing ORM engine and query service.

After the migration to the base of RDBMS, it is essential to create a prototype implementation of a persistence component that provides comparable data access capabilities as the legacy persistence API implementation. The prototype should provide access models in a JPA-compliant way and must adopt the essential functionalities from the legacy persistence API implementation.

The final step is to measure the system performance (in terms of object storage/retrieval speed) of the innovative persistence solution based on JPA and RDBMS and the measurement results will then be compared with the ones of the legacy persistence solution in order to find out whether the storage/retrieval speed can really be enhanced or not. In the worst case, i.e. the storage/retrieval speed is not improved or even decreased; another investigation on how to improve the system performance should be discussed.

4. Design, realization and implementation

This chapter first describes how to carry out the migration of the persistent store based on LDAP-compliant directory server to RDBMS and then presents the design, realization and implementation of the compatible persistence API as well as the essential functionalities (including version control and management of secondary object storage/retrieval). Furthermore, at the end of the chapter, several misbehaviors or shortcomings of JPA encountered during the implementation are also specified.

4.1. Migration to RDBMS

4.1.1. Introduction and requirement analysis

As described in the **part 3.1 (Problem)**, for the current persistent store which is the *Information Server* implemented based on LDAP-compliant directory server, the speed of writing of large amount of data to LDAP-compliant repository does not meet the expanded user requirements. Therefore, the system performance improvement (only regarding with the speed of object storage and retrieval) is planned with the first experiment of migrating the current persistent store to the base of RDBMS.

It is at first required that the alternative persistent store based on RDBMS must be comparable to the legacy one based on LDAP-compliant directory server. Particularly, the RDBMS-based persistent store must be able to support the basic persistence operations such as store/retrieve/update/delete[15] for only first-class objects[16] of the Java-based data model. This means that the secondary class objects are not allowed to store directly into the relational database[17]. The Java-based data model (including bean-style data access interfaces together with bean implementation classes) is generated from the XSD-based data model by the code generation process which is described briefly in the following part.

4.1.2. Solution

In order to sketch the solution of how to migrate from LDAP-compliant directory server base to RDBMS base, it is important to analyze the current generation process of the Java-based data model. From the **figure 4.1.1**, it is clear to see that the core of the generation process is the code generator which reads in the XSD-based data model, performs the processing[18] and then produces the Java-based data model (including bean-style data access interfaces together with bean implementation classes). In the scope of this work, it is not required and not necessary to analyze how and why the original data model (called ERM – Enterprise Reference Model) is defined in

---

[15] Other advanced persistence operations are not considered at this phase.
[16] Detail of object classification is described in the part **4.1.2.2** below
[17] More about management of secondary object storage/retrieval is presented in **part 4.3.2**
[18] Further detail is described in the following parts

the EXPRESS data description language as well as how the EXPRESS-based data model can be converted into the XSD-based data model.

To make it possible to migrate to RDBMS base, it is absolutely essential to create the relational database schema from the Java-based data model. Because it is required that the secondary class objects are not allowed to store directly into the relational database, all secondary classes must not be mapped to the relational database schema. The solution (illustrated in **figure 4.1.2**) is to modify the code generator in order to specify the appropriate JPA ORM annotations[19] directly in the implementation of the code generator so that the code generator can produce the Java-based data model decorated[20] with JPA annotations. The Java-based data model annotated with JPA mapping annotations can then be mapped to the relational database schema easily by the ORM engine (of the Hibernate framework). The following parts describe the analysis of the code generator and the Java-based data model before showing how to specify JPA annotations in the code generator implementation.

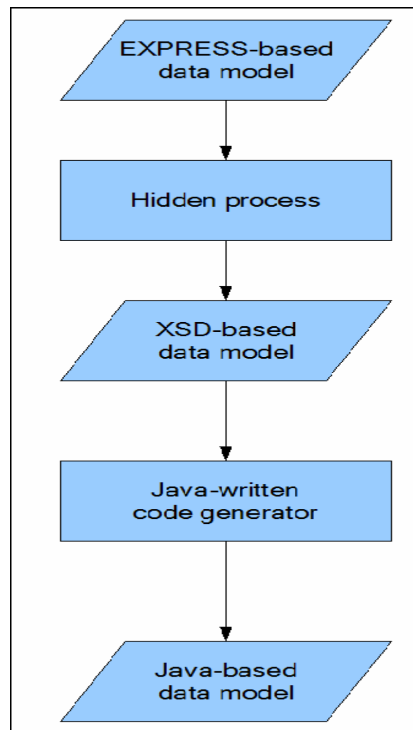

Figure 4.1. 1. Current generation process of Java-based data model

[19] JPA annotations description is in the part **2.2.3**

[20] Only the bean implementation classes (not the interfaces) are specified with JPA ORM annotations because JPA ORM annotations are not used with interfaces.

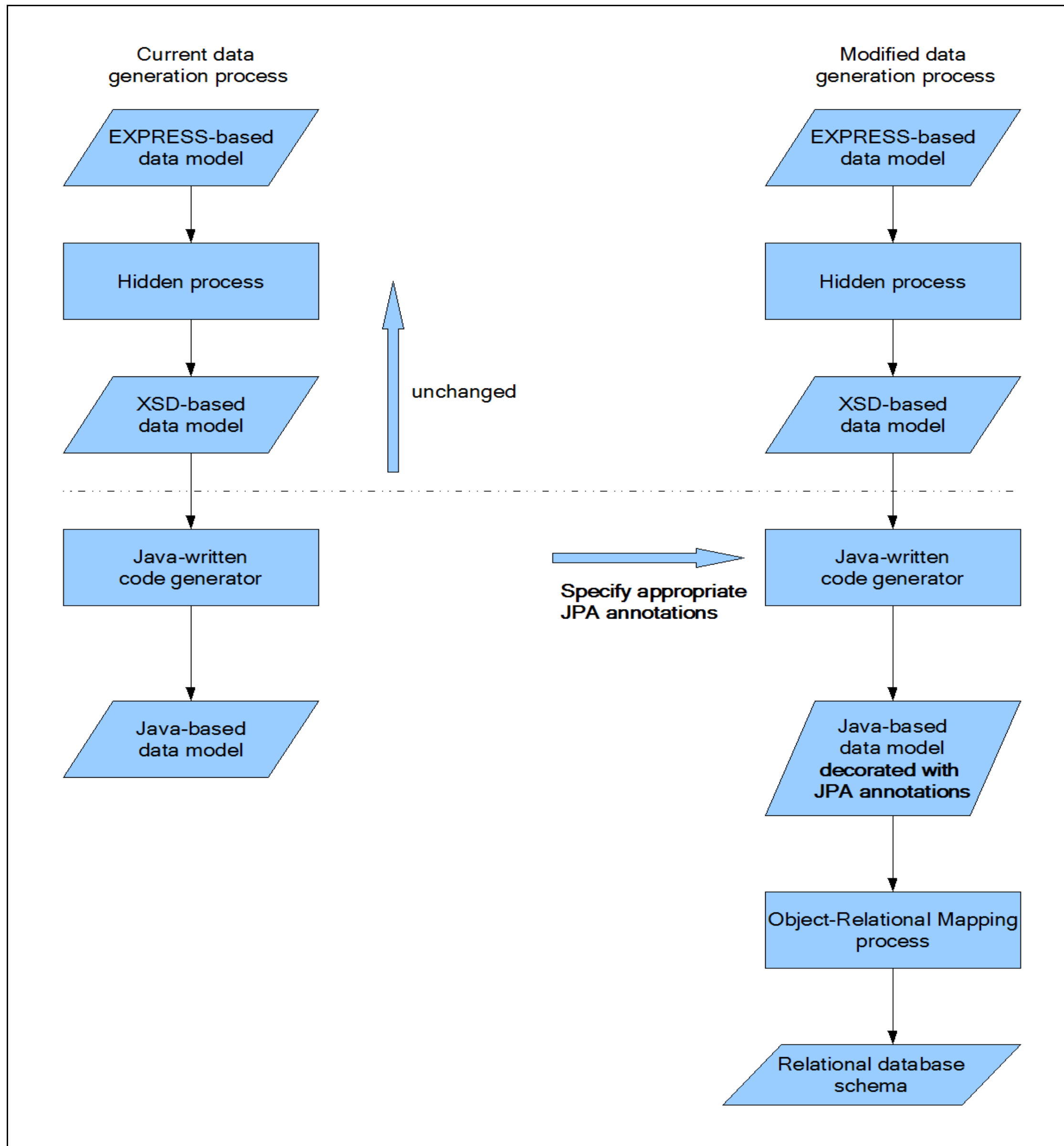

Figure 4.1. 2. Generation of relational database schema

4.1.2.1. Analysis of code generator

The legacy code generator working mechanism is depicted in the **figure 4.1.3**. Particularly, at the beginning some options have to be specified. For example, the option "-d" indicates the location or the package of the generated data model (interfaces and bean classes); the option "-i" signals that bean classes will be generated and if this option is disabled then only interfaces will be generated. One important note is that, the interfaces must be generated before generating the bean classes; the option "-p" is used to specify the package prefix name and the option "-file" specifies the location of the XSD-based data model.

The fact that how the XSD schema[21] (containing some hundreds of complex types) is converted into the collection of instances of *java.lang.Class* is not the scope of this work. Instead, the main focus lies at the *generateClass()* method (depicted in **figure 4.1.4**) which is the core of the code generator. In fact, the *generateClass()* method reads in and parses the instance of the *java.lang.Class* to construct a full Java class in String representation.

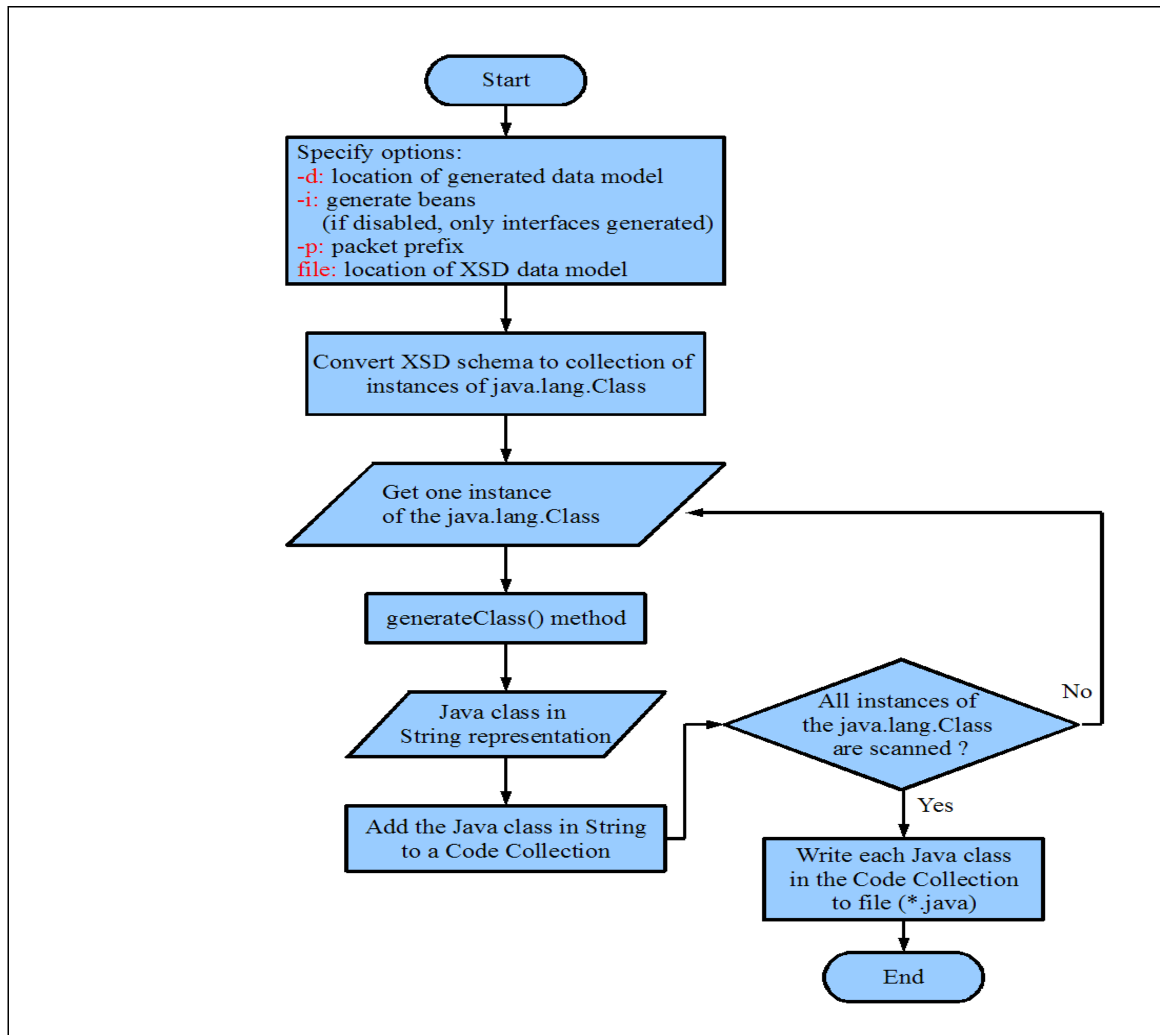

Figure 4.1. 3. Code generator working mechanism

From the **figure 4.1.4**, one can see that the *generateClass()* method can be used to create either the interfaces or the bean classes depending on the specified option "-i" mentioned above. One special notice is that the generated interfaces define only the signatures of the getter and setter methods in order to ensure the common data access interfaces. The generation of the Java class (in form of String representation) is a standard sequences. First, the class declaration is created, and then followed by the creation of the class body in which class attributes together with the

[21] Some portion of the complicated XSD file is shown in the **figure 4.1.5**

getter and setter methods are specified. During the Java class generation, the classes (user-defined or third party) in other packages are internally determined.

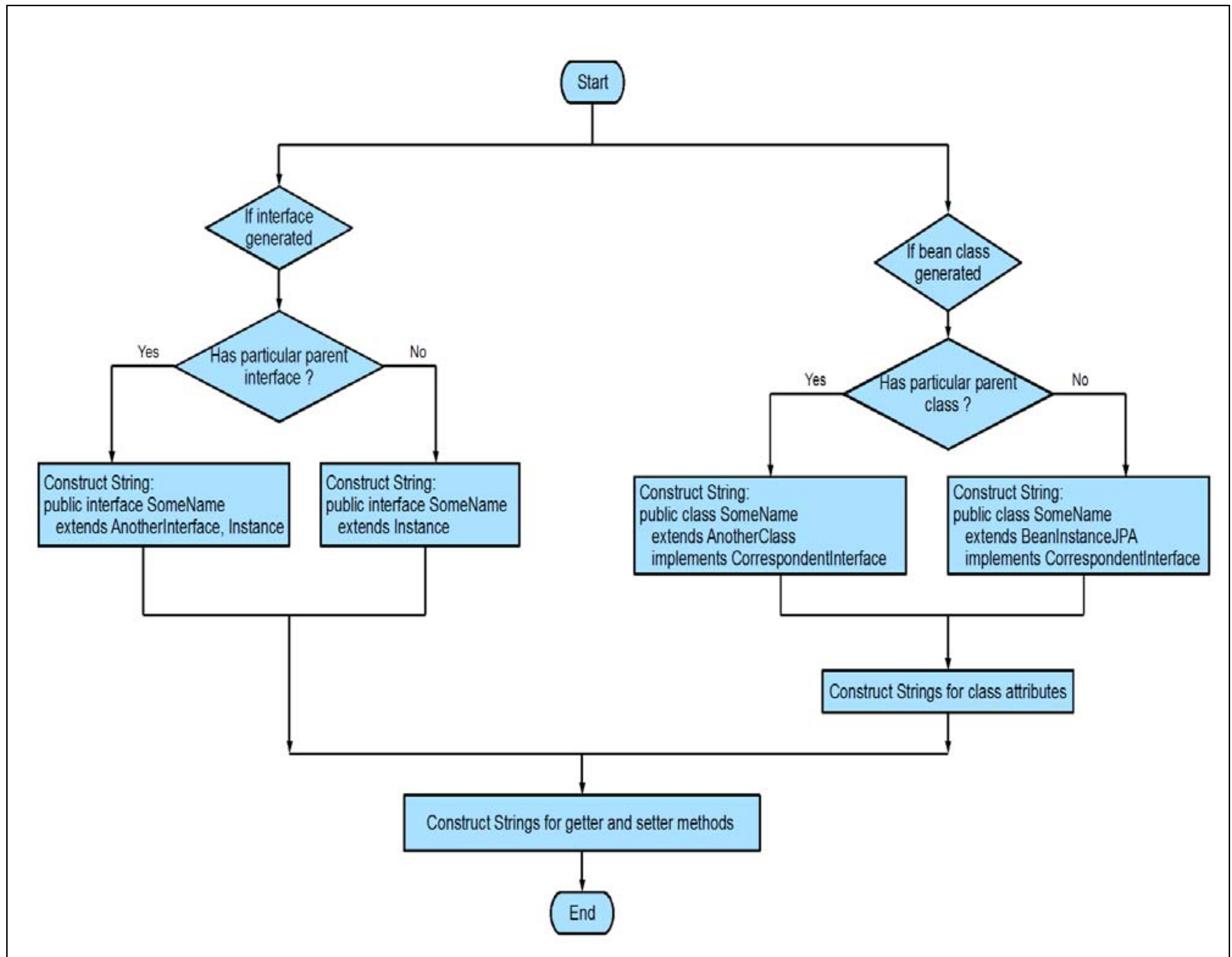

Figure 4.1. 4. Working mechanism of the method generateClass(..)

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema"
        xmlns:erm="http://www.topgallant.org/xsd/erm.xsd"
        id="erm"
        version="1169630168929">
  <xsd:complexType name="erm:cutting_description" >
    <xsd:element type="erm:separation_activity_description" />
  </xsd:complexType>

  <xsd:complexType name="erm:robot" >
    <xsd:element type="erm:production_equipment" />
    <xsd:element name="type" type="erm:robot_setup_type" />
    <xsd:element name="degreesOfFreedom" type="xsd:int" />
  </xsd:complexType>

  <xsd:simpleType name="erm:steel_design_structure_type" base="NMTOKEN">
    <enumeration value="BLOCK"/>
    <enumeration value="BRACKET"/>
    <enumeration value="BUILT_PROFILE"/>
    <enumeration value="CLIP"/>
    <enumeration value="CORRUGATED_STRUCTURE"/>
    <enumeration value="PANEL"/>
    <enumeration value="SECTION"/>
  </xsd:simpleType>

  <xsd:complexType name="erm:fitting" >
    <xsd:element type="erm:piping_part" />
    <xsd:element name="type" type="erm:fitting_type" />
    <xsd:element name="flangeType" type="erm:flange_type" />
    <xsd:element name="flangeConnType" type="erm:flange_conn_type" />
  </xsd:complexType>

  ............................................

</xsd:schema>
```

Figure 4.1. 5. Some portion of the XSD file

4.1.2.2. Analysis of the Java-based data model

From the **figure 4.1.2** above, it can be seen that the Java-based data model is generated from the XSD-based data model by the code generator. The high complexity of the Java-based data model (containing about 300 classes) is clearly shown via the class diagrams generated by using reverse engineer technique. The **figure 4.1.6** and **figure 4.1.7** show the inheritance hierarchy in hierarchical layout and symmetric layout respectively generated from the interfaces in the data model. From these two figures, it can be seen that the entities in the data model are classified into two main categories that are the first-class category and the secondary category. The first-class category is the biggest group with most of the interconnected entities having the common root parent class of *InformationObject*. In fact, if any entity has the root parent class which is this *InformationObject* class or in other words, if any entity is reached from this *InformationObject* class then that entity belongs to the first-class category. The secondary category contains the remaining entities which do not belong to the first-class one. Because of the fact that the interfaces in the data model define only the getter and setter methods, one can not see any associations between the entities in the same category as well as between the entities in different categories.

Instead, the complicated associations between the entities are shown in the **figure 4.1.8** which depicts the class diagram generated from the bean classes in the data model.

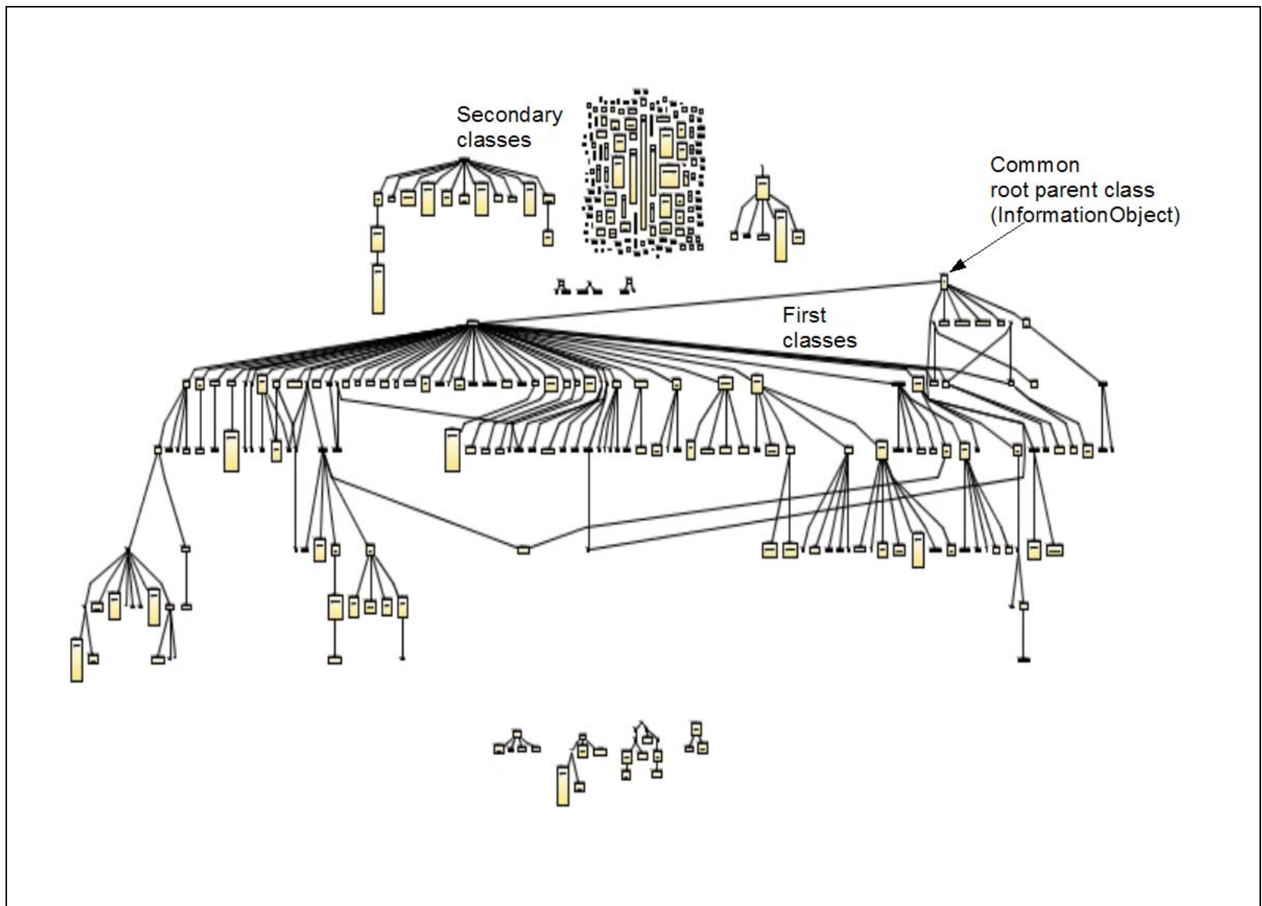

Figure 4.1. 6. Interface inheritance hierarchy (hierarchical layout) in data model

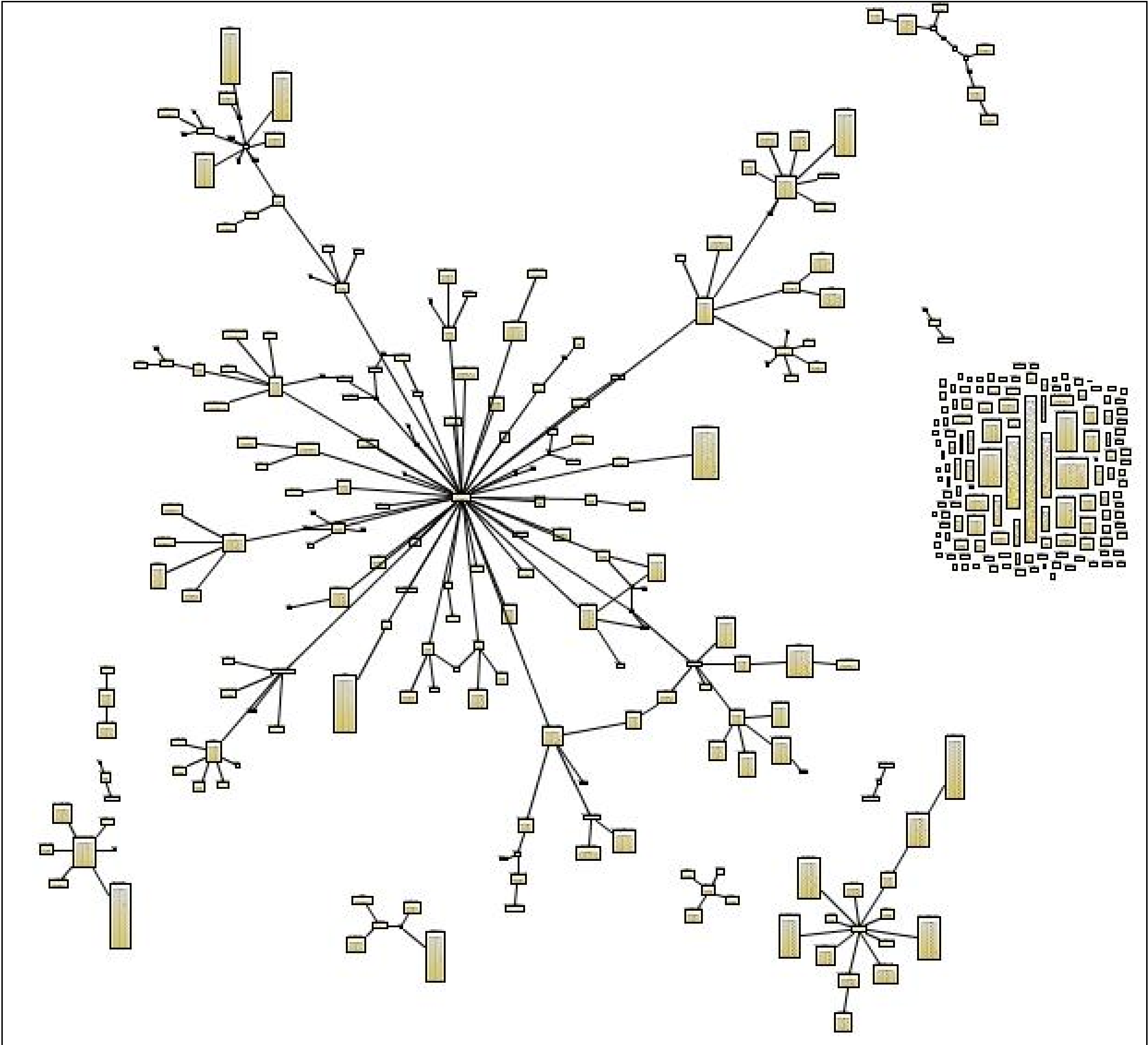

Figure 4.1. 7. Interface inheritance hierarchy (symmetric layout) in data model

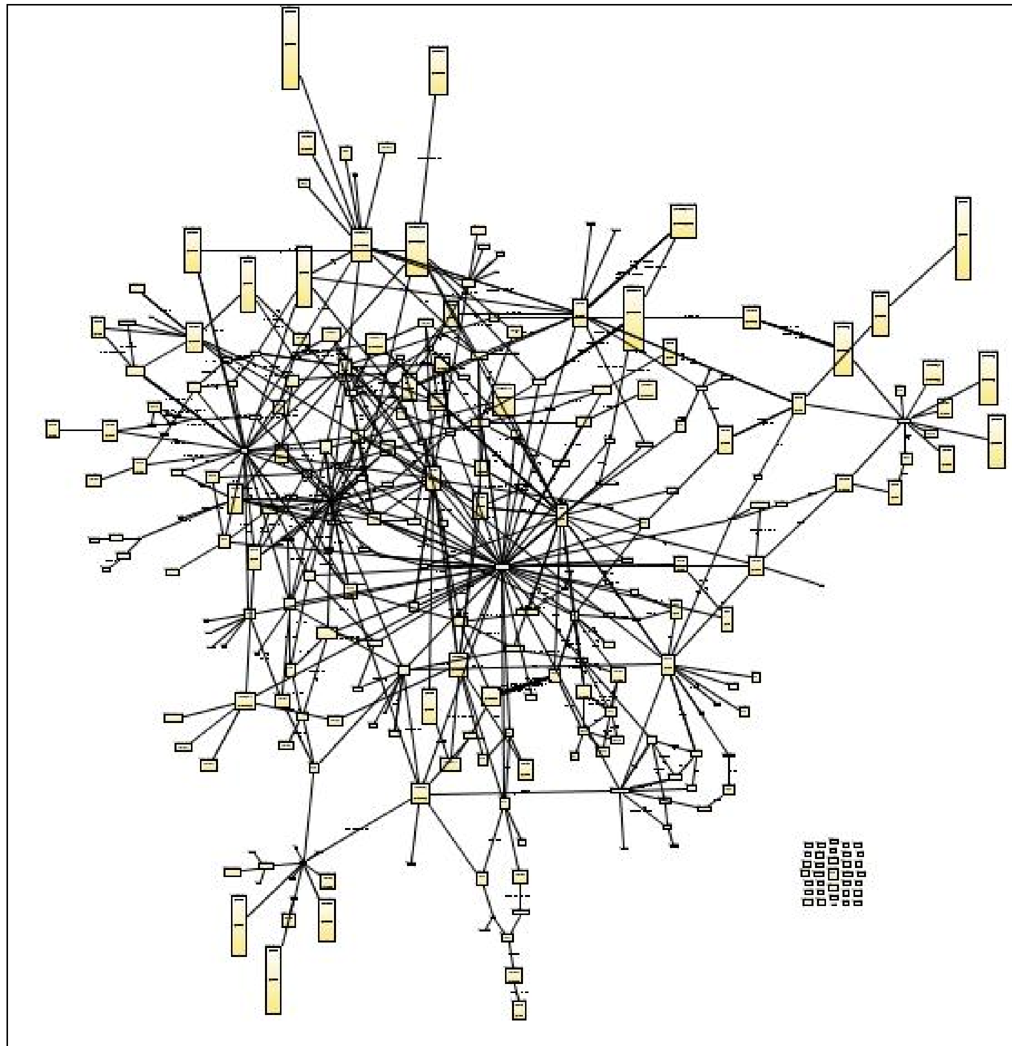

Figure 4.1. 8. Class diagram (symmetric layout) of the bean classes in data model

The complexity of the Java-based data model is clearly shown in the **figure 4.1.8**. One can realize that the data model complexity lies at the class inheritance and class associations. Therefore, the analysis of class inheritance and class association has to be performed before being able to specify JPA annotations[22] for mapping such complicated class inheritance and class associations to the relational database schema. At first, the inheritance tree of the first-class category[23] in the data model should be sketched because the **figure 4.1.6** via hierarchical layout indicates a multi-level class inheritance of the data model. In particular, there are six levels in class inheritance (depicted in **figure 4.1.9**). Each level is identified by the key class (e.g. the *ErmObject* on the level 1) and in addition to the key class, there are still many other classes indicated by the symbolic name (e.g. *OthersLevel1* on the level 1).

[22] As described in the **part 2.2.3**, JPA annotations are classified into the categories such as Inheritance Mappings Relationship Mappings (e.g. One-to-One, One-to-Many),

[23] The secondary category is not of interest because the its class inheritance is not complicated

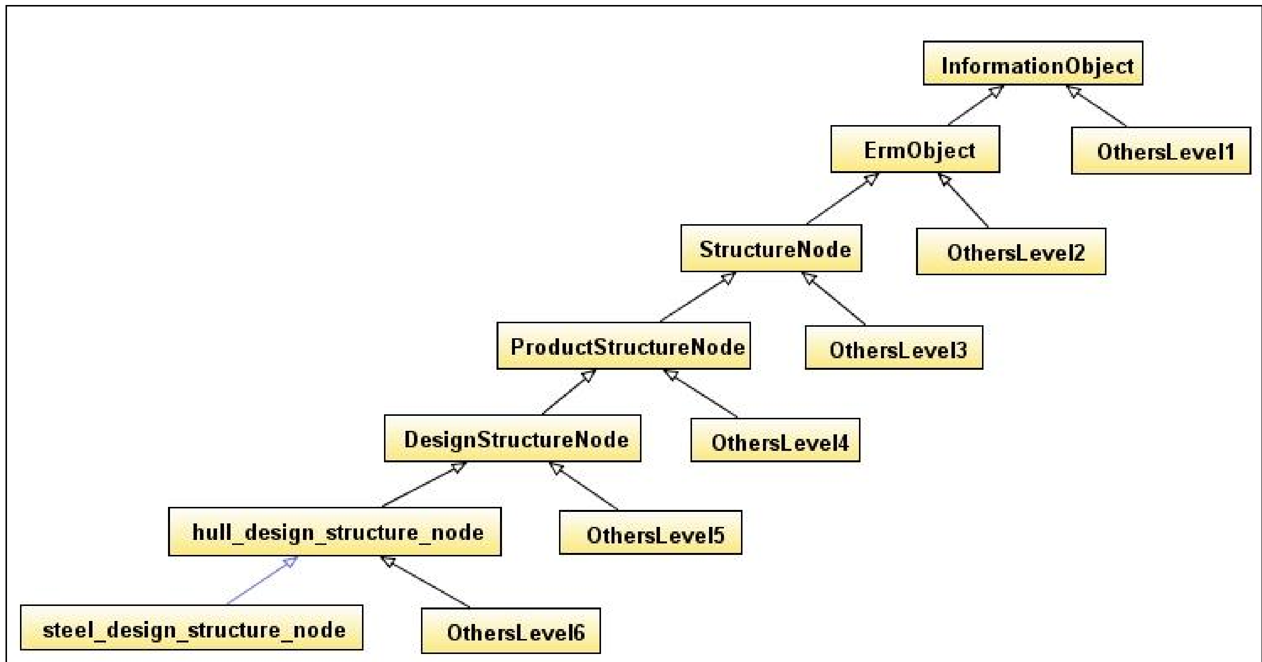

Figure 4.1. 9. Six-level class inheritance of the first-class category in the data model

After sketching the inheritance tree, the next step is to make a statistics[24] of the special data types in the data model. This is because standard JPA annotations[25] can not map directly some special data types[26] that are for example a collection of primitive type (e.g. List<String>), List<Integer>), an array of primitive type (e.g. Double[ ], String[ ], …). In addition, the complex type (user-defined type) and the type of byte array are also made statistics. The statistics results (written to text files) including data type and occurrences of the data type are:

- For special data types:
    - {List<Double>=5, List<String>=180, Double[]=5, List<Integer>=8}
- For type of byte array:
    - {byte[]=340}

---

[24] The statistics is made in a programmatic manner.
[25] The original intention is to use only standard JPA annotations for mapping.
[26] How to handle the special data types, which standard JPA annotations can not map directly, is described in the **part 4.4 (JPA shortcoming)**

Some portion of the statistics result of the complex type[27]:

```
List<KeyValueBean>
byte[]
RawMaterialBean
List<ActivityTemplateBean>
List<String>
List<InformationObjectBean>
List<ParameterSetBean>
List<InformationObjectBean>
ProductionEquipmentBean
...
```

The final analysis is to explore the class associations[28] of the data model. The typical class associations are: One-to-One, One-to-Many and Many-to-Many. Furthermore, the class associations can be bidirectional or unidirectional or self-referential (reflexive). The **figure 4.1.10** shows the bidirectional class associations in which class association between class A and class B is One-to-One, class association between class A and class C is One-to-Many and class association between class A and class D is Many-to-Many.

Based on the complex type of the class attribute, one can easily identify the class association. For example, if the complex type is object type then the class association is One-to-One and if the complex type is collection of object type then the class association can be One-to-Many or Many-to-Many. The fact that how to identify whether a class association is bidirectional or unidirectional or self-referential (reflexive) is as following. The difference between bidirectional- and unidirectional class association is that for bidirectional association, the owned object class has information about its correspondent owning object class but for unidirectional association, the owned object class does not know anything about its owning object class. For example, if the class A (in the **figure 4.1.10**) has a reference to class B and if class B also has reference back to class A, then the class association between them is bidirectional. In case, class B does not have reference back to class A, then the class association between them is unidirectional and in this case class A object is called owning object and class B object is called owned object. The self-referential class association is the case in which the class has a reference to itself.

For specifying the JPA annotations (of Relationship Mappings) into the code generator implementation, at the beginning, all types of class associations are supposed to appear in the data model.

[27] Every bean impl class has a suffix of „Bean“
[28] For class association analysis, only complex types are of interest (e.g. object type, collection of object type)

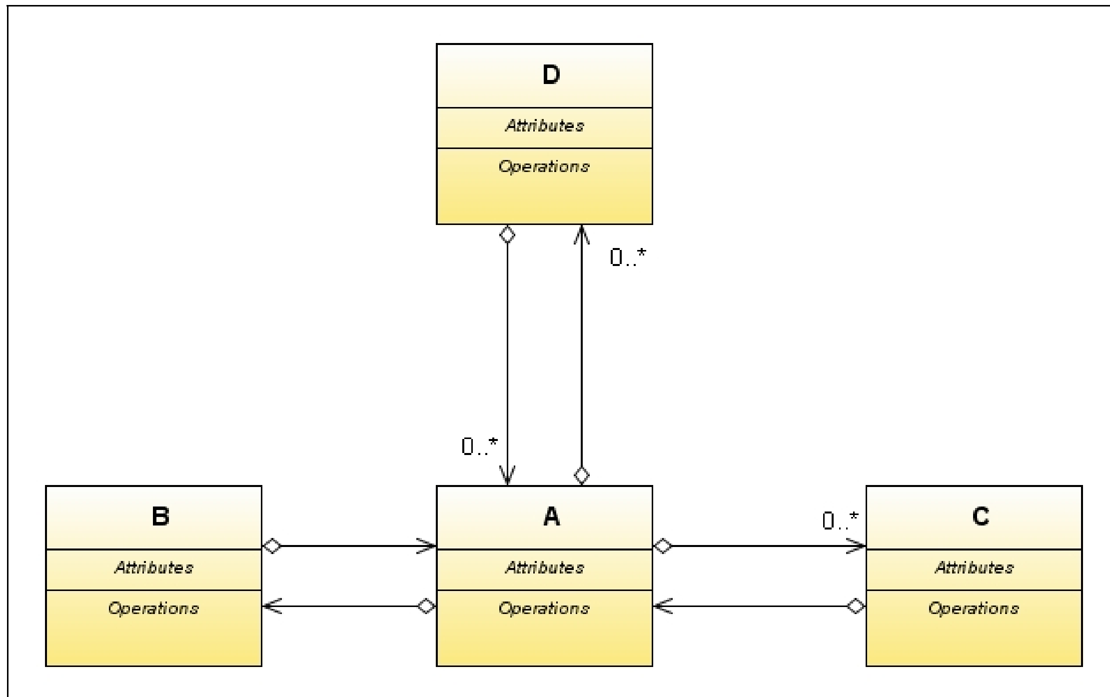

Figure 4.1. 10. Class associations

4.1.2.3. Code generator modification

The code generator works with the XSD-based data model (as the input and the output is the Java-based data model). Due to the high complexity of the real XSD-based data model, it is nearly impossible as well as not a good choice to apply at the beginning the real data model to the code generator for modification. Instead, the code generator modification is carried out stepwise with multiple artificial XSD-based data models whose complexity is increased from simple via complicated to real. Each artificial (XSD-based) data model is built in order to modify the code generator (for specifying JPA annotations) with respect to a particular mapping context such as class inheritance hierarchy, various class associations and so on. This is because JPA mapping annotations are classified into different categories[29] such as inheritance[30], relationships mappings[31] and so on. The purpose of applying artificial data models to the code generator is to ensure that the code generator can at least specify appropriately JPA annotations for fairly simple and specific cases.

[29] Further detail is presented in **part 2.2.3**
[30] JPA annotations of inheritance mapping are specified above the class name
[31] JPA annotations of relationships mapping are specified above class attribute

To map a class inheritance hierarchy, there are several different approaches:

- *@Inheritance(strategy = InheritanceType.JOINED)*: maps one class to one table regardless of abstract- or concrete class. The Foreign Key (FK) of the sub-class references the Primary Key (PK) of its nearest parent class (illustrated in **figure 4.1.11**).

- *@Inheritance(strategy = InheritanceType.SINGLE_TABLE)*: the whole class inheritance hierarchy is mapped to a single table. Particularly, only 1 table of the root parent class is created and it includes all attributes of the subclasses (illustrated in **figure 4.1.12**).

- *@MappedSuperclass*: map one concrete sub-class (except abstract parent class) to one table. Therefore, all attributes of the abstract parent class are included in each sub-class (illustrated in **figure 4.1.13**). This approach is often used in the case which an abstract class acts as parent class to other sub-classes.

- *@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)*: map one concrete sub-class (except the concrete parent class) to one table. Therefore, all attributes of the concrete parent class are included in each sub-class. This approach is often used to map a concrete class which acts as parent class to other sub-classes. The difference between this approach and the approach of *MappedSuperclass* is that parent class in this approach is concrete class while parent class in the approach of *MappedSuperclass* is abstract class.

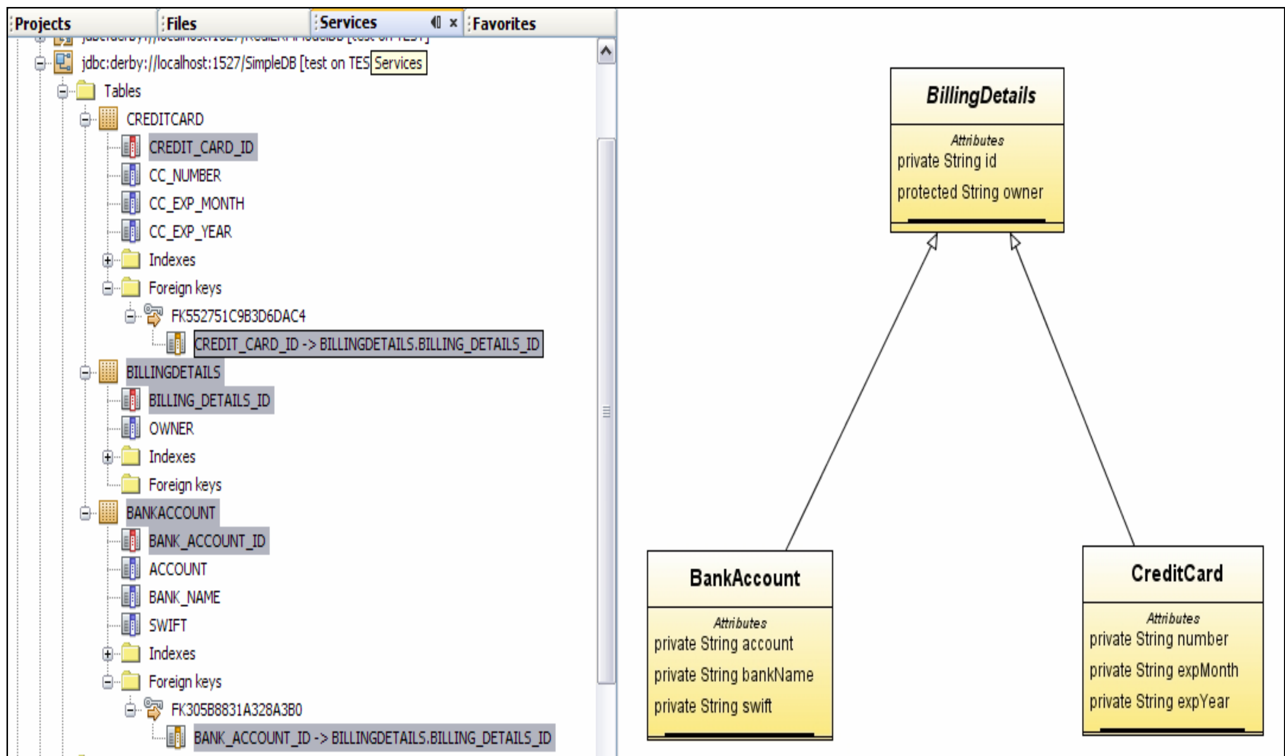

Figure 4.1. 11. InheritanceType.JOINED

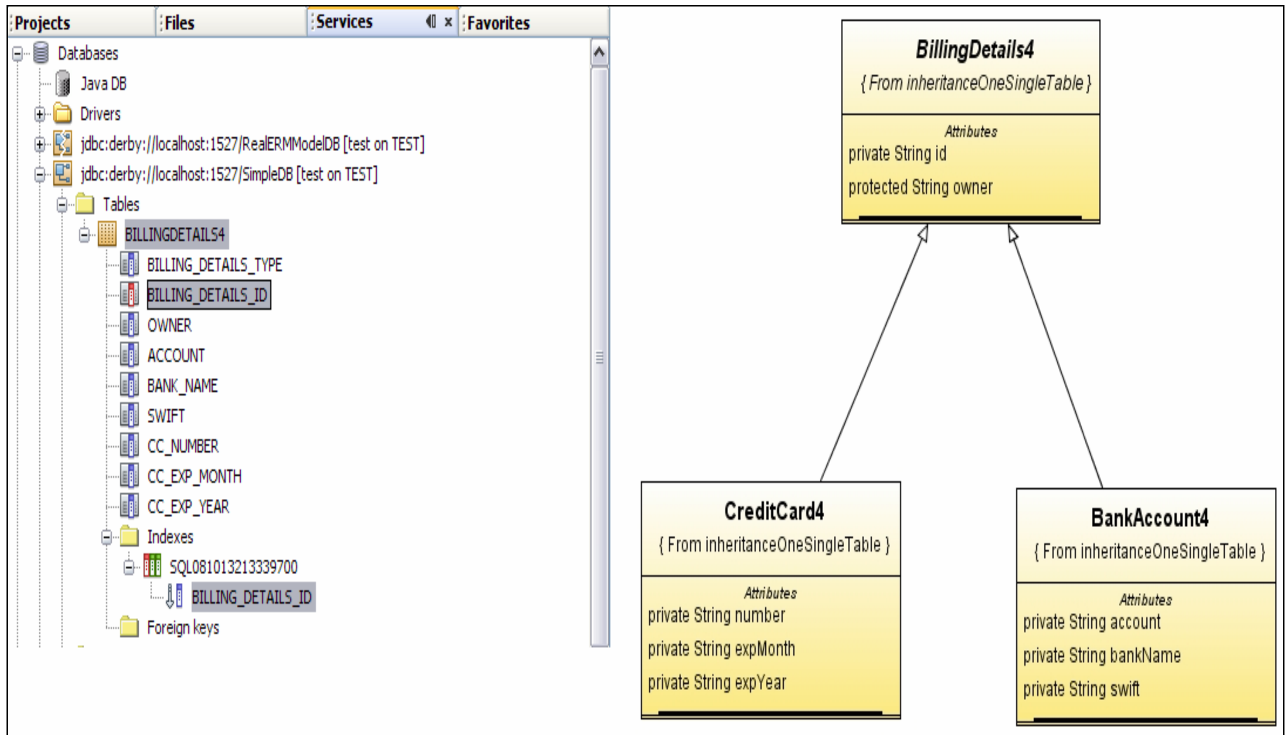

Figure 4.1. 12. InheritanceType.SINGLE_TABLE

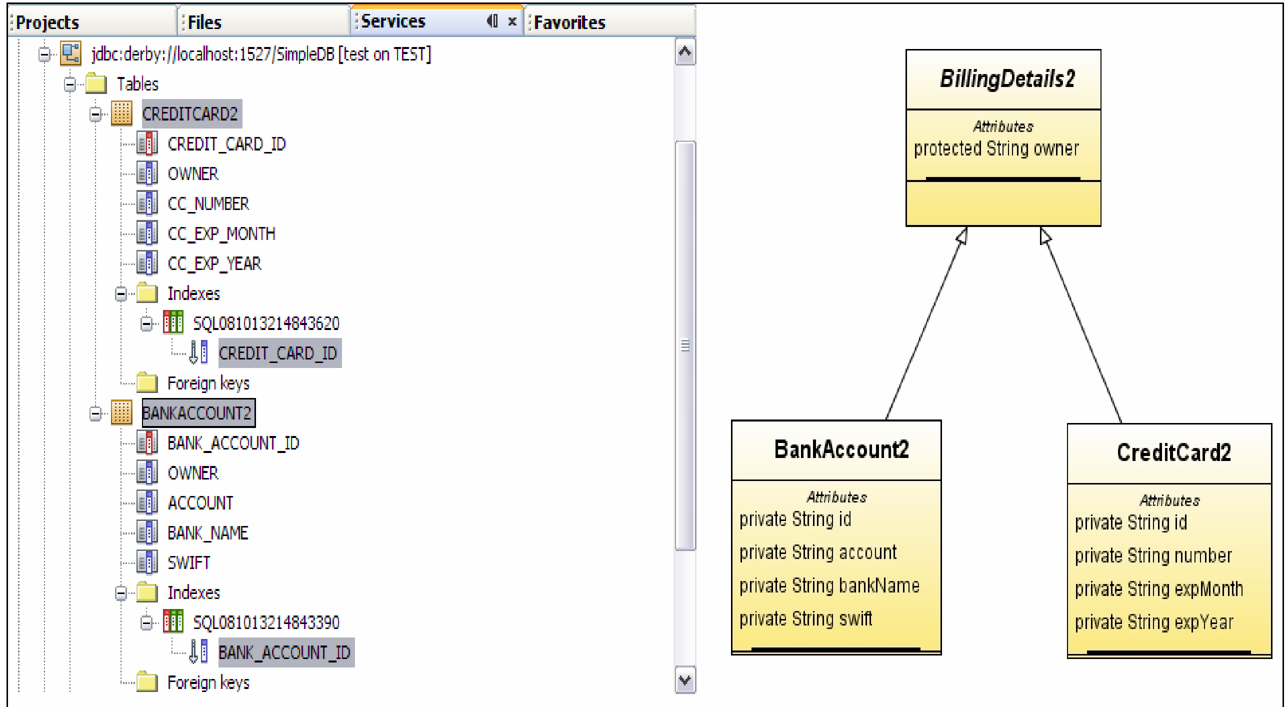

Figure 4.1. 13. MappedSuperclass

Among the different approaches, the approach of @Inheritance(strategy = InheritanceType.JOINED) is chosen to map the class inheritance hierarchy of the data model. This is because this approach allows to map each class[32] to a table in the relational database schema. Furthermore, this approach has a big advantage over the other approaches due to its uniform mapping strategy. In fact, with this approach, one needs only to specify the inheritance mapping annotation (particularly *@Inheritance(strategy = InheritanceType.JOINED)*) in the root parent class and then the other sub-classes are specified with the annotation *@PrimaryKeyJoinColumn* which defines a column acting as both PK and FK. The FK role is used to reference the PK of the upper parent class while the PK role is used to be referenced by the FK of the lower sub-class. Therefore, the entire class inheritance hierarchy can be mapped to relational database schema. This is illustrated in the **figure 4.1.14.**

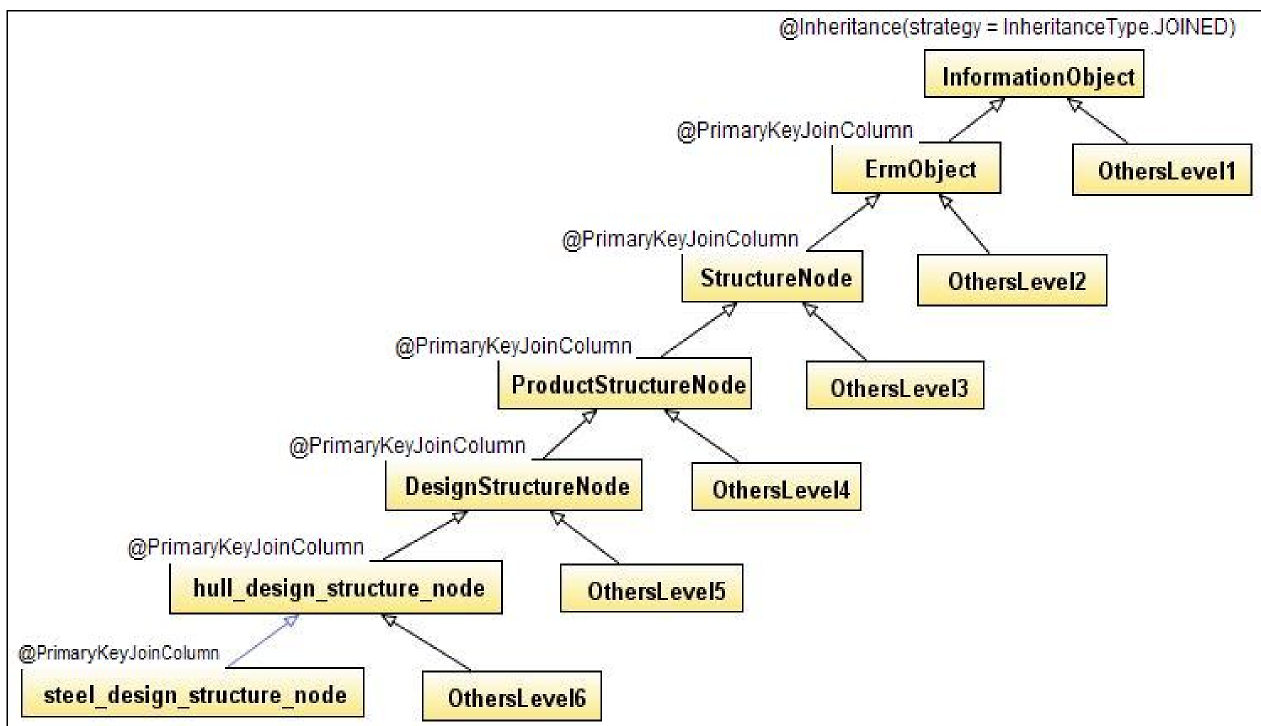

Figure 4.1. 14. Uniform mapping of the class inheritance hierarchy

From the **figure 4.1.14**, one can see that the root common parent class is the *InformationObject*. Any class, which can be reached from the *InformationObject* class or has the root parent class of *InformationObject*, is called first class and the classes, which do not have root parent class of *InformationObject*, are called secondary class. The secondary classes are not specified any annotations and thus they are not mapped to the relational database schema. Consequently, any attribute, whose type is of secondary class, in the first class is also excluded from the mapping process. This is simply done by specifying the annotation *@Transient* right above the attribute

After modifying the code generator for the class inheritance hierarchy mapping, the code generator modification is continued for the class association mapping. This is performed stepwise

[32] Secondary classes are not mapped to relational database schema as the requirement that secondary class objects are not stored directly in the database.

at first with the artificial data models which cover all possible class associations as analyzed above and then the real data model is applied. Mapping of some typical class associations of One-to-One and One-to-Many are presented as followings.

The class association of One-to-One is mapped by specifying the annotation *@OneToOne* at both owning side and owned side. Furthermore, the annotation *@JoinColumn* is specified to create a FK column at the owning side which references the PK column of the owned side. The owned side can reference back to the owning side via the attribute *mappedBy* of the annotation *@OneToOne*. Therefore, bidirectional association is established. If the attribute *mappedBy* is not specified, the association is only unidirectional. In the case of reflexive association, the FK references the PK of its own class. This is summarized in the **figure 4.1.15**. One should note that the annotation *@Entity* (specified above class name) indicates a persistent entity class which is normally mapped to a table in the database schema. The annotation *@Id* specifies a PK column.

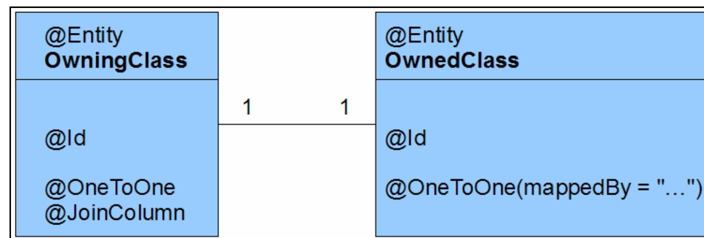

Figure 4.1. 15. One-to-One class association mapping

The class association of One-to-Many can be mapped by using an intermediate table or FK. The intermediate table contains two FKs referencing PKs of both sides of the association. The other possibility is to place FK in the "Many" side and thus multiple rows in the table of "Many" side can reference one PK of the row in the table of "One" side. The example of bidirectional One-to-Many association between the *Item* class and the *Bid* class (the "One" side is the *Item* table and the "Many" side is the *Bid* table) in the **figure 4.1.16** and **4.1.17** makes this clear. In fact, from the intermediate table "Item_Bid" in **figure 4.1.16** or from the FK column of table "Bid" in **figure 4.1.17**, one can easily see that the Item "i1" has two Bid "b1" and "b2" while the Item "i2" has only 1 Bid "b3". The mapping of One-to-Many association using intermediate table is summarized in **figure 4.1.18**. One should note that the annotation *@JoinTable* is specified at the "Many" side to create the intermediate table in the database schema. The annotation *@OneToMany* is specified at the "One" side while the annotation *@ManyToOne* is specified at the "Many" side.

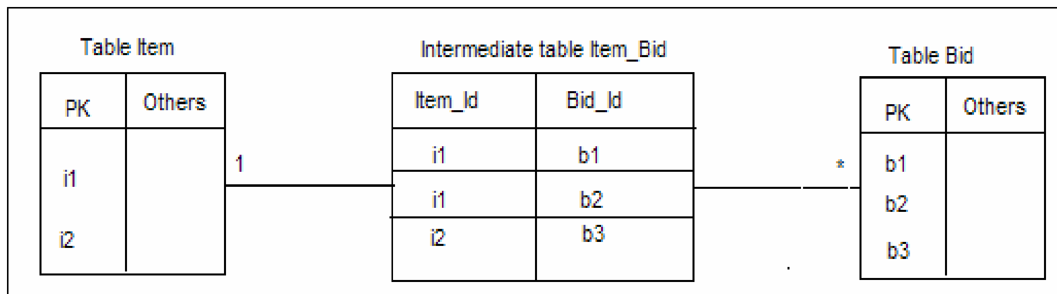

Figure 4.1. 16. One-to-Many association mapping result with intermediate table

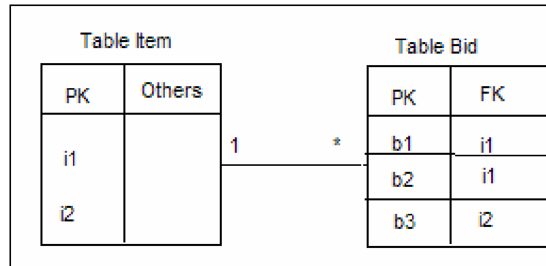

Figure 4.1. 17. One-to-Many association mapping result with FK

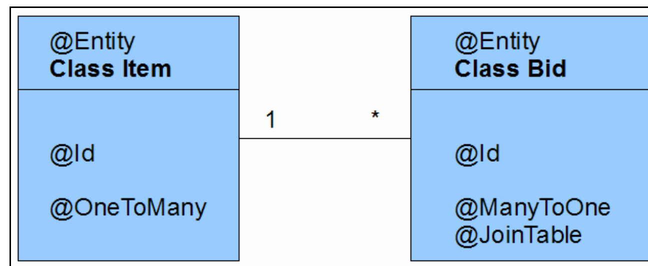

Figure 4.1. 18. One-to-Many association mapping using intermediate table

After having studied how to map various class association types, the next step is to build some artificial (XSD-based) data model to modify the code generator with specifying JPA mapping annotations. The **figure 4.1.19** illustrates an artificial model built to cover most of the important class associations as well as class inheritance. The bean classes, which are decorated with JPA annotations and are generated by the code generator, are then mapped to relational database schema (**figure 4.1.20**) by the ORM engine.

After having applied several artificial data models, the code generator were applied with the real (XSD-based) data model to try generating the bean classes decorated with JPA annotations. At the beginning, there were lots of compile errors with the generated bean classes and thus code generator was continued to be modified until the generated bean classes with JPA annotations can be mapped to relational database schema. The **figure 4.1.21** shows some portion of the relational database schema automatically generated (by the ORM engine) from the real (XSD-based) data model via the generated Java bean classes with JPA annotations.

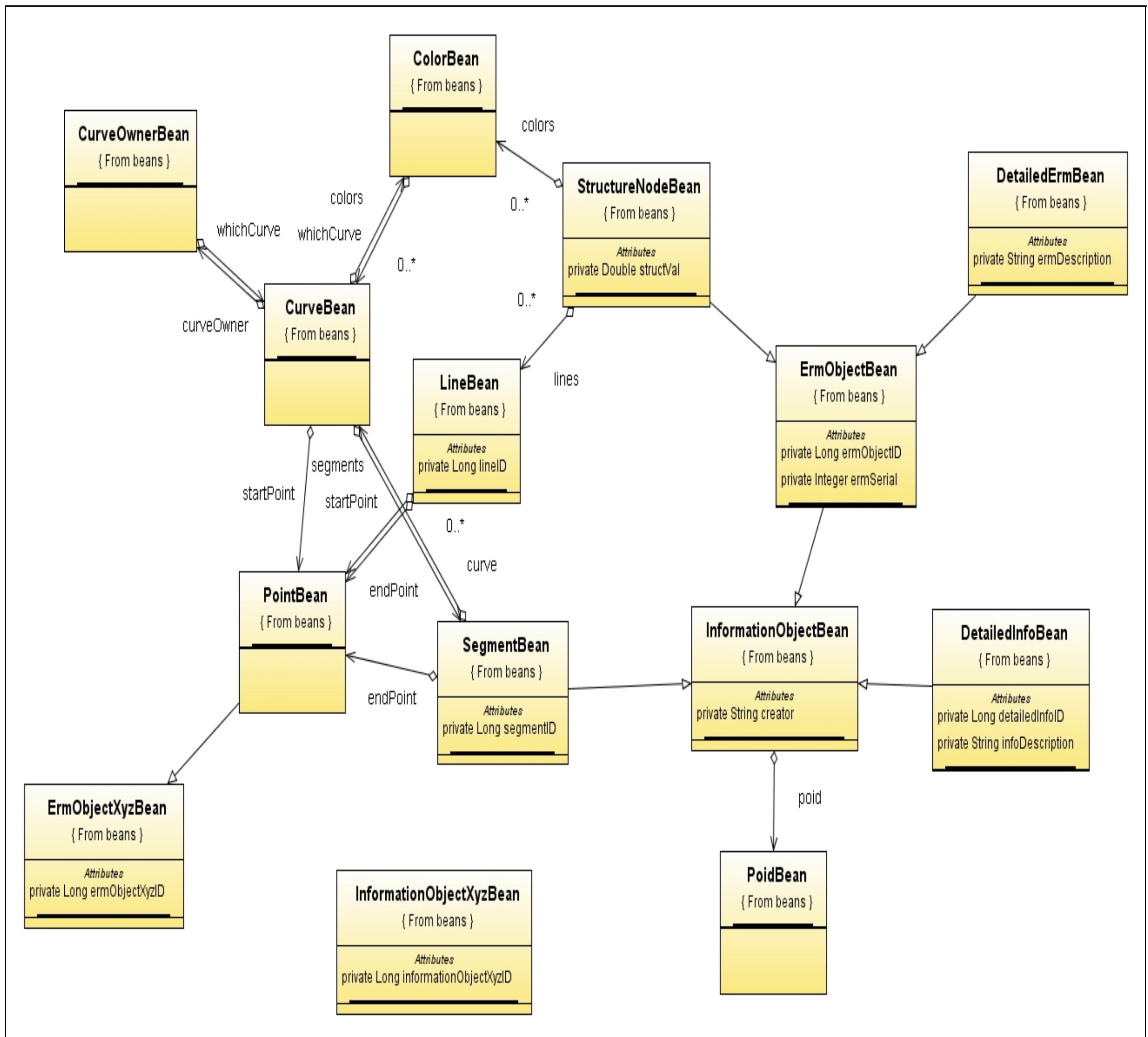

Figure 4.1. 19. Artificial model for class association mapping

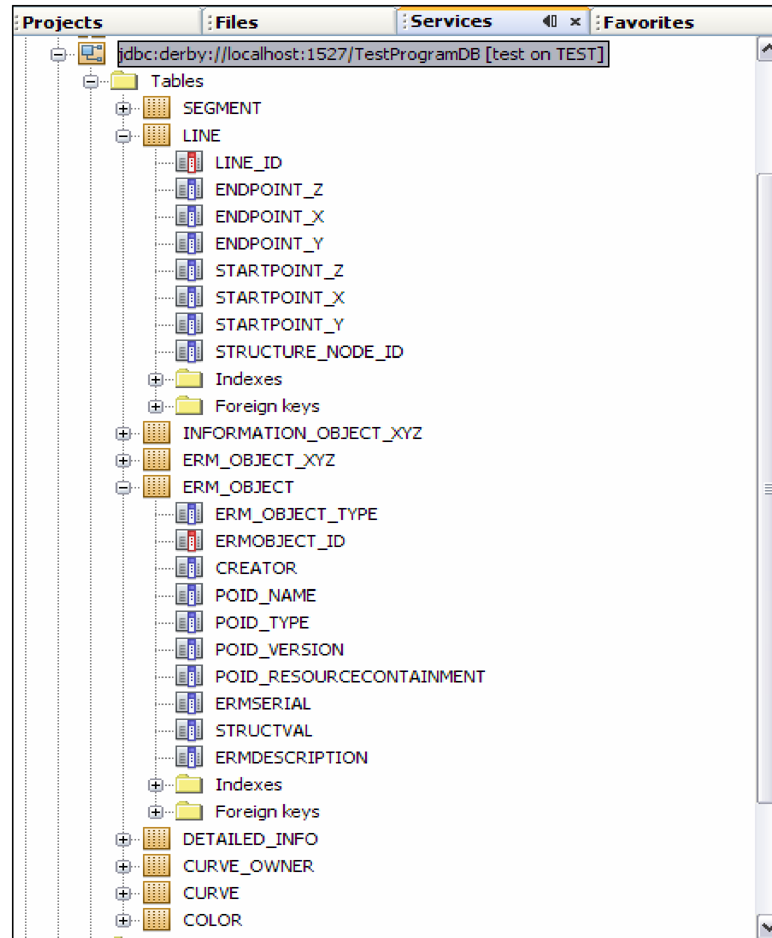

Figure 4.1. 20. Mapping result of the artificial model from figure 4.1.10

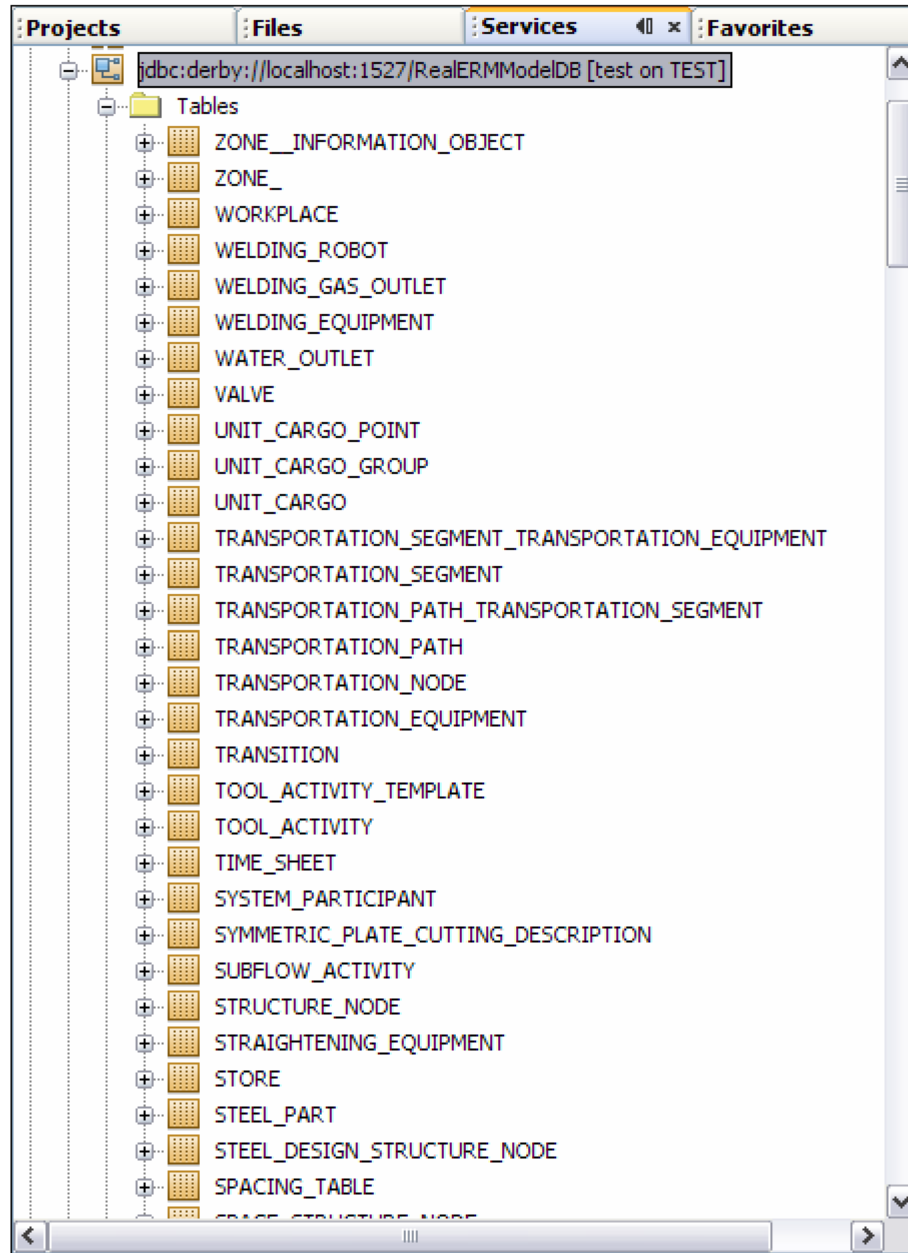

Figure 4.1. 21. Generated relational database schema

4.1.3. Result and verification

The generated relational database schema (**figure 4.1.21**) is validated by using the JPA Hibernate validator. The validator is specified as a property of the persistence unit dedicated for validation as shown below:

```
<persistence-unit name="DB_Schema_Update_Validate" transaction-type="RESOURCE_LOCAL">
 <provider>org.hibernate.ejb.HibernatePersistence</provider>
 <properties>
  <!-- Default DB parameters specified here will be over-written by the application -->
  <property name="hibernate.dialect" value="org.hibernate.dialect.DerbyDialect"/>
  <property name="hibernate.connection.url"
           value="jdbc:derby://localhost:1527/RealERMModelDB"/>
  <property name="hibernate.connection.driver_class"
           value="org.apache.derby.jdbc.ClientDriver"/>
  <property name="hibernate.connection.password" value="test"/>
  <property name="hibernate.connection.username" value="test"/>
  <property name="hibernate.hbm2ddl.auto" value="update"/>
  <property name="hibernate.hbm2ddl.auto" value="validate"/>
 </properties>
</persistence-unit>
```

The database schema validation works by fetching the database metadata and then comparing the object-relational mappings against the fetched database metadata. If a mismatch is encountered, exception will be thrown. The validation result is shown in the **figure 4.1.22**.

Some sample of the annotated bean class generated by the code generator is also shown here (**figure 4.1.23**).

After validating the generated database schema, the next step is to verify (using JUnit testing framework) the functional capability of the database schema (shown in **figure 4.1.24**). The testing sequence is as followings:

- Create and initialize some object. The chosen object of *ActivityBean* (from the generated Java-based data model) (depicted in **figure 4.1.25**) is complicated enough with a unidirectional One-to-One and One-to-Many associations in addition to the primitive data types.
- Store the object into database
- Retrieve the stored object based on one particular attribute of it
- Access (via getter) some attributes (including primitive types, object types and collection of object type) of the retrieved object and compare their values with the information provided at the object instantiation in order to check what has been stored.
- Modify (with setter) several attribute values of the retrieved object and then store the modified object into database.
- Retrieve that modified object and verify the modification

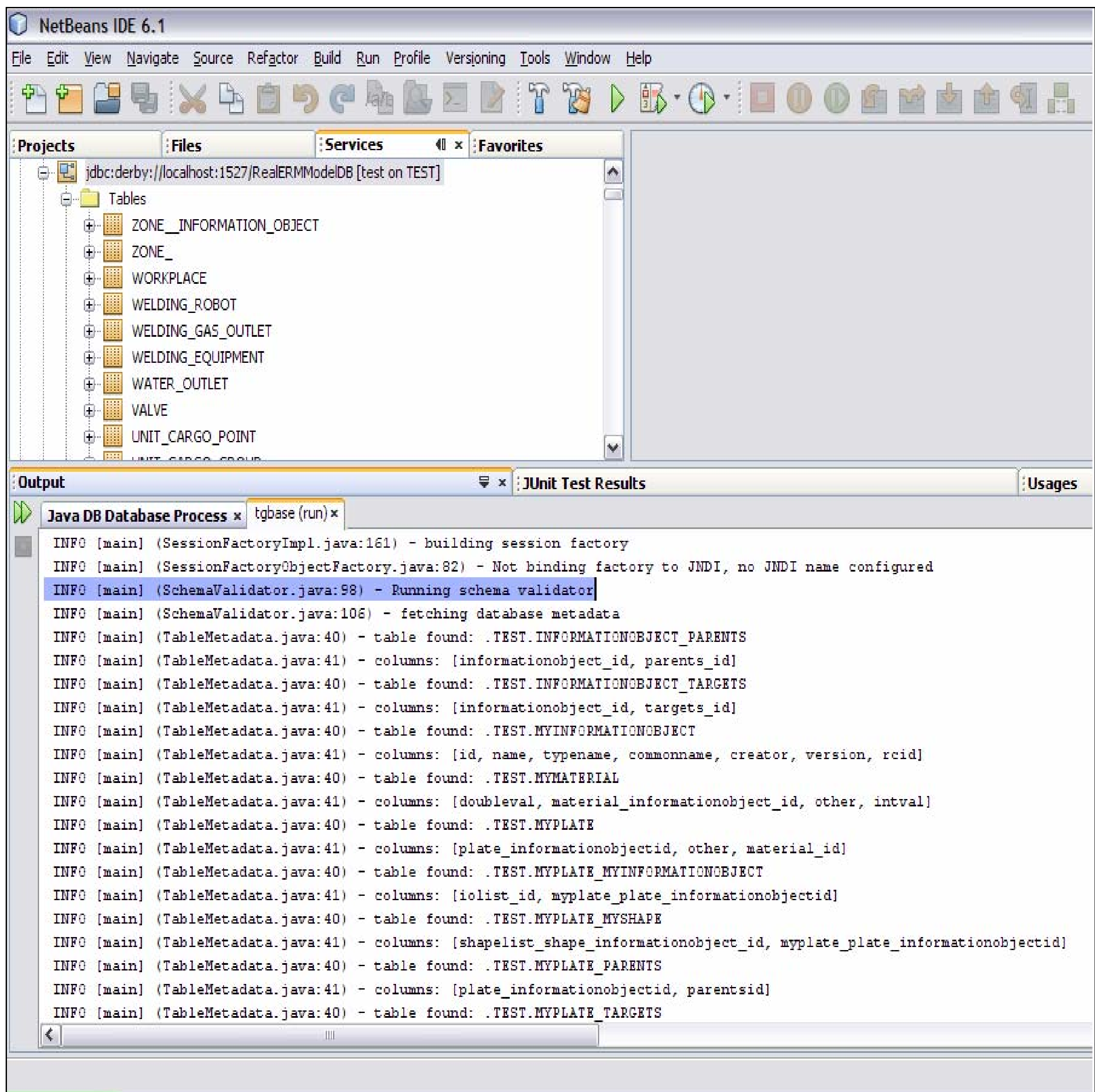

Figure 4.1. 22. Database schema validation result

Proj... | Files | Services | Favorites

- org.atlantec.jpa.databaseUtil
- org.atlantec.jpa.model.erm
- org.atlantec.jpa.model.erm.beans
  - AccelerationMeasureBean.java
  - AccessPolicyBean.java
  - ActivityBean.java
  - ActivityDescriptionBean.java
  - ActivityInstanceBean.java
  - ActivityTemplateBean.java
  - ActivityTypeBean.java
  - ActivityTypesBean.java
  - AdapterBean.java
  - AirflowVolumeMeasureBean.java
  - AngleMeasureBean.java
  - ApplicationEntityBean.java
  - ApplicationParametersBean.java
  - ApplicationProcessBean.java
  - AreaMeasureBean.java
  - AssemblyDescriptionBean.java
  - AssemblyLevelDefinitionBean.java
  - AssemblyNodeBean.java
  - AssemblyProductionPrepDescriptionBean.java
  - BSplineCurveBean.java
  - BSplineCurveWithKnotsBean.java
  - BSplineSurfaceBean.java
  - BSplineSurfaceWithKnotsBean.java
  - BlobBean.java

ActivityBean.java

```
40
41  @Entity
42  @PrimaryKeyJoinColumn(name="Activity_ActivityTemplate_ID")
43  @Table(name="activity")
    public class ActivityBean extends ActivityTemplateBean
45          implements Activity
46  {
47      @ManyToMany(
48          targetEntity = ParticipantBean.class,
49          cascade = { CascadeType.ALL } )
50      @JoinTable
51      @org.hibernate.annotations.Cascade(org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
52      private List<ParticipantBean>  participants = new ArrayList<ParticipantBean>();
53
54      @Column(name="type_")
55      private String  type;
56
57      @ManyToMany(
58          targetEntity = PartBean.class,
59          cascade = { CascadeType.ALL } )
60      @JoinTable
61      @org.hibernate.annotations.Cascade(org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
62      private List<PartBean>  parts = new ArrayList<PartBean>();
63
64      @Transient
65      private PeriodBean  validity;
66
```

Figure 4.1. 23. Annotated bean class generated by code generator

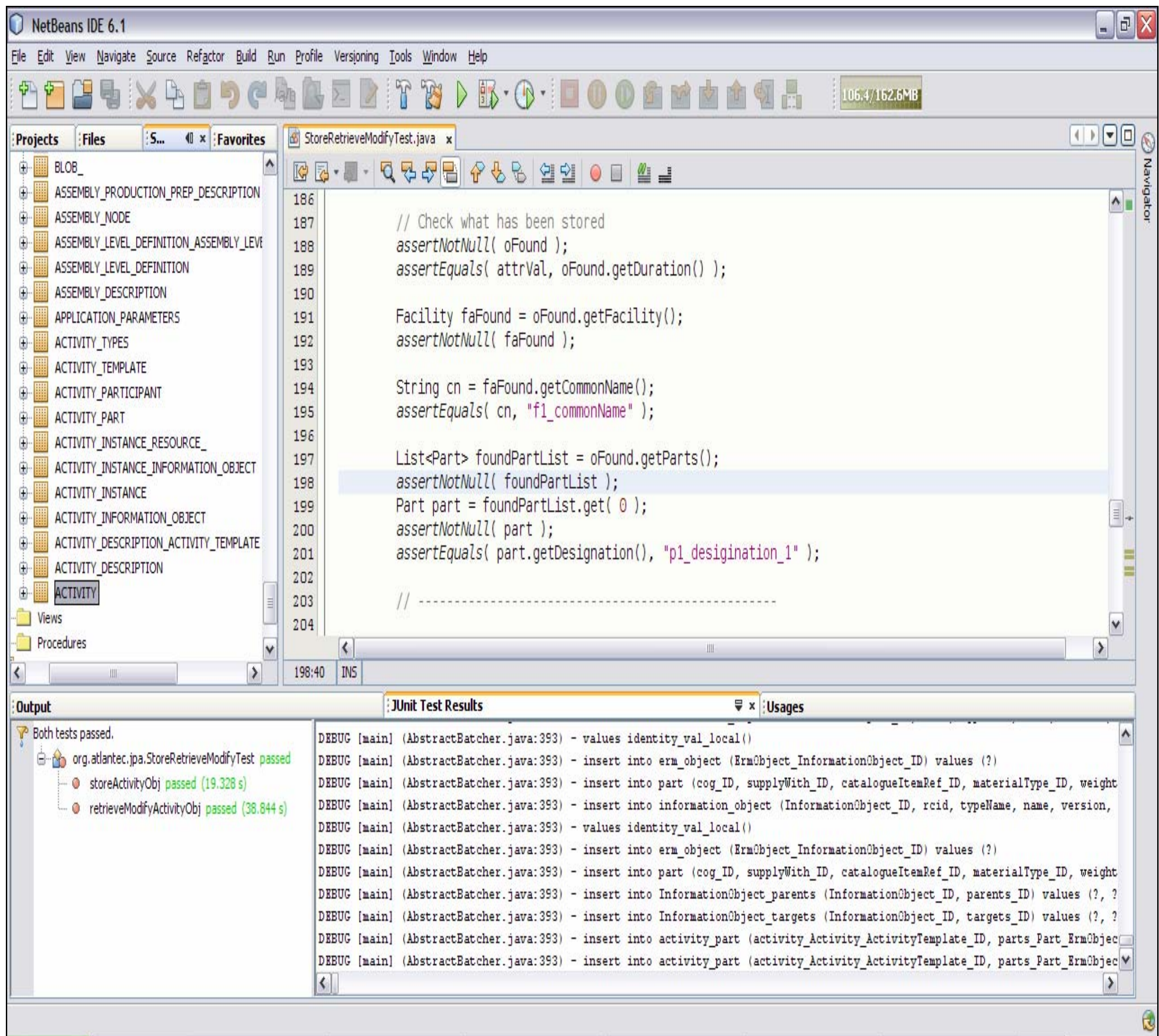

Figure 4.1. 24. Testing result of functional capability of database schema

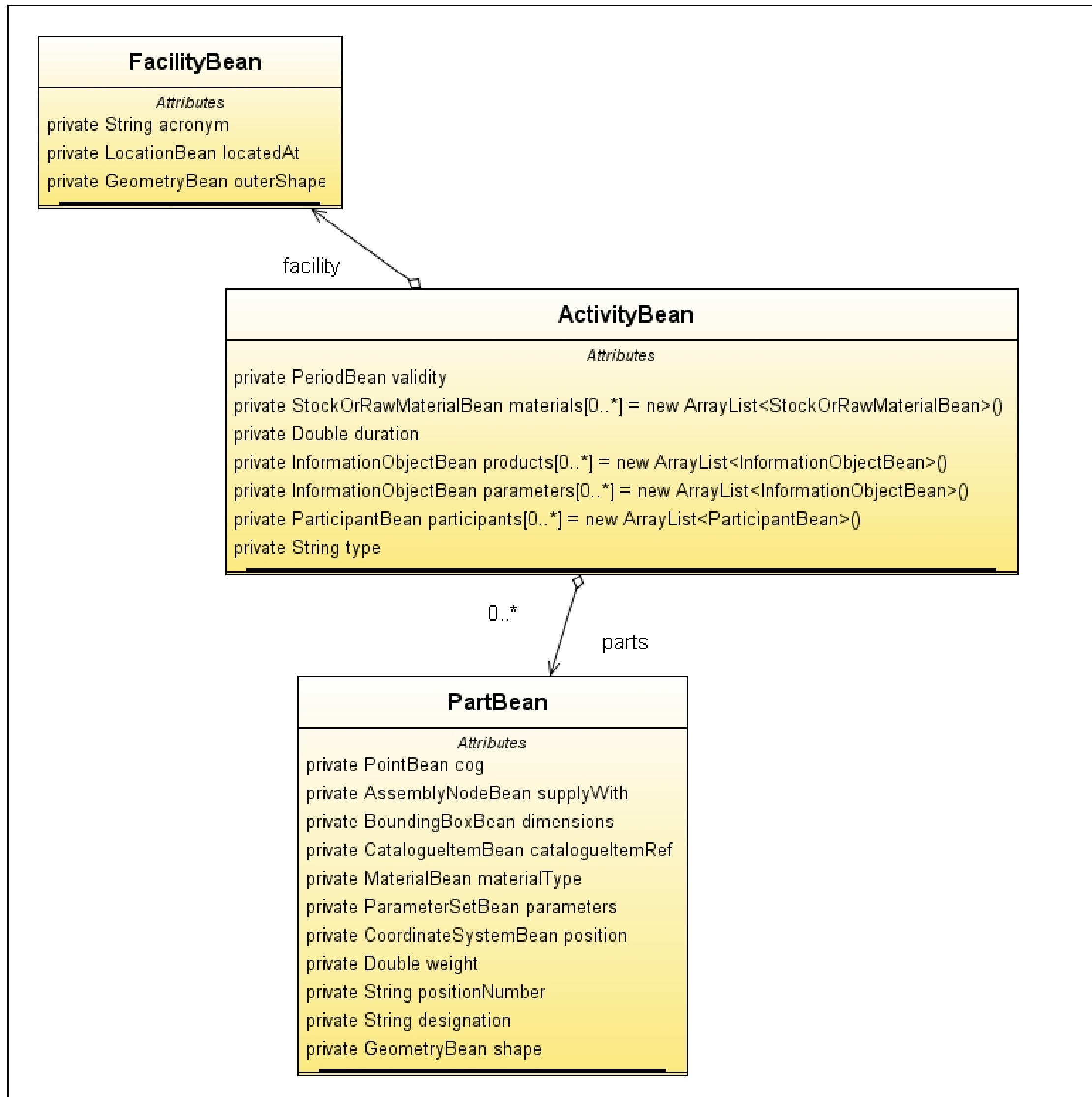

Figure 4.1. 25. Chosen class objects to verify the generated database's functional capability

In the **figure 4.1.25**, one should note that the inheritance indication of the three classes (*Activity*, *Part* and *Facility*) is not shown explicitly in the figure in order to show clearly class attributes and associations.

4.1.4. Conclusion

The fact that all secondary classes[33] are not mapped to relational database schema results in a light weight database schema. This is because the secondary classes occupy nearly half of the total number of classes in the data model.

After generating the Java-based data model with JPA mapping annotations, it is found out that all the class associations in the Java-based data model are only unidirectional[34]. This means that the owned object (referenced object) does not have any information about its owning object.

The original intention of using only standard JPA annotations (to be able to switch between different ORM engines without having to modify the annotated Java bean classes) could not be kept because one Hibernate specific annotation must be used in order to overcome the following problem. When an object of the class which contains a One-to-Many association with other class is stored into database, the collection of associated objects is also stored. The problem is when replacing the old (i.e. currently-stored in database) collection with the new collection via the setter method of the owning object (retrieved from database); the old collection still exists in database together with the new collection. It must be expected that the old collection is removed to leave space for the new collection. However, default behavior of standard JPA annotation for One-to-Many class association mapping does not remove the old collection. This problem as well as the detailed solution is presented in **part 4.4.3**. The following is the Hibernate specific annotation that must be specified additionally to the stand JPA annotations (*@OneToMany*, *@ManyToMany*) for mapping One-to-Many and Many-to-Many class associations:

```
@org.hibernate.annotations.Cascade(org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
```

The most critical problem which does not show up for artificial data models (due to low complexity) but occurs for the real data model is presented as followings. It is clear that the complexity of the real data model is very high not only because of multi-level inheritance hierarchy but also because of complicated class associations. In fact, as specified, all first classes inherit from the *InformationObject* class which contains two reflexive (self-referential) Many-to-Many associations:

```
private List<InformationObject> parents = new ArrayList<InformationObject>();
private List<InformationObject> targets = new ArrayList<InformationObject>();
```

This means that any first-class object in the data model contains at least two Lists of other first-class objects and the same is repeated for each of the other first-class objects. Therefore, the SELECT query, which is automatically generated (in the background by the ORM engine) when any List of associated objects is retrieved via the getter method of the owning object, is so huge. The measured size of the SELECT query generated (in plain text format) is about some hundreds of megabyte (MB). Therefore, the entire process is crashed whenever the owning object invokes

[33] Secondary class entities are not queried and thus it is not necessary to map them to database schema.
[34] It is not known in advance that the class associations in the Java-based data model are only unidirectional.

the getter of the List of associated objects. The major reason is because so many associated objects together with all of their attributes (primitive types, object types, collection of objects and so on) are loaded into memory when the owning object invokes the getter method of the List. The solution for this critical problem is to specify explicitly the LAZY fetching strategy for all class attributes regardless of data type. LAZY fetching strategy prevents the attributes (of any data type which can be primitive type, object type, collection of objects and so on) from being fetched when the owning object is retrieved from database and thus the attribute is only fetched on demand (via the getter method). The opposite fetching strategy, which is the default fetching strategy (if not specified), is the EAGER fetching strategy which fetches all attributes of the retrieved object.

To specify LAZY fetching strategy for attributes of primitive type (as well as of collection of values), the annotation *@Basic*[35] is used:

```
@Basic(fetch = FetchType.LAZY)
Some primitive attribute
```

To specify LAZY fetching strategy for attributes of collection of objects (e.g. One-to-Many association, Many-to-Many association) and of object type:

```
@OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)

@ManyToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)

@OneToOne(cascade=CascadeType.ALL, fetch = FetchType.LAZY)
```

One should note that, the "`cascade=CascadeType.ALL`" is used to specify that any persistence operations applied on owning object will also be applied on the associated objects.

In fact, after applying the LAZY fetching strategy for all class attributes (regardless of data types), the speed of object retrieval is significantly faster. This is because the generated SELECT query is significantly slimmer.

---

[35] If the annotation @Basic is specified above the non-primitive type attribute, then it has no effect and no conflict.

4.2. Compatible persistence API

4.2.1. Introduction and requirement analysis

After the persistent store has been migrated from LDAP-compliant directory server to relational database server, it is essential to implement the persistence API that is compatible to the working mechanism of the legacy applications. Particularly, the application compatibility is expressed via the mechanisms of:

- Object instantiation (not simply with the ***new*** operator)
- Created objects management
- Persistence and Query service

Furthermore, the compatible persistence API must be implemented in the standard JPA compliant way so that it is possible to switch between various JPA implementors in order to possibly achieve the highest performance. The JPA-based persistence API implementation must also adopt the essential functionalities[36] from the legacy persistence API such as version control, management of secondary object storage/retrieval.

4.2.2. Current working mechanism

At first, the analysis of current working mechanism of object instantiation, created objects management and persistence (on the base of LDAP-compliant directory server as the persistent store) is shown via the following pseudo code together with the correspondent sequence diagram (depicted in **figure 4.2.1**):

```
// 1. For data object instantiation
// Instantiate ObjectSet
ObjectSet os = ObjectSet.getInstance(...);

// Instantiate some data object
Plate p = ErmFactory.newPlate(os, rcid, name, version);

//--------------------------------------------

// 2. For storing data object
// Instantiate ObjectManager
ObjectManager objManager = ObjectManager.getInstance(...);

// Start Transaction obtained from ObjectManager instance
objManager.currentTransaction().begin();

// Put the object (to be stored) into persistent state
objManager.makePersistent(p);

// Commit the Transaction so that all objects in persistent state can be stored
objManager.currentTransaction().commit();
```

[36] Implementation of essential functionalities is presented in **part 4.3**

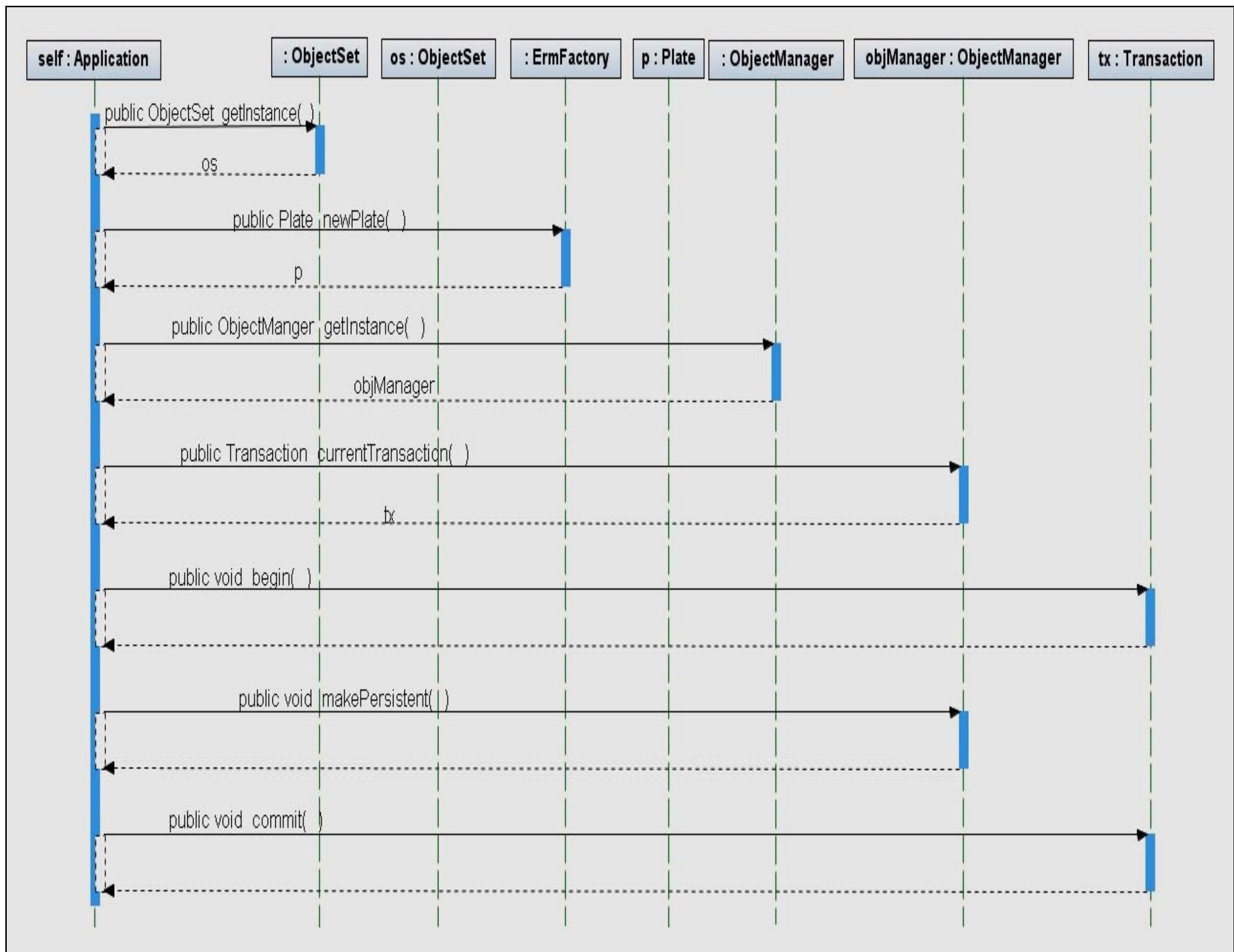

Figure 4.2. 1. Sequence diagram for current working mechanism of object instantiation, created object management and persistence.

Particularly, before the data object instantiation, the instance of the *ObjectSet* class must be created in advance. This is because the *ObjectSet* instance is always passed (as a parameter) to the instantiation of the data object so that any created object is put into the *ObjectSet* instance. Therefore, any created objects are stored and managed by the *ObjectSet* instance in order to enable the functionality of application-controllable clustering of secondary objects with first-class objects. It is important to note that the *ObjectSet* instantiation requires the explicit specification of the bean-style class implementation type. This is because the Java-based data model includes the interfaces and the bean classes which implement the interfaces. This makes it flexible to have different implementation types of the bean classes.

After instantiating the *ObjectSet*, the data object can be instantiated by invoking the static method of the class *ErmFactory*. The static method must be specific to the object type. This means that the class *ErmFactory* must provides as many static methods as the number of the object types in the data model. Therefore, the class *ErmFactory* implementation is very bulky because there are several hundreds of different object types in the data model. The fact that why the class *ErmFactory* was implemented in static manner is not discussed here (mainly because the

ErmFactory class was implemented long time ago and at that time Generics in Java was not yet available[37]). Furthermore, the object instantiation requires the passed parameters of *ObjectSet* instance (as explained above) and the parts of POID. According to the specification of Topgallant® Persistent Object IDs (POID), POID, which are globally unique key descriptors for data entities handled by the Topgallant infrastructure and related software components, has the following string representation[38]:

```
<resource-containment-id>?<type>?<name>?<version>
```

The different parts of POID are internally assembled into the POID as the object identifier. This means that any created object is assigned with a unique POID.

In order to store the created objects into the persistent stored (based on LDAP-compliant directory server), at first the *ObjectManager* instance must be created so that the *Transaction* instance can be obtained from the *ObjectManager* instance. The *Transaction* instance will then manage the persistence operations applied on the created objects.

4.2.3. Solution and implementation

After having analyzed the current working mechanism of object instantiation, created objects management and persistence, the compatible working mechanism (on the base of relational database server as persistent store) is carried out by re-implementing the *ObjectSet* class and *ObjectManager* class in a simplified manner. This is because the legacy implementation of *ObjectSet* and *ObjectManager* are bulky and operate based on LDAP-compliant directory server. The *ObjectManager* class is re-implemented in a completely different manner based on the JPA *EntityManager* which provides the basic persistence operations on the base of RDBMS. One should note that the *ObjectManager* instantiation also includes database connection and thus requires database connection parameters such as database URL, database username and password. The *ObjectSet* class is re-implemented in a similar way to the legacy one in order to adopt the specific implementation type of the bean-style classes with JPA annotations (the legacy *ObjectSet* implementation is designated to another implementation type of the bean classes).

For the legacy and bulky object instantiation (in static manner) using *ErmFactory*, it is replaced with the generic implementation of the *InterfaceFactory* class. In fact, the class *InterfaceFactory* provides a generic static method used to instantiate object of any type and thus the specific object class must be provided. The object instantiation also requires the passed parameters of *ObjectSet* instance and the parts of POID (the same as the legacy working mechanism). The *InterfaceFactory* class contains a collection of *ObjectSets* to hold all created *ObjectSet* instances and this makes it possible to retrieve all existing *ObjectSet* instances later[39].
The compatible working mechanism is briefly shown in the following pseudo code together with the correspondent sequence diagram (depicted in **figure 4.2.2**)

---

[37] Generics was added to the Java programming language in 2004.
[38] Further detail of POID is presented in **part 4.3.1**
[39] All existing *ObjectSet* instances need to be retrieved in the process of resolving from memory (described at the end of the **part 4.3.1.2**)

```
// 1. For data object instantiation
// Instantiate ObjectSet
ObjectSet os = ObjectSet.getInstance(...);

// Instantiate some data object
Plate p1 = InterfaceFactory.create( os, "p1_rcid", Plate.class, "p1", "v1" );

//------------------------------------------

// 2. For storing data object
// Instantiate ObjectManager
ObjectManager objManager = ObjectManager.getInstance(dbURL, dbUserName, dbPassword);

// Start Transaction obtained from ObjectManager instance
objManager.currentTransaction().begin();

// Put the object (to be stored) into persistent state
objManager.makePersistent(p);

// Commit the Transaction so that all objects in persistent state can be stored
objManager.currentTransaction().commit();
```

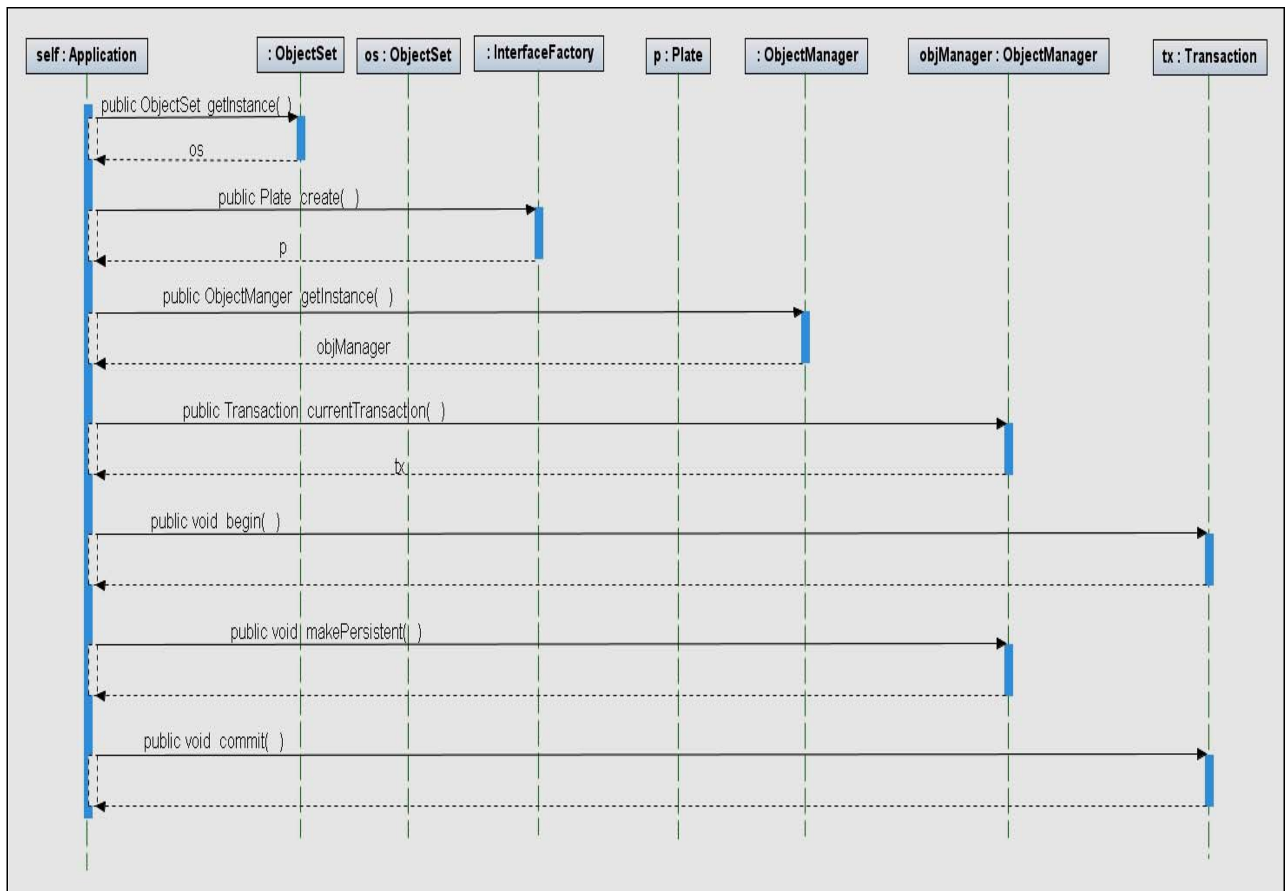

Figure 4.2. 2. Sequence diagram for the compatible working mechanism of object instantiation, created object management and persistence.

After implementing the compatible working mechanism of object instantiation, created objects management and persistence, the JPA-based persistence API implementation is carried out to provide advanced and easy-to-use persistence operations in addition to the basic persistence operations of the *ObjectManager* class (as mentioned above). The implemented persistence operations, which are based on JPQL[40] (JPA Query Language), mainly support retrieval of actual objects or object POIDs. For most of Topgallant® applications, the object POIDs (not the actual objects) will be returned from the retrieval operations because the retrieval purpose is often just to check the existence of objects. The actual objects are returned only on demand. One should note that the retrieval of object POIDs is much faster than the retrieval of actual objects. This is because the retrieval of actual objects results in the entire object graph in memory (including owning object and other associated objects of it).

4.2.4. Verification and conclusion

The compatible persistence API implementation is verified by re-using the testing scenario dedicated for verifying the functional capability of the generated relational database schema (presented in the **part 4.1.3**) with the following modification:

- Add the *ObjectSet* instantiation in order to replace the object instantiation using the *new* operator with the mechanism using *InterfaceFactory* class.
- Add the *ObjectManager* instantiation with database connection parameters such as database URL, database username and password. The *ObjectManager* instance is then used to store data objects into database.
- Replace the object retrieval using JPQL (JPA Query Language) with the developed methods in the compatible persistence API.
- Add one more test case for verifying the *ObjectSet* functionality of storing all the created objects: create several objects and then check the existence of the created objects in the *ObjectSet* instance.

The important conclusion is that the compatible persistence API implementation upgrades the current mechanism of object instantiation by replacing the bulky legacy *ErmFactory* (used to instantiate data object) with the generic implementation of *InterfaceFactory* which is much more light weight. Furthermore, when comparing the **figure 4.2.1** with the **figure 4.2.2**, it can also be seen that the required changes to the working mechanism of the legacy applications are not significant.

[40] Further detail of JPQL is presented at the end of the **part 2.2.1**

4.3. Essential functionalities

This chapter describes how to design, realize and implement the essential functionalities that are version control and management of secondary object storage/retrieval.

4.3.1. Version control

Version control functionality helps in tracking the changes over time to avoid general chaos so that it is easy to get back the previous working version or to get the latest version for synchronization. This functionality is definitely vital especially in the design- or production systems because the product components need to stored or saved as they are frequently modified or updated. This part begins with an overview introduction of version control together with the detailed analysis of version control requirements. After that, the solution and implementation is described and explained with clear figures and finally come the verification and the conclusion.

4.3.1.1. Introduction and requirement analysis

It is required that any stored entity is assigned with a time stamp version representing the point of time when the entity was created. According to the specification of Topgallant® Persistent Object IDs (POID), the version with hexadecimal representation is integrated in the POID which are globally unique key descriptors for data entities handled by the Topgallant infrastructure and related software components.

POID string representation form: `<resource-containment-id>?<type>?<name>?<`**`version`**`>`.

Resource-containment-id is the Base64 encoding of the internet address and hostname. *Type* is the fully-qualified Java type name (as a string) indicating type of the entity and *name* is the entity name which has to be unique within a resource/type context. In this thesis work, the fact that how the *resource-containment-id* is generated to guarantee the global uniqueness or how the POID generation done by a utility method of the company are not analyzed and specified.

In fact, version control requirement focuses mainly on the *version* as the last part in the POID. Particularly, version control solution has to ensure that if any entity loaded or retrieved from database is modified or updated, a new entity will be inserted into database with the POID the same as the loaded entity except the updated *version* and the modified content[41]. The following table makes it clear to see difference between the original entity and its various versions:

Entity name	POID	Entity content
Original entity	<resource-containment-id>?<type>?<name>?<version>	Original content
Version 1	<resource-containment-id>?<type>?<name>?<version_1>	Content 1
Version 2	<resource-containment-id>?<type>?<name>?<version_2>	Content 2

[41] Entity content mentioned here is actually the entity state or object state.

It is important that the only difference in POID between the original entity and its various versions is the *version* part and this enables to find out the latest entity based on the common *resource-containment-id*, *type* and *name* or to return a list of related entities with various versions. Further requirement is that no newly-versioned entity is inserted if the modified content is not different from content of the loaded entity and this implies a monitoring of real content change. Particularly, for primitive data types, different values lead to real content change and for complex data type, different object references (despite the same content) lead to real content change.

In addition to the general requirements specified above, there are other detailed requirements for the working mechanism of version control in the cases of actual object or **symbolic object** (also called **proxy**) and actual object reference or symbolic object reference. Particularly, modification (via setter method) and object attribute retrieval (via getter method) are handled straightforward for actual objects but complicated for proxies. This is because proxy works as a pointer, which contains only symbolic reference to a set of potential objects with some common POID parts and does not contain any concrete data. Therefore, getter and setter of the proxy have to be applied for some actual object resolved from the proxy (instead of from the proxy itself). This implies an internal resolving process inside the getter and setter methods. One should note that, the terminology **proxy** mentioned here is not the so-called proxy server in the Local Area Network (LAN) settings or in computer networks.

It is important to note that actual object (containing data) is identified by the unique POID and proxy (does not contain data) is identified by the wildcard POID whose one part or several parts are masked or not specified. Therefore, wildcard POID is used to reference a set of entities with some common POID parts. For example, the wildcard POID with the masked *version* part is used to reference the set of entities with the common *resource-containment-id*, *type* and *name* regardless of any versions.

Because of the fact that actual object is identified by the unique POID, if the object includes actual reference to another object, in version control context or in normal context, the owning object needs to resolve only the object it references to.

Version control context in database view for two different cases (actual object reference and symbolic object reference) is depicted in **figure 4.3.1** on the left hand side and the right hand side respectively. Furthermore, the equivalent application view is described in **figure 4.3.2**. The scenario is that, at the beginning, the Part entity named **p** is set with a reference to the Material entity named **m** (actual object reference is expressed with the continuous arrow and symbolic object reference is expressed with the dashed arrow) and the various versions are indicated with the suffix **v** followed by a number. The original version of the Part is **p_v1** and its different versions are **p_v2** and **p_v3** created as the version control general requirements (if any entity loaded from database is modified, a new version of it will be inserted into database) and similarly for the Material entity with a special notice that the entity **m_v?** is a proxy (only POID is initialized) which acts as a pointer referencing any **Material** entity with name **m** regardless of any version.

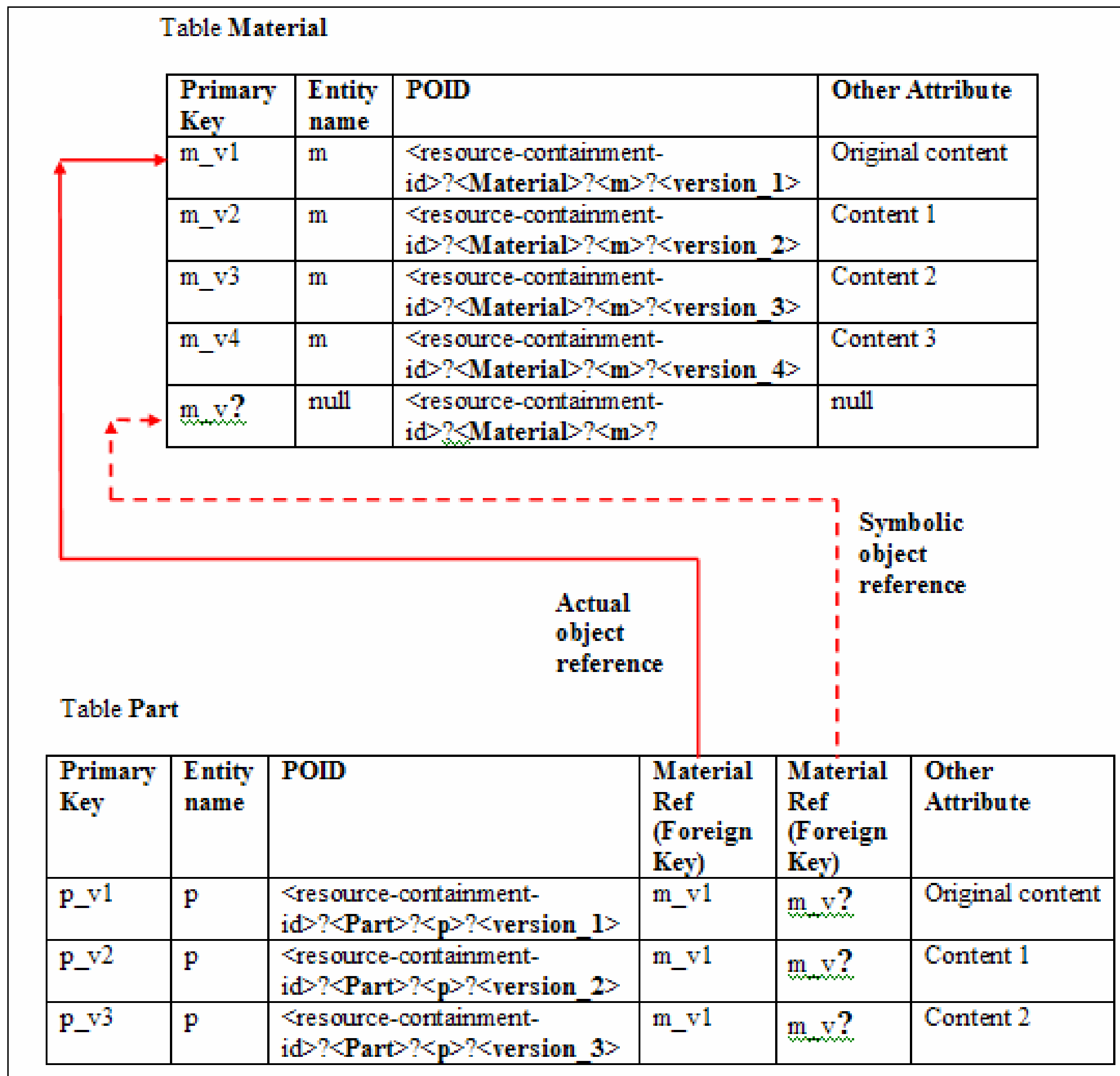

Table **Material**

Primary Key	Entity name	POID	Other Attribute
m_v1	m	<resource-containment-id>?<Material>?<m>?<version_1>	Original content
m_v2	m	<resource-containment-id>?<Material>?<m>?<version_2>	Content 1
m_v3	m	<resource-containment-id>?<Material>?<m>?<version_3>	Content 2
m_v4	m	<resource-containment-id>?<Material>?<m>?<version_4>	Content 3
m_v?	null	<resource-containment-id>?<Material>?<m>?	null

Table **Part**

Primary Key	Entity name	POID	Material Ref (Foreign Key)	Material Ref (Foreign Key)	Other Attribute
p_v1	p	<resource-containment-id>?<Part>?<p>?<version_1>	m_v1	m_v?	Original content
p_v2	p	<resource-containment-id>?<Part>?<p>?<version_2>	m_v1	m_v?	Content 1
p_v3	p	<resource-containment-id>?<Part>?<p>?<version_3>	m_v1	m_v?	Content 2

Figure 4.3. 1. Version control context (in database view) for actual object reference and symbolic object reference

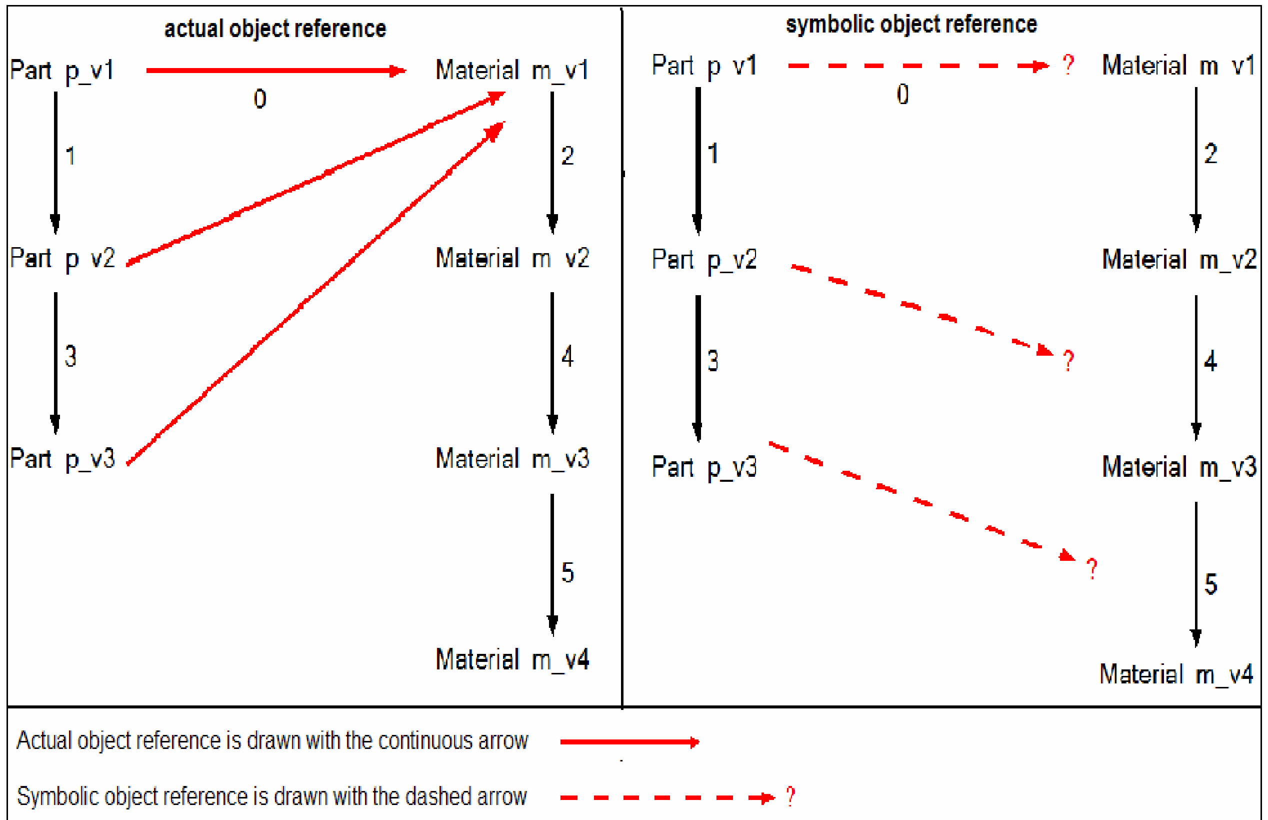

Figure 4.3. 2.Version control context (in application view) for actual object reference and symbolic object reference

Even though increasing the problem complexity in version control context, the symbolic reference (not the actual object reference) is the key concept throughout version control functionality.

In fact, from both **figures 4.3.1** and **4.3.2**, one can see that actual object reference is fixed and must be changed or reset manually, but symbolic object reference is dynamic and can refer to a set of matching entities from which the entity with latest version or some specified version can be determined.

4.3.1.2. Solution and implementation

Implementation of version control functionality is approached from simple case to real case. The simple case, which involves only actual objects and actual object references, is handled at first to reach the general requirements of version control. After that, the real case, which involves not only actual objects but also symbolic objects (proxies), is approached for full functionality of version control.

Flowchart for the simple case solution is depicted in the **figure 4.3.3**, which specifies generally the retrieval procedure and storing procedure in version control context. From the flowchart, one can see that the retrieved object has to be cloned manually and explicitly to some temporary object, on which the modification occurs, to prevent direct modification on the retrieved object.

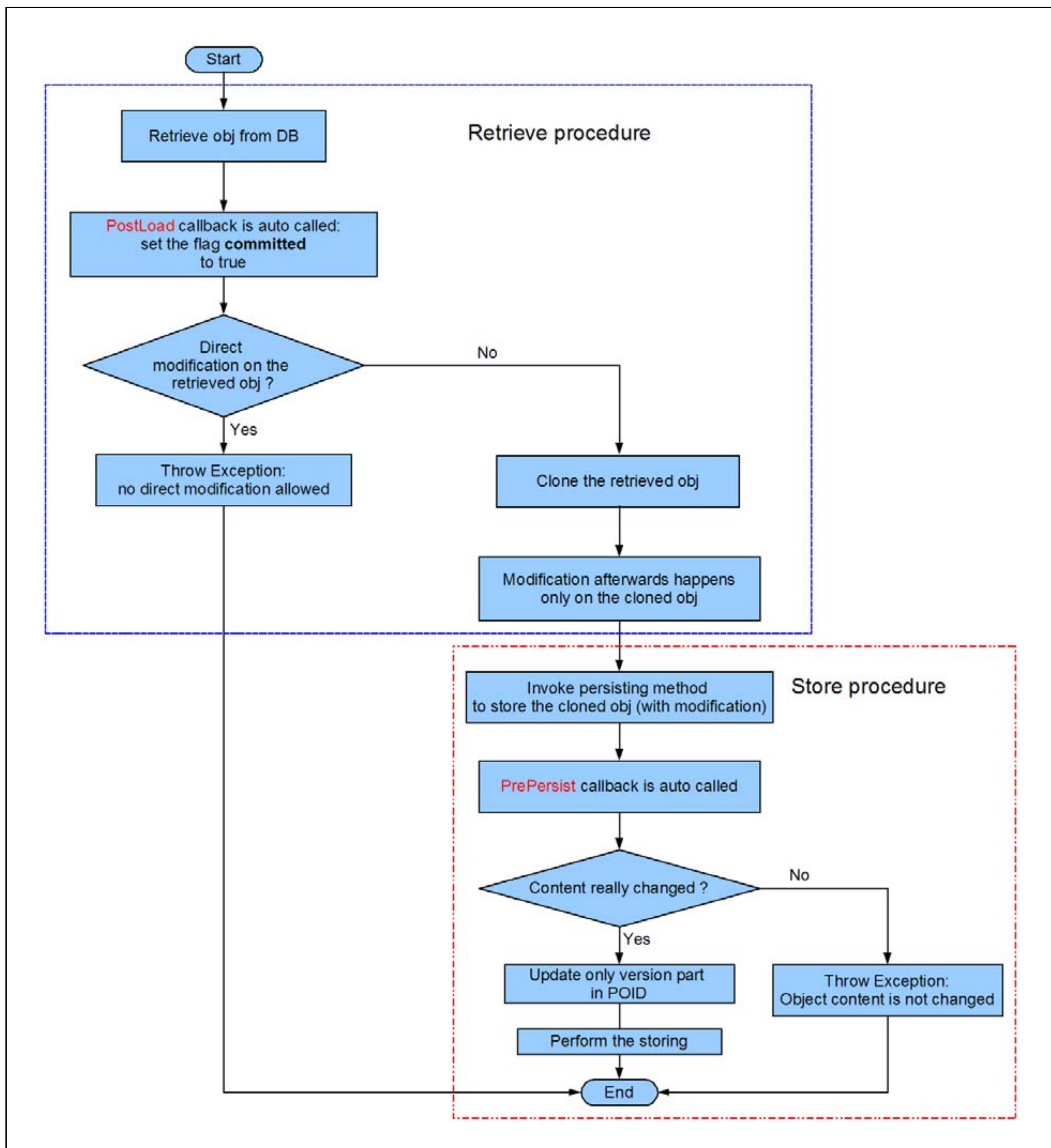

Figure 4.3. 3. Retrieval procedure and storing procedure in version control context in simple case

One of the functionalities supported by JPA is the callback mechanism (described in the **chapter 2**) utilized to solve the version control problem. Particularly, the Post-Load callback, which is automatically executed after the persistent entity is loaded from database into persistent context, is used to disable direct modification on the loaded object and the Pre-Persist callback, which is automatically executed before the object is stored into database, is used to monitor real content change and also to update only the version part in the POID. In fact, the version updating of the object to be stored is vital for version control functionality.

It is important to note that the above solution for the simple case is the fundamentals in solving version control problem for the real case, which requires additionally the internal resolving process to return the appropriate actual object referenced by the proxy.

The resolving process is needed by the fact that the getter and setter of the proxy are applied to the actual object resolved from the proxy. Furthermore, if the entity retrieved from database is a proxy, the resolving process will take place inside the cloning process. This is depicted in the *retrieve procedure* of the **figure 4.3.4** showing again the retrieval procedure and storing procedure in version control context, which is modified for dealing with proxy. From the figure, it can also be seen that if the entity to be stored is a proxy, the Pre-Persist callback will do nothing. This is because the proxy contains only wildcard POID referencing some actual entity and does not carry any concrete data. Via this, one can also see that proxy is stored as a normal entry in database. This means that proxy is instantiated as normal object with only the POID initialized.

As mentioned, proxy is the key concept throughout the version control functionality, thus, it is vital to understand how to set some attribute for the proxy or how to get some attribute of the proxy. From the **figure 4.3.5** describing the setter process, it is obvious to see that the setter method can receive an actual object reference or a symbolic object reference (i.e. wildcard POID). The setter method with wildcard POID enables the fact that the owning object can reference a set of other objects with common parts in the wildcard POID instead of a particular object as for actual object reference. Together with the **figure 4.3.6** showing the getter process, the core point in handling proxy is specified and that is whenever encountering the proxy, the first thing to do is to resolve the actual object referenced by the proxy.

Moreover, the setter process not only ensures that if the object state is committed (for example, after being loaded from database), no any modification can be done by throwing exception, but the setter process also monitors real content change. In deed, the setter process compares the value (either complex type or primitive type) received from outside with the current value of the object and if no real content change, the setter will end, otherwise the setter will set the flag *contentChanged* to true in order to signal the PrePersist callback to update the version part in POID before the object is stored into database.

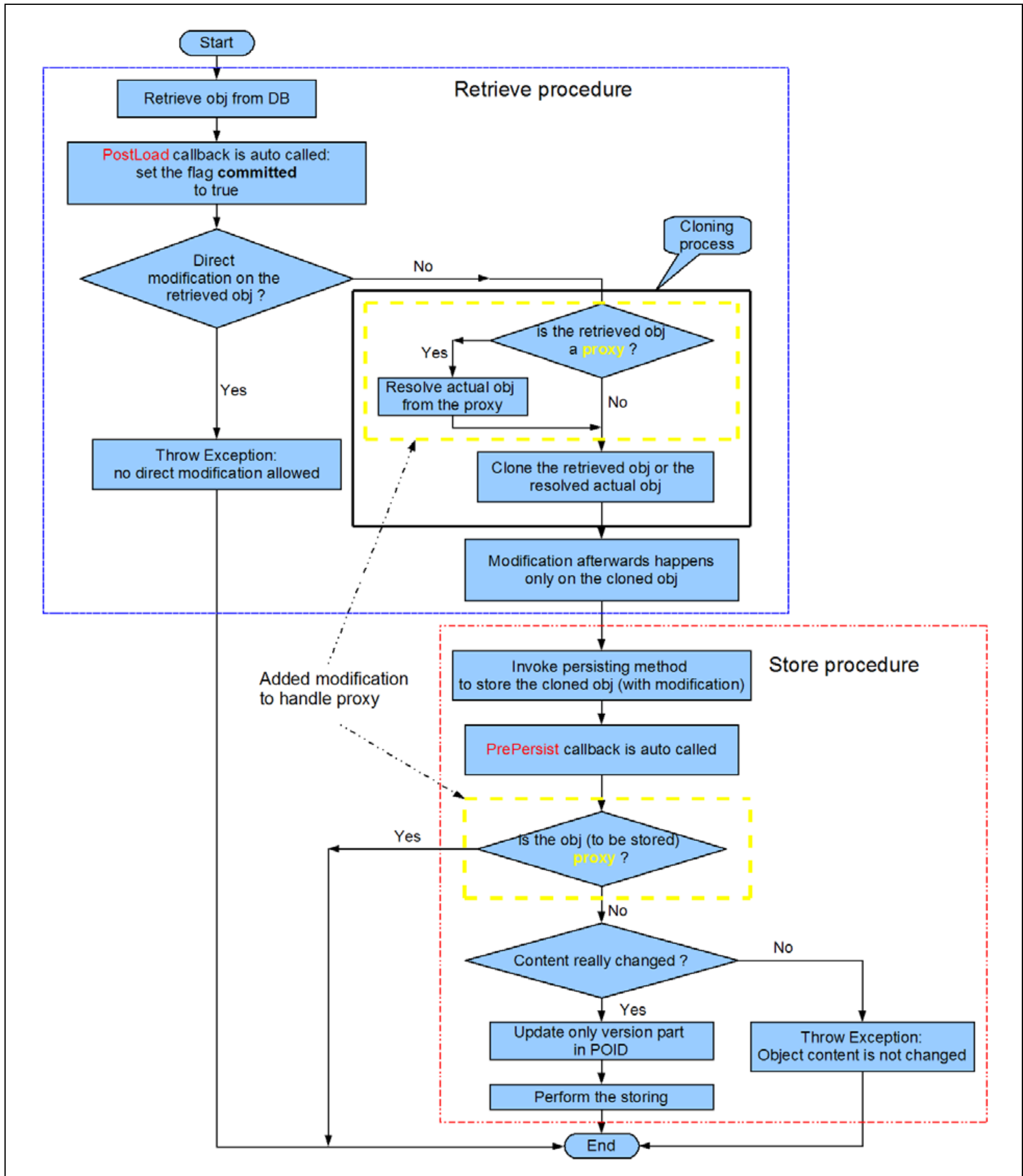

Figure 4.3. 4. Retrieval procedure and storing procedure in version control context in real case

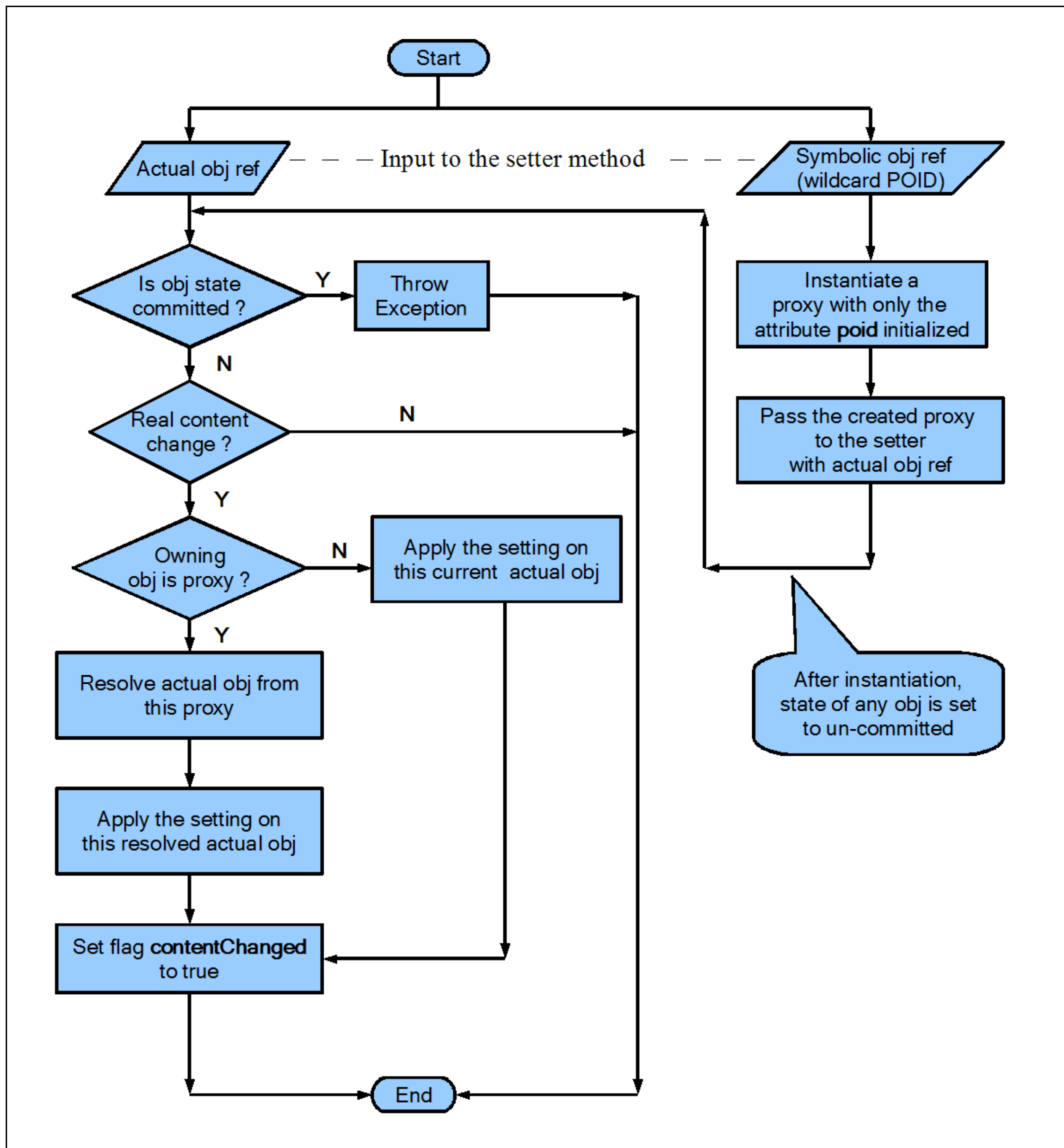

Figure 4.3. 5. Setter process

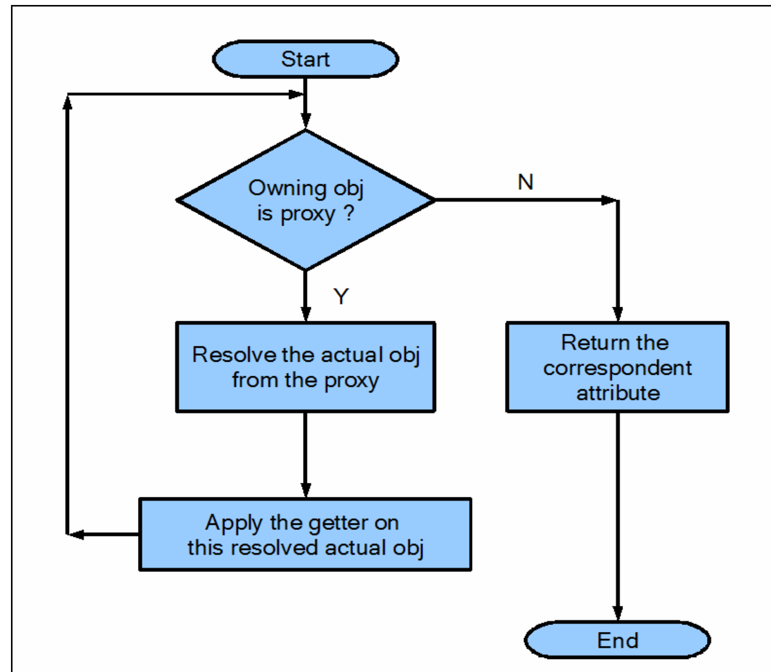

Figure 4.3. 6. Getter process

So far, the getter and setter process as well as the retrieval- and storing procedure in version control context have been specified. Via this, one can easily see that the resolving process is always involved to look for the actual object from the proxy. One should note that, it is only necessary to resolve the actual object if the setter and getter methods are invoked on the proxy in order to avoid the full object graph carried by the actual object.

It is required that the resolving process (depicted in the **figure 4.3.7**) at first looks for the actual object referred by the proxy from memory and then if no actual object found, the resolving process continues searching from database.

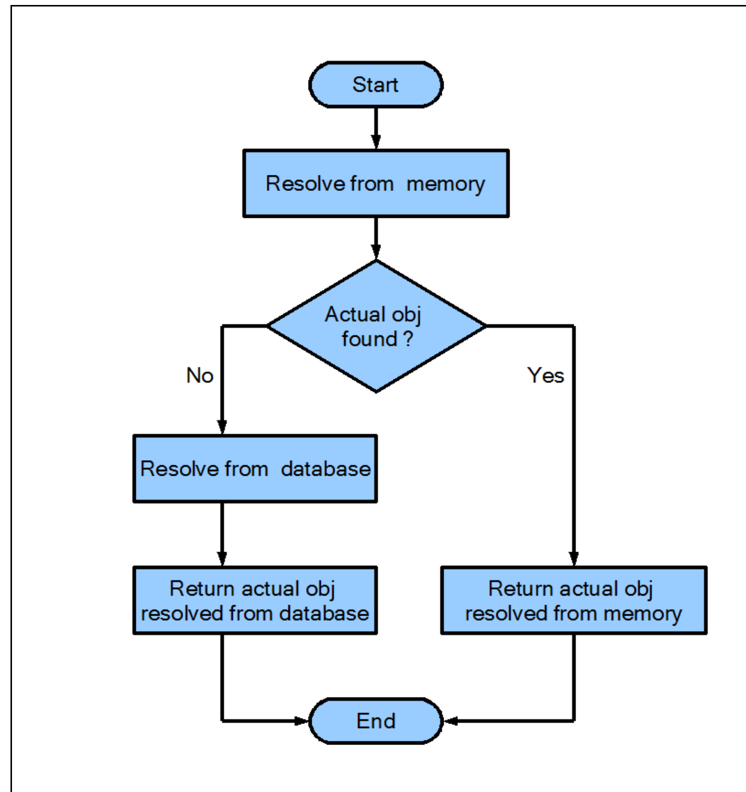

Figure 4.3. 7. General resolving process

The resolving process from database (depicted in the **figure 4.3.8**) is required to return to the client side only one actual object with the latest version instead of returning a set of objects matching the wildcard POID in order to avoid the fact that the client will have to perform extra process of determining the latest one from the returned set of objects. This is done by performing a nested SELECT query in which the inner SELECT query (executed first) is used to determine the latest version among the entities matching the wildcard POID and then the outer SELECT query is used to return the entity, which has the version determined above. The advantage of this resolving procedure is that the burden of searching is shifted to the server side thanks to the nested SELECT query.

Because of the fact that any entity, which is returned from the SELECT query, is always included by the *EntityManager* which creates that query, it is vital to perform the SELECT query using the appropriate *EntityManager* which contains the proxy object in order to ensure that the resolved actual entity will also be included by that appropriate *EntityManager*. Both proxy entity and its resolved actual entity must be included by the same *EntityManager*. This is especially vital in the case of proxy cloning because the cloning process eventually returns the resolved actual entity (**figure 4.3.3 and figure 4.3.4**) which is required to be attached to the current *EntityManager* of the application for the later coming store-procedure.

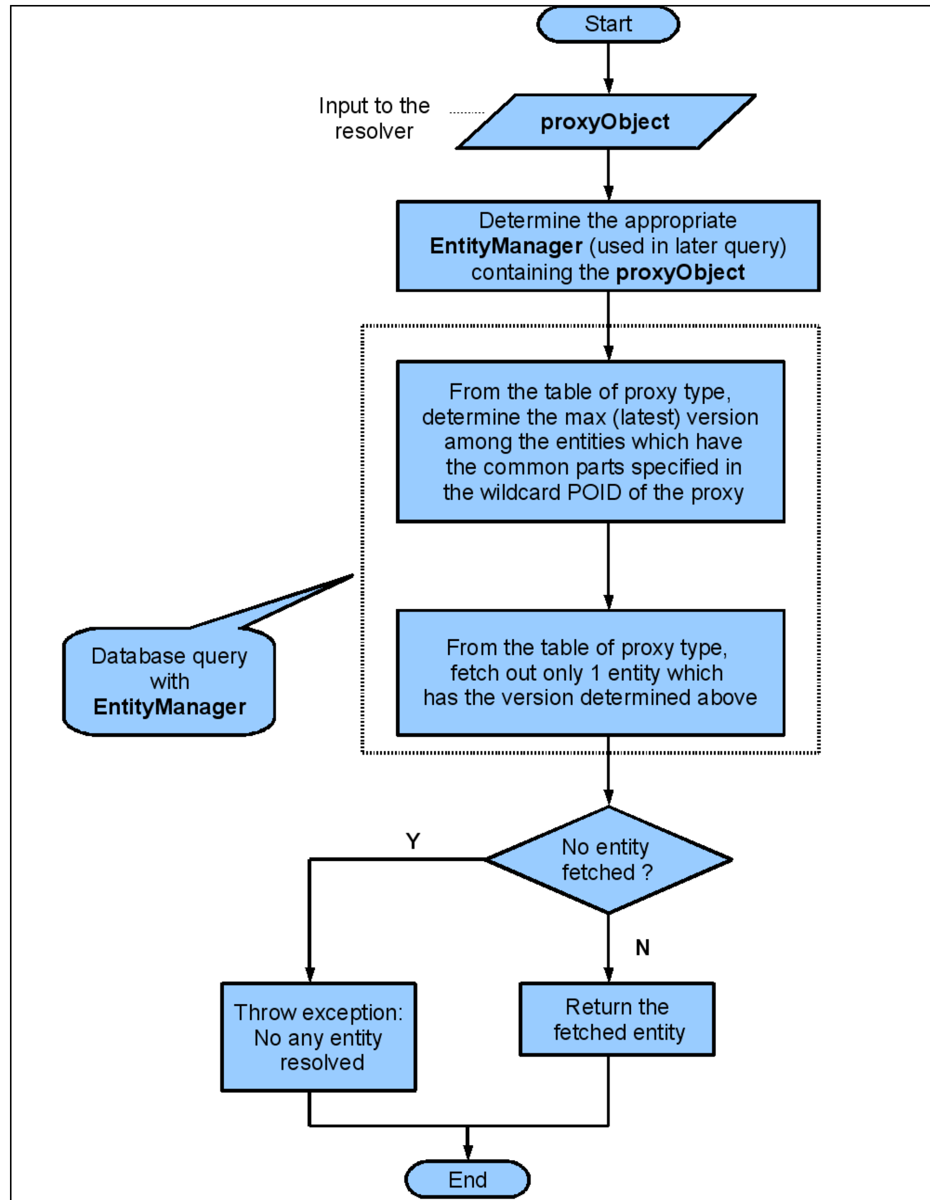

Figure 4.3. 8. Resolving process from database

The resolving process from memory (based on the proxy object) (depicted in **figure 4.3.9**) is executed before the resolving from database to avoid unnecessary database hits. In order for the resolving from memory to work, it is necessary to design the suitable container, which can hold all the currently managed and involved objects. The container (as a simple Java List) is included in the *ObjectSet*[42] class as an attribute. At the beginning, the instance of *ObjectSet* class is created and afterwards the *ObjectSet* instance is passed as parameter to the instantiation of any object so that any created object will automatically be put into the container (as a simple Java List) of this *ObjectSet* instance.

To sum up, the mechanism of resolving from memory is based on the fact that all created objects (e.g. by the application) are automatically stored in the container (a simple Java List) of the *ObjectSet* instance. After having the container, it is easy to determine the latest actual object (in the container) based on the fact that the latest actual object is the one which has the same type as the proxy object type and matches the wildcard POID of proxy object with the **max** version value.

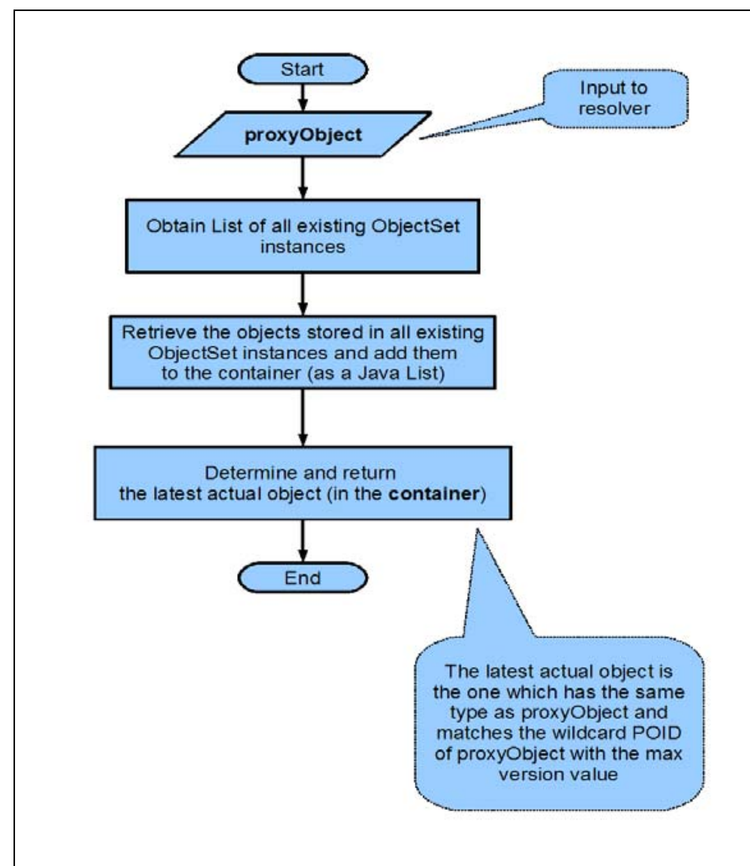

Figure 4.3. 9. Resolving process from memory

[42] Detail specification of ObjectSet is described in **part 4.2**

Last but not least, for the version control feasibility, that is the design of the diagram for (predefined) top classes including interfaces, abstract classes and base implementation classes specifying common attributes and methods for any entity instances of the data model. From the **figure 4.3.10**, one can easily see some common data attributes like *creator*, *POID* which are explicitly mapped to database schema and some must-have technical attributes used to implement the version control functionality like *contentChanged* (to monitor real content change of the object), *isCommitted* (to indicate object state for preventing direct modification on the object loaded from database). One should note that it is not necessary to map the technical (non-data) attributes to database schema because they do not carry information and they are used as helper attributes.

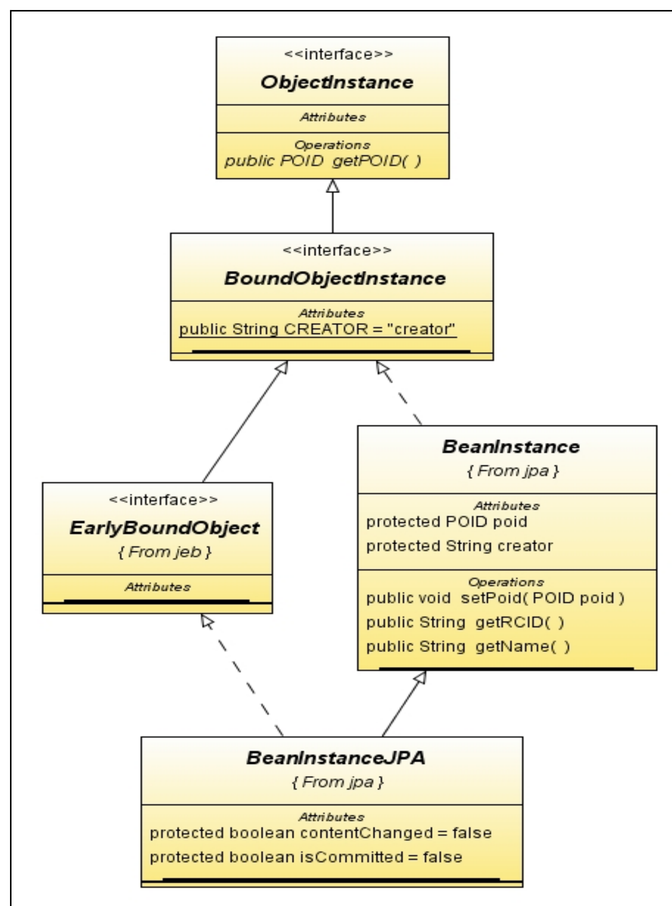

Figure 4.3. 10. Top base class inheritance hierarchy for version control

After the version control solution and implementation had been verified and tested (described in the coming **part 4.3.1.3**), the version control functionality was integrated to the Enterprise Reference Model (ERM) which is generated by the code generator (described in the **part 4.1**). Particularly, the *PrePersist* and *PostLoad* callbacks are associated with the root parent classes (in the ERM) so that any sub classes can also be associated with the callback features. The most

important is that the default simple getter and setter methods were replaced with the setter and getter process (**figure 4.3.5** and **4.3.6** respectively) in the version control context.

4.3.1.3. Verification

The version control implementation was first tested (using JUnit testing framework) with a simple artificial data model (depicted in the **figure 4.3.11**) in which the *InformationObject* is the root parent class inherited by the *Shape*, *Plate* and *Material* classes. Furthermore, it is clear to see the unidirectional One-to-One relationship between *Plate* and *Material*, the unidirectional One-to-Many relationship between *Plate* and *Shape* as well as the self-referential Many-to-Many relationship of the *Plate*. One should note that the data model has to inherit from the top class model (depicted in **figure 4.3.10**).

The tests include all the important cases of actual object and symbolic object (proxy) as well as actual object reference and symbolic reference (indicating proxy attribute). The test cases are as followings:

- Modify the actual entity (stored in database) to verify that a new version of it is inserted into database.
- Load the **actual** entity from database and then:
  - try retrieving some proxy attribute of it to verify the getter process.
  - or try overwriting some proxy attribute of it to verify the setter process.
- Load the **proxy** entity from database and then try modifying its attribute. The expected is that attribute of the correspondent actual entity resolved from the proxy is to be modified. In this test case, both resolving from memory and resolving from database processes are verified with the expectation that:
  - Resolving from memory must occur before the resolving from database
  - If the actual entity is already resolved from memory then the resolving from database must not occur.
  - If no actual entity can be resolved from memory then the resolving from database must take place. Finally, if no any actual entity can be resolved then exception must be thrown.

The above test cases are carried out for all common scenarios whose complexity is increased from unidirectional One-to-One relationship to unidirectional One-to-Many relationship as well as the self-referential Many-to-Many relationship.

Finally, the version control functionality was verified with the real Topgallant® ERM data model using again the test cases described above.

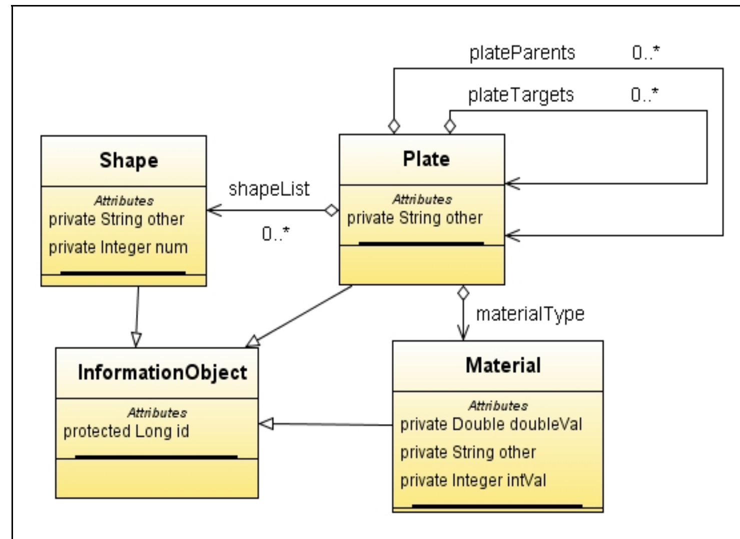

Figure 4.3. 11. Simple data model for version control testing

4.3.1.4. Conclusion

Version control functionality is not visible to the application- or client side because it is integrated into the Enterprise Reference Model (ERM) that is generated by the code generator. In fact, version control functionality is expressed invisibly via the object retrieval, object modification (using setter methods) and object storing. Particularly, after the object is retrieved from database, the *PostLoad* callback is automatically invoked to set the retrieved object status to “commited” in order to prevent direct modification on the retrieved object and thus, the retrieved object must be manually cloned for later modification. The modification and storing will then happen only on the cloned object otherwise exception will be thrown. Whenever the getter and setter methods are invoked, the internal resolving processes (from memory and then from database) come into play. The *PrePersist* callback is automatically invoked to update the object version before storing the object into database if the object state is really changed.

Symbolic object or proxy is one of the key concepts throughout the version control functionality whose working mechanism is mainly based on the specially-designed getter and setter process with the internal automatic resolving property. The advantage of using proxy is to avoid the same overhead as the actual object represented by the proxy. Furthermore, the callback mechanism (supported by JPA) particularly *PostLoad* and *PrePersist* is the prerequisite to implement the version control.

4.3.2. Management of secondary object storage/retrieval

4.3.2.1. Introduction and requirement analysis

From the inheritance hierarchy (depicted in **figure 4.3.2.1**) generated from the interfaces in the Topgallant® ERM data model, it can be seen that objects in the data model are classified into two main categories that are the first-class category and the secondary category. First-class objects are the information objects containing unique identification, type(s), origin and location, creator, timestamp/version, parental relationships, dependencies (to other objects through reference) and so on. The first-class category is the biggest group with most of the interconnected entities having the common root parent class of *InformationObject*. In fact, if any entity has the root parent class which is this *InformationObject* class or in other words, if any entity is reached from this *InformationObject* class then that entity belongs to the first-class category. The secondary category includes the remaining entities which do not belong to the first-class one.

The current working mechanism of management of secondary object storage/retrieval for the legacy persistence API based on LDAP-compliant directory server is presented as followings. It is important to note that the legacy persistent store based on LDAP-compliant directory server does not support direct storing of secondary objects. This is because secondary objects are not created independently and secondary objects are never queried. Only first-class objects are instantiated and then stored into the persistent data store for later query service. Mostly, first-class objects include reference of secondary objects. Once the first class object is stored into the persistent store, an XML representation of that first class object is automatically generated[43]. All attributes (regardless of types that can primitive type, other first-class references, other secondary-class references and so on) of the first-class object are included in the generated XML representation. The entire generated XML representation is bundled into a zip archive that is stored (in binary) in a special attribute of the stored first-class object (i.e. any first class has a special attribute reserved to hold the generated XML representation of its object state).

From the **figure 4.3.2.2** (showing first-class entries stored in LDAP directory), one can see that the attribute named "javaSerializedData" whose type is of BINARY is the special attribute holding the generated XML and from the **figure 4.3.2.3**, one can also see some portion of the generated XML representation of a *Plate* object (class *Plate* is the first class). Therefore, one can say that first-class objects (stored directly in the persistent store) act as containers for all related secondary objects.

Because the data of secondary objects is stored in the zipped XML representation (in binary), the retrieval of data of secondary objects is handled in such a special way that it is still possible to get the data of secondary objects via the getter method of the owning first-class object. Detail of how to handle the retrieval of secondary objects is not discussed in this work.

It is strongly required that the compatible JPA-based persistence API on the base of RDBMS must also adopt the same way of management of secondary object storage/retrieval as described

[43] Detail of how the XML representation can be automatically generated (using JDOM, www.jdom.org) is not of interest in this work.

above. Particularly, all secondary objects are not allowed to store directly into the relational database and the retrieval of data of secondary objects must be handled in such a way that it is still possible to get secondary object data by simply invoking the getter method of the owning first-class object.

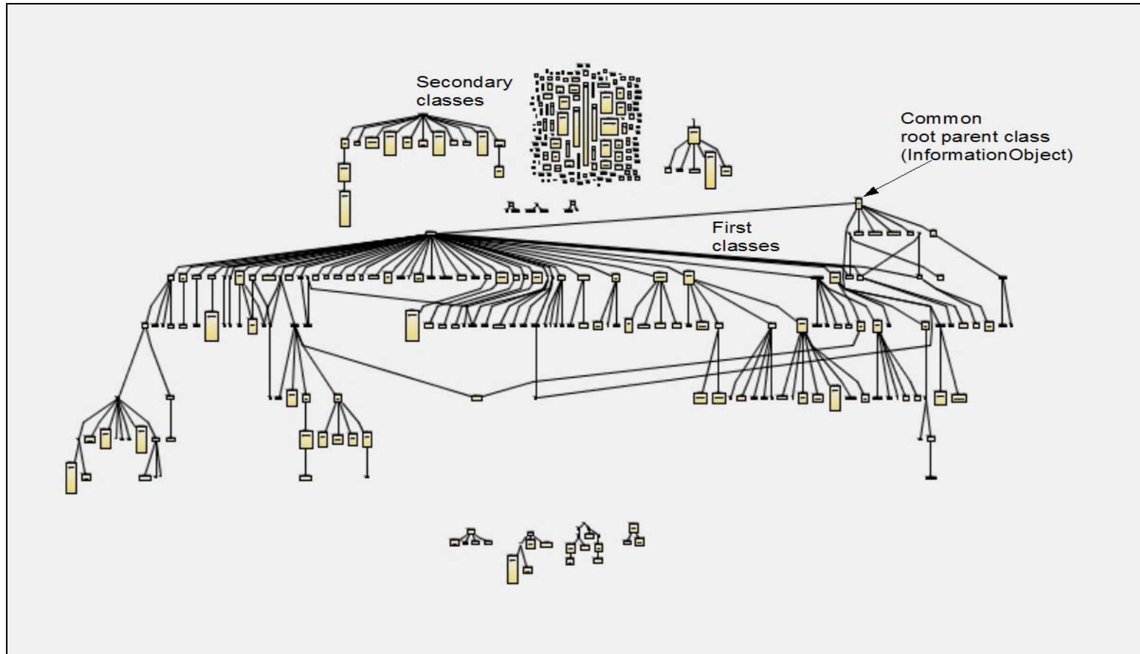

Figure 4.3.2. 1. Inheritance hierarchy of the interfaces in Topgallant® data model

LDAP Browser\Editor v2.8.2 - [ldap://tornado:11389/dc=QADTest]

File Edit View LDIF Help

dc=QADTest
cn=Genesis
tgisErmType=org.atlantec.binding.erm.AssemblyNode
tgisErmType=org.atlantec.binding.erm.AssemblyProductionPrepDescription
tgisErmType=org.atlantec.binding.erm.CurveGeometry
tgisErmType=org.atlantec.binding.erm.Material
tgisErmType=org.atlantec.binding.erm.Plate
cn=FV8561—D6AP—10P
tgisVkey=wKgqcfs5Dag.gzkf5qVUTS8$10985c235bc34000
cn=FV8561—D6AP—11P
cn=FV8561—D6AP—14P
cn=FV8561—D6AP—1P
cn=FV8561—D6AP—2P
cn=FV8561—D6AP—3P
cn=FV8561—D6AP—4P
cn=FV8561—D6AP—5P
cn=FV8561—D6AP—6P
cn=FV8561—D6AP—7P
cn=FV8561—D6AP—8P
cn=FV8561—D6AP—9P

Attribute	Value
cn	10P
javaSerializedData	BINARY (8Kb)
objectClass	tgisInformationObject
objectClass	top
tgisAlias	tribon.hprod.PARTNAME_LONG:8561-KT1-9P
tgisAlias	tribon:FV8561-D6AP-10P
tgisContributor	wKgqcfs5Dag.gzkf5qVUTS8//file:C:/devzone/T
tgisCreator	tk
tgisEffectivity	wKgqdfs5Dag.6ChkZGTiTtE?org.atlantec.bind
tgisErmType	org.atlantec.binding.erm.Plate
tgisExtendedAttr	topgallant.hprod.bevel.chamfer.exists:false
tgisExtendedAttr	topgallant.hprod.bevel.two_side.exists:false
tgisExtendedAttr	topgallant.hprod.marking.gsd.ts.cnt:0
tgisExtendedAttr	topgallant.hprod.bevel.one_side.exists:true
tgisExtendedAttr	topgallant.hprod.bevel.cvb.exists:false
tgisExtendedAttr	topgallant.hprod.bevel.one_side.length.min:0.
tgisExtendedAttr	topgallant.hprod.bevel.one_side.length.max:2.
tgisExtendedAttr	topgallant.hprod.bevel.one_side.angle.max:0.
tgisExtendedAttr	topgallant.hprod.bevel.one_side.angle.min:0.0
tgisExtendedAttr	tribon.hprod.GPS2:KT1
tgisExtendedAttr	topgallant.hprod.contains.knuckle:false
tgisExtendedAttr	tribon.hprod.PLATE_SIDE:PS

Figure 4.3.2. 2. First-class entries stored in LDAP directory

```
<?xml version="1.0" encoding="UTF-8" ?>
- <tg:ObjSet xmlns:tg="http://www.topgallant.org/jeb/tgeb" xmlns="http://www.topgallant.org/xsd/org.atlantec.binding.erm" tg:version="2" tg:replica="1"
   tg:vmodel="history" tg:creator="tk" tg:rcid="wKgqdfs5Dag.Xwug23eZGfQ">
 - <Plate tg:poid="wKgqcfs5Dag.gzkf5qVUTS8?.Plate?FV8561–D6AP–10P?10985c235bc34000" commonName="10P" position-
     r="wKgqdfs5Dag.Xwug23eZGfQ?.CoordinateSystem?26ca17?10af8064940a51b9" positionNumber="9" surfaceArea="37.76291231" materialThickness="0.0095"
     materialType-r="?.Material?AH36?=10985c235bc34000" type="STANDARD_PLATE" cog-r="wKgqdfs5Dag.Xwug23eZGfQ?.Point?26ca21?10af8064940a51c3"
     dimensions-r="wKgqdfs5Dag.Xwug23eZGfQ?.BoundingBox?26cca4?10af8064940a5447">
   - <tg:Sequence tg:name="contributors">
       <tg:Item tg:val="wKgqcfs5Dag.gzkf5qVUTS8//file:C:/devzone/Topgallant-1.4/adapter/tbspp/DSEB=ece7bcjava.awt.EventDispatchThread$e77
         $pool_5F2_5Fthread_5F7-10af8064940a51b5.xml" tg:t="St" />
     </tg:Sequence>
   + <tg:Sequence tg:name="aliases">
   + <tg:Sequence tg:name="extendedAttributes">
   - <tg:Sequence tg:name="parents">
       <SteelDesignStructureNode tg:r="?.SteelDesignStructureNode?FV8561–D6AP?=10985c235bc34000" />
     </tg:Sequence>
   + <tg:Sequence tg:name="features">
   - <tg:Sequence tg:name="manufacturingBoundary">
       <CurveGeometry tg:r="wKgqcfs5Dag.gzkf5qVUTS8?.CurveGeometry?FV8561-D6AP-10P.gen#burningContour#1?10985c235bc34000" />
     </tg:Sequence>
   </Plate>
   <KeyValue tg:poid="-?-?26ca16?10af8064940a51b8" value="FV8561-D6AP-10P" key="tribon" />
   <CoordinateSystem tg:poid="-?-?26ca17?10af8064940a51b9" origin-r="wKgqdfs5Dag.Xwug23eZGfQ?.Point?26ca18?10af8064940a51ba" axis1-
     r="wKgqdfs5Dag.Xwug23eZGfQ?.Vector?26ca19?10af8064940a51bb" axis2-r="wKgqdfs5Dag.Xwug23eZGfQ?.Vector?26ca1a?10af8064940a51bc" />
   <Point tg:poid="-?-?26ca18?10af8064940a51ba" value="{0.0 0.0 26.85}" />
   <Vector tg:poid="-?-?26ca19?10af8064940a51bb" value="{1.0 0.0 0.0}" />
   <Vector tg:poid="-?-?26ca1a?10af8064940a51bc" value="{0.0 1.0 0.0}" />
   <KeyValue tg:poid="-?-?26ca1b?10af8064940a51bd" value="8561-KT1-9P" key="tribon.hprod.PARTNAME_LONG" />
   <KeyValue tg:poid="-?-?26ca1c?10af8064940a51be" value="PS" key="tribon.hprod.PLATE_SIDE" />
   <KeyValue tg:poid="-?-?26ca1d?10af8064940a51bf" value="363" key="tribon.hprod.SHIP_NO" />
   <KeyValue tg:poid="-?-?26ca1e?10af8064940a51c0" value="KT1" key="tribon.hprod.GPS2" />
   <KeyValue tg:poid="-?-?26ca1f?10af8064940a51c1" value="8561" key="tribon.hprod.GPS4" />
   <KeyValue tg:poid="-?-?26ca20?10af8064940a51c2" value="1" key="tribon.hprod.MIRRORED" />
```


Figure 4.3.2. 3. Generated XML representation of a Plate object

4.3.2.2. Solution

At first, the solution for secondary object storage is described as followings. In the earlier task of migrating the legacy persistent store to RDBMS (described in the earlier **chapter 4.1**), the requirement analysis had also mentioned briefly about the fact that secondary class objects are not allowed to store directly into the relational database and thus the correspondent solution is to exclude the all secondary classes as well as all the secondary class typed attributes of first classes from the mapping to relational database schema by using the *@Transient* annotation (described in **part 4.1.2.3**). This means that, the relational database schema (generated from the Java-based data model) does not have any table of secondary class (recall that, in object-relational mapping, normally a class is mapped to a table) and thus, secondary class objects can never be stored directly into relational database. The completion of the task of migration to RDBMS also did help to prevent direct storing of secondary objects.

The remaining problem in handling of secondary object storage is how to keep data of secondary objects associated with the first-class object in the case the first-class object includes references

of secondary objects (i.e. the first class includes some secondary class typed attributes). From the illustration in **figure 4.3.2.4**, one can see that the *Plate* class (that belongs to first-class category) includes (unidirectional) reference of the *Feature* class. Because *Feature* class is secondary class, it does not have any correspondent table in database schema. Once the *Plate* instance is stored into database, the data of *Feature* instance referenced by its owner (the *Plate* instance) can not be kept because it is excluded from the storing of the *Plate* instance. Therefore, the *Plate* instance stored in database does not have any data of the *Feature* instance associated with it on the object instantiation and initialization.

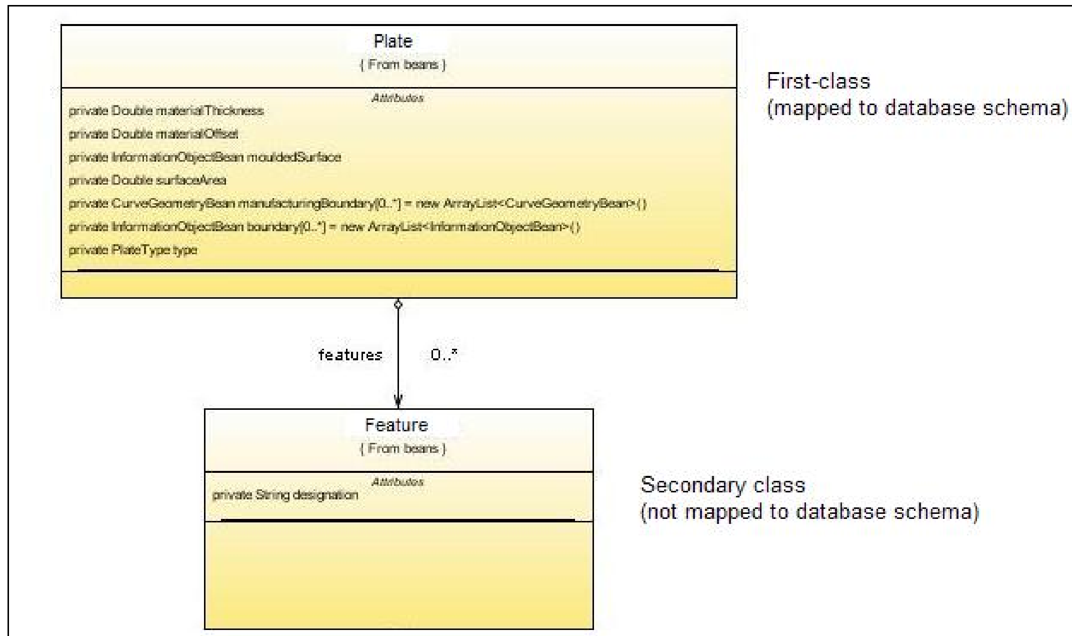

Figure 4.3.2. 4. First class includes reference of secondary class

The solution for keeping data of secondary objects associated with first-class objects (depicted in **figure 4.3.2.5**) is to make use of the *PrePersist* callback mechanism supported by JPA[44]. Particularly, right before the first-class object is stored into database; the *PrePersist* callback is automatically invoked to generate the XML representation of the first-class object. The generated XML contains the entire state of the first-class object, thus the data of secondary objects associated with the first-class object can be included in that XML representation. One should note that, the clustering of the objects (associated with the owning first-class object) to be included in the generated XML is not considered at this phase. Purpose of the object clustering is to have only the objects within the same scope included in the generated XML. In order to be compatible with the current mechanism of management of secondary object storage (described in **part 4.3.2.1** above), the generated XML representation must also be bundled into a zip that is stored (in hex) in a special attribute of the stored first-class object. Any first class has a special attribute (named "*xmlData*" of type byte array and mapped with *@Lob*[45] JPA annotation to relational database schema) reserved to hold the generated XML representation of its object state.

[44] Further detail of JPA callback is described in **part 2.2.4**
[45] *@Lob* annotation is used to map a byte array to a type of BLOB in relational database schema

Therefore, after generating the XML, the *PrePersist* callback continues to assign the generated zipped XML for the special attribute (named “*xmlData*”) of the first-class object and finally the first-class object is stored into database.
The XML generation is done by utilizing the marshalling capability of JAXB[46]. Particularly, any first class (including parent classes and sub-classes) is specified with JAXB annotation of *@XmlRootElement* so that the first class objects can be converted into XML representation which holds the entire object state. It is important that to note that in the Topgallant® ERM data model, there does not exist the case in which secondary class has reference of first class. Therefore, all secondary classes are **not** specified with any JAXB annotation because the secondary class object state needs not to be kept.

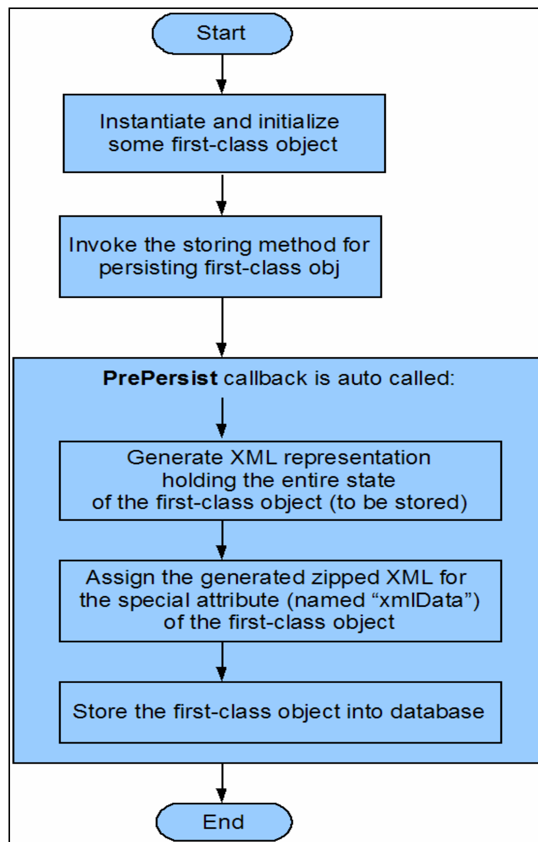

Figure 4.3.2. 5. Solution for keeping data of secondary objects associated with first-class objects

After having solved the problem of secondary object storage, the next focus is on how to get back the data of secondary objects via the getter method of the owning first-class object. It is clear to see that data of secondary objects is stored indirectly (in the XML representation) and thus the retrieval of data of secondary objects can not be handled in a straightforward manner. The solution for retrieving data of secondary objects is depicted in **figure 4.3.2.6**.

[46] Further detail of JAXB is described in **part 2.3**

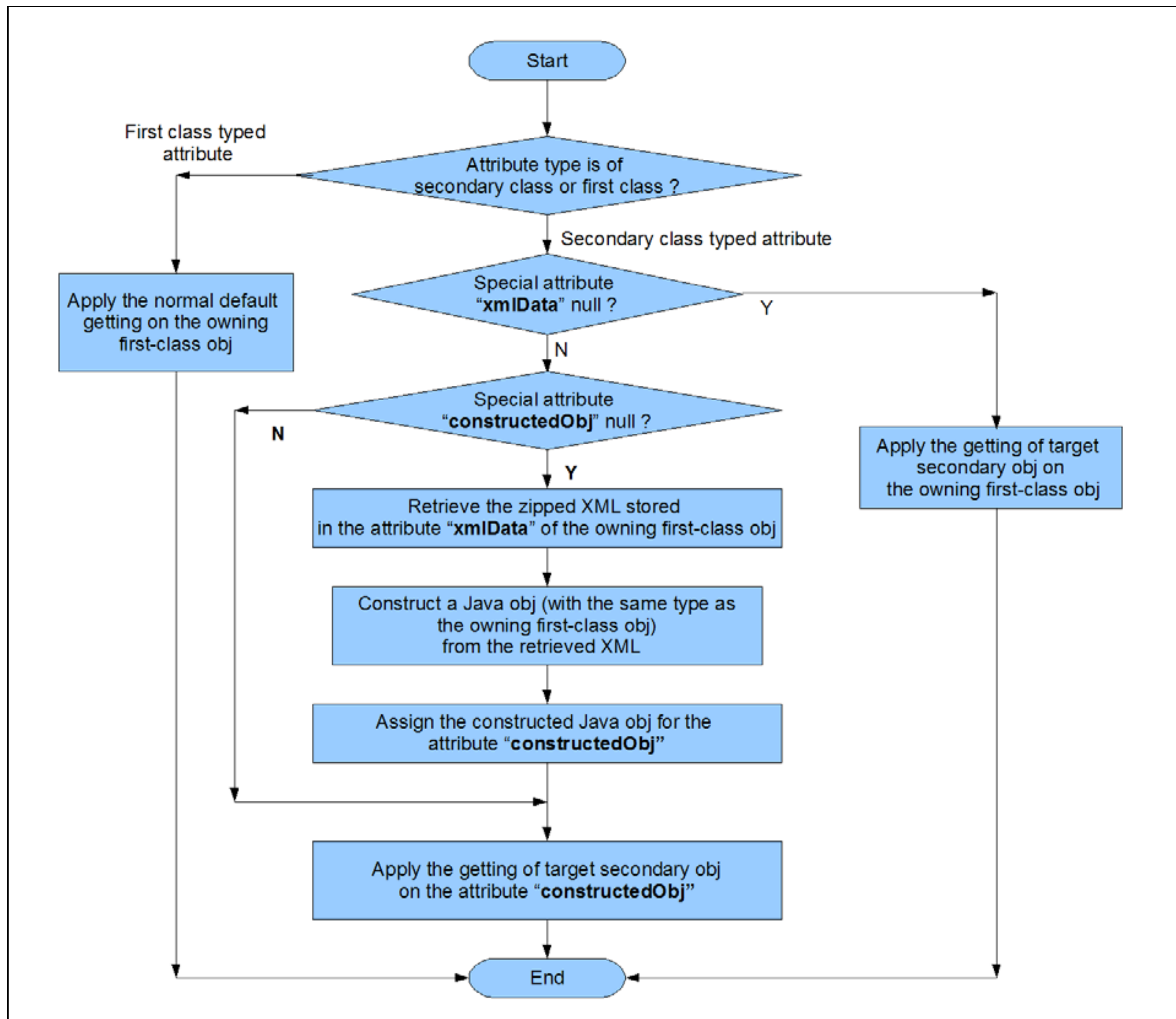

Figure 4.3.2. 6. Internal process of the owning first-class object's getter method for retrieving secondary object data

Particularly, the internal process of the getter method of the owning first-class object (loaded from database) for retrieving data of some target secondary object is as followings. At first, the zipped XML stored in the special attribute (named "*xmlData*") of the loaded first-class object is retrieved (by simply invoking the getter on that special attribute) and then the retrieved XML (in which data of the target secondary object is kept) is converted into some Java object (this process is called un-marshalling). From the Java object constructed from the XML, it is easy to return the target secondary object by invoking the getter method of that Java object. It is especially important to note that the process of retrieving the XML and then constructing the Java object from the retrieved XML occurs only one time on the first invocation of the getter method in order to avoid the expensive un-marshalling process (convert XML back to Java object). This is simply done by storing the Java object constructed from XML in the owning first-class object's attribute (named "*constructedObj*").

One should note that, if the special attribute is null (or not initialized) then it means that, the first-class object is still not stored into database and in that case, the getter method simply returns the target secondary object referenced by the owning first-class object. Furthermore, one should also note that, if the getter method is applied on the first-class typed attribute then the default getting process will simply be taken. First-class objects are defined as the ones having the common root parent class of *InformationObject* (mentioned above) and the other objects (that are not reachable from the *InformationObject* class) belong to the secondary-class category.

The following pseudo code illustrates an example (for better understanding of the solution) based on the **figure 4.3.2.4** above (which indicates that the *Plate* class as the first class includes a reference of the *Feature* class as the secondary class):

```
// Load some Plate instance from database
Plate loadedPlate = ... ;

// Invoke getter of the Feature (secondary class) to get data
Feature featureData = loadedPlate.getFeature();

// The internal process of the getFeature() method is as followings:
  // First, retrieve the zipped XML stored in the special
       // attribute of the "loadedPlate" object.
  // Name of the special attribute is "xmlData"
       retrievedXML = loadedPlate.getXmlData();

  // Then, convert XML back to some Java object
  Plate constructedObj = unMarshal(retrievedXML);

  // Apply the getting of Feature on the constructed Java obj
  return constructedObj.getFeature();
```

From the above pseudo code, it is clear to see that the `getFeature()` method is passed from the `loadedPlate` (as the owning first-class object) to the `constructedObj` (as the Java object constructed from the XML stored in the special attribute of the `loadedPlate`). One can conclude that, the retrieval of secondary object is actually performed by the getter method of the Java object constructed from the XML that holds the entire state of the owning first-class object.

In addition to the solution for retrieving data of secondary objects via the getter method of the owning first-class object, the solution for modifying data of secondary objects via the setter method of the owning first-class object is also essential. It turns out that, no any special design of the setter method is needed because the solution for keeping data of secondary objects associated with first-class objects (depicted in the **figure 4.3.2.5** above) did already help to update the object modification.

In fact, if any first-class object loaded from database is modified then right before that object is stored back into database, the *PrePersist* callback is automatically invoked to generate the XML representation that includes the new state (updated from the old state) of that object. After generating the XML, the *PrePersist* callback continues to assign the generated XML for the special attribute of that first-class object and finally that first-class object is stored into database.

It is clear to see that, the XML representation always hold the updated state of the first-class object and the special attribute of that first-class object is always assigned with the updated XML.

4.3.2.3. Verification

The implementation of management of secondary object storage/retrieval was tested (using JUnit testing framework) with a simple artificial data model (depicted in the **figure 4.3.2.7**) in which the *Plate* and *Material* classes are first classes because they have the common root parent class of *InformationObject*. *Shape* and *Feature* classes belong to the secondary-class category because they do not inherit from *InformationObject*. Recall that only first classes are mapped to relational database and thus only first-class objects can be stored directly into database. Therefore, data of secondary objects referenced by first-class object (set at the object initialization) can not be stored directly into database together with the owning first-class object. From the **figure 4.3.2.7**, one can see that the *Plate* (as first class) has a unidirectional One-To-Many association with the *Shape* (as secondary class) and a unidirectional One-to-One association with the *Feature* (as secondary class).

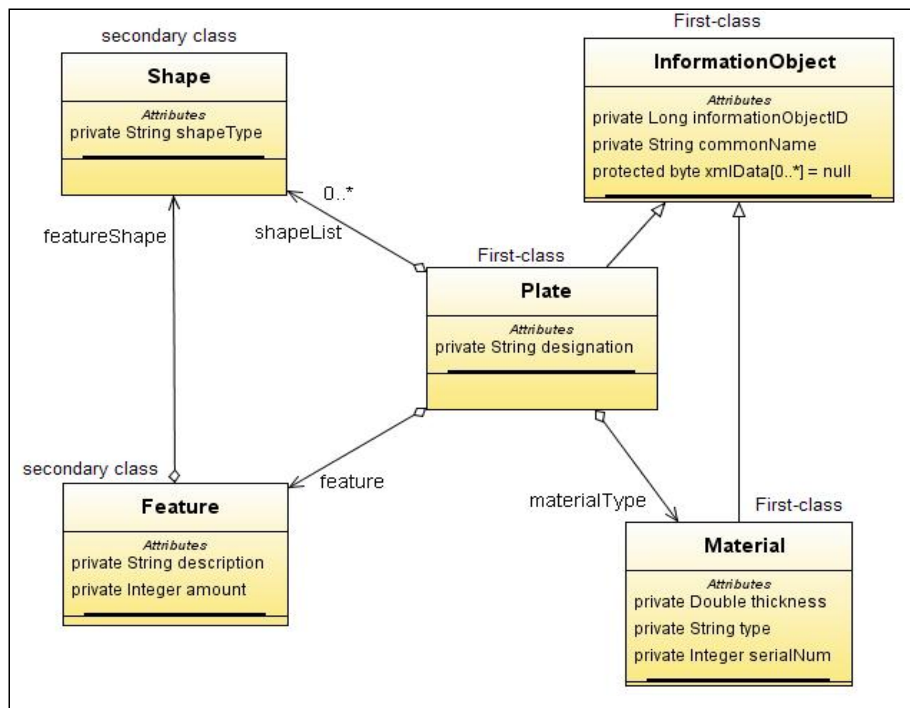

Figure 4.3.2. 7. Artificial model of first class and secondary class

The testing scenario is as followings:

- Create the *Plate* object (first-class object) and then initialize the created *Plate* object with both secondary objects (particularly, *Shape* objects and *Feature* object) and first-class object (particularly, *Material* object). Furthermore, the *Feature* object also has a reference of *Shape* object (i.e. secondary object has reference of other secondary object).

- Store the *Plate* object into database. One should note that, only data of *InformationObject*, *Plate* and *Material* are stored. Data of *Shape* and *Feature* are not stored.

- Retrieve the stored *Plate* object and verify that:
    - The initialized data of *Feature* object is returned correctly when invoking the getter method (**getFeature()**) of the retrieved *Plate* object. Furthermore, also verify that the initialized data of *Shape* object is returned correctly when invoking the getter method (**getFeatureShape()**) of the returned *Feature* object.
    - The initialized data of *Shape* objects is returned correctly when invoking the getter method (**getShapeList()**) of the retrieved Plate object.

- Modify the retrieved *Plate* object by setting a new *Feature* object and then store the modified *Plate* object back to database. After that, retrieve again that *Plate* object to verify the modification.

4.3.2.4. Conclusion

The JAXB capability of marshalling and un-marshalling plays a vital role in the implementation of the management of secondary object storage/retrieval. JAXB provides a set of annotations used to customize the binding of POJO-like objects to XML. Compared to JPA mapping annotations, JAXB annotations are much simpler. In fact, a Java class object can easily be marshaled into XML representation (by using JAXB marshalling capability) if the Java class is annotated with *@XmlRootElement* (right above the Java class name). The class attributes (whose types are for example primitive type, collection of primitive type, object type, collection of object type, …) normally need not to be specified with any JAXB annotations because the default binding behavior of JAXB is often fine.

In the Topgallant® ERM data model, there does not exist the case in which secondary class has reference of first class. Therefore, all secondary classes are **not** specified with any JAXB annotation because the secondary class object state needs not to be kept. Furthermore, any first class (including parent classes and sub-classes) is simply specified with JAXB annotation of *@XmlRootElement* so that the first class objects can be converted into XML representation which holds the entire object state.

The clustering of the objects (associated with the owning first-class object) to be included in the generated XML is not considered in this task. Purpose of the object clustering is to have only the objects within the same scope included in the generated XML. The functionality of management of secondary object storage/retrieval was implemented as a preparation step for the another functionality called clustering of objects.

4.4. JPA shortcoming

Throughout the implementation, several misbehaviors or shortcomings of JPA are encountered as followings.

4.4.1. Detaching

It is important to note that any entity loaded from database is in persistent state and thus it is managed by the persistence context of the entity manager (further detail about persistence context and entity manager is described in **part 2.2.1**). In the version control implementation, it was important to detach the loaded entity from the persistence context of the entity manager so that it is possible to change the entity identifier and thus the loaded entity (with the changed identifier) will later be inserted into database as a new entity. The **figure 4.4.1** shows the previous solution for version control which was bankrupted, because the detaching method of JPA does not work as expected.

In fact, it is desired to detach one particular entity, but the method *EntityManager*'s clear() detaches all the persistent entities previously-involved in the current persistence context and thus the operations applied to the detached entities will also be removed. Even though this misbehavior of JPA has been resolved by the method *Session*'s evict() of Hibernate, but because all persistence methods used in the implementation have to be of standard JPA, thus the method *Session*'s evict() of Hibernate could not be employed. Due to this fact, another solution for version control had to be investigated as described in the **part 4.3.1.2 (Version control)**.

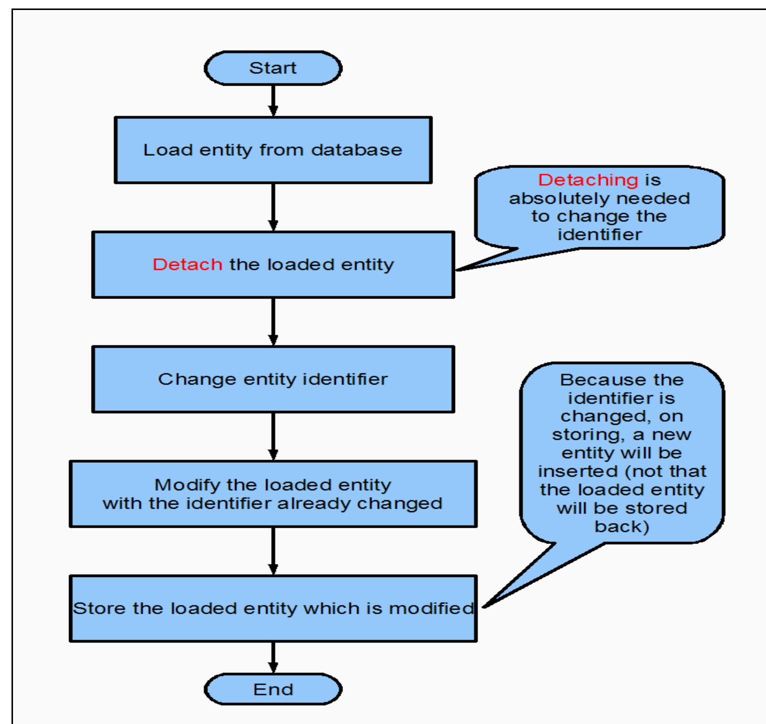

Figure 4.4. 1. Version control general solution based on detaching

4.4.2. Handling of special data types

Standard JPA mapping annotations do not support the mapping of array of primitive type such as Double[ ], String[ ] and so on. However, based on the fact that standard JPA annotations can map a byte array (byte[ ]) with the annotation *@Lob*[47], it is quite possible to use this annotation to map the array of primitive type as following:

```
@Lob
array_of_primitive_type (e.g. Double[ ], String[ ])
```

Nevertheless, because the *@Lob* annotation specifies that the (persistent) class attribute is to be persisted as a large object (i.e. the storing representation in database is hex string), it is nearly impossible to make query based on the *@Lob*-specified class attribute whose type is of array of primitive type. In fact, the entire array of values (if mapped using *@Lob*) is stored as a hex string in database, but when that array is accessed via the getter method of the retrieved owning entity, the representation of the accessed array (in the application) is in original representation (i.e. one can see the array of (human-readable) values (in the application).

Another mapping problem is that standard JPA annotations also do not support the mapping of collection of primitive type such as List<String>, Set<Double> and so on. However this problem can be overcome by simply replacing List with ArrayList and replacing Set with HashSet. The (persistent) class attribute of type ArrayList or HashSet can then be mapped without having to specify any mapping annotation. The trick is that ArrayList and HashSet are Java classes which implement the Serializable interface and default mapping behavior of JPA can map any field whose type implements the Serializable interface. Similar to the array of primitive type, the storing representation of ArrayList and HashSet in database is also in hex string.

4.4.3. Orphaned entity deletion

When an object of the class which contains a One-to-Many class association is stored into database, the collection of associated objects is also stored. The problem is when replacing the old (i.e. currently-stored in database) collection with the new collection via the setter method of the owning object (retrieved from database); the old collection still exists in database together with the newly-inserted collection. It must be expected that the old collection is removed to leave space for the new collection. However, default behavior of JPA does not remove the old collection which contains the so-called orphaned entities. An entity becomes orphaned if it is dereferenced by its owner (i.e. the orphaned entity is no longer referenced).

To overcome this problem, the following Hibernate-specific annotation must be specified additionally to the standard JPA annotation (*@OneToMany*) for mapping One-to-Many class association (one should note that, this problem does not occur for the One-to-One class association).

```
@org.hibernate.annotations.Cascade(org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
```

[47] *@Lob* annotation is used to map a byte array to a type of BLOB in relational database schema

In fact, the "`CascadeType.DELETE_ORPHAN`" of Hibernate enables automatic deletion of the orphaned entities.

Another solution for this problem is to manually perform a DELETE query to remove the old collection after invoking the setter method (of the owning object) for replacing the old collection with the new collection. However, this solution is not preferred in practice.

5. Evaluation

5.1. System performance measurement

The persistence solution, which includes the JPA-based persistence API and the persistent store based on RDBMS (particularly, Java DB), is evaluated for the system performance characteristics. The system performance in terms of object storage/retrieval speed is measured with the typical test scenario of Topgallant® application. At first, a large amount of Topgallant® objects are created, initialized and then stored into database for measuring the storing duration. The graph of created objects includes a number of parent objects (particularly, the *AssemblyNode*) and each parent object contains plenty of children (particularly, *Plate* objects and *Profiles* objects):

```
AssemblyNode1
      |---------- Plate1_1,    Plate1_2...
      |---------- Profile1_1, Profile1_2...

AssemblyNode2
      |---------- Plate2_1,    Plate2_2...
      |---------- Profile2_1, Profile2_2...

      .
      .
      .
AssemblyNodeN
      |---------- PlateN_1,    PlateN_2...
      |---------- ProfileN_1, ProfileN_2...
```

One should note that, each object is automatically assigned unique identifier once stored into database. The number of the created objects is increased for every measurement of storing duration.

After every measurement of storing duration, the measurement of retrieving duration is also carried out. It is especially important to note that only object POIDs[48] (not the actual objects) are returned from the retrieval operations because the main purpose of the retrieval is mostly to check the object existence and actual objects are only returned on demand. Furthermore, retrieval of object POID is much faster than the retrieval of actual object. Duration for retrieving all the stored *AssemblyNode* objects is measured and then duration for retrieving all children of all the retrieved *AssemblyNode* objects is measured too.

[48] Further detail of POID is described in **part 4.3.1.1**

The relational database server (particularly, Java DB version 10_4_1_3) with default configuration is running on the computer with: CPU of Intel Pentium 4 (3 GHz), RAM (2 GB) and the Operating System (Microsoft Windows Server 2003).

The measurement results (on the base of Java DB) are presented as followings:

<u>Case 1:</u> 10 *AssemblyNode* objects and each of *AssemblyNode* has 100 children (50 *Plates* + 50 *Profiles*) → totally 1010 created objects

Total number of *Assembly* objects:	10
Total number of *Plate* objects:	500
Total number of *Profile* objects:	500
Duration for storing 1010 objects:	813 milliseconds
Duration for retrieving 10 *AssemblyNode*:	297 milliseconds
Duration for retrieving 1000 children of 10 *AssemblyNode*:	328 milliseconds

<u>Case 2:</u> 10 *AssemblyNode* objects and each of *AssemblyNode* has 200 children (100 *Plates* + 100 *Profiles*) → totally 2010 created objects

Total number of *Assembly* objects:	10
Total number of *Plate* objects:	1000
Total number of *Profile* objects:	1000
Duration for storing 2010 objects:	2250 milliseconds
Duration for retrieving 10 *AssemblyNode*:	422 milliseconds
Duration for retrieving 2000 children of 10 *AssemblyNode*:	485 milliseconds

<u>Case 3:</u> 10 *AssemblyNode* objects and each of *AssemblyNode* has 400 children (200 *Plates* + 200 *Profiles*) → totally 4010 created objects

Total number of *Assembly* objects:	10
Total number of *Plate* objects:	2000
Total number of *Profile* objects:	2000
Duration for storing 4010 objects:	2438 milliseconds
Duration for retrieving 10 *AssemblyNode*:	297 milliseconds
Duration for retrieving 4000 children of 10 *AssemblyNode*:	719 milliseconds

Case 4: 10 *AssemblyNode* objects and each of *AssemblyNode* has 600 children (300 *Plates* + 300 *Profiles*) → totally 6010 created objects

Total number of Assembly objects:	10
Total number of Plate objects:	3000
Total number of Profile objects:	3000
Duration for storing 6010 objects:	3656 milliseconds
Duration for retrieving 10 AssemblyNode:	297 milliseconds
Duration for retrieving 6000 children of 10 AssemblyNode:	985 milliseconds

Case 5: 10 *AssemblyNode* objects and each of *AssemblyNode* has 800 children (400 *Plates* + 400 *Profiles*) → totally 8010 created objects

Total number of Assembly objects:	10
Total number of Plate objects:	4000
Total number of Profile objects:	4000
Duration for storing 8010 objects:	5437 milliseconds
Duration for retrieving 10 AssemblyNode:	421 milliseconds
Duration for retrieving 8000 children of 10 AssemblyNode:	1422 milliseconds

5.2. Comparison and conclusion

The measurement results (with the integrated functionality of version control) on the base of Java DB are then compared with the measurement results (with all integrated functionalities[49]) on the base of LDAP-compliant directory server.

Table 5.1: Duration for storing plenty of objects

Number of objects	Duration for storing (for Java DB)	Duration for storing (for LDAP-compliant directory server)
1010	813 ms = 0. 813 s	26031 ms = 26.031 s
2010	2250 ms = 2.250 s	87281 ms = 1.45 min
4010	2438 ms = 2.438 s	162844 ms = 2.71 min
6010	3656 ms = 3.656 s	358797 ms = 5.98 min
8010	5437 ms = 5.437 s	444313 ms = 7.4 min

[49] The other functionalities are for example clustering of objects, management of secondary object storage/retrieval and so on.

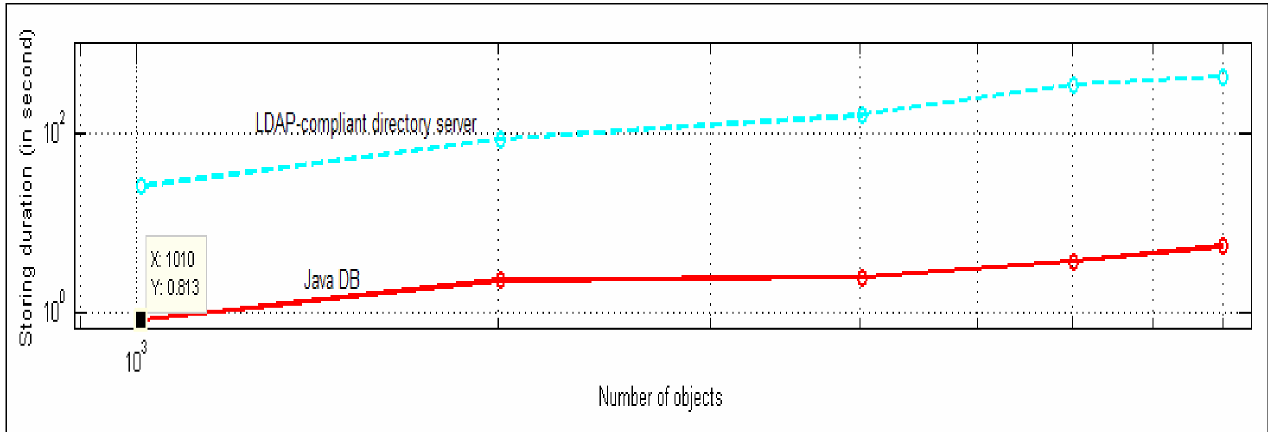

Figure 5. 1. Storing duration (in second) characteristics

Table 5.2: Duration for retrieving all children (of 10 parent objects)

Number of children	Duration for retrieving children (for Java DB)	Duration for retrieving children (for LDAP-compliant directory server)
1000	328 ms	250 ms
2000	485 ms	437 ms
4000	719 ms	765 ms
6000	985 ms	1094 ms
8000	1422 ms	1438 ms

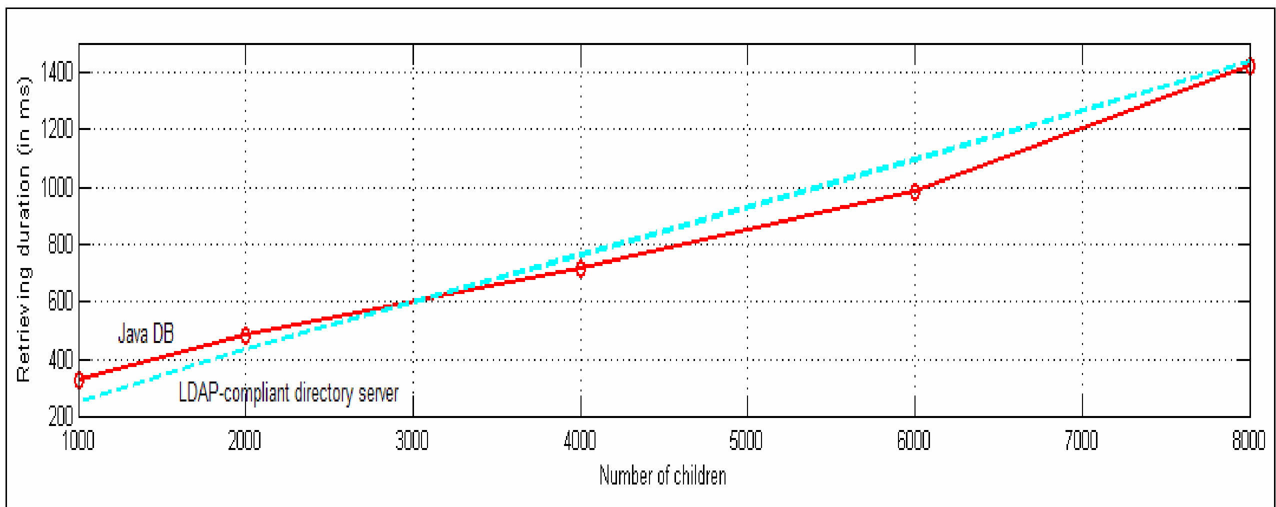

Figure 5. 2. Retrieving duration (in ms) characteristics

Table 5.3: Average duration for storing only one object

Duration for storing one object (for Java DB)	Duration for storing one object (for LDAP-compliant directory server)
813 ms / 1010 = 0.805 ms	26031 ms / 1010 = 25.77 ms
2250 ms / 2010 = 1.12 ms	87281 ms / 2010 = 43.42 ms
2438 ms / 4010 = 0.607 ms	162844 ms / 4010 = 40.6 ms
3656 ms / 6010 = 0.608 ms	358797 ms / 6010 = 59.7 ms
5437 ms / 8010 = 0.678 ms	444313 ms / 8010 = 55.47 ms
→ Average value: 0.764 ms	**→ Average value: 45 ms**

Table 5.4: Average duration for retrieving only one child

Duration for retrieving one child (for Java DB)	Duration for retrieving one child (for LDAP-compliant directory server)
328 ms / 1000 = 0.328 ms	250 ms / 1000 = 0.25 ms
485 ms / 2000 = 0.24 ms	437 ms / 2000 = 0.22 ms
719 ms / 4000 = 0.18 ms	765 ms / 4000 = 0.19 ms
985 ms / 6000 = 0.16 ms	1094 ms / 6000 = 0.182 ms
1422 ms / 8000 = 0.18 ms	1438 ms / 8000 = 0.179 ms
→ Average value: 0.22 ms	**→ Average value: 0.204 ms**

From the **figure 5.1**, it is quite obvious to see that the object storage speed on the base of Java DB is significantly faster than that on the base of LDAP-compliant directory server. Furthermore, from the **figure 5.2**, it is wonderful that the object retrieval speed on the base of Java DB is approximately equal to that on the base of LDAP-compliant directory server. Even more, for very large amount of objects (e.g. 6000 objects), the retrieval speed on the base of Java DB is faster than that on the base of LDAP-compliant directory server.

There is no doubt to conclude that the system performance with the innovative persistence solution (including the JPA-based persistence API and the underlying persistent store based on Java DB) is **significantly improved** in terms of object storage/retrieval speed.

6. Summary and conclusion

This chapter begins with the summary of the works carried out, followed by the specification of the further works and finally answers the questions specified in the part **Purpose** of the chapter 1 (**Introduction**) as well as gives some important conclusions.

6.1. Summary

With the support of standard persistence specification JPA combined with Hibernate ORM framework, the persistent store has been successfully migrated from LDAP-compliant directory server to relational database server. The migration to RDBMS base (**chapter 4.1**) is carried out by modifying the legacy code generator in order to specify the appropriate JPA ORM annotations directly in the implementation of the code generator so that the code generator can produce the Java-based data model decorated with JPA annotations. The Java-based data model annotated with JPA mapping annotations can then be mapped to the relational database schema easily by the ORM engine (of the Hibernate framework). The generated relational database schema is at first validated by using the JPA Hibernate validator and then the functional capability of the database schema is verified.

After the persistent store had been migrated from LDAP-compliant directory server to relational database server, the legacy persistence API was replaced with the compatible persistence API whose implementation is based on standard JPA-compliant way and operates on the base of RDBMS. The compatible persistence API implementation (**chapter 4.2**) is carried out by at first re-implementing (in a simplified manner) the current working mechanism of object instantiation, created objects management and persistence in order to satisfy the legacy application compatibility. After that, the JPA-based persistence API is implemented by using JPQL to provide convenient methods for query service. Furthermore, the JPA-based persistence API implementation also adopts the essential functionalities from the legacy persistence API such as version control and management of secondary object storage/retrieval.

The most essential functionality, which is the version control, is first implemented (**chapter 4.3.1**). Version control functionality helps in tracking the changes over time to avoid general chaos so that it is easy to get back the previous working version or to get the latest version for synchronization. This functionality is definitely vital especially in the design- or production systems because the product components need to stored or saved as they are frequently modified or updated.

General requirement of version control solution is that if any entity loaded or retrieved from database is modified or updated, a new entity will be inserted into database with the POID the same as the loaded entity except the updated *version* and the modified content (or modified state). Implementation of version control functionality is approached from simple case to real case. The simple case, which involves only actual objects and actual object references, is handled at first to reach the general requirement of version control. After that, the real case, which involves not only actual objects but also symbolic objects (proxies), is approached for full functionality of version control.

Version control functionality is not visible to the application- or client side because it is integrated into the Enterprise Reference Model (ERM) that is generated by the code generator. In fact, version control functionality is expressed invisibly via the object retrieval, object modification (using setter methods) and object storing. Symbolic object or proxy is one of the key concepts throughout the version control functionality whose working mechanism is mainly based on the specially-designed getter and setter process with the internal automatic resolving property. The advantage of using proxy is to avoid the same overhead as the actual object represented by the proxy. Furthermore, the callback mechanism (supported by JPA) particularly *PostLoad* and *PrePersist* is the prerequisite to implement the version control.

After the version control functionality implementation, the second important functionality, which is the management of secondary object storage/retrieval, is implemented (**chapter 4.3.2**). Recall that, in the Topgallant® ERM data model, objects are classified into first-class category and secondary-class category. Only the first-class objects are stored directly into database. This functionality enables the fact that even though all secondary objects are not stored directly into the relational database but the retrieval of secondary object data (set at the initialization of the owning first-class object) is still possible (in a normal way) by simply invoking the getter method of the owning first-class object.

The solution is that data of secondary objects associated with the owning first-class object is included in the XML representation that holds the entire state of the owning first-class object. The XML representation is automatically generated and then stored in a special attribute of the first-class object right before the first-class object is stored into database (this automatic mechanism is enabled by the *PrePersist* callback of JPA). When the getter (of the owning first-class object loaded from database) on some target secondary object is invoked, the XML is first retrieved and then converted back into some temporary Java object. Finally, the target secondary object is returned by the Java object constructed from the XML. It is clear to see that, the retrieval of secondary object is actually performed by the getter method of the Java object constructed from the XML that holds the entire state of the owning first-class object.

After the phase of design, realization and implementation, the innovative persistence solution consisting of the JPA-based persistence API and the persistent store based on RDBMS (particularly, Java DB), is evaluated for the system performance characteristics (**chapter 5**). The measurement results clearly show that the system performance in terms of object storage/retrieval speed is significantly improved.

6.2. Further work

In addition to the already-implemented functionalities that are version control and management of secondary object storage/retrieval, the other essential functionalities (such as application-controllable clustering of secondary objects with first-class objects, archive and streamed value persistence and so on) should also be implemented (in the later time) in order to bring the entire work into production.

Because of the fact that JPA offers a great ability of switching between different ORM implementors, besides the currently-used ORM implementor (that is the Hibernate), the other open source ORM implementors (such as OpenJPA [44], TopLink Essentials [45] and so on) should also be experimented in order to possibly find out which ORM implementor can result in the best performance.

Furthermore, the performance (of the database-involved application) also depends on the underlying persistent store (the currently-used is the Java DB), it is also desired to experiment with other kinds of RDBMS such as MySQL, Oracle RDBMS and so on.

6.3. Conclusion

The standardized persistence specification JPA (with Hibernate as JPA implementor) really supports the successful migration of the persistent store from LDAP-compliant directory server to RDBMS. In fact, the standard JPA mapping annotations are utilized to instruct the ORM engine to map the highly complex Topgallant® ERM Java-based data model to relational database schema. The generated relational database schema validation is approved by the JPA Hibernate validator and its functional capability is also verified via the test cases as presented in the **parts 4.1.3**, **4.3.1.3** and so on. However, due to the problem of orphaned entity deletion (presented in **part 4.4.3**); one Hibernate-specific annotation of “DELETE_ORPHAN” ought to be used to overcome the problem:

```
@org.hibernate.annotations.Cascade(org.hibernate.annotations.CascadeType.DELETE_ORPHAN)
```

Therefore, the original intention of using only standard JPA mapping annotations could not be kept. Consequentially, to be able to incorporate another ORM engine provided by different JPA implementor (e.g. OpenJPA, TopLink Essentials and so on), the Hibernate-specific annotation of “DELETE_ORPHAN” above must be replaced with the correspondent one provided by the target JPA implementor. Fortunately, most of the different JPA implementors are aware of the problem of orphaned entity deletion.

Most importantly, the innovative persistence solution consisting of the JPA-based persistence API and the persistent store based on RDBMS (particularly, Java DB) results in a good system performance. Particularly, the system performance in terms of object storage/retrieval speed is significantly improved. This is demonstrated via the comparison of the measurement results presented in **chapter 5**. Besides the advantages and usefulness demonstrated clearly by this thesis work, it is also clear to see that there are still a few disadvantages and shortcomings of JPA (presented in the **part 4.4**) which should be fixed and improved in the later version of JPA (the JPA version used in this thesis work is JPA 1.0). In fact, JPA 2.0 is promised to be an advanced and improved version compared to the JPA 1.0. One should keep in mind that the persistence specification JPA contains only interfaces which are implemented by a specific JPA implementor. Therefore, the later version of JPA is desired to be fully portable across various JPA implementors so that it is flexible and easy to switch between different JPA implementors without any modification.

References

[1] ATLANTEC ENTERPRISE SOLUTIONS; *Topgallant™ Adapters*; http://www.atlantec-es.com/products/topg-adapters.html; access date: 01.09.08

[2] ATLANTEC ENTERPRISE SOLUTIONS; *Topgallant™ Information Server*; http://www.atlantec-es.com/products/topg-is.html; access date: 01.09.08

[3] ATLANTEC ENTERPRISE SOLUTIONS; *Topgallant™ Application Server*; http://www.atlantec-es.com/products/topg-as.html; access date: 01.09.08

[4] ATLANTEC ENTERPRISE SOLUTIONS; *Topgallant Infrastruktur Specification*; 2006

[5] ATLANTEC ENTERPRISE SOLUTIONS; *Topgallant®Information Server Specification*; 2001

[6] Sun Microsystems; *Java Data Objects (JDO)*; http://java.sun.com/jdo/

[7] ATLANTEC ENTERPRISE SOLUTIONS; *Master thesis project proposal*; 2008

[8] Brian T. Kurotsuchi; *The Wonders of Java Object Serialization*; http://www.acm.org/crossroads/xrds4-2/serial.html

[9] Tim Cooper and Michael Wise; *Critique of Orthogonal Persistence*; University of Sydney – 2006; http://www.lsi.uvigo.es/lsi/erosello/imo/articulos/orthcritique.pdf **and** [PM04]

[PM04] Mick Jordan; *A Comparative Study of Persistence Mechanisms for the Java™ Platform*; Sun Microsystems Laboratories Technical Report; SMLI TR-2004-136; September 2004; http://research.sun.com/techrep/2004/smli_tr-2004-136.pdf

[10] Sun Microsystems; *Java SE Technologies – Database*; http://java.sun.com/javase/technologies/database/index.jsp **and**
Mindfire Solutions; *Hibernate Vs JDBC*; www.mindfiresolutions.com/mindfire/Java_Hibernate_JDBC.pdf

[11] Sun Microsystems; *Getting Started With Java Data Objects (JDO): A Standard Mechanism for Persisting Plain Java Technology Objects*; http://java.sun.com/developer/technicalArticles/J2SE/jdo/

[12] Sun Microsystems; *Advantages of the Enterprise JavaBeans Architecture*; http://java.sun.com/developer/Books/ejbarch/arch-adv.pdf **and**
IBM developer works; *Enterprise JavaBeans Fundamentals*; www.freejavaguide.com/ejb.pdf **and**
Humphrey Sheil; *To EJB or not to EJB*; http://www.javaworld.com/javaworld/jw-12-2001/jw-1207-yesnoejb.html

[13] BEA Systems, Inc; *JPQL Language Reference*; http://edocs.bea.com/kodo/docs40/full/html/ejb3_langref.html

[14] Sun Microsystems; *Java Persistence API FAQ*; http://java.sun.com/javaee/overview/faq/persistence.jsp

[15] Spring Framework; http://static.springframework.org/spring/docs/2.5.x/reference/orm.html

[16] ODBMS.ORG; http://www.odbms.org/

[17] Sun Microsystems; *Java DB at a Glance*; http://developers.sun.com/javadb/

[18] Sun Microsystems; *Java Persistence API FAQ*; http://java.sun.com/javaee/overview/faq/persistence.jsp

[19] Barry & Associates Inc; *Transparent persistence in object-relational mapping*; www.service-architecture.com/object-relational-mapping/articles/transparent_persistence.html

[20] Mario Van Damme; *Best Practices for Object/Relational Mapping and Persistence APIs*; www.developerdotstar.com/mag/articles/PDF/DevDotStar_VanDamme_ObjectRelational.pdf ; 2006

[21] Avaje homepage; *JPA API Issues*; http://www.avaje.org/jpaapi.html

[22] Hibernate; *Chapter 1. Architecture*; http://www.hibernate.org/hib_docs/entitymanager/reference/en/html/architecture.html

[23] Patrick Linskey (plinskey@bea.com); *Introduction to the EJB 3 Java Persistence API*; http://www.chariotsolutions.com/slides/pdfs/etc-jpa.pdf

[24] Christian Bauer, Gavin King; *Java Persistence with Hibernate*; ISBN: 1-932394-88-5; November 2006

[25] Hibernate; *Chapter 3. Working with objects*; http://www.hibernate.org/hib_docs/entitymanager/reference/en/html/objectstate.html

[26] Dwight Peltzer; *Providing a Complete Data Persistence Solution*; http://www.aspfree.com/c/a/.NET/Providing-a-Complete-Data-Persistence-Solution/

[27] Oracle; *TopLink JPA Annotation Reference*; www.oracle.com/technology/products/ias/toplink/JPA/resources/toplink-jpa-annotations.html

[28] Sun Microsystems; *Using JAXB*; http://java.sun.com/webservices/docs/2.0/tutorial/doc/JAXBUsing.html#wp78319

[29] Ed Ort and Bhakti Mehta; Java Architecture for XML Binding (JAXB); March 2003; http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html

[30] Øystein Grøvlen; *Sun Java DB*; Sun Tech Days 2006-2007; http://uk.sun.com/sunnews/events/2007/mar/revolution/techdays07/presentations/sun_java_db.pdf

[31] Peter Mikhalenko; *Get your feet wet with Sun's tiny Java DB*; January 2008; http://blogs.techrepublic.com.com/programming-and-development/?p=587

[32] Kohsuke Kawaguchi; *The JAXB API*; 2003 http://www.xml.com/pub/a/2003/01/08/jaxb-api.html

[33] *JAXB Introduction*; http://docs.rakeshv.org/java/jaxb/users-guide/jaxb-intro.html

[34] Bharath Mundlapudi; *Implementing High Performance Web Services Using JAX-WS 2.0*; August 2006 http://java.sun.com/developer/technicalArticles/WebServices/high_performance/

[35] Sun Microsystems; *Enterprise JavaBeans Technology*; http://java.sun.com/products/ejb/

[36] K. Zeilenga; LDAP: Directory Information Models; RFC 4512; Category: Standards Track; http://tools.ietf.org/html/rfc4512; June 2006

[37] Universität Mannheim:Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Julia Mildenberger, Guido Moerkotte, Robert Schiele, Till Westmann; *Anatomy of a Native XML Base Management System*; http://db.informatik.uni-mannheim.de/publications/TR-02-001.pdf; 2002

[38] Oracle; http://www.oracle.com/technology/documentation/index.html

[39] Dilip Kandlur, Dinesh Verma; IBM T.J. Watson Research Center; Measurement and Analysis of LDAP Performance; http://www.ee.sunysb.edu/~xwang/public/paper/16tnet01-wang.pdf; January 2008

[40] MySQL AB, 2008 Sun Microsystems, Inc; *The BDB (BerkeleyDB) Storage Engine*; http://dev.mysql.com/doc/refman/5.0/en/bdb-storage-engine.html; access date: October 2008

[41] Dave Minter, Jeff Linwood; *Pro Hibernate 3*; ISBN: 1-59059-511-4; 2005

[42] Hibernate; *Relational Persistence for Java and .NET*; http://www.hibernate.org/

[43] JPOX; *Java Persistent Objects*; http://www.jpox.org/

[44] Apache OpenJPA; *Java EE persistence project of the Apache Software Foundation*; http://openjpa.apache.org/

[45] Oracle; *TopLink JPA*; http://www.oracle.com/technology/products/ias/toplink/jpa/index.html

[46] THOUGHT Inc; *CocoBase JPA*; http://www.thoughtinc.com/index.html

[47] SAP AG; *SAP JPA*; https://www.sdn.sap.com/irj/sdn

Figures

Figure 2.1. 1. Topgallant®Infrastructure. Source [4] 9

Figure 2.2. 1. Sequence diagram for connecting to database 13
Figure 2.2. 2. Sequence diagram for storing object into database 14
Figure 2.2. 3. Sequence diagram for creating and executing query 15
Figure 2.2. 4. ORM with annotations 16

Figure 2.3. 1. JAXB main features. Source [29] 19

Figure 4.1. 1. Current generation process of Java-based data model 32
Figure 4.1. 2. Generation of relational database schema 33
Figure 4.1. 3. Code generator working mechanism 34
Figure 4.1. 4. Working mechanism of the method generateClass(..) 35
Figure 4.1. 5. Some portion of the XSD file 36
Figure 4.1. 6. Interface inheritance hierarchy (hierarchical layout) in data model 37
Figure 4.1. 7. Interface inheritance hierarchy (symmetric layout) in data model 38
Figure 4.1. 8. Class diagram (symmetric layout) of the bean classes in data model 39
Figure 4.1. 9. Six-level class inheritance of the first-class category in the data model 40
Figure 4.1. 10. Class associations 42
Figure 4.1. 11. InheritanceType.JOINED 43
Figure 4.1. 12. InheritanceType.SINGLE_TABLE 44
Figure 4.1. 13. MappedSuperclass 44
Figure 4.1. 14. Uniform mapping of the class inheritance hierarchy 45
Figure 4.1. 15. One-to-One class association mapping 46
Figure 4.1. 16. One-to-Many association mapping result with intermediate table 46
Figure 4.1. 17. One-to-Many association mapping result with FK 47
Figure 4.1. 18. One-to-Many association mapping using intermediate table 47
Figure 4.1. 19. Artificial model for class association mapping 48
Figure 4.1. 20. Mapping result of the artificial model from figure 4.1.10 49
Figure 4.1. 21. Generated relational database schema 50
Figure 4.1. 22. Database schema validation result 52
Figure 4.1. 23. Annotated bean class generated by code generator 53
Figure 4.1. 24. Testing result of functional capability of database schema 54
Figure 4.1. 25. Chosen class objects to verify the generated database's functional capability 55

Figure 4.2. 1. Sequence diagram for current working mechanism of object instantiation, created object management and persistence. 59
Figure 4.2. 2. Sequence diagram for the compatible working mechanism of object instantiation, created object management and persistence. 61

Figure 4.3. 1. Version control context (in database view) for actual object reference and symbolic object reference 65
Figure 4.3. 2.Version control context (in application view) for actual object reference and symbolic object reference 66

Figure 4.3. 3. Retrieval procedure and storing procedure in version control context in simple case 67
Figure 4.3. 4. Retrieval procedure and storing procedure in version control context in real case 69
Figure 4.3. 5. Setter process 70
Figure 4.3. 6. Getter process 71
Figure 4.3. 7. General resolving process 72
Figure 4.3. 8. Resolving process from database 73
Figure 4.3. 9. Resolving process from memory 74
Figure 4.3. 10. Top base class inheritance hierarchy for version control 75
Figure 4.3. 11. Simple data model for version control testing 77

Figure 4.3.2. 1. Inheritance hierarchy of the interfaces in Topgallant® data model 79
Figure 4.3.2. 2. First-class entries stored in LDAP directory 79
Figure 4.3.2. 3. Generated XML representation of a Plate object 80
Figure 4.3.2. 4. First class includes reference of secondary class 81
Figure 4.3.2. 5. Solution for keeping data of secondary objects associated with first-class objects 82
Figure 4.3.2. 6. Internal process of the owning first-class object's getter method for retrieving secondary object data 83
Figure 4.3.2. 7. Artificial model of first class and secondary class 85

Figure 5. 1. Storing duration (in second) characteristics 93
Figure 5. 2. Retrieving duration (in ms) characteristics 93

Abbreviations

API	Application Programming Interface
JPA	Java Persistence API
ORM	Object-Relational Mapping
POID	Persistent Object ID
ERM	Enterprise Reference Model
SPP	Shipbuilding Product and Process Information
JDO	Java Data Objects
JOS	Java Object Serialization
JBP	JavaBeans Persistence
OPJ	Orthogonal Persistence
JDBC	Java Database Connectivity
EJB	Enterprise JavaBeans
JSR	Java Specification Request
CMP	Container-Managed Persistence
OODB	Object Oriented Database
ODBMS	Object Database Management System
RDB	Relational Database
POJO	Plain Old Java Object
JPQL	Java Persistence Query Language

Appendix

Implementation explanation

This master thesis report contains an appendix of the implementation in a CD.

The entire implementation of this master thesis work was integrated into the base package named **tgbase** of Atlantec Enterprise Solutions GmbH, an international company providing software development and IT consulting services for the marine industry with the trademark **Topgallant®**. The base package **tgbase** includes all basic packages not only for working with Topgallant® applications but also for further development. All the basic packages in the **tgbase** have the prefix of **org.atlantec** followed by the specific package name.

The base package **tgbase** is not allowed to publish. Only the following packages, which were implemented in the scope of this master thesis work, are allowed to publish (inside each package, there is a txt file named "**read me.txt**" for the package explanation):

- **org.atlantec.jpa.codeGen** (not allowed to publish):
  - Includes the implementation of the migration to RDBMS (**chapter 4.1**).
  - The legacy code generator implementation was modified with the specification of JPA mapping annotations so that the code generator can produce the Java bean classes decorated with JPA mapping annotations. Finally, the JPA annotated Java bean classes can easily be mapped to the relational database schema.
  - Because the implementation was based on the legacy code generator implementation, all the implemented files are not allowed to publish. One just can see the file names (with **empty** content).

- **org.atlantec.jpa.databaseUtil**:
  - Includes the implementation of database utility methods such as creation of database schema from the JPA annotated Java classes, validation of the generated database schema and so on (**chapter 4.1**).

- **org.atlantec.jeb.jpaeb** and **org.atlantec.jpa.objects**:
  - Includes the implementation of the compatible persistence API (**chapter 4.2**).
  - The application compatibility is expressed via the mechanisms of object instantiation (not simply with the ***new*** operator), created objects management, persistence and query service.

- **org.atlantec.jpa.model** (partly published):
  - Includes all of the generated Java interfaces and bean classes by the code generator (located in the package **org.atlantec.jpa.codeGen** mentioned above) from the XSD-based data model. Only the bean classes are specified with JPA mapping annotations and are then mapped to the relational database schema by the ORM engine (**chapter 4.1**).

  - Because this is the Topgallant® data model, it is not allowed to publish. However, several typical interfaces and bean classes are published to demonstrate the JPA-annotated bean classes generated automatically by the code generator. Furthermore, one can also see the special design of getter and setter methods for the version control functionality (**chapter 4.3.1**).

- **org.atlantec.jpa.version**:
  - Include the implementation of the version control functionality (**chapter 4.3.1**).
  - The version control functionality is integrated into the Java-based data model. One can see the special design of getter and setter methods for the version control functionality in several Java bean classes published in the package **org.atlantec.jpa.model** mentioned above.

- **org.atlantec.jpa**:
  - Includes the base implementation classes (mentioned at the end of **part 4.3.1.2**).
  - These base implementation classes specify common attributes and methods that are needed by all classes in the Topgallant® data model.
  - Furthermore, the class **BeanInstanceJPA** includes the important methods necessary for the version control functionality.

- **org.atlantec.jpa.secondaryClass**:
  - Include the first implementation of the functionality of management of secondary object storage/retrieval (**chapter 4.3.2**).
  - This first implementation was verified with an artificial data model (in the sub-package named **testModel**).

In addition, the JPA persistence unit (default unit) is provided in "**META-INF\persistence.xml**". One should note that the parameters of database connection specified in this persistence unit will be overridden by the application.

Declaration

I declare within the meaning of section 25(4) of the Examination and Study Regulations of the International Degree Course Information Engineering that : this Master report has been completed by myself independently without outside help and only the defined sources and study aids were used. Sections that reflect the thoughts or work of others are made known through the definition of sources.

Hamburg, November 04th 2008 ____________________________

Trinh, Tien Trung