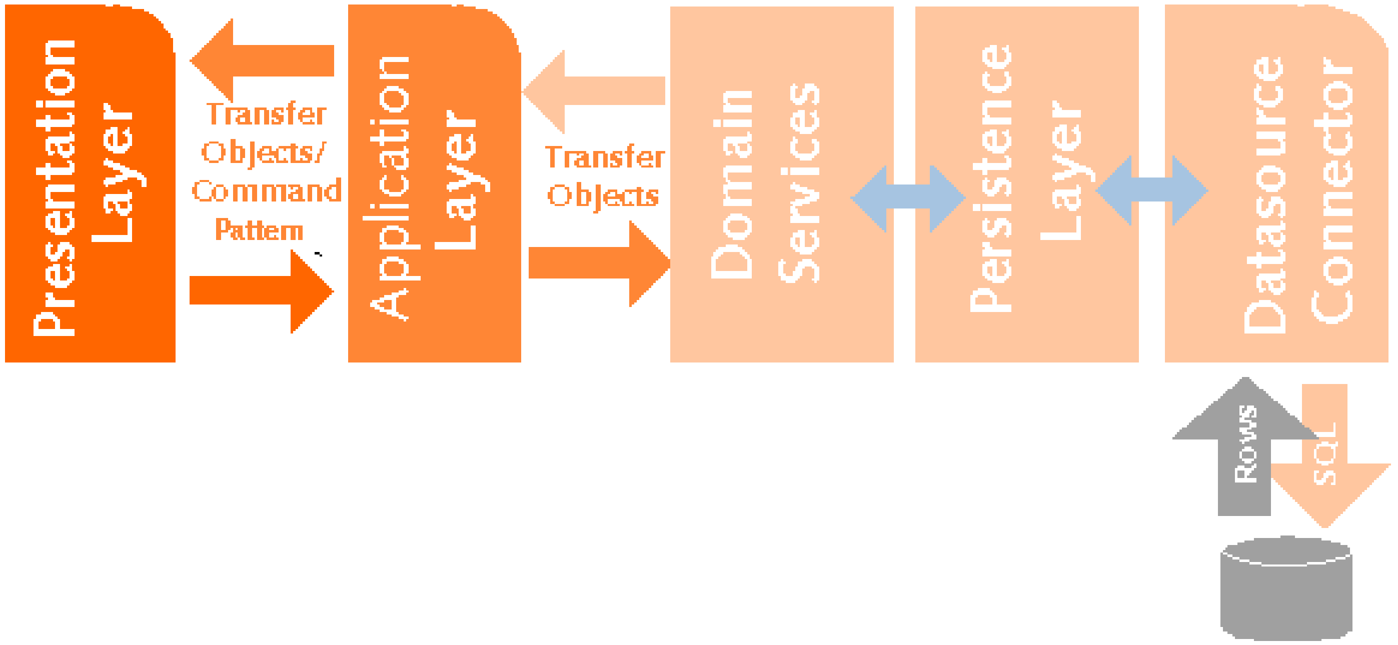
e-Platform

Object Persistence Design Guidelines

Motivation

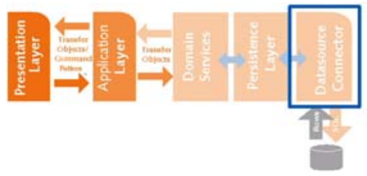
- Design guideline supports architects and developers in design and development issues of binding object-oriented applications to data sources
- The major task is to give an overview of the most important and common principles referred to literature, preferably in the internet



Contents

Common Design Principles

- Hard-coding SQL in your user-interface classes and/or domain/business classes results in a code that is difficult to maintain and extend. Therefore these classes should not directly access your persistence mechanisms.
- RDBMS-based referential integrity mechanisms are essential in most enterprise applications. We recommended that applications should not rely on the application's Java code as the sole guardian of data integrity.
- Views can be useful to simplify queries and enable O/R mapping. However the level of support depends on the underlying database, e.g. joint views are partially updateable in Oracle 7.3 and later. Cloudscape 3.6 does not support this feature.
- Don't try to write your own O/R mapping framework. If you need one, use an existing solution.

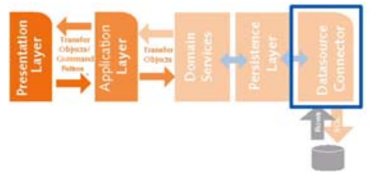


Datasource Connector (I/III)

- **Multi-object actions:** Persistence layer have to support multi-object actions, e.g.

```
PreparedStatement stmt = con.prepareStatement(
"INSERT INTO employees VALUES (?, ?)");
stmt.setInt(1, 2000);
stmt.setString(2, "Kelly Kaufmann");
stmt.addBatch();
stmt.setInt(1, 3000);
stmt.setString(2, "Bill Barnes");
stmt.addBatch();
// submit the batch for execution
int[] updateCounts = stmt.executeBatch();
```

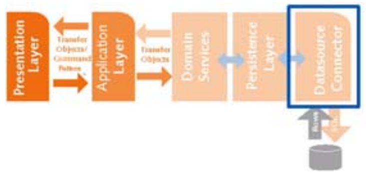
- **StoredProcedures:** We do not recommend the use of stored procedures in general. Technical and performance issues are exceptional situations for using stored procedures



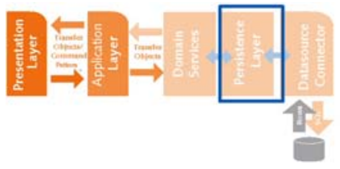
Datasource Connector (II/III)

- **Various RDBMS versions and/or Vendors:** Be careful about Date, Time and Timestamp and Boolean.
  - Oracle just has Date (which is really a timestamp to seconds)
  - Sybase has DateTime (which has milliseconds, but not microseconds)
  - DB/2 Date, Time and TimeStamp to microseconds
  - DB/2 does not support Boolean
- **Native and non-native drivers**
  - JDBC functionality may not be exactly as it appears in the ads
  - Not all drivers work with the same semantic
  - Problems with Date formats, BLOBS;
  - Differences in stored procedure support.
  - JDBC-ODBC looses precision beyond for NUMERIC > 15 digits
  - Lots of good drivers out there, just be aware
  - Different JDBC-Driver types
  - e-Platform recommends the use of the Sequal driver

Datasource Connector (III/III)



- **JDBC-Layer:**
  - The JDBC API is too low-level for using it directly as a viable option. When using JDBC, always use helper classes to simplify the application code.
  - **e-Platforms recommendation is to use the Spring JDBC FW (JDBC level persistence FW)**



Persistence Layer

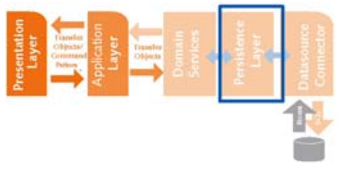
Persistence layers hide developers from the details of storing objects. They have to provide a common interface for the business/domains objects to support CRUD operations. Consequently the persistence layer is responsible, among others, for the

- **mapping of the objects to database tables**
- **SQL generation**
- **correct ordering of the database operations.**

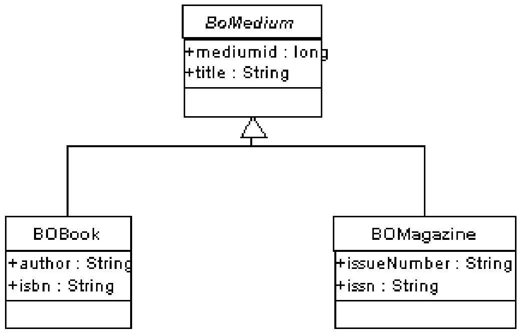
- A persistence layer is needed if an object model exists. Otherwise, in the case of a simple "window on data" application type a persistence layer is not needed. In this situation the direct use of the data source connector layer will suffice.

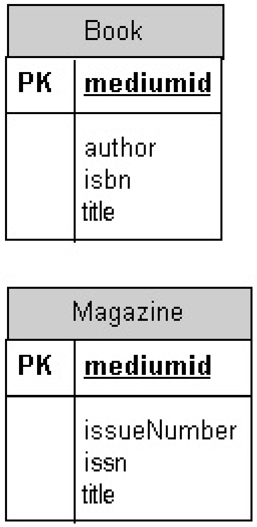
Persistence Layer-> Mapping

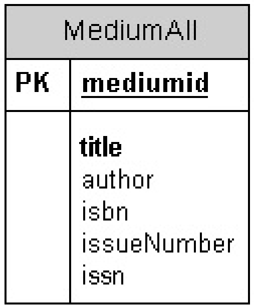
- Mapping objects to database table. Three basic approaches for the mapping exist:
- Use one table for an entire class hierarchy (collapsed mapping) - each abstract and concrete-class has data in one table. A separated type column is needed to distinguish the different sub-classes. A restriction to this solution is that all properties to subclasses have to map to database columns, where null values are allowed
- Use one table per concrete class (horizontal mapping) - each concrete class of an entire class hierachy has all the data in its own table
- Use one table per class (vertical mapping) - each class of the entire hierarchy has data in its own table



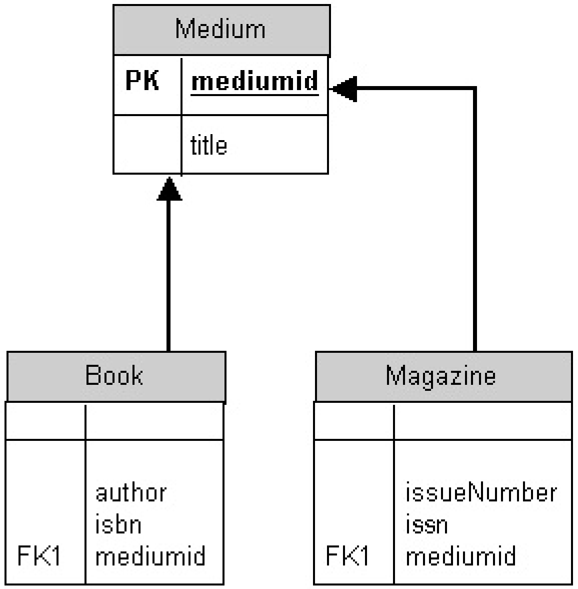
Persistence Layer-> Mapping







One table for an entire class hierarchy



One table per concrete class

One table per class

Persistence Layer-> Mapping

Factors to Consider	Use one table for an entire class hierarchy	Use one table per concrete class	Use one table per class
Ad-hoc reporting	Simple	Medium	Medium/Difficult
Ease of implementation	Simple	Medium	Difficult
Ease of data access	Simple	Simple	Medium/Simple
Coupling	Very high	High	Low
Speed of data access	Fast	Fast	Medium/Fast
Support for polymorphism	Medium	Low	High

Persistence Layer-> Mapping

- We recommend the following rules of thumb:
  - If possible, do not combine different mapping approaches. The result of a combination is the compexity of the DB operations
  - Your preferred basic mapping strategy is dependent on the object operations. If you access only concrete classes from your application and you are interested in avoiding a lot of unnecessary data space in your table then use the "Use one table per concrete class" strategy.
  - The "Use one table per class" approach is usually not recommendable because
    - it takes longer to read and write data since you need to access multiple tables
    - the resulting data model is often not accepted by the responsible enterprise data modelling group

Persistence Layer-> Editing and Writing Objects with Minimal Update

In general, a persistence layer should keep track of all persistent objects that have a changed state, and should make sure that they are all written back to the database. The problem is how to notify that an object has changed and needs to be saved to the database? The following solutions are possible:

- If you don’t keep track of what has changed, you have to delete everything that was in the collection and then re-write everything.
- Keep track of what has changed and determine the minimum that has to be written.
- Can make copies of the objects.
- Mark attributes dirty as they are changed

Persistence Layer-> Editing and Writing Objects with Minimal Update

In order to select the most appropriate solution for your need consider the following common observations:

- Most objects that are read never get modified.
- It is a waste of time to save unchanged objects.
- Developers often forget to save an object when it changes.
- If a write to the database is made every time an object changes, then another write or rollback would be required for cancellation requests by the user.
- Complex objects might only have one of their components changed, thus not requiring a complete write of all values to the database.
- Writing back objects that have not changed can make it hard to audit who actually last changed the object.

We recommend the solution with dirty markers. It reduces the database operations drastically if most of the objects read are never modified. To realise this solution the implementation of the "Dirty marker" pattern is useful, see Dirty Marker Strategy.

Persistence Layer-> Storage Order into the database

Storage Order into the database

The serialisation of object nets onto databases can, however, run into problems with referential integrity. Referential integrity is used to specify the logical requirement of the existence of a row before relationships can be made to it; and for related rows to be deleted (or re-related) before the target row can be deleted. This requires some technique onto the persistence layer to correctly arrange the storage into the database in order that the dependencies between the rows are maintained correctly. To arrange the objects of an object net correctly, the persistence layer often compares the objects in pairs. This brute force approach for ordering has a complexity of O(n*n), if n is the number of involved objects. This approach works well, if the object nets are small but it is a bottleneck if processing of large object nets is necessary. More intelligent algorithms are required. One approach is to use the knowledge of the mapping information to avoid the comparison of pairs and therefore reduce the complexity to O(n). The knowledge of how to do this is the intellectual property of persistence framework vendors

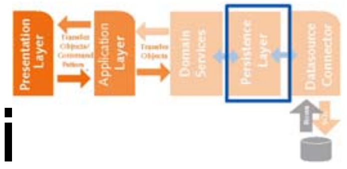
Persistence Layer-> Locking Mechanism

The persistence layer must allow multiple users to work on the same database, and protect data against being overwritten unknowingly. There are two main approaches to the problem

- Pessimistic Locking
- Optimistic Locking

- Pessimistic Locking
  - is used to process a piece of data which is read and update in the same transaction. After the read of data the rows are locked until a commit or rollback occurred.
  - Lock escalation on database is possible. Example DB2: If the numbers of locked rows increase a fixed level - DB2 is using page locking instead of row locking. The result is that all concurrent clients using the locked pages are blocked and can not execute any CRUD operations.

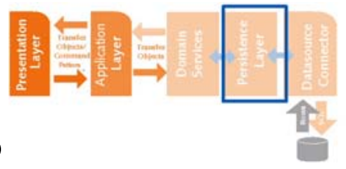
Persistence Layer-> Locking Mechani



- Optimistic Locking
  - is used to guarantee that data is not changed by other transactions when clients are updating information based on data that was read in a previous transaction. It is used in typical client/server applications when a user needs some time to process a piece of data.
  - Basically when an object is updated, if the version field has not changed since reading then the update is successful.
  - Usually the locking field is a timestamp. Other approaches are version number or hash values.
  - Set the new value and add the version field value to the where clause.
  - The query to the modify-statement must be adapted to the locking field in the where clause
  - Version numbering is an additional piece of software that works in the persistence layer and has in most cases influence into the data model (locking field)
  - transaction may be failed because of concurrency access
  - long term transactions usually done in dialog oriented applications

Persistence Layer-> Locking Mechanism

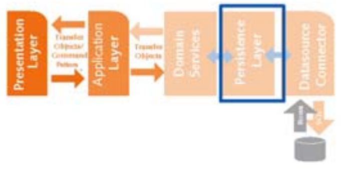
- Recommendations:
  - The use of an optimistic locking strategy is recommended in most client/server applications under the assumption that the data working sets of the different clients are disjoint.
  - In the case of batch processing, the pessimistic strategy seems better because dirty reads in so called "long term transactions" have to be avoided. A typical scenario are migration processes, e.g. the migration of contracts to a newer version.



Persistence Layer-> Object identifiers

- Since every object is unique in any object-oriented system, it is important to create unique identifiers for new objects. These unique identifiers can be used as primary key on a database. There exist three different approaches for the generation of unique identifiers:
  - Universal Unique Identifiers (UUID)
  - Sequence Blocks
  - Stored procedures for auto generated keys
- Recommendations
  - The use of UUIDS is useful in applications which work in offline mode and synchronise data later with a central data store. In this case UUIDs as primary keys is a simple way to guarantee unique identifiers in federated databases
  - Otherwise the use of sequence blocks or auto generated keys is preferred. An important restriction in the context of our service architecture is that business objects or domain objects created on the applications layer get their key value on the domain service layer
  - Use only Java primitive types to map SQL primary keys to Java and vice versa

Persistence Layer-> Caching



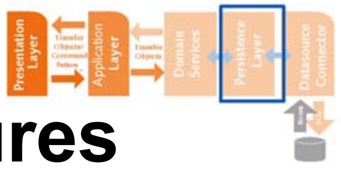
- Caching mechanisms are useful to reduce the number of database access operations and improve performance of an application.
  - Once read and built, an object can be saved in a cache.
  - Probably the most effective performance enhancement you will use.
  - The entry in the cache should be indexed by a unique property, usually the primary key is used
  - If a cached object is requested again, the copy in the cache is returned
- You may not be able to always check the cache first. From the query expression you need the primary key.
  - For most applications, if you are querying for more than one object, you have caching, costs memory to go to the database.
  - Cache can get stale, needs to be refreshed or purged.
  - Use caching options/policies at the class level:
    - Some classes should not be cached
    - Threshold for the class caches are defined by the application

Persistence Layer-> Caching&Identity

Identity support and caching are not the same thing although closely related because of the way they are implemented.

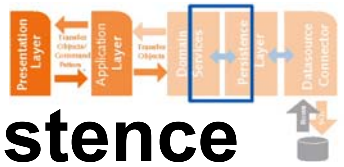
- Caching is for performance.
- Identity is for object integrity.
- Both must be supported to do any object application effectively with a relational database.
- Weak and soft references in JDK 1.2 (Java 2) available, so cache does not grow infinitely.

Persistence Layer->Record architectures



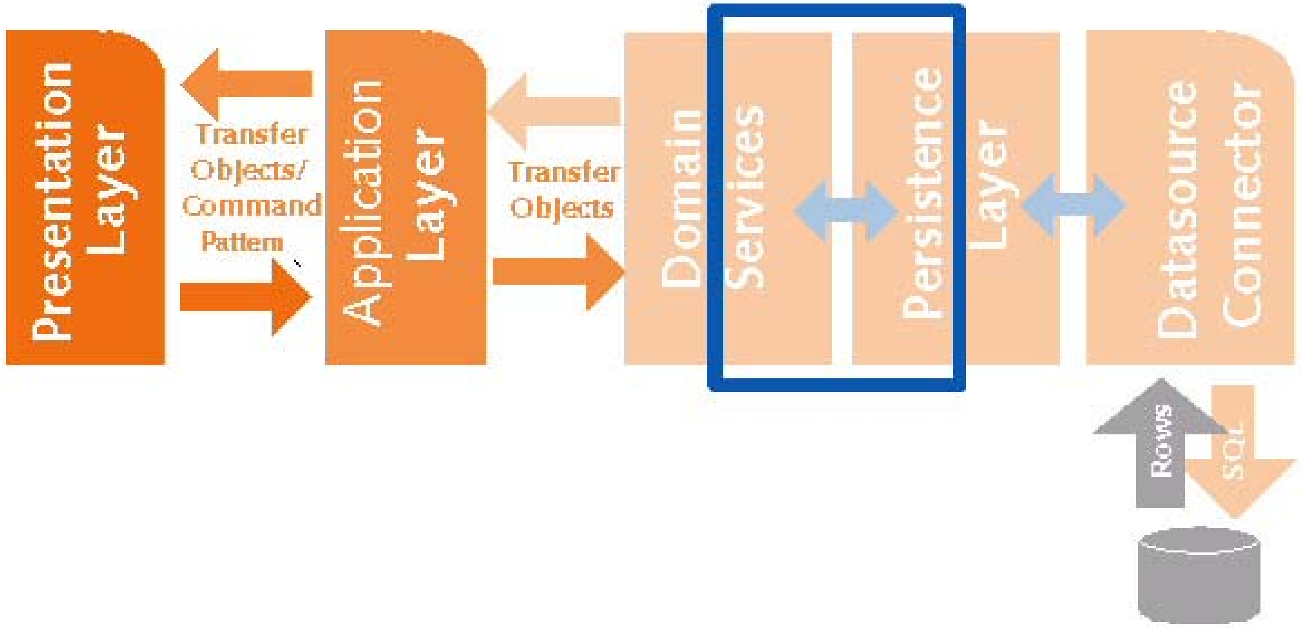
Users frequently desire lists of such top-level items as customers or products. The requirement is almost always for ID, name and brief summary information with read-only access; and similar requirements exist for reporting purposes. The basic approach to this is not to handle it with the main persistence mechanisms; instead guarantee the following features:

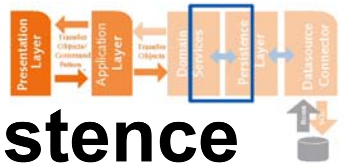
- A separate mechanism should be provided in the persistence layer for read-only bulk handling. This should return data in the form of rows or tuples, using packed value arrays for the queried columns. It also needs a design for 'search criteria' and cursor access to the returned rows.
- The persistence layer have to support the use of database cursors



Decoupling Business Logic and Persistence

For developing components, which implement non-trivial business logic, a good strategy for tackling complexity and improving maintainability is to design and implement a domain model, which is an object model of the application's problem domain. The domain model needs a persistence mechanism to retrieve and store data from databases.





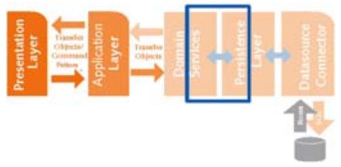
Decoupling Business Logic and Persistence

Each application has to provide a persistence layer if an object model exists. Otherwise, in the case of a simple “window on data” application type a persistence layer is not needed.

There are many choices for implementing the persistence layer in a J2EE application. Important choices that we'll consider in this chapter include:

- Data Access Object Pattern (DAO)
- O/R Mapping Frameworks, usually third-party persistence frameworks such as Object Frontier

Decoupling Business Logic and Persistence -> DAO



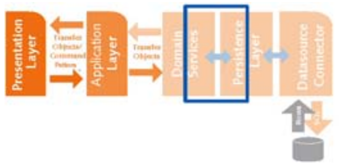
The Data Access Object (DAO) pattern is used to:

- Isolate data access from the underlying data store, like RDBMS or LDAP
- Decouple the client interface from underlying data access mechanics, e.g. connection pooling

Advantages and Drawbacks

+ It is a lightweight approach: it uses ordinary Java interfaces, rather than special infrastructure such as that associated with entity beans
+ Enables easier migration because it is data source independent: A layer of DAOs makes it easier for an application to migrate to a different database implementation. The business objects have no knowledge of the underlying data implementation. Thus, the migration involves changes only to the DAO layer.
+ Encapsulates proprietary APIs which are often used to improve performance

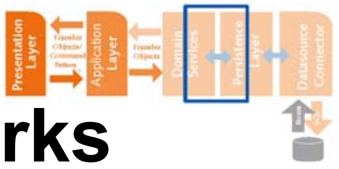
Decoupling Business Logic and Persistence -> DAO



Advantages and Drawbacks ….

- + Centralises all data access into a single layer: Because all data access operations are now delegated to the DAOs, the separate data access layer can be viewed as the layer that can isolate the rest of the application from the data access implementation. This centralisation makes the application easier to maintain and manage.
- - Adds an extra layer to architecture: The DAOs create an additional layer of objects between the data client and the data source that need to be designed and implemented to leverage the benefits of this pattern. But the benefit realised by choosing this approach pays off for the additional effort.
- - Typically the DAO layer is realised by the application. This has drawbacks as well as advantages. A major drawback is the amount of time for implementation. On the other hand, it is possible to build high performing data access layers but only if a fundamental data base knowledge is available- otherwise it is a high risk.

Decoupling Business Logic and Persistence -> O/R Mapping Frameworks



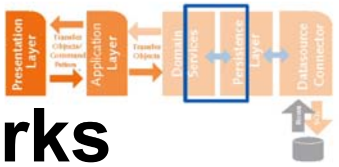
The most common features of O/R mapping tools:

- Generate SQL-Statements, derived from the mapping metadata
- Resolve circular identities, i.e.,

```
account == account.getCustomer().getAccount()
```

- Caching data in the object layer delivers excellent performance because RDBM access is reduced
- Serialisation of object nets onto databases, see Storage Order into the Database
- Support for optimistic and pessimistic locking
- Support O/R mapping with GUI-Tools
- Support one-to-one, one-to-many and many-to-many relations
- Support managed relations, for example cascading delete operations
- Support composite primary keys
- UI tool for mapping
- Importing and creating relational schema and table

Decoupling Business Logic and Persistence -> O/R Mapping Frameworks



Advantages and Drawbacks

+ O/R mapping provides a transparent persistence layer for handling the mapping of inheritance and relationships.
+ O/R mapping provides the binding to the low-level data access code. This reduces the development time.
+ Generates SQL-Code: After changes in the data model only the mapping metadata have to be changed to generate the appropriate SQL-Code.
+ The most O/R mapping solutions support additional freatures like: locking mechanism, cache strategies
- Applications may become business critical if the dependency degree of proprietary O/R solutions is high
- If data modelling is driven from an OO-perspective, it is possible that it results in a non-performing database schema, which is useless for other processes.
- Showstopper for projects due to an non-performing O/R Mapping solution caused by a product or inadequate usage
- Intensive training is necessary in order to achieve the O/R mapping skills.

Object Persistence Design Guidelines