# Data Object Graphs

*A Pragmatic Approach to MDA Generated DTO Graphs*

*Marcel Baumann*

*Version 1.0.10*

## Table of Contents

# 1 Introduction

Modern client-server applications manipulate complex graphs of data objects. For example a contract with its positions and optional amendments is represented as a hierarchy of data information. These graphs are transferred from and to the client application. The user views data, edits it and commits the changes to the database through server functions. The patterns commonly found in the literature describe thoroughly how to implement simple data transfer objects *DTO* but ignore the complexity how graphs of objects should efficiently be transmitted between server and client components. The intended audience is software developers interested in using DOG or implementing their own variant.

The framework described here provides pragmatic solutions to this common problem. This “state of the industry” approach efficiently transfers trees and graphs of value objects between client and server. Modern code generation technologies, based on the model driven architecture *MDA* approach are used to generate needed code artifacts. To streamline the integration of the framework into enterprise infrastructure, care was taken to provide a bridge to the enterprise blueprints described in the J2EE guidelines. The application architect can fully concentrate on the business analysis and avoid loosing time to solve infrastructure problems.

The document explains the various mechanisms implemented in the provided solution. The information reflects the status of the latest release. Intended audience is software developers who want to use the framework or extend it further. The chapter “Introduction“ describes the code generation framework used to create the data objects, the chapter “Update Mechanisms” explains the provided update mechanisms. The next section “Extensions” documents support for framework extensions such as lightweight types. A separate chapter “” introduces the validation framework used to verify the correctness of the graph. The last chapter “Future Directions” describes future directions of the framework.

The diagram below shows the various component of the persistence framework. Green components are components external to the framework. Care was taken that a complete set of open source components exist to create a running system. Naturally commercial products are also available. The open source community statement remains.

> “You pay more, you get less.”

The light gray components are the specific ones to the framework. They are responsible to manipulate data object graphs on client and on server side. When graphs are updated the components synchronize the caches. The dark gray components implements the communication between the client and server persistence framework. Often it is necessary to slightly adapt them to the application being developed. If the communication medium is CORBA 2.3 or higher, or a J2EE container is used, the implementation can be used as provided.

The source code of the data object graph manipulation framework provides detailed documentation of the classes and methods. The *Doc-Check* from Sun was used to verify the completeness of the documentation.

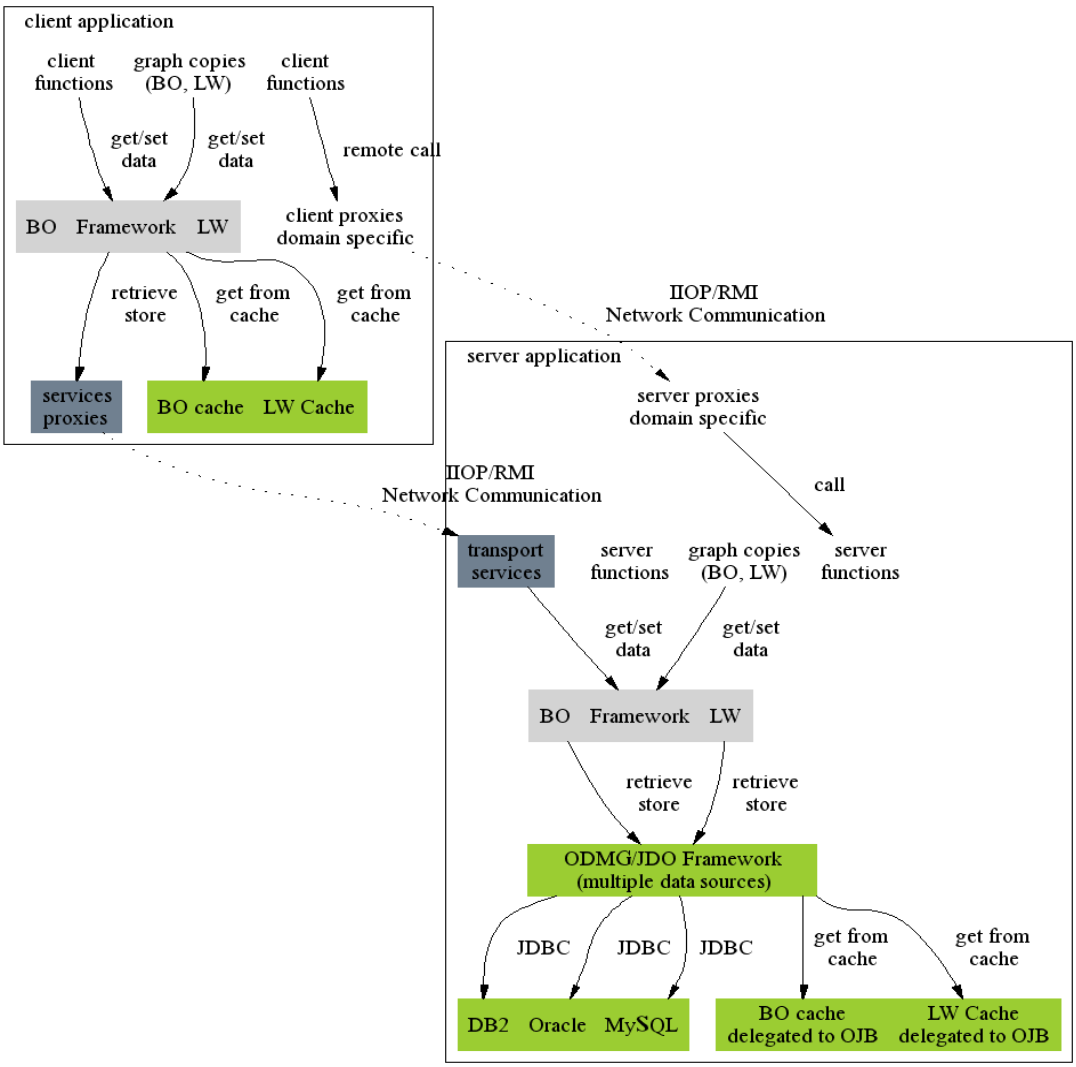


*Illustration 1 Client /Server Business Object Graphs*

A major goal when developing the framework is to follow existing or emerging standards such as “Service Data Object JSR235”.

# 2 J2EE Blueprints

The data object graph framework is an extreme variant of the "*Composite Entity"* blueprint defined J2EE patterns. The strategies to send efficiently objects over the wire and to retrieve graphs from the database are similar to the suggestions found in the blueprint and realizes the *"Transfer Object"* pattern. The main extension is that our framework does not need at all entity beans. They can be used but are not required. Below we show the advantages of this pattern as described in the guidelines of the J2EE platform.

All advantages described in the pattern are achieved. To harmonize the terminology, the reader should know that transfer object, dependent object and composite object described in the below text is coalesced to the term data object in this document.

- *Eliminates Inter-Entity Relationships:* Using the composite object pattern, the dependent objects are composed in a graph, eliminating all inter-entity-bean relationships. This pattern provides a central place to manage both relationships and object hierarchy.
- *Improves Manageability by Reducing Number of Entity Beans:* Using composite objects reduces the number of entity bean classes and code, and makes maintenance easier. It improves the manageability of the application by having fewer coarse-grained components.
- *Improve Network Performance:* Aggregation of the dependent objects improves overall performance. Aggregation eliminates all fine-grained communications between dependent objects across the network. If each dependent object were designed as a fine-grained entity bean, a huge network overhead would result due to inter-entity bean communications. Avoid network traffic because everything else is faster.
- *Reduces Database Schema Dependency:* The database schema is hidden from the clients, since the mapping of the entity bean to the schema is internal to the coarse grained composite object. Changes to the database schema may require changes to the composite entity beans. However, the clients are not affected since the composite entity beans do not expose th schema to the external world.
- *Increases Object Granularity:* With a composite entity, the client typically looks up a single entity bean instead of a large number of fine-grained entity beans. The client requests the composite entity for data. All objects are transferred to the client in a single remote method call. This reduces the chattiness between the client and the business tier.
- *Facilitates Composite Transfer Object Creation:* Although a transfer object returns all data in one remote call, the amount of data returned with this one call is much larger than the amount of data returned by separate remote calls to obtain individual entity bean properties. This trade-off works well when the goal is to avoid repeated remote calls and multiple lookups.
- *Overhead of Multi-level Dependent Object Graphs:* If the dependent objects graph managed by the composite entity has many levels, then the overhead of loading and storing the dependent objects increases. This can be reduced by using the optimization strategies for load and store, but then there may be an overhead

associated with checking the dirty objects to store and loading the required objects.

The persistence mechanism is an implementation of the "*Data Access Object"* pattern. The implementation is orthogonal to the composite entity pattern and does not request any modifications of the data object source code. Therefore the business logic is independent from the data access logic. Data access functions specific to the data object graph framework are implemented in separate classes not part of the business model.

# 3 MDA Generation

The *model driven architecture – MDA*- approach helps architects to concentrate on the domain modeled as *platform independent model – PIM* -. The transformation of this model into the *platform specific model - PSM* - is an automatic process done with MDA code generators tailored to the target platform.

This approach frees the developer from writing tedious source code for a set of data object types defining a domain model. These data objects often follow the Java bean conventions. For each property a setter, a getter, an optional change event and a vetoable event must be implemented. Support classes are written to provide standard patterns as the visitor, or the abstract factory. Additional mechanisms are often provided to manipulate reference codes or to connect validation rules on the data instances.

The PIM model describes features of the domain entities. Code generators are used to create source code artifacts used in the PSM model.

## *3.1 Features*

The provided code generator uses the concept defined in MDA and provides the following features.

- Accessors for simple or indexed properties. The visibility of the methods can be configured. The field is always private[1].
- Constructor initializing all collection fields. It is possible to define a collection interface for the visible part of an indexed property and a different implementation class to realize the container.
- Change and vetoable events if the property requests it. The implementation is fully compatible with the Java bean conventions.
- Handling of the modified flag when properties of a bean instance are updated.
- Generation of the OJB configuration file to support the persistence of the objects. The current persistence framework OJB supports ODMG 3.0 and JDO[2] standards.
- Generation of the value data object transformer to convert the data objects to a form, which can be transmitted over the communication protocol. Idiosyncrasies of older communication protocols, such as null value support in CORBA 2.1 are solved in the code generator [Brose2001].

The code generator belongs to the family of active tools. Modifications of the domain models are always performed in the MDA representation. The source code is generated each time the model is changed. Mechanisms are provided to preserve customer extensions of methods in the generated classes.

The persistence framework directly manipulates the generated data objects. The diagram below shows the overall flow of transformation from the descriptive model to the final applications. The separation of concerns between the server and client is shown too.

1 This convention follows best practices of object-oriented development.
2 The current implementation is based on the reference implementation of Sun and should not be used in productive environments.

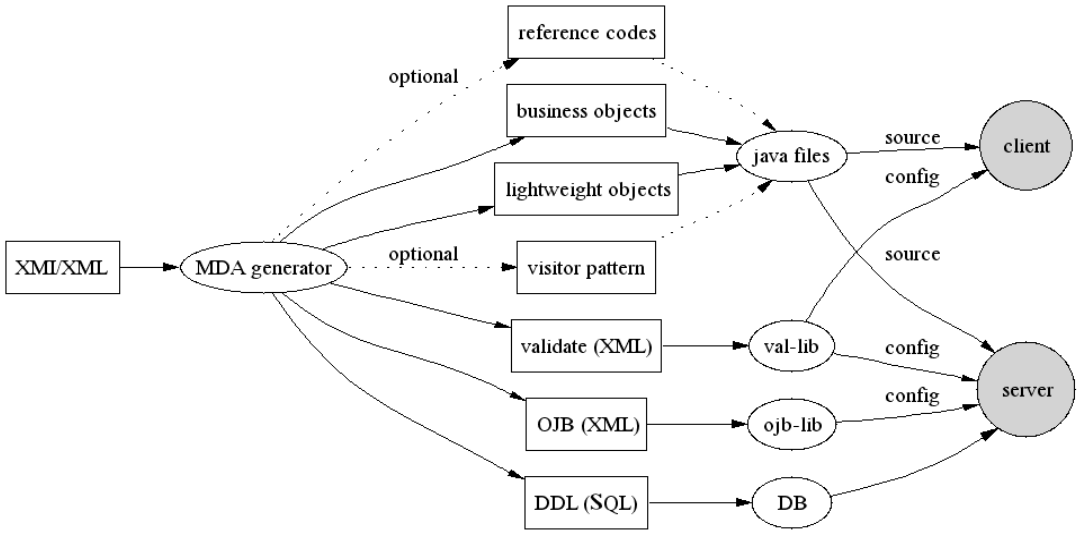


*Illustration 2 Transformation Process*

## *3.2 MDA Model*

The business domain is described in a XML file. The initial version uses a modified Torque based XML data definition type. The new version is XMI compliant and stores the framework specific attributes as stereotypes and tagged values. Therefore this file can be viewed and edited with any UML compliant modeling tool. This approach provides compatibility with commercial and open source tools.

The drawback of this approach is the fact that editing tagged values is a cumbersome activity in actual UML tools. The concept of profiles exists in UML but it only simplifies the definition of the meta-model, not its manipulation.

The code generator instructions are defined as Velocity templates. Using MDA terminology these templates are the cartridges of our source code generation framework.

## *3.3 Semantics*

The model describes the semantic of different aspects of a complex application.

- Persistence mapping between business layer and relational database. The attributes are mainly information for the database mapping. The information is used to create the mapping files and the definition file for the database.
- Transformer mapping describing which attributes are sent over the wire and if null values should be supported. This information is almost derived from the persistence mapping.

In the future the model will describe the view model and mapping information between the view model and the user interface.

This mapping attributes are defined as tagged values. Each tag has an implicit type, a set of legal values and often a default value. The set of all tags defines a domain specific meta model in the MOF terminology. Therefore all tags are formally

documented in a reference document.

# 4 Update Mechanisms

The framework provides mechanisms to retrieve, update, insert and delete individual objects or whole subgraphs. Two groups of functions are provided. The first group of methods manipulates a shallow object copy without any children. These methods are useful to change simple properties in one object. The second set of procedures manipulates graphs of objects.

Newly inserted objects are automatically detected in the framework because their identifier is null. Deleted objects are inferred from the difference between the copied graph and the original one stored in the cache. Updated objects are discovered because their flag "*modified*" is set to true. Objects are updated either because a property value was changed or a child was added or removed[3].

The two diagrams below show left the original graph and right the updated one. The node labeled with "C" are clean ones, with "M" are modified ones, with "I" are inserted ones, and with "D" are deleted nodes. The algorithm traverse both graph and collects the nodes to identify the deleted ones. After the traversal we simply check if an original node is in the collection of the nodes in the modified graph. Updated and inserted nodes can directly be identified in the modified graph.

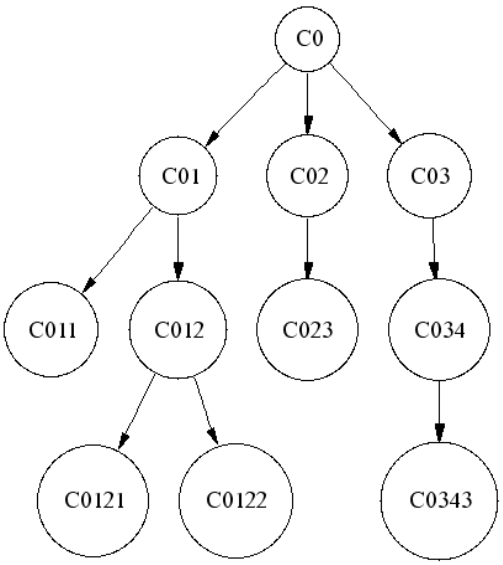


*Illustration 3 Original Object Graph*

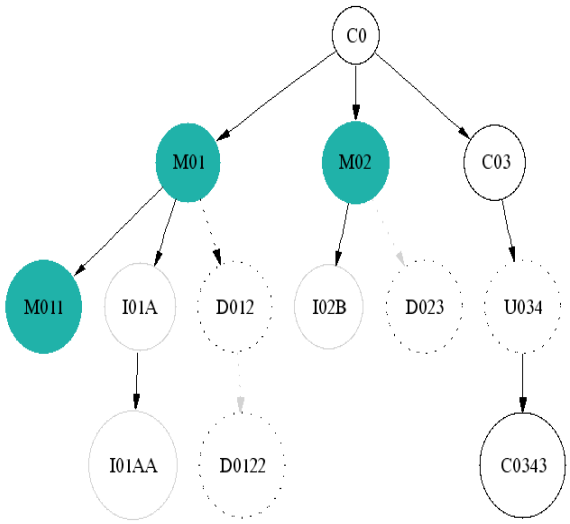


*Illustration 4 Modified Object Graph*

The routine completes its work in O(3 n) where n is the number of nodes in the graph[4]. Both graphs are first traversed to infer differences. The last traversal commits the changes in a transactional context.

3 In Java object-oriented terminology children are indexed properties. Therefore inserting or removing a child modifies the indexed property of the father.

4 This algorithm is fast on standard personal computer with graphs containing up to twenty thousand nodes.

## *4.1 Assumptions*

The framework architecture is optimal if the following assumptions are true for your application.

- **Identity:** An object is identified with the information pair (qualified class name, identifier). Newly created objects can only be identified after they are stored in the database. The identifier is created through services of OJB. Best practice is to use the auto increment feature of the underlying database[5].
  Using database terminology the identifier is a technical primary key composed of a unique numeric column in the table containing the instance.
- **Functions:** The data object classes must implement the visitor pattern. This pattern is used to traverse graphs of data objects.
- **Optimistic Locking:** Client applications seldom modify the same data objects concurrently. The framework uses optimistic locking to detect conflicts[6]. The framework implementation is optimized for load patterns where services seldom modifies the same objects concurrently.
- **Completeness:** A graph of data objects is transmitted completely. The application can decide the level of the root but all children are sent. Therefore the size of the graph should not exceed the transmission capacities available between the client and the server.
- **Network Latency:** The framework assumes that the cost transmitting data between the client and the server is not negligible. Or more precisely, the latency is non-zero, the bandwidth is finite and the network is not always reliable.
- **Service Oriented Architecture:** Complex business logic is not part of the data objects. These functions should be realized in specialized services. This approach is called service oriented architecture. For example you model sales contracts, the functions computing your margin or customer discounts should be realized as services. Often these services have to be changed to reflect new company rules.
- **Modified Flag:** The generated data objects internally set the flag "*modified"* each time a property value has changed.

The framework is easier to use if the developers follow build-in axioms how functions should be accessed or programmed.

- The application developer never set the identifier of a data object. The handling of identifiers is performed inside the framework.
- The application developer always accesses reference code properties through the associated code type. The identifier of the code is never used to directly modified such a property.
- Local business logic can be implemented in the data object class as application extensions. Such extensions are preserved when the classes are generated.

---

5 Application identities, visible to the users should be created through a business service. The framework has no knowledge about these identities and handles them as normal properties.

6 Pessimistic locking is seldom in distributed applications. To implement it the server must keep the context of the client call until it commit or rollback its transaction. Transactional contexts open for a long time is quite expensive and should be avoided.

Complex business rules should always be implemented in an orthogonal structure. This approach is the one promoted in the service oriented architecture approach.

## *4.2 Retrieve Objects*

The OJB framework retrieves graph of connected objects in one single request. The framework does not need to write additional code for this operation. The retrieve criteria is the identifier of the root object. The server side programmers can define additional queries to load specific object graphs. The full power of the OQL language is available to them.

Retrieval operations are provided for each data object type being a root. When a root is retrieved, the transitive closure of the graph is retrieved.

The transformers translate the graph to the transmission format. This new object is sent to the client where the same transformer classes perform the reverse operation. Now the client has a copy of the data object graph read from the database.

## *4.3 Insert Objects*

Business objects can be created and inserted in the graph using the setter methods of properties. Complex domain models often provide a factory to create and initialize objects. The framework is not involved in the creation of objects. It will take care of the new objects during the store phase. All newly created objects, and only such objects, have a null identifier. The framework uses this assumption to identify new object.

The initialization of a new data object is not handled in the persistence framework. The application should provide a factory responsible for the creation and initialization of new data objects. The factory can apply default values and user preferences on the newly created objects. It can also define complex graph creation services.

## *4.4 Modify Objects*

The client can manipulate the received copy of the data object graph. Changing a simple property modifies the object owning it. The generated code detects the modification and set the boolean flag "*modified"* to true. The flag is set each time any property of the object is modified. Parent and children nodes are not modified through such operations.

Changing an indexed property modifies the object owning it. A change is either adding new objects or removing existing ones from the list containing all values of the indexed property. Just updating an attribute of an object part of the indexed property does not change the owner of the property.

The persistence framework handles all modifications, insertions and deletion in the store request. Therefore changes are canceled if the application never calls the store operation for this graph of objects.

## *4.5 Delete Objects*

When data objects are removed from an indexed property or the value of a simple property containing a data object is set to null, the associated object is implicitly

removed from the graph. These operations are performed on the graph of objects. The framework is not involved in the deletion of objects. It will take care of the deleted objects during the store phase.

Complex domain models often provide services to remove objects and perform the associated house-keeping tasks. These operations are wrappers to the functions offered in the data objects.

## *4.6 Store Objects*

The store operation is the moment when the framework performs most of its activities. The following steps are done.

1. The client checks if the edited graph contains updated or newly created objects. If modifications were detected it sends the graph to the server.
2. The server compares the received graph with the original one. All deleted, newly inserted, and the modified objects are identified and collected. The function is based on the visitor pattern. An object is deleted if it is part of the original graph and absent in the updated graph. It is inserted if its identifier is null. It is updated if its flag "*modified"* is set to true.
3. The server cache is updated with the object received from the client. The old version does not reflect the actual property values and are garbage collected.
4. The deleted objects are removed from the database. The new ones are inserted in the database and the updated ones are committed to the database. The persistence layer updates the fields used for optimistic locking – either timestamp or version counter -.
5. The field values used for optimistic locking of newly inserted or updated items are collected on the server side and sent back to the client. If the server had to modify properties due to business logic, these changes are also sent to the client through an application specific protocol. This protocol is simple because each object can uniquely be identified by its identifier and qualified class name now.
6. The client updates its copy of the data objects with the received updates. The original values stored in the cache are discarded from the cache and a new copy of these objects is stored in the client cache.
   This step is only performed in client applications having a client side cache.

The store service persists a set of data object graphs where all roots have the same data object type. The transactional context is the same for all graphs.

## *4.7 Transactional Context*

Server applications handle parallel requests and must provide transactional integrity on the data objects. The data object framework provides a simple but powerful approach to concurrent updates. The goal is that two applications cannot modify simultaneously the same data objects.

1. Graphs of data objects are retrieved from the database. The persistence layer insures that the retrieval is atomic.
2. The graph of objects is sent to a client for visualization or modification.

3. The graph of objects is either received from a client or directly manipulated in the server. The request performs changes on the objects. Each running session has its own copy of persistent objects. Race conditions are detected at commit time through optimistic locking.
4. The graphs of modified objects are committed to the database. The persistence layer insures that the update operation is atomic. If another client has already committed changes on the same objects an error occurs and the above changes are discarded[7].
5. The local copies are persistent objects are discarded.

This approach guarantees the consistency of a server wide object cache and very good performance as long as contention is seldom as stated in the base assumptions of the framework.

The approach decouples database transactional integrity from service transactional integrity. It provides an optimal solution for service oriented approach. A very small set of services sometimes need sophisticated transactional control. These cases can be implemented with the above approach and nested transactions.

## *4.8 Server Cache*

The server component provides caching mechanisms to improve responsiveness of the application. Fetching data from the cache is a magnitude faster as retrieving the same information from the database. A server side cache has the following requirements.

1. Objects are identified through their identity as defined in assumption *Identity*.
2. The cache in a multi threaded environment with concurrent accesses and updates. Either a session based cache or a distributed object cache should be used. Distributed caches should be compliant with emerging standards as JSR-107.
3. The cache should rollback changes if the transaction is aborted.

The chosen solution implements the following rules.

- Objects are local to each session in the server[8]. A server can be clustered.
- All retrieve operations are performed against the session cache. The server assumes that the cache content reflects the objects stored in the database. Therefore at most one functional server is sole owner of the stored instances. This approach still allows clustering but discourages design where multiple servers manipulate the same objects.
- At the start of an update operation, the cache is updated to contain the modified objects. If the transaction is successful the cache already contains the expected objects. If the transaction is aborted, the objects are simply flushed from the cache. Therefore the workflow can continue in the same session without side effects.

The swap of the original objects with the newly modified during the update and the removal of them after a rollback is realized with the help of the visitor pattern. The

7 This approach is optimal under the constraint that concurrent updates are seldom.
8 Global caches can also be configured. This feature is available in JSR-107 but no used in th is framework.

specialized form of the functor[9] visitor is used to implement the required routines. Naturally this solution is so simple to implement only because the framework assumption about completeness exists.

## *4.9 Client Cache*

The client component provides caching mechanisms to improve responsiveness of the application. Again fetching data from the cache is a magnitude faster as retrieving the same information from the server. A client side cache has the requirements similar to the server side. The major difference is that server cache is mandatory for the framework correct behavior but the client cache is optional.

1. Objects are identified through their identity as defined in assumption *Identity.*
2. The cache should work in a multi-threaded environment with concurrent accesses and updates. Modern client applications are often multi-threaded.
3. The cache should rollback changes if the server request is aborted. Usually failures are seldom therefore a simple solution is to remove all involved objects from the cache and request them from the server[10].
4. The cache should be compliant with emerging standards as JSR-107.

The cache can also be used as a cheap mechanism to undo changes and revert to original values. The developer only needs to do a deep copy of the graph and return it to the user interface instead of returning the graph received from the server.

The same mechanism and implementation is used to update the cache with committed modified objects. An abort of the transaction does not automatically flush the cache. The client application logic must explicitly request this operation.

## *4.10 OJB Dependencies*

The design of the framework follows the recommendations of the authors of OJB. The ODMG transaction, query, and database are wrapped in framework specific classes. All these classes are defined in one package.

This indirection layer allows to easily switch from ODMG to JDO standard. The same layer provides support for switching from OJB to other products such as *Hibernate* or *Top-Link*.

Facade abstractions are provided to support multiple logical and physical databases. In the integration business, communication with other systems is still often implemented with stagging tables. The facade hides the used databases and provides a clean logical view for the client components.

9 A functor is an object-oriented representation of a function. The framework uses the functor package of the Jakarta project.

10 The best approach is to let the application decides that is the best reaction. The framework provides basic operations to revert local changes and reload from the server.

# 5 Extensions

## *5.1 Lightweight Types*

Data objects can be quite heavy. It is suboptimal to transmit them to create a navigation tree or an overview of a graph. A tree has a lot of nodes and displays a very small number of properties and the hierarchical links between nodes. The lightweight object pattern provides an efficient way to transmit this information to the clients. A lightweight representation of each data object type displayed in an overview is defined. It contains only the data shown on the screen. Often these lightweight objects are read-only. The main design consideration of lightweight objects is retrieval speed and partial graph representation especially when complex queries are used. Lightweight instances should never have a reference to their data object to avoid to transmit them over the wire when the lightweight graphs are sent.

The MDA code generator described in “MDA Generation” transparently generates all defined lightweight classes and the associated transformation classes. The transform classes create the lightweight representation of a data object graph.

Because lightweight objects are separate classes, they can be mapped to their subset of attributes on the database tables. OJB supports directly multiple mapping on the same physical table. The application is responsible to synchronize the two representations when modifications are performed on one instance. The best approach is never to cache any lightweight object because they could represent partial graphs depending on the query used to compute them. Instead of trying to transform complex topologies through an injective function, the information is retrieved each time from the database.

Lightweight types should only provide read access to their properties.

### 5.1.1 Framework Dependencies

The lightweight representation has common attributes with its data object. The transformation classes are available to synchronize the lightweight instances with the data object instances. The framework never references lightweight instances.

### 5.1.2 Restrictions

If the application supports editing of lightweight items, it must synchronize the data objects in the program. The preferred way is to update the data objects with the modifications. Upon completion the above described mechanism is used to update the data objects and synchronize the lightweight representations.

Applications can hand code the retrieval of lightweight objects to increase the speed of the application. Such custom operations should avoid to store the objects in the cache. This strategy completely shields the framework from the application specific retrievals and no special measures need to be taken to synchronize the various views on the database.

## *5.2 Shallow Objects*

A new requirement was found during productive use of the framework. Some

developers need to update one data object without having to retrieve or send the complete graph to the server. This requirement opens interesting challenges. Now the framework must replace one object in a graph and conserve its topological structure.

The framework manipulates three kinds of objects: data objects, lightweight representations and the shallow view on an object. The table below shows which functions are available for each kind.

| ***Function*** | ***Full*** | ***Light*** | ***Shallow*** | ***Comments*** |
|---|---|---|---|---|
| Retrieve | yes | yes | yes | The result of a query request |
| Insert | yes | no | no | Creation of a new object |
| Store | yes | no | yes | Commit of all changes |
| Delete | yes | yes | no | Deletion of an object in database |

*Table 1 Shallow & Full Functions*

Store operations on shallow objects are supported but are seldom in real applications. Insert operations on shallow object are dangerous and therefore prohibited because the object could not have mandatory nodes.

Ergonomic studies show that in place edition of attributes in the navigation tree is too complex for casual users. Therefore the lightweight objects can only be retrieved or deleted.

The shallow feature could easily be implemented with the help of the MDA generator. A shallow property extract and update service must be generated for each data object type supporting shallow operation. The extract operation creates a copy of the object with all relevant simple and indexed properties. If a property is relevant or not is data user definition. The update operation performs the inverse function. These two operations synchronize the shallow copy of the object with the original one.

The only drawback is that shallow update operations must separately be programmed otherwise the framework will assume that the missing objects have been deleted. Therefore support of shallow operations duplicates the number of CRUD methods for a specific data object type.

## *5.3 Queries*

Complex applications require domain specific queries to retrieve a set of relevant data. Often a user selects only the data objects relevant for the current business case he is working on. The server should provide mechanisms to realize custom specific inquiries and return the set of found objects.

Queries should return domain objects and not columns. Lightweight objects can be used to return a subset of the attributes. Queries can also be used to return incomplete lightweight subgraphs as search results. Lightweight representation is preferred because partial graph editing is not well supported because the semantics of the delete operation is not clear.

Reports are realized with reporting tools and should be separated from the core functionality of the application. As much as possible reports should be off line.

## *5.4 Derived Attributes*

Attributes of a data object can be the result of a computation. A computation is a formula with variables. A variable is another attribute either computed or not and mathematical operators.

Simple formulas such as aggregation or propagation of attributes can directly be declared in the data objects. More complex formulas should always be declared in a service class as suggested in the SOA approach.

A framework to implement a powerful yet simple computations is beyond the goals of this article. The description of such a framework can be found in the article “Data Objects Derived Attributes” [mb-attributes-2003].

## *5.5 Reference Codes*

A reference code is an enumeration of values defining a domain relevant type and its legal values. For example the set of currencies defined in the related ISO standard is a reference code. Each code is identified by the type it belongs to – represented as a Java class - and a unique code identifier. Reference codes can have a hierarchical structure. For example the department structure of a worldwide company can be represented through a hierarchical reference code.

The framework knows the concept of reference code and uses the reference code manager to access values of codes. The implementation of the framework never stores reference code objects or send them over the wire. Only the identifier of the code is stored or transmitted. Reference code objects are always transient ones. The mapping to the type is done implicitly.

- The database definition connects the identifier of the code to the reference code table. Therefore persistent properties can only contain persistent reference codes. Technically codes are mapped to 1-0..1 or 1-1 relationships.
- The transformer factory knows the dependency between identifier and type. The name of the identifier is hard coded in the transformation routines. This approach is congruent with the concept of a reference code being a typed business enumeration.
- Transient reference codes are build in the application logic and are not part of the data object graph framework.
- The framework provides services to transfer reference codes from the server to clients. Incremental updates are available to minimize network overhead.

# 6 Architecture

The architecture describes the structure of our frameworks and the main abstractions constituting it. The core of the framework is two interfaces defining the responsibility of a data object class and the handler manipulating these objects, and a set of classes providing default implementation of the interfaces for the client and for the server. Support functions are provided in the data objects and are generated through MDA technology and the adequate cartridges responsible to generate the source code and configuration artifacts.

The major goal of the framework is to minimize the number of constraints on the data object classes. Therefore the approach using a common ancestor is not an option. The interface variant was selected. Because the source code of the classes is generated no coding overhead is introduced to implement the interface.

The second goal was to minimize the number of concepts used for the realization of the above described mechanisms. Two main patterns are webbed in the source code. The visitor pattern is used to traverse trees and perform operations on them. The facade pattern hides the underlying layers from the application. This design is a nice example of a clean and lean architecture.

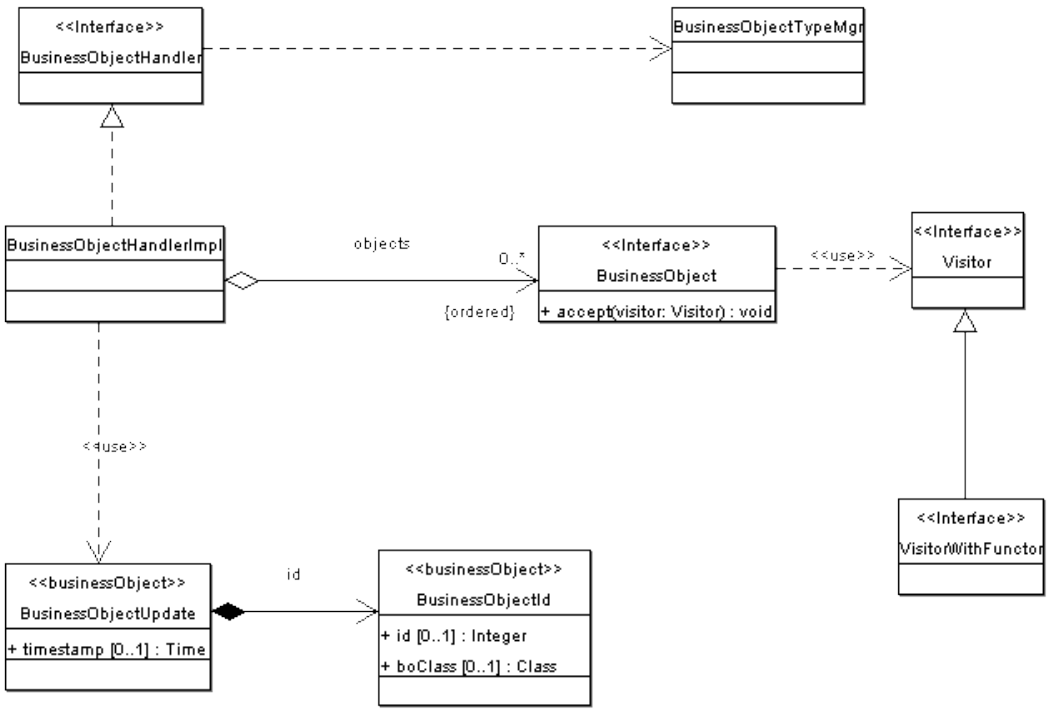


*Illustration 5 Business Objects Overview*

## *6.1 Data Object Identifier*

A data object identifier uniquely identifies a data object in the application or in the database. The class is immutable to reflect its role as identifier.

- *Class:* The property defines the class of the data object.

- *Identifier:* The identifier is the unique key for a data object of the above type.

## *6.2 Data Object Type*

A data object type defines the characteristics of a data object type. The following attributes are defined.

- The class of the data object class.
- The class of the lightweight representation class of the data object.
- An optional class defines the interface of the data object type.
- A flag indicating if data object type instances are used as roots or not.

## *6.3 Data Object Interface*

Business object classes must implement the data object interface. This small interface allows the framework to manipulate instances.

- *Identifier property:* The property is read-only. The framework uses reflection to modify it. The user application has not possibility to change it.
- *Modified property*: The property is modifiable. An update operation sets the modified flag of the interface. Immutable data object properties do not influence the interface.
- *Timestamp property*: The property is modifiable. The framework uses reflection to modify it. The user application has not possibility to manipulate it.
- *Foreign key properties*: Foreign keys should be defined as anonymous fields. They are not visible in the data object.

The interface can be extended to support additional properties common to all data objects of an application. For example the identifier of the user having performed the last update on the object is such an attribute.

## *6.4 Lightweight Data Object Interface*

Lightweight data object classes must implement the lightweight data object interface. This small interface allows the framework to manipulate lightweight instances.

- *Identifier property:* The property is read-only. The framework uses reflection to modify it. The user application has not possibility to change it.
- *Foreign key properties*: Foreign keys should be defined as anonymous fields. They are not visible in the data object.

## *6.5 Reference Code Interface*

Business objects often has properties with a domain model type. These types represent a set of legal values for a domain specific type system. Our framework provides an elegant mechanism to model these types. Two interfaces are provided.

- The reference code interface for regular reference code. A reference code has an identifier, a short description, a extended distribution, and a sort order.
- The hierarchical reference code interface for hierarchical reference code. The

interface is an extension of the regular reference code. It has an additional property defining the owner of the code.

## *6.6 Data Object Handler Interface*

The data object handler provides services to retrieve, store, and delete a graph of data objects given its root. It also manages the content of the data object cache.

- Retrieve the graph of data objects given the identifier and the class of the root object. The handler first tries to retrieve the graph from its cache. If not found it forwards the request to the underlying layer for execution.
- Retrieve the graph of lightweight objects given the identifier and the class of the root object. The handler first tries to retrieve the lightweight graph from its cache. If not found it forwards the request to the underlying layer for execution.
- Store the graph of data objects in the database. Deleted objects are first removed, new objects are inserted and modified ones are updated in the persistent store. The new objects receive their unique identifier. Timestamps are updated. Upon successful completion the new identifiers and timestamps are sent back to the client to synchronize its copy of the graph with the one now stored in the database.
- Remove the graph of data objects with the given root identifier. All objects are deleted in the database. The objects are removed from the server cache.
- Create a deep copy of a graph of data objects. This function ensures that clients never manipulate the original objects.
- Update or deletion of lightweight objects are not provided because the semantic of these operations cannot be defined without side effects.
- Manage the cache where the objects are cached. It is possible to find out if an object is in the cache and to flush the cache.

The handler has very few constraints on the object it manages.

- The managed objects must be either data object or lightweight objects.
- Each data object type must register its visitor with functor class. This visitor is used to perform all manipulations needed to synchronize data object graphs with their persistent representation. The visitor must provide a default constructor. Because it is in general automatically generated through our MDA tools, the user has not additional work.
- The data object must provide a public method accept with one parameter of type visitor. The handler calls this method through reflection.

At least two implementations of the handler exist. The server side implementation maps the services to the persistence methods of the OJB component. The client side implementation maps the same services to the transport layer to propagate the requests to the server. This approach allows to plug tailored implementation for example to minimize the network latency.

Both implementation uses the visitor pattern to traverse the graph. Variants of the visitor functor specialization are used to perform the required actions on the nodes. The used Java environment does not support generic therefore the implementation

must manipulate classes implementing the visitor pattern without knowing them at development time. The current solution uses the reflection package and the assumption that the method used to traverse the graph has the signature.

```
void accept(Visitor visitor);
```

The handler is configured with the corresponding visitor with functor functor and call the accept method on the data object to manipulate. The logic is defined in a strategy class. Lazy evaluation of relations is delegated to the persistence layer, e.g. OJB.

### 6.6.1 Cascaded Deletion

The UML standard defines the concepts of composition and of aggregation. Composition implies cascaded deleted of the owned objects when they are removed from the owning object. Aggregation leaves the decision to the application. The data object handler always delete owned objects from the database when they are removed from the owning object. This approach follows the closure rules of ODMG and JDO standards.

This mechanism is not adequate for all applications. Two solutions are provided. First the developers can extend the strategy class to handle these situations. Second an extension of the framework support reflective behavior through a meta model describing the data objects. This solution is extremely flexible but more complex to deploy.

## *6.7 Persistence Manager*

The persistence manager defines a thin abstraction layer above the chosen persistence framework. It heavily uses visitors and function objects to fulfill its responsibilities. Additionally it hides the idiosyncrasies how multiple databases can be accessed concurrently.

The package provides wrapper for the transaction and query classes used in the persistence framework. Delegation is used to hide these classes and method signatures. Therefore if the framework is exchanged the client application does not need to edit its source code but only recompile all classes.

The query classes are used to program application specific search functions to retrieve specific graph of data objects.

## *6.8 Reference Code Manager Interface*

The reference code manager interface defines the services provided to manipulate a set of reference codes. A manager can delegate the management of reference code types to another reference code manager, building a hierarchical structure of managers. This approach scales up and allow the integration of multiple data object models.

Reference codes are always loaded when the application is started to enhance responsiveness of the application. No differences exist between reference codes and regular enumeration types.

## *6.9 Caches*

Caches are no more part of our framework but are provided as pluggable components. The framework communicates with the cache to ensure they are in sync with the data layer and the database. These management activities are necessary because the manipulated graphs of data objects are always copies of the ones stored in the cache.

## *6.10 Detailed Design*

The detailed design is defined in the JavaDoc comments of the source code. The documentation is available in XHTML and PDF formats.

The documentation and source code are checked for completeness and quality assurance with DocCheck, PMD, JavaNCSS and Jdepend.

# 7 Best Practices

## *7.1 Models*

Domain models should be object-oriented and cleanly defined. Commonality should explicitly modeled as interface and abstract classes. Programmers often defines too many attributes when using code generators. Care should be taken to define only properties needed in the application.

Each entity should have a primary technical key used only in the application and not visible to the users. This approach guarantees that any instance can be uniquely be identified and hinders the users to connect their domain model to a technical key.

Framework functions should access properties through reflection mechanisms and avoid calling the getters and setters. User interface programmers often tend to hide data functions in accessors. This practice is bad design but a fact of life. The framework should be resilient to such designs if the cost is not too high.

The developer is responsible to connect new root objects with objects owning them. An approach is defined in the explorer framework.

## *7.2 Coding Conventions*

Business object classes should use well-known naming conventions. The authors encourage the use of Java beans conventions. Indent the source code following the rules defined by Sun Inc.

When using collections instead of arrays for indexed properties use the conventions defined in the Jakarta “Bean Utils” project.

## *7.3 Queries*

Complex queries, in particular complex searches of relevant data containing only matching objects are best implemented with hand coded SQL statements. The speed improvement is tremendous. In a management application for insurance contracts the gain was a factor hundred. Regular retrieval operations should always be realized within the framework. This approach minimizes maintenance costs without impeding responsiveness of the application[11].

Best practice is to retrieve most of the fields for all objects in one single query to minimize database overhead. The hierarchy of lightweight data objects is constructed in the program. Instantiation of objects and associated relations are best done in the code. This approach is an extension to the “*Fast Lane Reader”* pattern described in the J2EE blueprints. The difference is that our variant returns a forest of data objects instead of a tabular representation.

As stated above the constructed lightweight objects are not cached on server side. This design decision eliminates the design of a coherence mechanism between the

11 If the query statements are written using for example the criteria classes available in OJB, they would be portable between various databases. OJB will take care to translate the abstract query into the specific SQL dialect of the target database. Simple online reporting tasks should also realized with the same mechanism.

lightweight and data application models[12].

## *7.4 Framework Manipulations*

The framework often manipulates data object instances for example when transforming the instances into CORBA objects. The preferred way to access properties should be through reflection. Calling the manipulators methods either directly or through introspection could trigger undesirable side effects[13].

## *7.5 Database*

Modern databases have advance features for automatic generation of technical primary keys. These features should be used instead of creating in the code the identity keys.

12 This decision was refined during the development and deployment of a large contract management application for insurance companies. Practice showed that trying to synchronize the lightweight model with the business model was very complex and did not bring any advantages. Not caching lightweight objects simplifies the design in the context of a threaded client server application.

13 This approach is used in OJB to insure best conversion between business object and database. If the design of the application guarantees no side effect introspection can be used.

# 8 Future Directions

## *8.1 Introduction*

A detailed discussion of learned lessons in one project using this framework can be found in “Lessons learned in Insurance Project”. One major finding in the project is that performance is NOT a problem. Here some empirical measurements.

- Queries were used to retrieve a forest of lightweight objects fulfilling the arguments of the request. For testing purposes we defined general queries returning a set of about one thousand trees of lightweight objects. The number or returned lightweight instances is about twenty thousand and each instance has an average of four attributes and one link to another object.
  The retrieval of the data for a DB2 database through OJB, post processing of the tree to compute derived attributes, transformation of all instances to their CORBA representation and transmission to the client application took in average 8 seconds on a personal computer. The bottlenecks are the database itself, and the CORBA layer. In real environment the bottleneck will often be the network.
- Retrieval of a tree of about two hundred of data objects with a total of four thousand attributes needs in average four seconds on the same personal computer.
- The same application servers run on a productive UNIX server are a factor two to four faster. Exact measurements were not possible due to lacking performance monitoring tools in the production environment.

The server application development trend is to delegate more and more functions to frameworks and libraries similar to the one described in this text. Integration of our data object graph framework in controlled environments such as JDO, J2EE and Tomcat will increase this trend.

## *8.2 Improvements*

The framework was used in a complex mission critical contract management application in the insurance and banking domain. The code should be refactored, documented and released. The framework currently tracked more precisely the state of a data object, and uses an internal model instead of a visitor based mechanism to walk through the graph. This exact tracking is superfluous.

Caching mechanisms should be based on the emerging JSR standards. Open source implementation are available and should be introduced in the next release of the product.

The framework ideas should be published in technical editorials and presented in internal workshops. The actual version is used in mission critical applications. A extensive search in the literature and in Internet has revealed that no similar framework is currently available as open source.

## *8.3 Additional Mechanisms*

The query support of the framework is minimal. Mechanisms should be designed to support generic search queries for all data object types used as roots. For example “*query by criteria*” of OJB could be used as a basis.

The generator could be extended to generate a generic client interface to view graphs for a given domain. This client simplifies the verification of the data model before full-fledged client applications are available.

The generator could also be extended to generate user interface support classes.

### 8.3.1 Graph Visualization

Often it would be helpful to visualize complex graph structures during development or maintenance. Visual representations greatly simplify analysis of problems in the application or in the database. The visualization tool should display any graph without programming effort. The user should only need to configure its data object graph structure.

The explorer framework could be used to realize this visualization tool.

### 8.3.2 Distributed Transactions

Currently it is not possible to define a transaction spanning multiple physical databases. Transactions work only on one database or on a set of logical databases all located in the same physical database. Some applications have requirements to support two phases commit on multiple physical data sources.

The simplest solution will be to deploy the server side part of the framework in a managed environment such as a J2EE container. These environments provide a transaction monitor with two phase commit features. The used persistence layer OJB already provides extensions for such controlled environments. The effort to implement this solution is about one week including testing and packaging.

# 9 Implementation

## *9.1 OJB Support*

The OJB library hooks its own collection classes when retrieving relationships from the database. These classes tracks changes to handle object removal. The framework has similar features. The integration of OJB must avoid interferences between the two frameworks.

- The DOG framework should carefully manipulates attributes containing foreign key values. The best approach is to defined such fields as anonymous to avoid tampering of their values in the application. OJB uses these fields to generate appropriate insert, update and delete statements for dependent objects – meaning part of a relationship with cardinality higher than one -.
- The DOG framework always manipulates copies of object trees retrieved through OJB. Data objects all support deep copies of graphs. The implementation of the clone operation removes the OJB specific removal aware collections.
- Trees available in clients or sent from the client are always copies and not orginal ones. The client never needs to perform a deep copy before manipulating objects through the DOG framework. As a bonus no dependencies exist between client applications and OJB libraries.
- Server functions should always make a deep copy of the tree before manipulating it using DOG services. If not DOG functions are used the deep copy is optional.
- Shallow operations always copy modified data back to the deep copy of the original tree before calling store operations.
- Each time a copied tree of object is stored, the original objects should be removed from the cache and replaced with the copied instance. This action guarantees that OJB cache never contains obsolete objects. The DOG concept of root object is used to remove the transitive closure of all objects in the graph and replace them with their copies. Before the changes are propagated to the client the changes must be flushed to the database to guarantee that optimistic locking timestamps or counters and new identifiers are generated[14].

Analysis is under way to determine if a tighter integration would diminish the persistence overhead.

## *9.2 Hibernate Support*

To be written.

## *9.3 C++ Support*

The DOG framework is also available in a C++ implementation. The coding guidelines are similar to the ones of Java.

- Package names are mapped to name spaces. The directory structure is the same. A class is defined in a header file and an implementation file. The public declarations

14 This approach is compatible with container managed transactions. The framework does not need any knowledge of the transaction mode of the executed service.

of a package are provided in a separate file name including the public and protected classes but not the package and private classes.

- Method names are the same. Java methods implementing operations available as operators in C++ are mapped to C++ operators.

# 10 Glossary

| | |
|---|---|
| CORBA | *C*ommon *O*bject *R*equest *B*roker *A*rchitecture |
| CRUD | *C*reate, *R*ead, *U*pdate, *D*elete |
| DAO | *D*ata *A*ccess *O*bject |
| DOG | Data *O*bject *G*raph |
| DTD | *D*ata *T*ype *D*efinition |
| DTO | *D*ata *T*ransfer *O*bject – the older terminology for this pattern was value object – The pattern is sometimes called Transfer Object |
| IDL | *I*nterface *D*escription *L*anguage |
| J2EE | *J*ava 2 *E*nterprise *E*dition |
| J2SE | *J*ava 2 *S*tandard *E*dition |
| JDBC | *J*ava *D*ata *B*ase *C*onnection |
| JDK | *J*ava *D*evelopment *K*it |
| JDO | *J*ava *D*ata *O*bject |
| JMS | *J*ava *M*essage *S*ervice |
| JRE | *J*ava *R*untime *E*ngine |
| JSP | *J*ava *S*erver *P*age |
| MDA | *M*odel *D*riven *A*rchitecture |
| MOF | *M*eta-*O*bject *F*acility |
| ODMG | *O*bject *D*atabase *M*anagement *G*roup |
| OJB | *O*bject/*R*elational *B*ridge or *O*bject *J*ava *B*ridge |
| OMG | *O*bject *M*anagement *G*roup |
| O/R | Object / *R*elational |
| POJO | *P*lain *O*ld *J*ava *O*bject |
| PIM | *P*latform *I*ndependent *M*odel |
| PSM | *P*latform *S*pecific *M*odel |
| SDO | *S*ervice *D*ata *O*bject |
| SOA | *S*ervice *O*riented *A*rchitecture |
| UML | *U*nified *M*odeling *L*anguage |
| USDP | *U*nified *S*oftware *D*evelopment *P*rocess |
| XMI | *X*ML *M*eta-Data *I*nterchange |

# 11 References

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

| | |
|---|---|
| BeanUtils | manipulation library for Java beans<br>http://jakarta.apache.org/commons/beanutils |
| Commons Log | Common logging interface package for Java Applications<br>http://jakarta.apache.org/commons/logging |
| DocCheck | quality insurance for detailed design documentation<br>http://java.sun.com |
| Eclipse | Java development environment<br>http://www.eclipse.com |
| Graphviz | creates the technical graphs used in this document<br>http://www.research.att.com/sw/tools/graphviz |
| Log4J | logging package for Java applications. Variants are provided for C++, PHP, etc.<br>http://www.apache.org/logging |
| OJB | ODMG 3.0 and JDO compatible persistence layer<br>http://db.apache.org/ojb |
| Validator | validation framework<br>http://jakarta.apache.org/commons/validator |
| Velocity | code generation template engine<br>http://jakarta.apache.org/velocity |

# Bibliography

Brose2001: Gerald Brose, Andreas Vogel, and Keith Duddy, Java Programming with CORBA, 2001
mb-attributes-2003: Marcel Baumann, Business Object Derived Attributes, 2003

# Index of Tables



# Illustration Index