Data Object Attributes

How to Initialize and Compute Attributes

Marcel Baumann

Version 1.0.1

Table of Contents

1 Introduction

When developing an application that implements 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.

Business logic, in a very broad sense, is the set of procedures or methods used to manage a specific business function. Taking the object-oriented approach enables the developer to decompose a business function into a set of components or elements called *business objects*. Like other objects, business objects have identity, state and behavior.

To manage a business problem you must provide the desired functionality. The set of business-specific rules that help identify the structure and behavior of the business objects, along with the pre- and post-conditions that must be met when an object exposes its behavior to other objects in the system is known as *business logic.*

Complex domain model contains complex business object topologies and numerous properties defining their state. These properties describe aspects of the domain model and are tremendous importance to the users of the application. Some of them are the result of a mathematical computation. Such attributes are often called *derived attributes*.

The business logic provides major functions to

- Validate attributes, business objects and graphs of business objects. Only complete and consistent graphs should be stored or handed over to external programs.
- Compute the value of derived attributes. Computation is defined with formula having dynamic variables and mathematical operators. Variable are attributes of the same or other business objects. These variables can again be either computed or not. All computations are handled as executable formulas applied on a set of connected business objects.

The following questions must be answered in such an environment.

- Which is the validation rules to compute for a selected attribute?
- What is the event triggering the validation of an attribute?
- Which is the formula to compute a selected derived attribute?
- What is the event triggering the computation of a derived attribute?
- How are derived attributes initialized?
- Should derived attributes be stored in the database?

This document tries to provide pragmatic answers to the above questions. It describes an approach to streamline the implementation of derived attributes and their computation.

Intended audience is software developers who want to use the framework or extend it further.

2 Assumptions

The described approach was developed to respect a set of hypothesizes. The following assumptions are fulfilled in the target applications.

- The validation rules and formulas can be changed during the lifetime of the application. For example another computation rule must be applied if a business status in the object has a new value.
- The architecture of the framework uses the SOA approach. Computation rules are often defined as business rules and change upon time. A common requirement of business objects is that they be reusable by different components of the same application or by different applications. Business objects are able to be used by various components when they are developed in a standard way and run in an environment that abides by these standards.
- Simple formulas can directly be implemented in the business object classes to augment the legibility of the source code.
- A common characteristic of business objects is that they often operate on shared data. In this case, measures must be taken to provide concurrency control and appropriate levels of isolation when computing derived attributes.
- The domain objects implement the Java bean coding conventions. They provides accessors methods to their properties and support change event propagation.

The following terminology is used in this document.

- *Attribute*: A field of a business object containing a value. It is also called property.
- *Derived Attribute*: An attribute, which value is computed from other attributes and a mathematical formula.
- *Control Attribute*: An attribute, which value defines if a set of attributes is active or not. The set is said to be dependent of the control attribute value. At any time at most one control attribute owns any attribute.
- *Inactive Attribute*: An attribute which is currently disabled through its control attribute. An inactive attribute should not be displayed or stored in the database. Ideally its value should be reset to null. An inactive attribute is never visible on the user interface.
- *Active Attribute:* An attribute currently in use in the application is active. When an attribute changes its state from inactive to active, it must be initialize to its default value. If the attribute is a derived one, its value must be computed.
- *Constrained Attribute:* A constrained attribute has its legal value range constrained by the value of a control attribute.
  A constrained attribute can also be a control attribute.
- *Formula:* A formula is a mathematical function performed on a set of attributes and returning the new value of a derived attribute. A design variant is to group all the formulas of a business object type together.
- *Validation:* The validation is a boolean function checking if the value of the attribute respects the constraints of the business domain model. Validation rules

never change the value of any attribute. No validation rules should exist for a derived attribute.

- *Lexical and Syntax Validation:* The lexical and syntax validation verify if the attribute is mandatory or not and if the attribute value is within the requested range. This information is often used in the interface to provide visual feedback to the user. Lexical validation should be performed during user input to provide adequate feedback to him.
- *Semantic Validation:* Once the lexical validation is successful, the attribute values can be validated against domain business rules defining constraints between attributes. This information is often used in the interface to provide explicit error messages to the user and provides hints how to correct the problem. Semantic validation should be performed in the client and server applications.

- *Reference Code Type:* Reference code types define an enumeration of legal values seen from a business view point. Examples of reference code types are ISO currencies or ISO paper sizes. These sets of values often change over time and a mechanism must be provide to select only the values currently active. Reference codes are the ideal construct to define a range of values for an business object attribute.

Complex application models are often modeled with derived control attributes. In other words the domain model defines master control attributes for the overall behavior, and local constrained control attributes for specialized behavior. If the analyst is using state machine to model his domain the above logic can be mapped to a hierarchical state machine.

The user interface needs information such as grouping of attributes or if an attribute is editable or not. This information is part of the meta model of the user interface but not of the meta model of the business layer.

3 Use Cases

The following use cases were identified. Each of them requires derived attribute functionality.

- Compute new values when a property is changed. Each time a given property changes a set of formulas are computed and their result is written to the associated derived attribute.
- Initialize a set of derived attributes when a control property is changed. Each time a given control property changes a set of properties is initialized. The value of the derived attribute is defined through a formula. The result can be a default value, the result of a computation, or a flag indicating the property is disabled.
  Only the set of newly activated attributes should be initialized. Values already inputed from a user should never be overwritten.
- Initialize a graph of newly created business objects. Upon creation of a set of connected business objects, they must be initialized. This initialization could trigger the computation of derived attributes.

The following use cases are not only for derived attributes but consider similar

situations.

- Apply user preferences to a graph of connected business objects. The mechanisms to initialize business objects can be used to apply user preferences.

Overall initialization should be performed with the help of an initialization visitor. Property initialization upon the change of a control property should be performed with derived attribute mechanism.

4 Concepts

4.1 Attribute Kinds

4.1.1 Regular Attributes

Regular attributes are fields containing data which can be edited from the user. An attribute has always a type.

A subset of regular attributes has reference code as their type, so their value range is predefined. The user interface often uses drop down lists to select a legal value.

4.1.2 Derived Attributes

Derived attributes can be identified either statically or dynamically. Static derived attributes are class fields, which are always computed.

Dynamic derived attributes are business object fields, which are sometimes computed. The decision if they are computed or not is defined in a business rule and can only be evaluated during runtime. Care should be taken to insure that the decision if an attribute is derived or not can always be computed without implicit knowledge of the history of the object. The business rule uses values of other properties to generate its output.

Derived fields should be transient. The only exception is when the data is required in the database for reporting purposes.

As a remark static derived attributes are a lot easier to verify against the requirements and maintain than dynamic ones.

4.1.3 Control Attributes

The main feature of a control attribute is the function defining which attributes are activated or deactivated when its value is changed. The framework assumes that an attribute is controlled by at most one control attribute.

Set of controlled attributes = function(state)

Set of activated attributes = function(new state) – function(old state)

Set of deactivated attributes = function (old state) – function (new state)

The framework must provide support for the following functions.

- Retrieve the set of control attributes for a given state of a control attribute
- Retrieve the set of activated attributes when changing from an old state to a new

state.

- Retrieve the set of deactivated attributes when changing from an old state to a new state.
- Set the value of all deactivated attributes to null.
- Initialize and apply user preferences to the set of activated attributes, which are not derived attributes. Activate the computation formula and triggers the computation for all activated attributes being derived ones.

It is not of the responsibility of the framework to detect computation formulas requiring attributes being currently deactivated.

4.2 Validation Rules

To be written.

4.3 Formulas

Formulas implement business logic at property level. This business logic should be implemented in services following the service oriented architecture paradigm. If wanted the computation of a business rule can be requested through the service. But registration mechanisms hugely simplify the firing of all involved business rules if triggers are defined in various components of the applications. As a standard example the rules should be applied in the client and in the server applications.

The following rules were identified.

- Formulas are defined on attributes. Attributes belongs to business object types.
- Trigger sources are attributes. Attributes belongs to business object types.
- Paths are relative from the point of declaration. Paths use positional notations. Special constructs are available to define paths applicable on all instances of a collection.

4.3.1 Initializations

To be written.

4.3.2 Derived Attribute Formulas

Aggregations

An aggregation formula computes the sum of a attribute value over a set of business objects. Instances are stored in a collection. Below some formula examples are given.

Aggregated attribute = sum (a(i)), 0 <= i < size()

Aggregated attribute = min (a(i)), 0 <= i < size()

Aggregated attribute = max (a(i)), 0 <= i < size()

The owner of the aggregated property also owns all instances involved in the computation. The accessed indexed properties are read using the provided accessors.

The trigger is the modification of the involved attribute in one of the owned object

and is propagated using the Java bean updated event.

Propagations

A propagation formula computes the new value of an attribute over a set of business objects. The instances are stored in a collection.

Propagated attribute(i) = function (lead attribute, size()), 0 <= i < size(0)

The owner of the lead property also owns all instances which attribute is updated. The accessed indexed properties are written using the provided accessors.

The trigger is the modification of the lead attribute and is propagated using the Java bean updated event.

Computations

Computations are mathematical formulas of any complexity. The only natural restriction is that circular references are prohibited. Usually the computations are coded as Java methods in the application. Hooks are provided to scripting languages to enable developers to formulate the computations in scripts. We recommend the use of the "*Bean Shell*" language to write complex scripts.

All functions are provides as instances of various functor classes. The Jakarta functor package is used to provide this functionality. Compositions of primitive functions in a more complex one is supported as operators of functors. The functor package provides all services to implement a functional programming model.

4.3.3 User Interface Hooks

A user interface dialog or view contains a set of properties. Two modes exist. Either the application computes the derived attributes each time a property is modified or only upon successful completion of editing activities. Both variants are realized with the same code. The first approach uses a copy of the business object, the second one directly manipulates the original object.

Each time a property is modified, a change event is triggered and propagated to the derived attribute calculator. The manager selects all functors registered for the property belonging to the business object type. Each functor is executed. The order of execution is undefined.

This mechanism also works for in line editing in table views.

4.4 Business Object Initialization

4.4.1 Creation of Business Objects

The service oriented architecture requests a factory for the creation and initialization of new business objects. The factory is also responsible to create more complex business object graphs.

The creation process instantiates new objects, initialize them and applies user preferences on them. These operations should be under full control of the factory to insure consistent data upon a creation request.

The user preferences aspect must be an optional one to allow creation of systems from over systems or through a workflow system without an user being logged in.

4.4.2 Initialization

The initialization of newly created business objects under responsibility of the factory. The requirements should specify for each attribute its initial value.

If necessary the derived attribute computation rules must be activated before the initialization process is started to insure consistent data. Initialization is orthogonal to the business object architecture and derived attributes framework.

The initialization process should be implemented as a visitor traversing the tree of business objects and setting each attribute to its default value. This approach is more flexible than to set the values in the constructor. Complex applications often require different initialization based on the origin of data. Examples are migration of data from other systems could require different initialization rules or different initializations based on control attributes of the owning business object.

4.4.3 User Preferences

User preferences are default values overwriting the application default values. The same approach as for initialization should be used. A visitor traverses the tree of business objects and overwrites the attributes for which the user has defined a custom default value. Preferences are orthogonal to the business object architecture and derived attributes framework.

Preferences can be organized in hierarchies. For example the department of the employee can provide preferences to avoid that each user has to define the same values in his user preferences but still allow a user to overwrite the department values.

4.4.4 Reset Default Values

User friendly applications often provide a mechanism to allow a user to reset the values of a business objects to the default ones. The default ones are the initial values or his user preferences if defined. The reset operation is realized by applying the initialize and user preference visitors on the business object to be reseted.

The factory must provide services to reset individual attribute to their default value if the application enables and disables dynamically attributes. Either each attribute or each set of related attributes can be set back to its default value.

5 Meta-Model Definition

5.1 User Input

The software developer should input the description of the meta-model with all its instances, rules and discriminators. To minimize the initial investment for the requirements capture tool a user readable format based on XML was selected.

The major complexity is the description of the rules and discriminators. The rule description must respect the following conventions.

- Properties are described using the syntax of bean utility and velocity packages.

Each property starts with a dollar sign and can uses the optional braces as in velocity. The notation of the property is the one used in the bean utility framework.

- All properties used in a formula are are relative to the current business object instance.
- The remaining elements of the formula must comply to the Java syntax.

The discriminator definition must respect the following conventions.

- The discriminator is a list of actual parameters.
- Each parameter has a name, and a value. The user is responsible that the value is compatible with the type defined in the associated formal parameter.
- The attribute for which the rules are specified must have a compatible discriminator type. It is compatible if each parameter name is defined in the type as a formal parameter. A formal parameter has a name and a bean utility or a bean shell expression and a type. The type information is used to instantiate the expected Java object.

Currently formal parameter types are either wrappers of primitive type, strings, or reference codes.

5.2 Best Practices

The following advices simplify requirements specification. The information should be available in tabular form and not be spread in various documents.

- Clearly identify all derived attributes, the triggers for their computations and the formulas.
- Clearly identify all control attributes and the set of attributes they manipulate.
- Clearly identify all attributes, which are dynamically enabled or disabled.
- Verify the completeness all formulas for all control attributes. All dependent attributes should be identified.

The following advices streamline the implementation of the above identified requirements.

- A formula is realized as a function provided through a service. This approach respects the service oriented architecture paradigm advocated in this document.
- The trigger of the computation of formulas should be centralized in a formula calculator and distributed in the business object instances. The calculator function can be activated or disabled from the application when the use cases request it. Computation can be performed for an individual attribute or a business object instance.
- The trigger of the computation of the validation rules should be centralized in a validation rules engine. The validation function can be activated or disabled from the application when the use cases request it. Validation should be performed on a business object and not on individual attributes. The object-oriented approach encourages to have consistent objects.

6 Architecture

6.1 Meta Model Approach

As discussed above, the architecture must provide fine grained initialization and computation mechanism if attributes can be enabled and disabled dynamically in the application.

The meta model is either available as runtime engine or hard coded in the application.

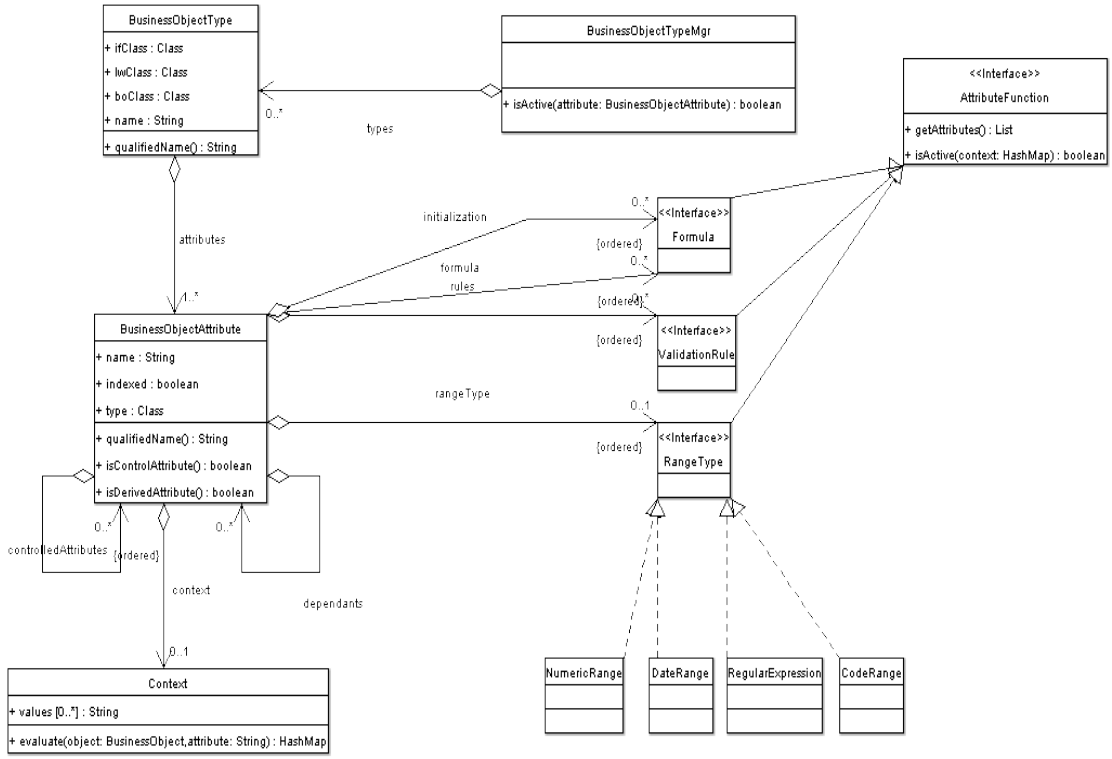

Illustration 1 Attributes Meta-Model

The choice of a variant is a tactical decision based on the requirements of the application. We suggest to use a meta model and to avoid hand coding as soon as a few hundreds of rules or attributes are identified.

The meta model defines the following entities.

- *Attribute:* A attribute is a property with a value and a type. Each attribute belongs to exactly one business object type.
- *Business Object Type:* The class defining a business object type. A type contains attributes.
- *Context Definition:* Set of attributes an attribute needs to compute its state. The state is used to define the set of currently controlled attributes. Only controlled attributes are triggered when the attribute changes. The context is also used to activate attributes added to the set of controlled attributes and to deactivate the ones removed from the set.
  The dependent relation defines the static set of all attribute possibly controlled.

- *Context:* The current values of all attributes defined in a context definition.
- *Formula:* An expression to execute. All referenced attributes in the formula are defined through a path expression. Formulas are used to initialize attributes when a new business object is created and to compute the value of derived attributes.
- *Path:* A expression using positional notation to define the path from the current context to a set of attributes.
- *Predicate:* A predicate is used to compute the current set of controlled attributes. A predicate only needs access to the attribute object and its current context.
- *Trigger:* A trigger is a relation between a control attribute and a derived attribute. All formulas defined on the derived attribute will be executed each time the trigger attribute changes. The trigger defines the path to the derived attribute. The path can contains collections without index to indicate that all items of the collection are part of the set of derived attributes.
- *Validation Rule:* A validation rule checks that an attribute has a consistent state. An attribute can have a set of validation rules.

A user interface meta model could be defined to provide the following information, which is not part of the business layer meta model.

- Is an attribute visible or not?
- Is an attribute mandatory or not?
- Has the attribute a range check associated with it?

6.2 Attribute Functions

An attribute function is a computation executed on behalf of a business object attribute. A function can be associated with a context. A context has two dimensions. The first dimension defines the set of attributes belonging to it. This dimension is the static definition of the context. The second dimension defines the value of each attribute in the context. This dimension is the runtime definition of the context.

Attributes functions are used to perform the following activities.

- As an initialization formula they compute the initial value of a given business attribute. The formula can be constrained through a context meaning multiple formulas can be defined as initialization function. The contexts must be disjoint.
- As a computation formula they compute the value of a derived attributes when one of the trigger modifying the result is changed. The formula can be constrained through a context meaning multiple formulas can be defined as computation function. The contexts must be disjoint.
- As validation rules they verify that the new value of an attribute respect the business rules defined on the domain model. The validation rules can be constrained through a context meaning multiple validation rules can be defined as validation function. The contexts must be disjoint.
- As a range type they define the range of legal values for an attribute.

Each attribute function needs a context in which it performs its computation. The

context contains the following information.

- The list of all attributes with their values in the domain context where the function is executed. Because this information is complex to retrieve in a tree containing multiple instances of the same type the list of attributes and their value is coded in the formula itself and not part of the meta-model.
  An elegant approach is to use the bean utility package to introduce a flexible programming model.
- The attribute itself, its old value and its new value. This information allow the use of the same formula for multiple attributes.
- The context of the attribute, meaning the business object instance owning the attribute instance.

The framework must construct the context for any attribute function defined in the system. To allow such a feature it must be able to retrieve from an attribute function the set of attributes used in the function. Currently it uses the services of the Jakarta bean utilities to perform this task.

The meta-model associates formulas with attributes. The same mechanism is used to connect initialization rules, computation formulas, validation rules and definition of sets of dependent attributes. The framework must now provide the algorithm to select the correct set of formulas during runtime. The algorithm is as follow.

1. Compute the context for the selection of formula of a given type. The context is defined through the set of attributes and their values. The values are computed using the bean utility expression defined in the context. The result of the computation is a hash map. The framework requests that all contexts for the selection contains the same set of attributes.
2. Select the set of formulas defined for this context. For each context the values of the attributes are compared with the ones defined in the formula context. If all values are equivalent the set is selected. If none is found the default set is selected if defined.
3. Execute the set of formula returned in the previous step. The order of execution is not defined. The parameters of the execute method is the business object instance, the name of the property. No other parameters are passed. It is not the responsibility of the computation framework to provide mechanisms to parametrize

6.3 Registration of Formulas

Each computation is declared as an instance of a functor. The functor is registered for a given business object type and a property. A convenience method registers the functor for all properties defined in a business object type.

A special version of functors can infers if they should be fired or not. This kind of formulas are used for dynamic derived attributes. This variant is application specific and is hidden from the derived attribute manager.

6.4 Registration of Business Objects

A business object tree is registered to the manager when the derived attributes should

be computed. The manager uses a register functor and the available visitor to traverse the tree and register itself on each relevant property in each visited node.

6.5 Visitors

The difficulty with the visitors is that the derived attribute manager must manipulate them without knowing their class. The solution is to use the reflection package to call the visitor accept method of the business object instance and give the visitor as parameter of the method. The operations specific to the manager are encapsulated in a functor. The visitor with functor interface provides the mechanisms to connect the package specific functors to classes unknown at compile time.

Here the details of the algorithm.

- Each business object class register its visitor, which must implement the interface *"Visitor With Functor*". The derived attribute manager defines the functor it needs to fulfill its responsibilities.
- If an instance of the visitor class does not already exist, it is instantiated the first time it is needed. The functor is set based on the function, which should be executed.
- The business object to traverse is inspected and its accept method is inferred with the help of the introspection package. The method is called with the visitor as unique parameter.

6.6 Code Generation

The meta-model describing the business objects, their attributes and all formula is also the reference model used to generate code for the validation and computation of attributes. The code generators should provide the following artifacts.

- Validator forms for the Jakarta validation framework
- XML declaration file for all rules and their context.

6.6.1 Jakarta Framework Integration

A goal of tremendous importance for an architect is to reuse existing components instead of reinventing the wheel. A well-known validation framework is available from the Apache foundation.

The framework uses a declarative approach based on a XML configuration file. This configuration describes the meta model of business objects and properties and the defined validation rules.

6.7 User Interface Meta Data

The user interface is interested in attribute characteristics.

- Is an attribute active? Only active attributes should be displayed.
- Is an attribute mandatory? Mandatory attributes are often graphically identified.
- Is an attribute read-only or can it be modified? Only changeable attributes have editable fields.

7 Solutions

7.1 Pragmatic Approach without Meta-Model

7.1.1 Concepts

Often the implementation of a meta-model is an expensive task and should only be performed if the additional functionality is required in the application. Otherwise a hard coded approach should be preferred.

The initialization and preferences visitor are hand coded. The formulas are also written by hand. This approach has limitations as soon as the number of attributes and rules in the system becomes high.

7.1.2 Realization

The complete meta-model is coded in the derived attribute manager responsible to compute the derived attributes. The formulas are declared in services following the SOA convention.

- Initialization and preferences settings are implemented as a set of visitors. These visitors are applied to a business object subgraphs upon creation. This approach works for any subgraph of business objects.
- Each formula is defined as a method in a service. Some applications delegate the method to an application specific domain defining an homomorphic hierarchy to the domain model.
- Each trigger of a formula, is mapped to a change event listener registration of the involved attribute in the business object after its creation. This mechanism computes all derived attributes and is compatible with Java bean conventions.
- A special case is when a control attribute changes its value. A initialization method must exist to deactivate the no more controlled attributes if necessary and initializes the newly activated or controlled attribute. The trigger mechanism is the same as the one used for formula computation.

This approach can only be maintain if the documentation of all attributes and their formulas are well-documented and cross-references are provided for efficient traceability. The documentation replaces the explicit meta-model and should have the same level of detail and quality.

7.2 Persistent Derived Attributes

7.2.1 Concepts

Offline reporting tools and external systems reads information from the database. These applications have a need to display derived attributes values without implementing the business logic used to compute them.

Therefore such requirements require that derived attributes are stored in the database each time they are computed. Derived attributes are computed each time one of the trigger attributes is modified.

7.2.2 Realization

7.3 Replication

7.3.1 Concepts

The replication component provides functions to replicate graphs of business objects and their attributes. The replication operation can be a simple copy of an object, or a partial replication of attributes as found in copy as, renew, and initialization based on a prototype. The semantics of these functions are defined in the business requirements of the application.

These services are often fine tuned during the life cycle of the application. The chosen implementation should allow efficient modifications of which attributes are replicated.

7.3.2 Realization

The realization of the services is heavily based on the use of the visitor pattern and the methods provided through the code generator. All business objects have a deep copy function. The replication server first creates a clone of the graph to replicate. Second a visitor is used to modify selected attributes or reset their values to default ones. The visitor implements a subtract algorithm. Attributes which are not replicated need update. This approach is efficient if the majority of the attributes are simply copied as is.

Each kind of replication is implemented as a visitor class. A common abstract class is provided to provide common features. These features insure that the replicated graph of business objects is recognized as a new set of business objects in the persistence framework.

The replication manager implements the most complex business requirements. For example it generates a new name for the replicated root, adapt date ranges, insert the replicated graph in an existing business object. These operations often use parameters provided by the client. The replication manager is the single entry point for the replication component. All others classes are private to the package.

8 Glossary

CORBA	*C*ommon *O*bject *R*equest *B*roker *A*rchitecture
J2EE	*J*ava 2 *E*nterprise *E*dition
J2SE	*J*ava 2 *S*tandard *E*dition
JDK	*J*ava *D*evelopment *K*it
JRE	*J*ava *R*untime *E*ngine
MDA	*M*odel *D*riven *A*rchitecture
SOA	*S*ervice *O*riented *A*rchitecture
UML	*U*nified *M*odeling *L*anguage

9 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 http://jakarta.apache.org/commons/beanutils
Validator	validation framework http://jakarta.apache.org/commons/validator

Illustration Index