



Distributed Object Graph Traversal, Preparation, and Transport

Prepared by: Jeremy Chan jeremy.chan@jonahgroup.com

The Jonah Group, Ltd.
80 Spadina Avenue, Suite 304
Toronto, Ontario
Canada, M5V 2J4

Tel: (416) 304-0860 x112 (Toronto)
Fax: (416) 946-1929
URL: <http://www.jonahgroup.com>

Goal: To describe a solution to the general problem of object graph traversal, preparation, and transport within a distributed application.



Revision History

Revision Number	Date	Description of Changes	Author
1	Oct 14, 1999	Created	Jeremy Chan
2	Dec 01, 1999	Revised for Java Report	Jeremy Chan
3	Jan 13, 2000	Revised based on comments from Dwight Duego of Java Report	Jeremy Chan

Table of Contents

1	INTRODUCTION	1
1.1	BACKGROUND / MOTIVATION	1
1.2	A SOLUTION	2
2	SYSTEM OVERVIEW	3
2.1	PLACEMENT	3
2.2	SAMPLE MAPPING	3
2.3	ARCHITECTURAL IMPACT	5
3	FILTER STRINGS	7
3.1	REQUIREMENTS	7
3.2	GRAMMAR	8
3.2.1	EBNF SPECIFICATION	8
3.2.2	SAMPLE FILTER STRINGS	9
4	CONCLUSION	12

1 Introduction

1.1 Background / Motivation

All distributed applications require some protocol for the transport of data between communicating processes. The nature of the application determines the extent to which the efficiency of the protocol is favored over its ease of use. Typically, the data in which the client software is interested resides at the server in the form of business objects. Protocols that allow the developer to interact with these high-level objects, rather than the underlying attributes / relationships of which they are comprised, are generally favored.

Communication with a remote object can be achieved using the following two general techniques:

- By interacting with a local proxy to the object
- By transporting a representation of the object to the client and interacting with it locally.

In practice, the application designer usually uses some combination of these two techniques. To minimize the number of client/server interactions, structurally rich data can be transported to the client. Having complex local data structures at the client allows greater responsiveness and richness on the client side, at the expense of added data transport overhead per client/server interaction. Distributed data consistency issues also become more problematic in the case where updates to the local (client) structures need to be reflected at the server, or where client-side objects become stale due to updates at the server. Nonetheless, the requirements of the application may dictate that (relatively) rich data structures reside locally on the client. This introduces a number of additional issues. The developer must decide on:

- A transportable representation for server objects
- A transport protocol for this representation
- An expressive way of specifying exactly which objects are retrieved/updated on the server, as well as the semantics behind these operations; the more complex the transported data structures, the more important this issue becomes. In other words, given a sufficiently complex graph (or tree) of logically related objects, developers need general facilities to:
 - iterate over objects in the graph
 - specify precise subsets of the graph upon which various lifecycle (create / update / persist / delete) operations will be applied
 - specify the precise semantics of these operations

Together, these issues comprise the general problem of "object graph traversal", which is encountered in many applications.

At first glance, the business objects themselves seem to be an ideal transportable representation. Not only do they encapsulate the structure of the object model, but they also provide appropriate interfaces to this encapsulated data. However, this technique is problematic for two reasons.



Architecturally speaking, this violates the modern multi-tier architecture, where business logic resides on the server and the client(s) implements a thin presentation layer for this logic. Architectures with one or more thin clients accessing services that reside on the application server are more scaleable and re-useable. Direct transport of business objects to the client engenders the additional requirement that the objects are dynamically linkable on both client and server.

Secondly, programmatic constraints may prevent the transport of business objects to the client. For example, business objects are often either statically or dynamically linked to a persistence framework. Clearly it would be ill advised to send all of the framework's classes to the client along with the business objects to which they're linked. Security constraints may also exist, which dictate that proprietary business logic not be available outside of the corporate firewall.

1.2 A Solution

For these reasons, an opportunity exists for the creation of a subsystem that maps business objects to distinct transportable representations and back again (bullet 1, above), *without the need for programmer intervention*. It quickly becomes apparent that the usefulness of such a technology is significantly enhanced by a declarative syntax for expressing precisely which objects are mapped and how these mappings take place (bullet 3, above).

The Java package **com.mtnlake.pl.attributeFactory** is an implementation of such a subsystem. The single public interface to this package is the class **com.mtnlake.pl.attributeFactory.AttributeFactory**. The name "AttributeFactory" is derived from the fact that an AttributeFactory instance implements something similar to the Abstract Factory pattern, which is applicable when the creator of an object or group of objects cannot anticipate the class of objects that it will create (Gamma et al, 1995). We define the term "Attribute Object" as the in-memory transportable representation of a business object (and any specified related objects). Given a server-side business object and a specification of which related objects to create, The AttributeFactory creates a corresponding graph of Attribute Objects whose types are determined at runtime. It also performs the reverse mapping so that any changes to the objects can be persisted.

The default AttributeFactory implementation uses serializable Java objects as the transportable representation. Thus, any transport protocol supporting reliable binary data streams (e.g. TCP) is supported by this implementation. A Java client and server are assumed in the default implementation. Since the transportable representation and the transport protocol normally go hand in hand, a separate implementation (or subclass) of AttributeFactory must be created for each supported transport protocol / data representation (not a difficult task). For example, CORBA could be used for transport and a subclass of AttributeFactory could provide the necessary mapping protocol between business objects and CORBA structs. The emergence of XML as a standard for platform-independent data exchange makes it another candidate for the transportable representation.

Throughout this discussion, we assume the existence of a persistence framework to which the business objects are linked. The framework is responsible for the business object lifecycle, including creation, update, deletion, and storage. It is also responsible for the assignment of unique keys or object identifiers (OIDs) to business objects as they are created.

2 System Overview

2.1 Placement

Figure 1 shows the location of the AttributeFactory subsystem in a distributed application. Note that the factory performs data conversions in both directions (transportable representation < -- > business object).

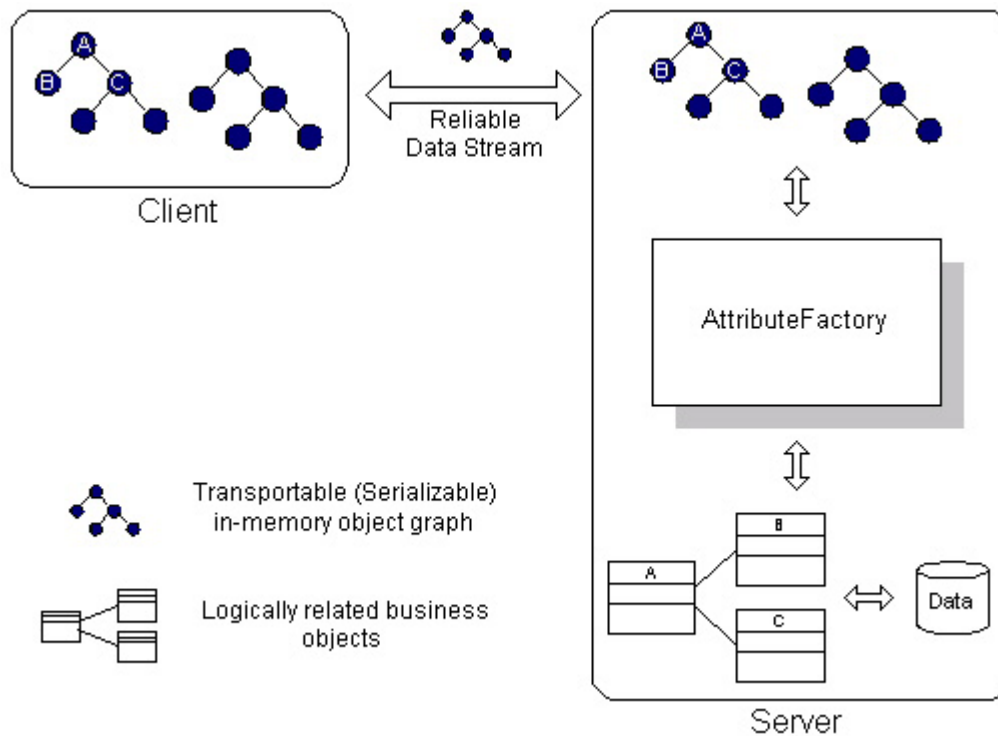


Figure 1 - Placement of the AttributeFactory subsystem within a typical client/server application

The AttributeFactory uses the Java reflection API to both discover object properties and to perform the conversion between Attribute Objects and business objects. This part of the implementation is easily replaceable, such that other discovery mechanisms (e.g. Introspection using BeanInfo classes or some other manual method) can be substituted. In the default implementation, an Attribute Object contains a public field for each business object attribute and for each related object reference. We assume the existence of accessor methods on business objects for each of their attributes and related objects. The class definitions for all of these objects are generated from a common schema specification.

2.2 Sample Mapping

For example, consider a one-to-many relationship between the business objects Customer and User. A Customer stores one private attribute ("name") in memory, and can retrieve its related list of Users (presumably through some interface to the data store) when the accessor getUsers() is called. A User

stores two private attributes (firstName and lastName) and can retrieve its related Customer object through getCustomer(). The corresponding CustomerAttributes object contains two public attributes: a name attribute (corresponding to the Customer name), and a users attribute (corresponding to the Customer's related list of users). During a "read" operation, the factory first discovers which properties are attributes and which are relationships (using reflection), and then fills in the appropriate fields in the AttributeObject using information retrieved from the business object. The reverse process occurs during a "write". Write operations presumably affect the data store.

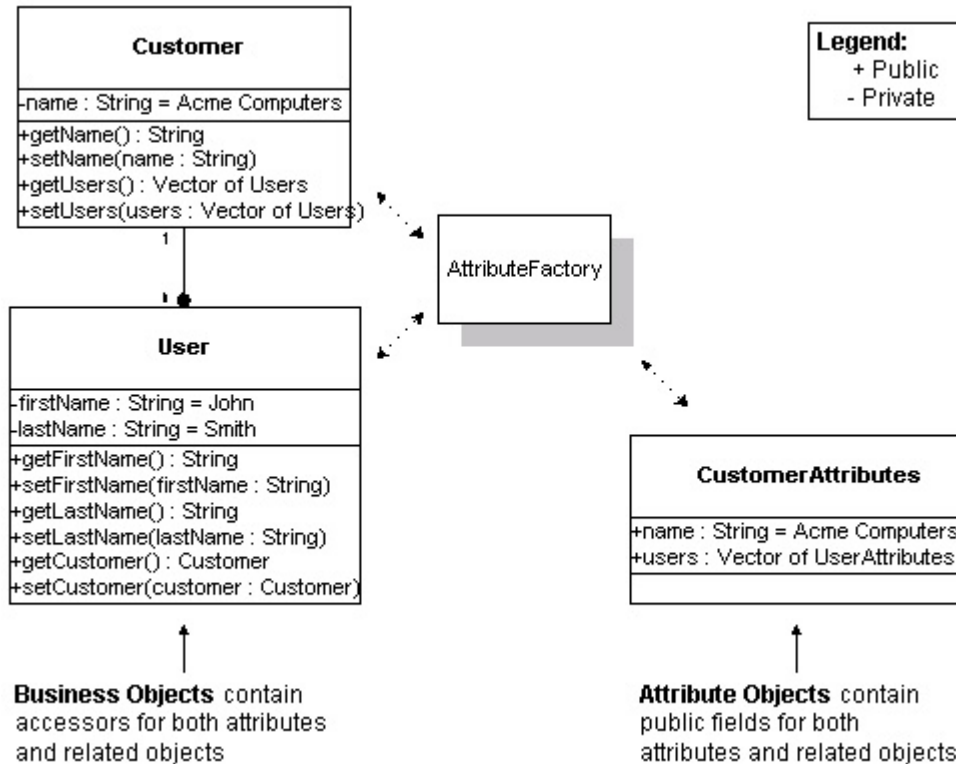


Figure 2 - Sample mapping between business objects and attribute objects

The question now becomes "how does the programmer specify which attributes and relationships he/she is interested in?" We define the term "filter string" to represent such a specification. Filter strings can theoretically be provided for both attributes and relationships, but in most situations we found that the attributes of an object are generally updated all at once when an object is being persisted – and usually into a single row in a relational table. In many cases, there is no significant performance gain in updating a single attribute in a table versus the entire row of attributes. In addition, requiring a filter string specification for each attribute of each object would be an onerous task for the developer. Thus, the filter strings used in this system only specify the role names of relationships; all object attributes encountered in the traversal of the relationship graph are translated.



As an example, consider the situation where the developer wishes to retrieve the Customer object along with that customer's list of users, modify the data at the client, and then persist it. The syntax used is as follows:

```
Customer customer = <code to retrieve the customer business
                    object>;
CustomerAttributes customerAttr =
    factory.select(customer, "users");

sendToClient(customerAttr);
.
.
<modify customer object & related user objects at client>
.
.
sendToServer(customerAttr);
factory.update(customerAttr, "users");
```

The string "users", above, is the filter string. It is taken from the "users" role name of the customer/user relationship.

Suppose each user was related to a list of read/write permission objects, each of which was related to a portfolio object. The code to retrieve, modify, and update all of these objects together would look exactly the same as above, except that the filter string would be "users.permissions.portfolio" instead of simply "users".

A more detailed discussion of **Filter Strings** can be found in Section 3.

2.3 Architectural Impact

It should be noted that the chosen discovery/ mapping mechanism (in our case, reflection) should not in any way impact the business object model, aside from imposing naming conventions on any property accessors the developer wishes to expose (by prepending "get" or "set" prefixes as per the JavaBeans specification). This is not problematic: if the developer does not want to expose a public accessor for a business object attribute at the server, then he/she would almost certainly not want the same attribute exposed within the Attribute Object at the client, obviating the need for its discovery using reflection¹.

Our approach does engender some subtle changes in the way object properties are accessed at the client. In this implementation, Attribute Object properties are public fields that are accessed directly to retrieve/update information in memory. The rationale behind this approach stems from three implementation-dependent criteria:

- Attribute Objects only expose publicly accessible business object data, and thus require no access-level protection.

¹ That said, there are mechanisms within Java 2 for a program using reflection to circumvent access level verification by the VM. This involves signing a jar file containing the server code, and associating a policy file which grants "reflectPermission>>suppressAccessChecks". This is what a beanbox application might do (in the absence of an associated BeanInfo class) to tool a bean containing private attributes.



- In our application, access to Attribute Object properties is not multi-threaded, and thus does not require the monitor lock protection that might be provided with accessor methods.
- Attribute Objects do not expose business logic, except by providing a key (the OID field) from which the associated server-side business object can be instantiated and invoked. This supports an n-tier architecture where business logic resides only at the server. However, instead of invoking this logic using a remote protocol like RMI, it is invoked by marshalling a generic request containing both the attribute object instance and the operation name, and then sending this request to the server for processing.

Since all three of these criteria were satisfied in our application, there was no real reason to define both public accessors and private members in Attribute Objects. However, the implementation does not depend on the fact that the properties are public. Indeed, the developer could make them private and include accessor methods if he/she so desired. These accessors might also be provided simply to mimic the way that data is accessed from the business object on the server.

3 Filter Strings

3.1 Requirements

For the rest of the examples in this article, consider the object model sample in **Figure 3**, containing objects one might find in a database of portfolios and financial instruments.

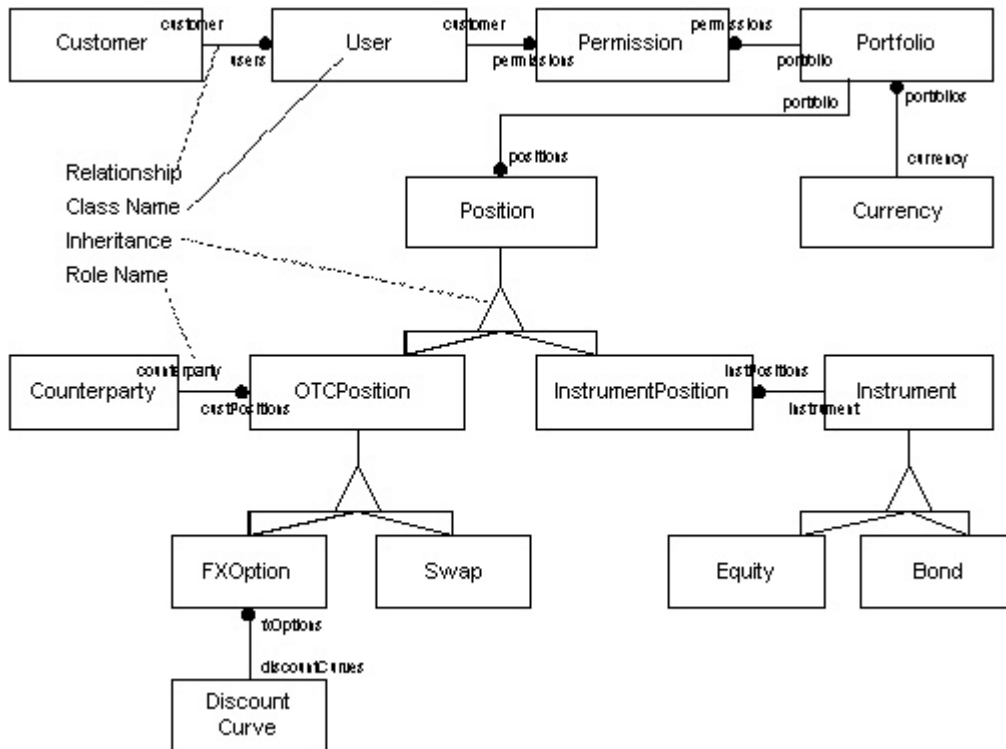


Figure 3 – Sample financial object model

As we will see, the requirements for filter strings quickly become more elaborate than might have been indicated previously. For example, from the *Portfolio* node, how would the developer specify both the *positions* and *currency* relationships in the same filter string? Furthermore, how does one conditionally traverse the *counterparty* relationship from the *Position* node without knowledge of the runtime class of the retrieved positions? (e.g. "positions.counterparty" would be invalid for all retrieved *InstrumentPositions*). Finally, what are the exact semantics with respect to *write* operations? For example, if we invoke:

```
factory.update(customerAttr, "users.permissions.portfolio");
```

Should the existing portfolios in the database be updated or replaced, or should any newly provided ones simply be added? If replaced, should the old ones be deleted? To complicate matters further, the desired semantics are usually different for each node in the filter string (e.g. the developer may wish to create new users and permissions, but reference existing related portfolios!). The developer needs more tools to specify these semantics.



For these reasons, the filter strings used by the AttributeFactory have evolved into a kind of shorthand declarative object query language with a simple associated **Grammar**. As a convenience, the developer is also given the ability to embed arbitrary query predicates within the filter string. These (optional) predicate portions of the filter are applied to the class of objects identified by the role name within the filter.

A more detailed discussion of these capabilities is given below.

3.2 Grammar

3.2.1 EBNF specification

EBNF symbols:

```
{ }    zero or more occurrences
[ ]    optional (zero or one) occurrence
( )    grouping (precedence)
```

The following grammar specifies syntactically valid relationship filters:

Tokens:

```
'.'    node separator (indicates node traversal branch)
'{'    begin subclass grouping
'}'    end subclass grouping
'|'    subclass specification separator
'('    begin multiple traversal grouping
')'    end multiple traversal grouping
'&'    multiple traversal node separator
'!'    indicates "delete related object(s)"
'%'    indicates "replacement" - unlink whatever objects are present in
the DB during "write" and replace with supplied objects. NOTE, if "!" is
used in conjunction with "%", the supplied objects must NOT contain any of
the related objects being unlinked/deleted.
'~'    force "create new" - good for "copy" operations.
IDENTIFIER  identifier token - any valid Java Identifier
PREDICATE   sql predicate token
EOF        End of file token
```

Productions:

```
FilterString      ::= topNodeRelationships

topNodeRelationships ::=
    [ ( subclassSpecifics {'&' identRelationships} ) |
      ( identRelationships {'&' identRelationships} ) ] EOF

nodeRelationships ::=
    [ subclassSpecifics ] [ '.' immediateRelationships ]

subclassSpecifics ::=
    '{' subclassRelationships {'|' subclassRelationships}
    '}'

subclassRelationships ::= className nodeRelationships;
```



```

immediateRelationships ::=
    identRelationships |
    '(' identRelationships {'&' identRelationships} ')'

identRelationships ::=
    ['!'] ['%'] ['~'] roleName [PREDICATE]
    nodeRelationships

className                ::= IDENTIFIER

roleName                 ::= IDENTIFIER

```

3.2.2 Sample Filter Strings

The language engendered by this grammar gives the developer a rich declarative syntax for specifying object traversal semantics during database read/write operations.

To specify the traversal of multiple relationships per node, the **(,)**, and **&** symbols are used. Below, the developer specifies the traversal of both the position and currency relationships of the portfolio class:

```
"portfolio.(positions & currency)"
```

To specify the conditional traversal of relationships based on runtime type, the subclass grouping separators (**{ , }**, and **|**) are used. Below, the *counterparty* role is traversed for all *OverTheCounter (OTC)* positions and the *instrument* role is traversed for all *InstrumentPositions*.

```
"portfolio.positions
{
    OTCPosition.counterparty |
    InstrumentPosition.instrument
}"
```

Arbitrary predicate strings can be used to further constrain the set of retrieved/updated objects, in the same way they would be constrained within the WHERE clause of a SQL statement. The predicate is delimited by the **[** and **]** characters, and can be placed at any identifier node within the filter. The AttributeFactory does not itself process this part of the string: a separate, query engine is invoked instead (in the default implementation, this is the job of the persistence framework). As such, any language (such as SQL, used in the default implementation) can be used to specify the predicate. In the example below, the filter predicate specifies that only those positions whose *amount* attribute is greater than 35.7 should be retrieved.

```
"portfolio.positions[amount > 35.7]"
```

The **!**, **%**, and **~** operators allow the developer to precisely dictate the semantics of *update* operations. The strength of this approach is its flexibility: operators can be *independently applied at each node in which they occur*. Operators are placed immediately before the relationship identifier in a relationship filter string. A discussion of these three operators can be found in the subsections below.

The semantics of *update* operations additionally depend on two related variables. These are:

1. the multiplicity of the relationship. Traversal toward the *one* side of a relationship causes the operation to be applied to the related object. Traversal toward the *many* side of a relationship applies the operation to each of the related objects.



2. the presence or absence of a database object identifier (OID) within the attribute object. The OID is a unique key that identifies each object in the persistent store. An attribute object in memory may or may not have a valid OID. During any *update* operation, objects with null OIDs are *created* in the database. Objects with valid OIDs are *updated* in the database². The sole exception to this rule is when the **!** operator occurs by itself. See Section **3.2.2.2** for details.

3.2.2.1 No operator

When an operator is not present during an *update* operation, then supplied objects with null OIDs are created (and linked) in the database. When they have valid OIDs, the database objects are updated with data from the position. In this way, a developer can both edit existing positions and create new ones with a single operation. For example, to create/update positions in a portfolio, the developer supplies the appropriate PortfolioAttributes object, whose *positions* member references the Vector of PositionAttributes objects to be created/updated. He/she then invokes

```
factory.update(portfolioAttr, "positions");
```

3.2.2.2 The deletion ("!") operator

When used by itself during an *update* operation, the presence of the deletion operator indicates that all of the supplied objects should be deleted. If any of the supplied objects have null OIDs, an *InvalidRequestException* is thrown.

For example, to delete some of the positions in a portfolio, the developer supplies a PortfolioAttributes object that references a Vector of PositionAttributes objects to be deleted, invoking

```
factory.update(portfolioAttr, "!positions");
```

to delete the positions.

3.2.2.3 The replacement ("%") operator

When used by itself during an *update* operation, the presence of the replacement operator indicates that the supplied objects should replace the currently related objects in the database. The semantics are the same as if no operator was used, except for the fact that any objects in the database that aren't represented in the supplied vector of attribute objects are unlinked (but not deleted) in the database.

```
factory.update(portfolioAttr, "%positions");
```

If the deletion operator is used in conjunction with the replacement operator, all *replaced* (unlinked) objects are also deleted from the database. When the replaced objects are no longer required, this form is used.

```
factory.update(portfolioAttr, "!%positions");
```

² The assignment of object identifiers to new objects is the responsibility of the persistence framework, not the AttributeFactory



The supplied objects may have null OIDs (indeed, in many situations they will, since during replacement old objects are often replaced with new objects).

3.2.2.4 The 'create new' (“~”) operator

This operator allows the developer to force the factory to create a new object on "update" operations. It does this by simply nullifying the object id before the write occurs (remember that when no operator is used, a null OID causes the factory to create a new object in the database). This is useful in duplication operations, during which some objects are created new and some are simply referenced.

Suppose that the user wishes to duplicate a *portfolio* and its contained *positions* and *discountCurves*, but wishes the new positions to share references to existing *counterparties* and *instruments* with the old positions. The user would supply the appropriate attribute object graph rooted at the portfolio node, along with the filter string as follows:

```
String portfolioDupFilter =
    "~positions
    {
        OTCPosition
        {
            FXOption.~discountCurve
        }.counterparty |
        InstrumentPosition.instrument
    }";

factory.update(portfolioAttr, portfolioDupFilter);
```

The tildes indicate that positions and discountCurves should be created anew, whereas all other objects in the database should simply be referenced.

Operator	Operation	Action
<none>	<i>select()</i>	For each business object, create the corresponding attribute object in memory
<none>	<i>update()</i>	For each attribute object, create the corresponding business object (causes persistent store to be updated). If the OID is valid, update the existing object in the database. If the OID is null, create a new object in the database.
!	<i>update()</i>	For each supplied attribute object with a valid OID, delete the object in the database.
%	<i>update()</i>	Unlink all related objects in the database. If "!" is present, also delete these objects. Replace the unlinked/deleted objects with the supplied objects. Supplying an empty vector simply unlinks the current related set of objects.
~	<i>update()</i>	For each attribute object, force the creation of a new business object (causes persistent store to be updated).

Table 1 – Summary of AttributeFactory filter string operator semantics

4 Conclusion

AttributeFactory is not meant to be used instead of middleware that enables the use of local object proxies (e.g. CORBA, RMI, DCOM). It is useful when data needs to be transported by value between client and server. To the extent that the object bus supports pass-by-value semantics, AttributeFactory can be added as a complementary technology.

The query language specified by the filter string grammar presented in this article affords the developer a convenient syntax for expressing both *object graph traversal* and persistent object *update semantics*. As an analogy, strings in the language are akin to flexible object graph “cookie cutters”, which not only determine the shape of the cookie, but also imprint suggestions on how the cookie should be eaten. The AttributeFactory employs this cookie cutter in customizing database access, and in preparing objects for transport by value. In this way, the language is used in the preparation and persistence of “pruned” object graphs.

However, the language is potentially useful in more general contexts than might be indicated by this alone. Indeed, any instance of the class of operations that result from the ability to express object graph traversal is a potential new use for the language. Such new behaviors would result from the replacement of the “client” of the parse tree traversal (in this discussion, AttributeFactory is the default “client” of the traversal). Some novel behaviors might be:

- writing select parts of the object graph out to a file / stream
- limiting public access to specific (e.g. public) parts of the graph
- distinguishing/specifying the unique “treatment” of objects in the graph on a node-by-node basis; new ‘operators’ could be added to the language to support novel treatments.

AttributeFactory facilitates common operations that are important in the development of many distributed applications, particularly those that require the transport of rich data structures between client and server. It was initially developed for a secure, multi-tier Java financial risk management / assessment application called **Risk Direct**, and plays a central role in the application server of this system. Risk Direct is owned and operated by [Standard and Poor’s](#) and [Algorithmics](#), and can be accessed at <http://www.riskdirect.com>.

Jeremy Chan is a Principal and Technical Architect at The Jonah Group, Ltd. He can be reached at jeremy.chan@jonahgroup.com.

References

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Longman Inc., Reading, MA, 1995.2