Data Access and Integration Layer

An Integration Oriented Data Access Layer

2nd version

By: Capt. Gal Kahana

Table of Contents:

1. Introduction 3
2. A DAL 4
3. XML 4
4. The Problem 5
5. The Solution - DAIL 7
6. Transformers 7
7. AbstractDAIL class 8
7.1 retrieve method 8
7.2 modify method 9
7.3 getDataSource method 10
8. IDAILDataSource interface 10
8.1 retrieve method 10
8.2 modify method 12
9. ITransformer interface 12
9.1 transform method 12
10. The Transformation 12
10.1 A Track template 13
10.2 Track Parameters 14
10.2.1 "intersect" parameter 14
10.2.2 "paramRetrieve" parameter 15
10.2.3 "dontMerge" parameter 17
10.2.4 "pipe" parameter 17
11. Duplicate Fields 18
12. Inflating Fields 20
13. Some final words on how to implement a Data Source driver 22
14. Error handling 22
15. Summery 23

1. Introduction

An IT application built on a 3-tier methodology has 3 or more layers, each responsible for a different part of the application functionality.
The classic 3-tier implementation uses 3 main layers:

1. A Client layer. This is where the user can view data and push buttons to view it or change it. In MS DNA/.Net this is normally the Explorer web browser. The language of development is ASP/HTML.

2. A business logic layer. This is a layer of objects, where most of them symbol entities in the application (for example: an employee, a package etc.) and accept requests - data acquisition or manipulation. The requests are transformed to actual data storage manipulation commands and moved to the next layer. In MS DNA/.Net this is achieved by creating COM components or .Net classes in either VB/VC/C# or any other language and placing them on an application server (which is usually….the web server itself).

3. A data storage layer. This is a layer that holds the actual data and allows manipulation on it. A database server is such a layer. In MS DNA/.Net one may choose to work with many types of Databases via ADO interface, ODBC interface, or other database specific interfaces using SQL as the description language for queries.

This essay introduces a 4th layer - DAIL, Data Access and integration layer - that should ease the conversation between the 2nd and 3rd layers. NO, this is not a DAL (for those of you who know what a DAL is). Actually it is also a DAL, but much more.
First, there will be a display of the common communication methods between tiers in a DNA/.Net application. Then the problem will be presented, and then the solution (and of course an explanation of the library this is all about).

A note for the 2nd version:
This version uses a new concept - a transformer. The transformer is used to transform the requests to raw data that data storages can use. The 1st version forced the use of XSL files for transformation, which denied the ability to cache a transformation and required IO. It also enables he user to create a less sophisticated method of transformation, that is cheaper then XSL, for certain usages.

2. A DAL

The data storage layer is normally a database server, but it may also be a web-service placed in another site, or a legacy mainframe application to which the communication is via a queue or even an active directory implementation.
What is required in such an environment is a tool that will take care of creating the initial interface with those data storages, transfer the requests and return the responses - Since this is normally a repeating code.
For this there exists the DAL. The DAL (data access layer) is an object that is an object that handles all call to different types of data sources in an application, sometimes exposing a single interface for all of them to ease the usage.
For ADO supporting databases, for example, it can create connections, activate queries and perform calls via commands, specifically fitted to the application - so that the business object does not have to know the exact user and password to the database, or how the insides of a command creation looks like etc.
The DAL is therefore a 4th layer that stands between the business objects and the data storage.
Where I work (and I imagine that this exists in many companies) there's a certain quest for a "generic" method of doing a DAL that will fit to every application. I have a reason to believe that DAIL should answer that.

3. XML

XML is a simple string based method to transfer parameters in a very flexible fashion. This is why many business objects accept XML even if XML communication (via XMLHTTP) is not necessary - this is because only XML has such flexibility and ease of use (via DOM objects, or the new .Net parsers). XML has another value for integration purposes in the form of XSL transformations. XSL can translate XML data into HTML - so that the data fetched from the business layer may be displayed in a nice way. It may also be used to translate XML requests to SQL code when communicating the business layer to the database layer. One more use (that I like the least) is to translate an XML of one entity liking to another XML for communicating with a different entity. This should be avoided by using a standard XML. A little coordination works magnificently on performance.

MS DNA (and .Net) uses ADO/ADO.net and work XML to serialize data for datasets and other objects. Those XMLs represent both scheme and data of a retrieved data set and via methods and classes one may change this XML so that when it is being sent back to the business objects it may automatically use that for submitting modifications to the database. The methodology presented here avoids using the dataset object for those purposes in favor of a simple XML standard. I tend to believe that the client normally

requires only the properties of the entities which it asked for, and not what their types are. It should know this in advance. I also believe that data manipulation commands should only include the manipulation - and not all the data twice (as in the before-after method). Hence when I update an entity I want only to send its ID and the property I changed (and maybe also the original value, but only that). I also believe that the dataset manipulation method is not flexible enough (using commands). An XSL (for example) is a far better method to express the translation from the XML to a good and effective SQL.

Therefore a simple XML request/response standard is used in the methodology - it is presented in the essay "Request-Response standard". You should know it by heart. Just kidding. But it is advised strongly to take a peek at that essay, since this one takes up from there. To be simplistic about it, a request looks something like this:

<Request id="*typeID*" action="*actionID*">
    <Data>
        action parameters for the data
    </Data>
    <Profile>
        definition of the items on which the
        action should take place
    </Profile>
</Request>

The "id" and "action" property in the "Request" tag define what the entity type is (Person, Pizza reservation etc.) and the action that is being made (data retrieval, entity creation etc.). The Profile tag includes definition of the entities on which the action should be made, and the Data tag includes definition of the parameters for the action.

A response looks like this:

<Response id="*typeID*" action="*actionID*">
    <Item id="*itemID*"> *item data* </Item>
    ….
</Response>

The response is a list of Items. For a retrieve request it will hold the items retrieved, and in each one the data that was requested for each entity. For a modification request it will return an arbitrary value indicating the success status and additional data.

4.The Problem

Building a business object normally involves doing 2 things:

1. How to include permission handling in the code
2. How to modify the XML request to SQL code, communicate with the data storage layer and return it as XML that may be easily read (or bounded).

The permission handling is sometimes handed over to Com+, but Com+ while still being good for authentication is not good enough for applying flexible permissions (being only a Role-Based method, and not ,for example, hierarchy based, or owner based). The PAL (Permission Analysis Layer) component is the part of the presented methodology handling permissions and it is discussed in a different paper.

The second problem is normally handled by having a method that looks at the XML and build an SQL string out of it. The resulted SQL is passed to the database and processed. Someone has to also take care to how to handle commands and how to bring back the data in a neat form. (Of course, you can use an object-based model like ADO.NET which hides some of this, but it's basically has the same functionalities)
In the project I've been recently working on I came up with even more complexities to that problem:

What happens if I have more then a single data storage that deals with the same entity? For example, what if the salary of an employee is handled in one database, while her name is in stored in another data source, and I want to display both in the same table on the GUI?
How do I get both SQLs to run, and merge the result? (One more complexity - in that project I didn't have a simple protocol with one of that databases - I couldn't simply use a dblink, because the handshake with it was non-trivial). Worse - what happens if I want to profile (describe the properties of…) the entities I'm working on by using one of the data sources data but fetch a field from the other data source?

What's more - what if my modification action should run 2 SQLs instead of 1? I need to run those 2 SQLs. So this means 2 calls to the DAL. I'd much prefer a single call for both. Oops, it may even be that the 2 calls run to 2 different data sources - one is a database and the other is web service. Oh my. What a mess. And I have to do all that in my business logic object.

As if that wasn't enough, what happens if I wish to add a property to my entity? I would like to only add the SQL code to fetch that property, and maybe even to avoid compiling. XSL may take care of that.

There seemed to be a strong feeling that all of those problems have common solution, and should not be placed inside the business object code, but in the DAL or another object. The BO (business object) should only be aware to the final result of a request from the DB (not including a transformation to a standard XML), and not how it came up to be. This is the job of the DAL.

There exists an approach that says - let the DAL do the conversions. Just give it the XML and it will do all the conversion to SQL (or other language - even a parameters call for a web service) in its own code. This is good for extremely small applications or where the conversion is algorithmically trivial - but not for large applications that handle data of many different entities.

You probably have many other problems in your project that were not drawn here for this specific area of data accessing in the business objects. All in all, data accessing is a pain doing in the BO, and should be centralized - but yet it should allow flexible building of the data manipulation/retrieval code.

5. The Solution - DAIL

DAIL solves most of those problems. The component completely hides all data access from the business object.

The philosophy behind DAIL is that each application defines entities. Each entity is represented by one or more data representation which resides in one or more data sources, varying in type and access. In order to integrate data source access and generalize it, DAIL users the Request/Response standard in the high-level. As for the specific data source access DAIL defines data source drivers that know how to accept raw data (like SQL) and respond with a standard response. DAIL itself accepts requests in the request standard and translates them to action data using transformers. This way, outside the DAIL, everybody deal with standard request and response.

To perform that magic, a DAIL library exists. The library main class is an abstract class, having 2 methods: retrieve and modify.
The reason it is abstract, is because a protected method is required to be implemented for DAIL to be used. The method returns a Data Source Driver for a given label. A data source driver is a class that handles the access to a single data source. The reason for its abstractness is because different applications have different data source and a library can't tell.
A data source class implements retrieve and modify methods, each taking a string in a data source specific format (SQL for example) and returning a response as defined by the Request/Response standard. This way, the DAIL can integrate easily responses from different data sources.
Each method of DAIL main class accepts an XML string which is a request string (normally the BO will just hand it the request XML it was given) and a transformer. The transformer translates the XML into "tracks". A track is a call to a single data source. The track includes a label of the data source (DAIL uses this label with the above mentioned method, to get the data source driver), some flags and the data string that should be handed to the data source driver. To make things interesting, tracks can communicate with each other, so that one may use a track response, for another track to work.

6. Transformers

One of the main participants in the DAIL solution is the transformer. A transformer can take a request string and transform it to a series of tracks. The method of doing the transformation may be varied. For example, one can implement a transformer that uses

XSL transformation to create the tracks from the request. In fact, the first version of DAIL, which did not use transformers, used XSL files specifically. A Simpler implementation that may be used with data sources that know how to handle standard requests may simple append a header with the data source label to the request string. Use whichever transformation that is good for you for the specific needs.
DAIL library supplies the 2 above mentioned examples as transformer classes.

The next sections will discuss each of those solution components in great detail, so don't worry if you feel a little lost now.

7. AbstractDAIL class

AbstractDAIL is the main class in the DAIL library. It has 2 public methods and a single protected abstract method. Clients should subclass this class and implement the abstract method. The business objects should call the subclass 2 public methods.

7.1 retrieve method

The retrieve method signature is:

```
public string retrieve(
      string strXML,
      ITransformer transformer,
      string strID,
      string strAction)
```

The retrieve method is used when one wishes to retrieve data about a list of entities (or a single entity). The XML request (in the Request/Response standard) is passed as strXML, the transformer class defining the appropriate transformation for this action is passed as transformer. strID and strAction will be used by the data sources drivers when the responses will be created - they'll be the "action" and the "id" properties values.
The return value is an XML string in the form of a Standard response, which is the result of the query.

The algorithm of the retrieve method is:

- Activate the tranformer on the XML
- Loop the created tracks:
    - Get the data source driver labeled by the track header
    - Call the retrieve method of the data source driver with the track content
    - Save the response
- Merge the responses:
    - For track response j and track response j + 1 produce response j*:
        - Loop the items listed in both responses, for each matching item replace with and item holding both fields
        - If the merge is marked as an "intersection merge" save only the items that indeed are included in both responses. Otherwise save all items

        - j* is a new response holding all saved items
    - j = j*
    - subtract 1 from j and continue
- return the merged result

The algorithm uses the term "matching items". Matching items are "Item" tags in the Response that has the same value in the "id" attribute of them.
There are some more complexities in the algorithm including dealing with fields that appear in both responses, how communication is achieved between tracks etc. This will be explained later.

7.2 modify method

The modify method signature is:

```
public string modify(
      string strXML,
      ITransformer transformer,
      string strID,
      string strAction)
```

The parameters have the same meaning as in retrieve. The difference is that while the retrieve method is used for retrieving data, the modify method is used for performing data manipulation. The returning value is a standard response, and its content is the string returned from the last track. As oppose to the retrieve method which result is completely determined by the data source, the result of the modify method is fixed and looks like this:

<Response id="*$strID*" action="*$strAction*">
    <Item>
        <OK>last *track response*</OK>
        parameters retrieve results
    </Item>
</Response>

Replace $strID and $strAction with the input values of the variables strID and strAction in accordance. Apart from the last track response, the method also returns a collection of parameter retrieves result. "Parameter retrieve" is a notion used for retrieve tracks that retrieve parameters. It is also used in retrieve method, and will be explained later.

The algorithm of the modify method looks like this:

- Activate the transformer on the XML
- Loop the created tracks:
    - Get the data source driver labeled by the track header

    - If this is a parameter retrieve call the retrieve method of the data source driver with the track content, and save the retrieve parameters. Otherwise call the modify method of the data source driver, and save the response string (this may not be a standard response)
- Build the response string and return it

It is suggested that the modify method of the data sources will return the number of modified records - as in the notion of ADO - but it's just a suggestion.

7.3 getDataSource method

The getDataSource method signature is:

```
protected abstract IDAILDataSource getDataSource(
      string strDSLabel)
```

This method should be implemented by the subclass to return an IDAILDataSource class - a data source driver class - that matches the label handed by strDSLabel.
If the method does not recognize the given label it should throw an XBLIP.DAIL.UnknownDataSourceException exception.

8. IDAILDataSource interface

IDAILDataSource interface should be implemented by data source driver classes. The drivers drive a specific application data source, and their purpose is to enable the DAIL to integrate the usage of those data sources. The IDAILDataSource interface has 2 methods, retrieve and modify.

8.1 retrieve method

The retrieve method signature is:

```
string retrieve(
      string strTrackContent,
      string strPipe,
      string strID,
      string strAction)
```

strTrackContent is the raw data that is produced in the track that triggered the call to this data source. It may be an SQL query for a database server; it may be an XML request for another web service, or any other string that will make that data source tick.
The second parameter, strPipe, is used when the track signaled that it wishes to use a pipe. This issue will be explained later.
The expected result of the retrieve method is an XML string in the form of a standard response which id attribute (in the response tag) is strID, and the action attribute is

strAction. strID and strAction will contain valid values only in a retrieve action of AbstractDAL and they are given so that the final response will have those id and action.

Now comes an important part, so read carefully: As explained in the essay on the Request/Response standard it is an important property that the order of the fields in each item in the response will be the same as the order of the fields that appeared in the request "Data" tag. In order to maintain that, each data source driver must return a response in which the fields in each item are ordered in the order given by the data. While this is mainly the job of the transformer (for example, when using SQL all the transformer has to do is to order the "select" fields in the order of the appearance in the XML Data tag) it's important to also note that here, since the class driver is the producer of the response string.
You may say - what happens if there are more then a single data source, and each of them only returns some of the fields for each item/entity. AbstractDAIL takes care of that, just make sure that the fields in the response returned from a single data source are ordered correctly in relation to one another, and the AbstractDAIL will merge the fields in the correct order (it does that by looking at the "Data" tag of the request). The reason for ordering the fields in each data source response, because this way the merge do not have to actually sort - this way its an O(n) algorithm for bringing the ordered result, and not O(nlogn) as for a regular sort.
For example, assume that the following request was issued:

```
<Request id="employee" action="retrieve">
        <Data>
                <Name/>
                <Salary/>
                <LastPaymentDate/>
                <Address/>
        </Data>
        <Profile>
                <Department><Item>R&D</Item></Department>
        </Profile>
</Request>
```

Assume that one data source deals with employee personal data, and the other handles salaries data. In this case the resulting 2 responses should look like this:

```
<Response id="employee" action="retrieve">
        <Item id="1234">
                <Name>joe</Name>
                <Address>not far from here</Address>
        </Item>
        ... More items
</Response>

<Response id="employee" action="retrieve">
```

```
        <Item id="1234">
                <Salary>4000</Salary>
                <LastPaymentDate>3/4/2002</LastPaymentDate>
        </Item>
        ... More items
</Response>
```

Notice that "Name" comes before "Address" in the first response - as in the request - and that "Salary" comes before "LastPaymentDate". Were the order in the request different, the order of the fields in the resulting responses would have been different accordingly.

8.2 modify method

The modify method signature is:

```
string modify(
      string strTrackContent)
```

The modify method executes the track content (handed by strTrackContent) and return a string result. The string may be arbitrary. As explained before, it is suggested to return the number of modified entities.

9.ITransformer interface

ITransformer interface handles transformation of a string (request in the case of DAIL) to another string. The interface should implement a single method - transform. DAIL library supplies (under XBLIP.Transformers) 2 implementations, one uses XSL as means of transformation, and the other one appends a header to the given request.

9.1 transform method

The transform method signature is:

```
string transform(string toTransform)
```

Simply return a string, which is a function of the given string.

10. The Transformation

This section is probably the most important. The transformer class should produce a string that includes either 0 or more tracks descriptions, where each track is a single data source action. More then simply labeling the data source and giving the raw data to send to the data source driver flags may also be defined to describe the integration of tracks, for example how to pass the result of a single track, for another track to use in its query.

10.1 A Track template

A single track string looks like this:

QUERY_SEPERATOR-dbLabel-*parameters*
Track raw data

That’s it. Simple, right?
The "QUERY_SEPERATOR" label that starts the track is used as the track header (plus it separates the tracks - hence the name). After it come a hyphen, and later the data source label. To signal the label end comes another hyphen and then some optional parameters. The flags are predefined by DAIL and will be explained soon.
After the flags comes the raw data. This data is passed as is to the data source driver, handling this track.

For example, the following track is a simple date selection from my_database data source:

QUERY_SEPERATOR-my_database-
select sysdate from dual

The data source driver should take this request, execute it, and return its response in a response standard. Pay attention that although the SQL command will return a data set it is expected that my_database will transform that dataset to a standard response

The next example displays 2 tracks; one retrieves the employee name from 1 table, and the other, the employee salary

QUERY_SEPERATOR-persons_database-
select id , name from persons where department = 'R&D'

QUERY_SEPERATOR-salaries_database-
select id , salary from salaries where department = 'R&D'

The above example will result in those 2 tracks to execute, and if the drivers will return a response string where the "id" field is set to the id value returned from the query, we should be able to get a single response with a list of persons, and for each we will get the name and the salary.

Modification queries look the same:

QUERY_SEPERATOR-persons_database-
update persons set name ="joe"
where id="1234" and name = "joee"

The example displays a query where the name of "joee" who is identified by the id = 1234 is updated to "joe"

10.2 Track Parameters

Pre-defined parameters enable DAIL to be more then just - execute all those tracks and return - but to give actual value.

The pre-defined parameters are (I'll explain what they mean later):

1. intersect
2. paramRetrieve
3. pipe
4. dontMerge

In order to use a parameter append its name to the track header. Prefix it with an underscore. For example, to use the "intersect" parameter, do the following:

QUERY_SEPERATOR-persons_database-**_intersect**
select id , name from persons where department = 'R&D'

One may use more then 1 parameter by concatenating the parameters. For example to use both "dontMerge" and "paramRetrieve":

QUERY_SEPERATOR-persons_database-**_dontMerge_paramRetrieve**
select id , name from persons where department = 'R&D'

10.2.1 "intersect" parameter

"intersect" parameter may be used in a track that is used in a retrieve method call where a merge exists.
The merge algorithm goes from the last response to the first response, and merges each time 2 responses to a single response, which in turn is merged with the next response and so forth until the first response is merged to the accumulated response to produce the final response. The method of merging 2 responses is by creating a new response holding both responses items. For each item that is found on both responses the item in the new response will hold both source items fields in the order instructed by the request.
By applying the "intersect" parameter, one makes sure that only those items that exist on both responses will be in the merged response.
Using this method one may create a transformation that allows for retrieving fields from 2 data sources and in the same time retrieving only the items that answer to a profile defined by parameters found on both data sources.
For example: Assume that you wish to retrieve the names and salaries (as before) of the employees, but only those that their salary is more then 2000 and are of the R&D department (let's assume that the department property exists only in persons table, and

that salary is only on salaries table). Again - join is not used, since no dblink exists, so one has to use DAIL merge:

QUERY_SEPERATOR-persons_database-_intersect
select id , name from persons where department = 'R&D'

QUERY_SEPERATOR-salaries_database-
select id , salary from salaries where salary > 2000

Were we not using the "intersect" parameter we would get a list of all employees but only for the R&D people we would get the name, and only for the people whose salary is larger then 2000 would we get the salary.

10.2.2 "paramRetrieve" parameter

The "paramRetrieve" parameter signals a retrieve query that retrieves parameters that may be used in other tracks. The "paramRetrieve" tracks may appear both in modify and retrieve queries, to enable parameter passing.
A track that is a "paramRetrieve" query has to retrieve a single "Item" in its response (not more!). The query fields are then saved and from that track on, they may be used. Each parameter that was retrieved by a "paramRetrieve" track has an ordinal starting from 0 and going up. In order to refer to a parameter a track places inside it contents (where it wants the parameter value to appear) the following string template:
PARAM_REF_*parameterIndex*_

Before the execution of a certain track, the track parser replaces all instances of the above template with the matching parameters to achieve the effect.

For example, the next example retrieves the salary of all persons that are in the same department as "joe" (Assuming that only 1 like this exists):

QUERY_SEPERATOR-persons_database-_paramRetrieve
select id,department from persons where name="joe"

QUERY_SEPERATOR-salaries_database-
select id , salary from salaries where department = PARAM_REF_0_

The first track is assumed to return a response in which the only field for each item (the only item) is the department. This means that only the department parameter is created. The second track refers to this parameter via the label "PARAM_REF_0_", to retrieve all the salaries of the persons in the department of joe. (The result of this response will also include joe, but he will have also the "department" property, unlike everyone else - this is the result of the merge).

Normally paramRetrieve tracks should be ignored in the merge. Use the dontMerge parameter that will be explained later, for that.

An interesting side effect to the usage of parameters tracks is in the modify method. When running tracks in the modify method and using the parameters track, apart from being able to use those parameters in the tracks, they are also returned to the user.
A response for a call to modify in the AbstractDAIL subclass, when using parameters will look like this:

```
<Response id="itemID" action="actionID">
        <Item>
                <OK>response from the last modify track</OK>
                <Params>
                        <Param id="0">param 0 value</Param>
                        <Param id="1">param 1 value</Param>
                        ……
                </Params>
        </Item>
</Response>
```

The parameters are listed in the response. For each parameter there exists a "Param" tag, which id is the ordinal given to it. The value of the node is the parameter value.
This particular feature is very useful when modification data is required in the client code. For example, normally when a new entity is created and its ID is made by a sequence the client would like to know that sequence, in order to later perform updates or any other action. To achieve that retrieve the sequence in parameter retrieve track, and use the parameter for the track that creates the entity. For example, in order to create a new employee entity, do the following:

```
QUERY_SEPERATOR-persons_database-_paramRetrieve
select persons_sequance.nextVal from dual

QUERY_SEPERATOR-salaries_database-
insert into persons (id,name)
values (PARAM_REF_0_,"john")
```

The first track will retrieve the sequence value, which the 2nd track uses for insertion. The response from this "modify" action will look something like this:

```
<Response id="Person" action="Create">
        <Item>
                <OK>1</OK>
                <Params>
                        <Param id="0">614</Param>
                </Params>
</Response>
```

The OK tag holds the value of 1 - assuming that the data source driver returns the number of modified entities - and the "Param" tag holds the value of the sequence, which is 614.

It is allowed to use more then a single parameter retrieve track in a series of tracks, the ordinals for the parameters will be given in the sequence of appearance. For example the ordinals for the parameters of the 2nd parameter retrieve track will start from 1 more then the last parameter of the 1st parameter retrieve track, and so on.

10.2.3 "dontMerge" parameter

"dontMerge" parameter is used in a retrieve method call tracks. A track that puts that parameter is not merged with the other tracks. This is normally good for parameter retrieve tracks and pipes (which will be explained next).
For example, if we take the example using paramRetreive, and use dontMerge with it…

QUERY_SEPERATOR-persons_database-_paramRetrieve_dontMerge
select id,department from persons where name="joe"

QUERY_SEPERATOR-salaries_database-
select id , salary from salaries where department = PARAM_REF_0_

… The final response will hold only the response of the second track, meaning that "joe" will not have a "department" field, this time.

10.2.4 "pipe" parameter

"pipe" parameter is used in a retrieve method call tracks.
"pipe" allows the usage of a previous track response in the current track. One may use the "pipe" parameters in 2 fashions, one for usage of the response of the track before it, and one for using a response via an index.

The first form looks like this:
QUERY_SEPERATOR-*dbLabel*-_pipe_
raw data

The second form looks like this:
QUERY_SEPERATOR-*dbLabel*-_pipe***n***_
raw data

The n is the number of track which response should be used. n is starting from 0.
The pipe response is given to the data source driver as the 2nd parameter, allowing for the driver to determine what to do with the pipe.
This is very useful when there are 2 data sources, and one of them accepts as a handshake a list of properties, for example - id's. The first track is used for retrieving the id's by a certain profile and the second may use pipe, to get extended data, based on the first track list of id's.

11. Duplicate Fields

"Duplicate fields policy" defines what to do when 2 data sources return responses, and for a certain item (or all, or some) both responses contain the same field name. This may happen when we would like to integrate both values into a single field, or to prioritize them.
For example, assume that 2 data sources contain the phone numbers of the employees: one holds the cell-phones numbers and the other holds their home phone number. We would like to create a field that holds both, for each person. In order to do that simply name the fields in the same name, DAIL will take care of the rest. For example, if the 2 following responses are to be merged:

```
<Response id="employess" action="retrieve">
        <Item id="123">
                <Name>joe</Name>
                <Phones>
                        <Phone>123456</Phone>
                        <Phone>344353</Phone>
                </Phones>
        </Item>
        …. More items
</Response>

<Response id="employess" action="retrieve">
        <Item id="123">
                <Phones>
                        <Phone>456456</Phone>
                        <Phone>234233</Phone>
                </Phones>
        </Item>
        …. More items
</Response>
```

The result coming out of DAIL will look like this:

```
<Response id="employess" action="retrieve">
        <Item id="123">
                <Name>joe</Name>
                <Phones>
                        <Phone>123456</Phone>
                        <Phone>344353</Phone>
                        <Phone>456456</Phone>
                        <Phone>234233</Phone>
                </Phones>
        </Item>
        …. More items
```

</Response>

The result holds the first track phones on top of the second track phones.

This is the default behavior. One may wish instead of appending the contents of the fields, to sum them. To do so, use the attribute "type" on the fields in the track coming before the other. For example the following 2 responses:

```
<Response id="countPersonsA" action="retrieve">
        <Item id="count">
                <Number type="sum">5</Number>
        </Item>
</Response>

<Response id="countPersonsA" action="retrieve">
        <Item id="count">
                <Number>3</Number>
        </Item>
</Response>
```

Produce the following response when merged:

```
<Response id="countPersonsA" action="retrieve">
        <Item id="count">
                <Number>8</Number>
        </Item>
</Response>
```

A more sophisticated usage of this feature is when one data source contains certain data and the other contains modifications on that data.
For example, one data source holds the addresses of the employees as input in their first day of work, and the other data source holds non-nulls for only the addresses that were modified. In this case we would like to merge 2 responses, each coming from another data source, in a fashion that will produce a list of the current addresses for all employees - meaning that for each employee whose address was not modified the original address will be displayed, and for each employee whose address was changed, the new address will be given.
For this there exist priorities. A field that is duplicated may have a "priority" attribute which contains a number. If both fields have content (non-empty) then the value of the field with the higher number is taken. Otherwise the field that has content is taken.
Consider the following 2 responses:

```
<Response id="anItem" action="anAction">
        <Item id="anID">
                <A priority="2">a</A>
                <B priority="1">b</B>
```

```
        </Item>
</Response>

<Response id="anItem" action="anAction">
        <Item id="anID">
                <A priority="1">A</A>
                <B priority="2"></B>
        </Item>
</Response>
```

The resulting response when those 2 merge is:

```
<Response id="anItem" action="anAction">
        <Item id="anID">
                <A priority="2">a</A>
                <B priority="1">b</B>
        </Item>
</Response>
```

The A field takes the value of "a" because 2 is larger then 1.
The B field takes the value of "b", even though it has a lower priority, because the field with the higher priority does not have a value.

12. Inflating Fields

The issue of inflating fields has been discussed in the Request/Response standard essay. "Inflating field" is a key name for fields that are represented in the response by more then a single field. We may use the example of the phones to show that. Assume that instead of returning a single "Phones" field, the response would hold a "Phone" field for each phone a person has. For example:

```
<Response id="employess" action="retrieve">
        <Item id="123">
                <Name>joe</Name>
                <Phone type="home1">123456</Phone>
                <Phone type="home2">344353</Phone>
        </Item>
        …. More items
</Response>
```

Attributes are used to differentiate between the 2 phones, but this is not a must for inflating fields. The only requirement is that the fields of an inflating field, in a response, will be adjacent - this is a result of the requirement for maintaining the order of the fields in the request.
A problem with inflating fields is when 2 responses hold inflating fields with the same name - i.e. duplicate inflating fields. The DAIL should know, when merging those 2

responses, which of the sub-fields of an inflating field to place before the other. In order to do so, one must define (only when an inflating field is spread over more then 1 response!) an "order" property in each of the inflating-fields subfields. This way DAIL will know how to order the subfields in the resulting response. For example, let's return the previous phones example:

```
<Response id="employess" action="retrieve">
        <Item id="123">
                <Name>joe</Name>
                <Phone type="homePhoneA" order="1">123456</Phone>
                <Phone type="homePhoneB" order="3">344353</Phone>
        </Item>
        …. More items
</Response>

<Response id="employess" action="retrieve">
        <Item id="123">
                <Phone type="cellPhoneA" order="2">456456</Phone>
                <Phone type="cellPhoneB" order="4">234233</Phone>
        </Item>
        …. More items
</Response>
```

The resulting response will look like this:

```
<Response id="employess" action="retrieve">
        <Item id="123">
                <Name>joe</Name>
                <Phone type="homePhoneA" order="1">123456</Phone>
                <Phone type="cellPhoneA" order="2">456456</Phone>
                <Phone type="homePhoneB" order="3">344353</Phone>
                <Phone type="cellPhoneB" order="4">234233</Phone>
        </Item>
        …. More items
</Response>
```

Notice how the "order" property set the order of the inflating-field subfields in the resulting response. Also notice that in the original 2 responses the sub fields are ordered relatively - 1 appears before 3 and 2 appears before 4, this is a result of the requirement for maintaining the field order in each separate response.
One may decide to give the same order to some of the sub fields, in this case, the Duplicate Fields policy will be used to determine the result for those sub fields.

13. Some final words on how to implement a Data Source driver

An important aspect of what the client has to do is to create a data source driver implementation for each data source used in the application. How to implement them is determined by the type of the data source, and the usage of it in the application.
For example, in order to build a data source for a simple ADO data base one should do the following:

retrieve method:

- have a contract with the transformation that they produce SQL
- take the SQL and execute it on a command
- transform the result to a standard response, using XSL or whatever other method
- return the response

modify method:

- have a contract with the transformation that they produce SQL
- take the SQL and execute it on a command
- return the number of modified entities

In order to use bind variables queries or stored procedure a more sophisticated implementation should be used to describe the query elements: parameters of the command, returning value etc. It is advised to have an XML standard defined for this sort of query, and that a certain label signal that this is such a query. This way the data source driver will be able to build an appropriate command and dispatch the stored procedure or bind variables command.

The DAIL library gives some basic implementations of a few data source, one may use them directly (or subclass) for ones own usage.

14. Error handling

The DAIL library implementation assumes that the inputs are valid, so it does not validate the XSL or XML. Errors resulting from that are for the user to fix.

As mentioned before, when an unfamiliar data source label is encountered a

XBLIP.DAIL.UnknownDataSourceException should be thrown by the getDataSource method, handing the label of the unfamiliar data source.

When an exception occurs within the data source, DAIL throws an XBLIP.DAIL.DataSourceActionFailedException exception and in it the exception that originated that exception. Handling of that should be made in a higher level. This approach originates from a tendency to determine a unit method for handling data exceptions in projects.

15. Summery

This essay has introduced a methodology (and a library) for handling data access in a flexible and centralized fashion.
The gains from using such a methodology are many, some listed below:

- The usage of a standard Request/Response allows for a full integration of multiple data sources

- Transformation (for example via XSL) allows for a very flexible method of building queries efficiently from XML requests. A benefit of using XSL files is that when modified no one has to recompile, so no shut down of the application is required.

- Apart from the transformation class, the Business object is not aware of the data source type to use the data. More then that, the Business object never knows how many data sources participated in the handling of the request

The standard Request/Response allows also for a generalized implementation of permissions handling, as described in the "PAL component for effective permission handling" essay. It also allows for quite interesting dynamic GUI building methods.
That's IT.