

Repositories for Data Management A Data Access Pattern Language

White Paper produced by
Mphasis Corporation

April 2005
Version 1.0

© Mphasis. All rights reserved

Document Contact Information

Name	Bert Hooyman, Senior Architect Mphasis UK Ltd. Southbank House Black Prince Road London SE1 7SJ, United Kingdom
Email	bert.hooyman@mphasis.com
Business Phone	+31 618 780 559
Business Fax	+31 302 967 801

Table 1 **Contact Details**

Note From The Author

The data access pattern described in this white paper is the result of exposure to numerous recurrences of the same data management requirements. We believe the pattern language covers all the essentials of data management. Having said that, we find that reality is always more complex than we are led to believe. Many of these 'reality checks' have found their way in this white paper. We invite all readers to actively contribute additional reality checks and contradictory findings, with the purpose of further enhancing the quality of this work.

Acknowledgements

I have enjoyed the valuable feedback from the following Mphasis engineers: Niranjan Sathe and Mohan Borwankar. I thank them both for helping put this white paper together.

Contents

Document Contact Information	i
Note From The Author	i
Acknowledgements	i
1 Executive Summary	1
Audience	1
Objectives	1
Prerequisites	2
Further Reading	2
2 Access Patterns for Data Management	3
2.1 Principal Features of a Repository	3
3 Repository Requirements Analysis	5
3.1 Create, Update, Delete Capabilities	5
3.2 Criteria-Based Queries	5
3.3 Filtering	5
3.4 Attribute Population	6
3.5 Pagination & Scrolling	6
3.6 Sorting	8
3.7 Screen Population	8
3.8 Metadata	9
3.9 Separating Responsibilities	9
4 Solution Outline	10
4.1 Create, Update & Delete	10
4.2 Data Retrieval	10
4.3 Screen Population	11
4.4 Metadata	11
5 Manage a Whatever	12
5.1 How Repository Access aligns with the User Request Pattern	12
6 Extensions to Repository	15
6.1 Soft Delete	15
6.2 Effectivity or Temporal Property	15
6.3 Auditing	15
7 Conclusions	16
Appendix A - References	17
Appendix B - Value Objects for Repository Access	18

1 Executive Summary

This MphasiS white paper addresses the management of data in web-enabled business applications. The scope of this research is all those parts of a web-enabled application that are light on behavior and heavy on data. This includes, but is not limited to, screen population, data validation and entitlement management. In many cases, the primary business function of an application is indeed little more than data management. An example of this is the web-enabling of corporate ERP solutions, where the business process automation, the workflows and the resource planning activities are centralized in the ERP system but data entry and query is pushed out to the corporate intranet.

In an earlier whitepaper (see ref. [1]), a requirements analysis and *generally applicable use cases* were presented. These use cases together describe a pattern which is known as "Manage a Whatever". In a subsequent whitepaper (see ref. [2]), *canonical web pages* and a *page flow pattern* have been identified that apply to Manage a Whatever. A recurring pattern of *user requests* emerges from the web page flows. Following the user request pattern, the processing of these user requests is further analyzed in ref. [3] with the purpose of identifying a *generic design for request processors* in the data management domain.

In this white paper, the pattern language is extended with an analysis of the patterns of data access that occur in data management applications. These patterns are based on the request processing patterns, but they are also applicable in different types of data management designs such as a service-provider approach.

The resulting data access design is well suited for building service providers for business data access.

Audience

This white paper is written for MphasiS clients. It describes a pattern language for accessing composite business domain data. Using the pattern language, MphasiS architects and development leads can roll out streams of web-enabled data management modules for their client projects in an efficient manner.

Objectives

The goal of this white paper is to bring a practical implementation of reusable data management code closer to reality. The repository design that is elaborated here is not directly tied to any technology, although it is clearly based on an object-oriented style of software design.

Once a project has selected its target technology, a baseline repository implementation can serve as the starting point for all data access components required in the project. Across projects, it is expected a 'best practice' design would emerge for a repository, including O/R mapping, caching, optimized performance and unified behavior.

Prerequisites

This white paper refers to terminology that was introduced in earlier parts (ref. [1], [2] and [3]). In particular, reference is made to the pattern of uses requests identified in [2] and elaborated in [3].

Further Reading

There are several good publications that discuss data access strategies in general or specific data access patterns in particular. The notion of a Repository is introduced in Fowler's Patterns of Enterprise Application Architecture (ref. [8], pp. 322-327) and in Evans' Domain-Driven Design (ref. [9], pp. 147-159).

New strategies for efficient pagination appear on the web almost every week. The most valuable publications appear in the context of O/R Mapping tools, as pagination is best implemented at that level.

2 Access Patterns for Data Management

There are many ways to encapsulate data access features and extensive literature on the subject is available, in particular reference [10].¹

At a low level, Data Access Objects (DAO) have been introduced to simplify access to relational data from business logic. More sophisticated approaches use Object/Relational mappings (ORM) to bridge the semantic gap between relational data and object-oriented data models.

In other cases, software architects have found that data management applications may not even require an object-oriented approach; their designs expose a row-based data model to the business logic tier or even to the presentation tier. This model has become especially popular in the .NET world.

In the context of the Manage a Whatever pattern, it is worthwhile to spend some upfront effort in the design of a solid data access approach to come up with a truly reusable solution that offers flexibility, extensibility and controllable performance. Hence, the design for a generic data access component is based on the notion of a *Repository*.

The repository pattern is discussed in [8], pp. 322-327 and [9], pp. 147-159.

2.1 Principal Features of a Repository

Access to data via a repository is based on the following preconceptions:

- The repository presents an *object model* of an underlying (relational) persistence model.
- The repository presents an *in-memory view* of all instances of a data entity. The repository may implement lazy fetching as well as data caching strategies for *performance optimization* purposes.
- The managed entity is a *rich data model*; the repository typically encapsulates access to a least 3 to 10 database tables.
- The repository understands the concept of *enumerated attributes* and is capable of providing all relevant data for screen population.
- All filtering and searching of entities is based on *criteria-based matching*. Using Criteria objects, the repository retrieves the matching instances from the underlying database. See [8] pp. 316-321 for an introduction to Query Objects and criteria-based matching.
- The repository supports *pagination*; a pagination object specifies which subset of matching instances is requested. Whenever a match operation is applied, the repository sets the matching instance count.
- The repository supports *sorting*; a sort object specifies the sort attribute and the sort order to be applied.

¹ All references are provided in Appendix A - "References" on page 17

- The repository is responsible for *inserting, updating and deleting* data in the underlying database. It may also be made responsible for some simple referential integrity, although this is best delegated to the RDBMS.
- The repository implements a *smart update policy*; when an update request does not actually change any attributes, no database activity will take place. This prevents spurious database updates.

Clearly, this style of data access is much richer than a DAO pattern. The repository interface is meant to be all-inclusive and fully supportive of interactive data management applications as well as (web) service provisioning.

The repository provides an interface for Create, Update and Delete functions at the entity level; it hides any cascading database operations to supporting database tables. The repository also provides a rich interface for data retrieval, including retrieval by identifier and through criteria matching. Efficient pagination and sorting is best dealt with in the repository or perhaps even at lower layers, so the repository exposes a sortable pagination model for entity collections.

The implementation of a repository can be based on .NET, Java, or any other technology platform.

An implementation for the Java world would ideally be based on some underlying object/relational mapping (ORM) tool; Hibernate is a recommended open source candidate.

3 Repository Requirements Analysis

3.1 Create, Update, Delete Capabilities

No repository is complete without providing methods for persisting newly created entities, updating existing entity data as well as permanently removing entities from the underlying persistence layer.

A repository should implement *smart updates*; that is it must prevent actual database activities when effectively no data has been changed. A good repository is further capable of determining on its own which tables require updates given an entity passed into the repository. As long as all classes are navigable from the entity class, the repository must be capable of persisting changes in all such classes.

In reality, ORM tools provide all this functionality, so a repository design based on an ORM tool is a distinct advantage.

3.2 Criteria-Based Queries

A criteria-based query interface replaces the multitude of `findWhateverBySomeAttribute` methods that are so common in current designs. Instead, the criteria-based interface lets users communicate selection criteria in an object-oriented fashion. The repository encapsulates the transformation of a Criteria object to an (SQL) query string.

A Criteria object also serves another purpose as it can represent a 'current collection'. Assume a user has entered some search criteria in a search form, and the form elements have been transformed into a Criteria object. From that point on, the Criteria object implicitly represents the collection of matching entities. This object can be serialized and held as part of a user's session data, so the current collection can always be rebuilt from session data.

Note that criteria objects also represent customized searches that application users may wish to save for later re-use. In this way criteria-based queries provide a simple solution to recurring custom reporting requirements.

3.3 Filtering

Filtering, or Browsing, is an alternative to searching. In a search form, users enter data values to search on and the system transforms the form data into a Criteria object. In filtering, the system presents a list of *used* values to the user, the user selects one particular value and the system transforms that into a Criteria object. Filtering is always on the value of a single attribute at a time, and by definition it never results in an empty collection. This is because the system created the list of values based on actual usage in the first place.

3.4 Attribute Population

The repository must be aware of two distinct styles of data retrieval. One style is known as *summary-level information*, the other style is *full entity population*.

Summary-level information is called for whenever the repository is used to provide lists of entities; in a list mode only a few attributes per entity are needed. Conversely, full entity population is needed when a single entity is asked for (for viewing or updating purposes).

As an example, in an online ordering system, the order summary data includes total order value, customer ID and name, order date and maybe a few other details. The detailed data for an order includes all order header data, plus all the order items, their prices, any discount scheme in place, delivery plan dates, payment details and so forth.

The repository must have clearly distinguishable interfaces for both styles of data access. The summary-level data retrieval must be optimized to include all required attributes and no more than that. In particular, as few table joins as possible must be incorporated in the summary views.

Ideally, a lazy-fetching strategy is available to completely resolve this issue. Using lazy fetching, the summary-level attributes are marked for eager fetching, and any other attributes are only retrieved on request. As the business logic refers to such lazy attributes, the repository transparently fetches the missing attributes. Note that lazy fetching is a feature of several ORM tools, including Hibernate.

3.5 Pagination & Scrolling

Pagination is used to limit the display of matching entities to a predetermined maximum. Ideally, pagination requests are delegated all the way back to the RDBMS to avoid any unnecessary retrieval of data.

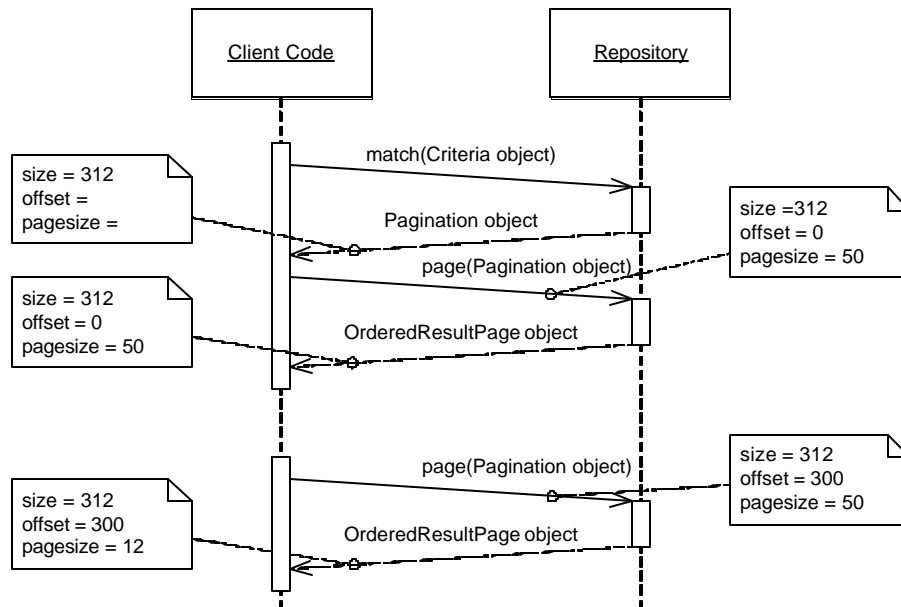


Figure 1 Sample interaction between Repository and Client Code

Following a successful match against a Criteria object, the business logic can ask for a single page of matching results using a *Pagination* object.

The *Pagination* object must provide the starting offset in the ordered collection of matching entities as well as the number of entities to be returned (the *page size*).

Upon return the repository will provide another page object that specifies the offset of the first entity (should match the starting offset that was asked for), the count of matching entities (should be same as what was returned initially when the match was done) and the count of entities returned (should be same as the requested page size, except for the final page which may be shorter).

3.5.1 Scrolling

Scrolling is in many ways identical to pagination. Scrolling also works on a current collection, i.e. it is preceded by a match operation that has returned a match count.

The difference is that the page size is implied to be '1', that is only a single instance is requested. Further, the client code expects to receive the full entity details, as scrolling is defined as stepping through the entity details one instance at a time. The page object can be used for scrolling as well, in order to allow client code to scroll to any particular instance in the collection. During scrolling operations, the page size attribute of the page object is simply ignored.

3.5.2 Concurrency Considerations

The outlined approach for pagination and scrolling may have some unpleasant side effects in highly dynamic environments where data is continuously inserted, updated and removed. With every new access to the repository, the query criteria will be evaluated again, and the matching result list can be different for each request, as newly added entities appear in the result and other entities no longer are part of the result.

It is tempting to consider alternative solutions that provide a more 'stable' view on the underlying data, but in reality such solutions introduce the risk of exposing stale data. Instead, application designers must consider to what extent highly dynamic data collections should be accessed using a web-enabled data management tool in the first place.

3.6 Sorting

Collections of matching entities are typically sorted, in other words they can be implemented using Java Lists. The business requirements for collection access to a repository include the need to provide matching instances in a predetermined order. Client code must be allowed to specify an attribute to sort on as well as a sort order.

Note that several ORM implementations include sorting in the Criteria interface, obviating the need for a separate Sorting class.

3.7 Screen Population

For interactive applications, helper classes are often introduced to offload the main code of screen population responsibilities. Screen population is at least in part a responsibility of the repository, as repositories will understand optimizations such as caching of key/value lists.

In order to support screen population, the repository must provide an interface to retrieve all *allowed* values for any enumerated attribute of the entity. In many cases, these allowed values will come from reference data tables or property files. Either way, the repository must be capable of presenting such lists of key/value pairs to the client code in a uniform manner.

3.7.1 Adding Hyperlinks to Data Elements

In data management applications, there are many opportunities for an enhanced user experience through the use of extensive hyper-linking. Any displayed data item can be hyperlinked to an implicit search for other records matching that attribute value. This is true in Filter, List and View pages. In the List page, hyper-linking can also be used to drill down to one record in the list, typically this is accomplished through hyper-linking one or more unique attributes. In that case, the implicit search returns a single matching entity which is subsequently displayed in View mode.

3.8 Metadata

For truly extensible designs, the repository must have a way to communicate the object model of the entity it represents. In Java, introspection is one way to do this, using the `java.lang.reflect` package. Alternatively, XML descriptions may be considered. Note that an initial repository implementation need not have a metadata query interface to function; this is only needed when the underlying business domain model is highly dynamic.

3.9 Separating Responsibilities

Given the requirements as stated above, which responsibilities of data access and data management in general belong where?

3.9.1 O/R Mapping

An O/R Mapping tool excels at transforming relational data into an object model and back. Hibernate, as one example, talks about persistent classes and the responsibility of Hibernate is to ensure that all changes to such classes are actually persisted. So, ORM is positioned as delivering database content as objects, and taking objects to persist them in a database. ORM has no more responsibilities than doing that efficiently.

3.9.2 Repository

The repository builds on the O/R Mapping layer and adds to that some business-model awareness, such as the ability to provide screen population lists (both lists of *used* and lists of *allowed* values). Repository also has a responsibility to distinguish summary-level information requests from detail-level information requests, and implement a pagination model on both types.

3.9.3 Repository Client Code

Client code that uses repositories is responsible for translating any user requests into appropriate repository calls. Also, the user interface may need information from more than one entity simultaneously. In this case, the client code must make repeated calls to one or more repositories before a complete page can be posted to the user.

The client code is also responsible for transaction management and for maintaining user session state (processing context).

4 Solution Outline

Based on the requirements analysis of Chapter 3, a minimal interface to a Repository object is outlined here. In this interface, various custom data types are used. An overview of these data types is provided in Appendix B - "Value Objects for Repository Access".

4.1 Create, Update & Delete

For these features, three distinct methods will be needed:

```
Whatever Repository.insert(Whatever aWhateverInstance);  
Whatever Repository.update(Whatever aWhateverInstance);  
void Repository.delete(Long aWhateverInstanceID);
```

Both `Repository.insert()` and `Repository.update()` take an instance of `Whatever` and return a possibly modified instance. In between, the repository may extend the instance with synthetic data such as a unique identifier, a timestamp, a version number or any other type of system information as needed in the application context.

4.2 Data Retrieval

For retrieval purposes, four distinct methods will be needed:

```
Whatever Repository.get(Long aWhateverInstanceID);  
Long Repository.match(Criteria aSelectionCriteria, SortingOrder);  
OrderedResultSet Repository.page(Pagination aPageSpec);  
OrderedResultSet Repository.scroll(Pagination aPageSpec);
```

`Repository.get()` returns the full details of a single instance of `Whatever`, as identified by a unique instance identifier.

`Repository.match()` takes a query in the form of a `Criteria` object, combined with a specification of the requested sort order. It returns a matching instance count.

`Repository.page()` and `Repository.scroll()` both take a `Pagination` object to specify their interest in a specific subset of the instances of `Whatever` that match the current criteria. This implies that both these methods must be preceded by `Repository.match()` to define the criteria. Both `Repository.page()` and `Repository.scroll()` return their results as an `OrderedResultSet`, which includes a (List of) matching `Whatevers`, the `Criteria` used to identify these, and the `Sorting` and `Pagination` that applies. The difference between the two methods is that `Repository.page()` returns only a minimally sufficient subset of attributes (summary data), and `Repository.scroll()` returns full details. `Repository.scroll()` further ignores the page size attribute of the `Pagination` argument as it only returns data on a single instance.

4.3 Screen Population

Screen population refers to providing lists of key/value pairs for various attributes of a Whatever. Two methods suffice:

```
List Repository.domain(String attributeName);  
List Repository.range(String attributeName);
```

`Repository.domain()` provides a list of key/value pairs of all possible values for an attribute. This applies only to enumerated attributes.

`Repository.range()` provides a list of key/value pairs of all values that are currently in use. It is intended for enumerated attributes but also works on free-text attributes.

4.4 Metadata

An optional interface for retrieving a domain model specification may be required. The specification may be in any desired form.

```
String Repository.model();
```

5 Manage a Whatever

The requirements outlined in chapter 3 appear fairly generic, but in reality they have been set up to align with the generic data management patterns of Manage a Whatever.

This pattern language covers use cases of data management, described in [1], common page flows in web-enabled data management applications, described in [2], and recurring patterns of HTTP request processing, as described in [3].

In particular, the data access patterns align with the user requests identified in [3]. This may make the initial design of the repository access patterns slightly biased towards web-enabled interactive data management applications. However, the resulting pattern is equally applicable to other styles of (remote) data access, including the encapsulation of enterprise business systems such as ERP applications. Repositories provide the basis for an elegant *service layer* that can be used to build industrial-strength web service interfaces.

5.1 How Repository Access aligns with the User Request Pattern

The user request pattern of Manage a Whatever is summarized in Table 2 below. Full details on the User Request pattern are presented in ref. [3].

Request	Description	Repository Responsibilities
new whatever	display an empty edit page in create mode.	Screen population
copy whatever	display a pre-populated edit page in create mode. Requires identifier of the whatever to copy.	Retrieval of source object, screen population
update whatever	display a pre-populated edit page in update mode. Requires identifier of the whatever.	Retrieval of source object, screen population
save whatever	process form data as either update or insert and revert to view whatever page.	Persisting new or modified data, returning saved object
cancel	cancel edit task and revert to view whatever page.	No specific responsibility
delete whatever	delete current whatever from persistent storage and revert to list whatever page. Requires identifier of a whatever.	Removal of identified object
view whatever	view details of selected whatever. Requires identifier of the whatever. Whatever is identified by their index in the current selection of whatever (the only way to drill down to whatever details is via a selection from a list of whatever).	Retrieval of requested object

Request	Description	Repository Re-sponsibilities
	Note that as a design decision, a multitabbed GUI of View Whatever Details may be implemented using multiple view whatever requests, each one requesting a different subset of information. This also applies to update whatever.	
list whatever	display list page with a overview of all whatever. In practice, often interpreted as 'list my whatever' (i.e. for any user a predefined notion of ownership applies).	Retrieval of first page of object summaries
sort whatever	re-display the list page, applying the indicated sort order and column. Begin on page 1 in case of a paginated list.	Retrieval of first page of object summaries in a specified sort order
page whatever	re-display the list page, showing the selected page (subset) of whatever. Maintain existing sort order.	Retrieval of specified page of object summaries in a specified sort order
search whatever	display the search page.	Screen population
find whatever	process form data as a query by example and revert to either list or view details page with search results, or to search page in case no matching whatever were found.	Retrieval of either a page of object summaries or details of a target object
filter whatever	display a filter page.	Screen population
match attribute	process selected attribute value as a query and revert to either list or view details page with query results (the case of no matching whatever should be an exception). Note that this request is only described as part of the filter whatever use case. However, there is another possible use for this request, as any suitable attribute in a list or view details page can be a hyperlink to navigate to a list of all whatever that share the attribute value. This extension makes for a highly interactive experience at relatively low increase of complexity.	Retrieval of either a page of object summaries or details of a target object
print whatever details, print what-	these are examples of services that apply to whatever. These may or may not be part of the application. All these services require an	Any

Request	Description	Repository Re- sponsibilities
ever list, ex- port what- evers	identifier of the whatever(s).	

Table 2 Summary of the User Request Pattern

6 Extensions to Repository

The core repository interface can be extended in several ways to enhance its versatility.

6.1 Soft Delete

In some cases, company policies stipulate that no data is ever to be physically removed; it should only be marked as expired. All queries to the database must be augmented to suppress expired instance data. Obviously, this can easily be encapsulated in a repository.

6.2 Effectivity or Temporal Property

Extending the notion of soft delete, many situations call for the use of versioned data. In this case, data is not only never deleted, it is also never updated. Instead, an update to a database row is implemented as an insertion of a new database row. The original row is marked as expired, using `fromDate` and `toDate` fields.

Now, each entity will have its own history trail. This can be exposed explicitly by the repository, as part of the data model of the entity, or an *effectivity* parameter can be used. With effectivity, all criteria passed to the repository are extended with a clause that specifies an *effective date*. The repository will now return entity data as it was known on the specified date. This is an implicit use of a history trail.

6.3 Auditing

The repository is a good place to implement auditing, as it has full awareness of all inserts, updates and deletes. However, the repository typically is unaware of the identity of the application user causing the transaction to happen. Therefore, repository *client* code is typically more appropriate for auditing responsibilities. A *service layer* which sits on top of the Repository is probably the most appropriate place for implementation of auditing.

To properly support auditing, however, the repository will have to track if any data was modified, and if so, precisely what data was modified. As the repository is responsible for implementing smart updates, it may well be that a request for update never translates into an actual physical database update. Such a situation does not call for an audit entry.

7

Conclusions

To integrate with the user request pattern of Manage a Whatever, a Repository interface with only nine methods will suffice. Combining a Repository design with a suitable ORM tool such as Hibernate greatly simplifies the design of the Repository as many responsibilities can be delegated to the ORM tool.

A well-performing implementation of a repository should consider an optimized implementation of `Repository.match()` as this is a frequently used and potentially expensive operation. In particular, `Repository.match()` must provide a count of matching instances without necessitating a full retrieval. Worst case, it can always issue an SQL `count()` query. Some RDBMS providers may have more efficient solutions knowing that at least some data will be retrieved immediately following `Repository.match()`. From the repository access patterns detailed in ref. [3], it follows that any call to `Repository.match()` is followed by either `Repository.page()` or `Repository.scroll()` except for some isolated cases.

Another optimization that repository designers should consider are the frequently recurring calls to `Repository.domain()`. These calls are issued whenever a data entry screen must be populated, so for all Find, New, Copy and Update Whatever use cases.

Appendix A - References

- [1] "Data Management Requirements – A Use Case Pattern Language", Version 1.1, White Paper by Bert Hooyman, Mphasis UK Ltd., March 2005
- [2] "Data Management Information Architecture – A Page Flow Pattern Language", Version 1.1, White Paper by Bert Hooyman, Mphasis UK Ltd., March 2005
- [3] "Data Management Request Processing - An MVC Pattern Language", Version 1.0, White Paper by Bert Hooyman, Mphasis UK Ltd., April 2005
- [4] "Designing Easy-to-use Websites", Vanessa Donnelly, Addison-Wesley, 2001. ISBN 0-201-67468-8
- [5] "Designing Data-Intensive Web Applications", Stefano Ceri, Piero Fraternali, Aldo Bongio, Marco Brambilla, Sara Comai, Maristella Matera, Morgan Kaufmann Publishers, 2003. ISBN 1-558-60843-5
- [6] "Web Database Applications with PHP & MySQL – Building Effective Database-Driven Web Sites", Hugh E. Williams and Davis Lane, O'Reilly 2002. ISBN 0-596-00041-3
- [7] "Web Application Design Handbook : Best Practices for Web-Based Software", Susan L. Fowler and Victor R. Stanwick, Morgan Kaufmann Publishers, 2004. ISBN 1-558-60752-8
- [8] "Patterns of Enterprise Application Architecture", Martin Fowler, Addison-Wesley, 2003. ISBN 0-321-12742-0
- [9] "Domain -Driven Design", Eric Evans, Addison-Wesley 2004. ISBN 0-321-12521-5
- [10] "Data Access Patterns", Clifton Nock, Addison-Wesley 2004. ISBN 0-131-40157-2
- [11] "Introducing Hibernate", on the web as: <http://www.hibernate.org/4.html>
- [12] "JDBC Cursor Performance with a Very Large Database", Michael S. Schwartz, IBM, available on the web as: <http://www.idug.org/idug/member/journal/-aug02/battle.cfm>

Appendix B - Value Objects for Repository Access

The sample code presented below extends the solution outline presented in Chapter 4.

```
class Criteria implements Serializable {
    // the precise details of the Criteria class are not relevant at this point
    ...
    ...
}

class Pagination implements Serializable {
    public Integer size; // match count of current collection
    public Integer offset; // starting offset into the ordered collection of matching
                        // entities
    public Integer pagesize; // number of entities to include on a page
}

class Sorting implements Serializable {
    // assume Criteria does not support sorting so this must be specified separately
    public String attributeName;
    public String sortOrder;
}

class OrderedResultPage implements Serializable {
    public Criteria matchCriteria;
    public List sortDetails; // a List of Sorting objects
    public Pagination pageDetails;
    public List results; // a List of Whatever
}

class KeyValuePair {
    public String Key; // the instance identifier
    public String Value; // the human-readable instance value
}
```