Openwings
Data Services Specification
Beta Ver 0.81

COPYRIGHT ©2000-2002 GENERAL DYNAMICS DECISION SYSTEMS, INC.
ALL RIGHTS RESERVED
This document is subject to "Terms of Use" as described at http://www.openwings.org.

PREFACE

About the Open Specification Process
A brief overview of the community process for developing Openwings Specifications is discussed below. This process was developed by General Dynamics Decision Systems and is similar to the community process for developing Java specifications.

A Process Management Organization (PMO) oversees development and maintenance of all Openwings Specifications. Any company, organization or individual can join the Specification development team; team members are called Participants. The latest Openwings Specifications will be maintained on a Public Web Site.

Participants can submit requests to the PMO to modify existing or develop new Specifications. If the PMO accepts the request, the PMO will then begin the process of forming an Expert Team and will appoint a Specification Lead.

The Specification Lead will direct the Expert Team in developing the new Specification. Once the team agrees on a draft of the Specification, it will be posted on the Public Web Site for public review. A member of the Expert Team will also begin development of a Reference Implementation and Compatibility Test Suite for the new Specification.

Based on public comments, a final release of the Specification will be produced by the Expert Team. Concurrently, the Reference Implementation and Compatibility Test Suite will be completed. Once the Final Public Release of the Specification is completed and posted on the Public Web Site, the Expert Team will disband.

This process is intended to provide rich consensus based Specifications.

Contributors
We would like to recognize and thank the following people for their contributions to this document.

Author(s):
Christopher Williamson

RFI Contributor(s):
N/A

RFC Contributor(s):
Mitch Branting, Dale Clark, Guy Bieber, Jeff Carpenter, Wade Wassenberg, Jon Lammers

1. INTRODUCTION 1

1.1 PURPOSE 1
1.2 SCOPE 1
1.3 DEFINITIONS 1
1.4 OVERVIEW 2
1.4.1 Openwings Overview 2
1.4.2 Document Overview 2

2. GOALS / REQUIREMENTS 3

3. USE CASES 4

3.1 INSERT DATA 4
3.2 DELETE DATA 4
3.3 UPDATE DATA 4
3.4 RETRIEVE DATA 4
3.5 MANAGE DATASERVER 4

4. ARCHITECTURE 5

4.1 OBJECT-RELATIONAL MAPPING 6
4.1.1 OR-Mapping Tools 6
4.1.2 OR-Mapping File 6
4.2 QUERYING LANGUAGE 7
4.2.1 Filters 7
4.3 TRANSACTIONS 7
4.3.1 Data Integrity vs. Performance 8
4.4 AVAILABILITY 8
4.4.1 DataServer Discovery 8
4.4.2 Replication 9
4.4.3 Object Locality 9
4.5 MULTIPLE USERS 9
4.5.1 Threads 9
4.5.2 Connection Pools 9
4.5.3 Locks 10
4.6 ASYNCHRONOUS VS. SYNCHRONOUS 10
4.7 MESSAGING INTERFACE 10
4.8 KEY GENERATION 11
4.9 MANAGEMENT INTERFACE 11
4.10 CONFIGURATION 11
4.10.1 DataServer Policies 11
4.10.2 DataServerProxy Policies 11

5. INTERFACES 12

5.1 DATASERVERPROXY 14
5.2 DATASERVER 18
5.3 MESSAGEDATASERVER 21
5.4 DATASERVERLISTENER 25
5.5 FILTERCONVERTER 26
5.6 KEYGENERATOR 27
5.7 DATASERVERMANAGEMENT 28
5.8 DATASERVERSTATISTICS 30
5.9 DATASERVERPOLICY 32
5.10 DATASERVEREXCEPTION 34

5.11 AbortRequestException 35
5.12 ConnectionException 36
5.13 DatabaseException 37
5.14 DeadlockException 38
5.15 DuplicateDataException 39
5.16 FileSystemFullException 40
5.17 FilterConverterException 41
5.18 InvalidRequestException 42
5.19 KeyAllocationException 43
5.20 KeyGeneratorException 44
5.21 MalformedQueryException 45
5.22 ObjectNotFoundException 46

6. TOOLS 47

6.1 CodeGenerator 47
6.1.1 Command Line Interface 47
6.1.2 XML and Template file formats 47
6.2 DataServerManager 49
6.2.1 Command Line Interface 49
DataServerManager GUI 49

7. EXAMPLES 50

8. COMPLIANCE 51

9. REFERENCES AND FURTHER READING 52

Figure 1: Definitions 1
Figure 2: Processing Service Goals / Requirements Table 3
Figure 3: Main use cases 4
Figure 4: Data Services Interfaces 6
Figure 5: Message Based Data Server Interfaces 10
Figure 6: Interface Roles 12
Figure 7: Class Diagram 13
Figure 8: Exception Hierarchy 13
Figure 9: DataServerManager GUI 49
Figure 10: Compliance Table 51

1. Introduction

1.1 Purpose

The purpose of the document is to define the persistence abstraction used in Openwings.

1.2 Scope

This document will describe what is required to build a compliant DataServer. Users, implementers and administrators are the three roles, which are defined to use DataServices. Users use DataServices interfaces to read and write data. Implementers design and write the code to implement DataServices interfaces. Administrators use DataServer interfaces to maintain DataServers.

1.3 Definitions

Here are some definitions to provide context to this section.

Term	Definition
DataServer (DS)	The middle-tier object in Data Services used to persist a specific type of data.
DataServerProxy (DSP)	The first-tier object in Data Services, which is used to locate DataServers and interface with them. Proxies are also used to monitor clients.
Database Management System (DBMS)	The third-tier in DataServices, that provides access to databases.
Deadlock	A deadlock occurs during transactions where applications lock tables and wait for each other to unlock each other's tables before continuing.
Key	In this document, a key generally means the primary key. A primary key is the identifier for an object or a relational entity.
Key Generator	An object used to generate a key for a particular data type.
Lock	A method of concurrency control used to control accesss to data in a multi-user database. Locks prevent simultaneous updates to shared data.
Object-Oriented Database Management System (OODBMS)	A database system that uses an object-oriented model as its form of using and storing data.
Object-Relational Database Management System (ORDBMS)	A database system that uses an object oriented model as its form of using data but uses a relational model to store the data. A conversion needs to be done to map the object model to the relational model.
Object Query Language (OQL)	A language used to send commands to database management systems using object attributes.
Query	A database management system command.
Structured Query Language (SQL)	A language used to send commands to database management systems using relational fields.
Transaction	A collection of related operations performed as a single unit on the database. None of the operations are permanent until they are all committed.

Figure 1: Definitions

1.4 Overview

1.4.1 Openwings Overview

Openwings™ is an open community, non-proprietary effort to define specifications for self-forming, self-healing systems. Motorola (now General Dynamics Decision Systems) and Sun Microsystems established the Openwings™ consortium in June of 1999. Since then, over 100 companies have registered to help mature its development, and more companies are registering daily.

The core Openwings™ framework is designed to incorporate existing commercial standards and is intended for use in both commercial and military environments. Openwings™ is being developed using a community development process modeled after the very successful Java Community Process. Anyone may join the Openwings™ community, participate on expert teams, or use the resulting specifications free of charge by merely signing up at the community web site (<http://www.openwings.org>).

The Openwings™ Architecture provides a framework for building plug-and-play, service-oriented, network-centric, self-forming, self-healing systems that are independent of middleware, databases, platforms, and deployment contexts. Openwings™ has a special focus on issues of availability, security, and interoperability. Openwings™ is the embodiment of a new movement in the software engineering community towards a paradigm known as Service-Oriented Programming (SOP). For an introduction to SOP, refer to <http://www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf>.

1.4.2 Document Overview

Data Services provides persistence services for user data. This document will address the persistence requirements, use cases, architecture, and interfaces that will be used for Data Services.

This specification follows the basic format of all of the Openwings specifications:

- Goals and Requirements – Describes requirements of this specification.
- Use Cases – Describes UML use cases for this specification.
- Architecture – Describes the various architectural aspects of this specification.
- Interfaces – Describes interfaces required for this specification.
- Tools – Describes various tools and utilities provided with the specification.
- Examples – Provides representative real world use cases for this specification.
- Compliance – This section describes what is required to comply with this specification.

2. Goals / Requirements

Here are the goals / requirement for data services:

#	Goals / Requirements	Req Ref	Defer
1	The DataServer shall support mapping inheritance, association, and collections.	5	-
2	Shall support the use of object-oriented databases, relational databases, and flat files	5	-
3	Shall support connection management.	5	-
4	Shall support 2-tier transaction management.	5	Defer
5	Shall use object-based queries.	5	-
6	Shall hide database security and use the Component Services security model.	5	-
7	The APIs shall be vendor independent.	23	-
8	Shall provide a tool for automated database management that will backup, purge old data.	8	Defer
9	Shall provide a capability to push filtered incoming data to consumers.	5	Defer

Figure 2: Processing Service Goals / Requirements Table

3. Use Cases

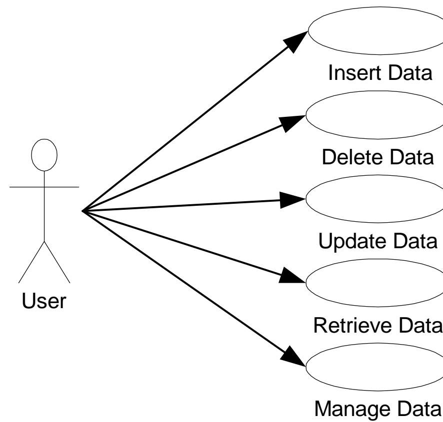

Figure 3: Main use cases

3.1 Insert Data

The insert data use case persists data using a DataServer. There are two variants of inserting data with a DataServer. Data can be inserted one object at a time or multiple objects at a time.

3.2 Delete Data

The delete data use case removes data using a DataServer. There are three variants of removing data with a DataServer. Data can be removed one object at a time using an ID, removed multiple objects at a time using the objects' IDs, and remove multiple objects using filters which specifies which objects to remove.

3.3 Update Data

The update data use case persists updated data that is in the database using a DataServer. There are two variants of updating data with a DataServer. Data can be updated one object at a time using the object's ID, or update multiple objects at a time using the objects' IDs.

3.4 Retrieve Data

The retrieve data use case selects data from the database using a DataServer. There are three variants of retrieving data with the DataServer. Data can be retrieved using filters to specify which objects to retrieve. Data can also be retrieved in the form of iterators using a filter. Data can also be retrieved data using object ID's to specify which objects to get.

3.5 Manage DataServer

The manage DataServer use case manages DataServers using the DataServer Manager. Logging, performance data, backup and purge functions are part of this use case.

4. Architecture

Data Services takes a three-tier approach to storing data. One of the main benefits of Data Services is it abstracts database technology out of a component and provides a standard interface components use to store data. Other components shouldn't care how the data is stored in the backend, they just want to read and write the proper data. By using a middle tier, Data Services centralizes all persistence code into its own component that is managed separately from other components. This makes managing persistence code easier as other components are abstracted from the persistence code. A three-tiered model will also allow for more flexibility with the implementation decisions and will not be limited by other components. Thin clients can also be used with this architecture. Thin clients make it easier on application machines, as the middle-tier can be run on different machines thus reducing the processing load on the client machine.

The DataServerProxy interface is the first tier of Data Services. It is used to manage DataServer lookups, track performance, and configure logging on a client. When using the DataServerProxy, an application is relieved of lookup responsibilities. The proxy communicates with Component Services to get a list of the DataServers available. The DataServerProxy then uses logic plugged in by the user to decide which DataServer to use. . A Class object is used to help determine which DataServer to lookup. The DataServer is then stored in a Hashmap for later use. Therefore a user that uses a proxy does not have to worry about most persistence specific issues. A user just needs to give the object to the proxy along with the Class of the data and the proxy will take care of the rest. An object's class could be determined using inspection. However, it is not used to determine the class because the class parameter provides a mechanism for inheritance. This class parameter allows the user to specify exactly what class of DataServer to use in an inheritance hierarchy. This allows data to be stored at different levels depending on the need. The proxy also has built in support for making synchronous and asynchronous calls. A user can either wait to see if the call was properly executed before continuing (synchronous) or continue on without caring if the call executed properly (asynchronous).

The DataServer is a middle-tier component that provides access to a persistence store for a particular type of data. There will not be one DataServer interface, but several type-dependent DataServer interfaces. The interfaces are generated using templates and the CodeGenerator tool. The interfaces will all look the same except they will use different data types in their functions. So large systems with several data types will have several DataServers on a system each with its own interface representing a data type. Data is distributed among several type-dependent DataServers for better performance and better reliability. DataServers can be tuned individually to enhance performance. The DataServer implementation is also specific to a particular persistence storage device, but this should not be able to be seen through the interface. Different backends can be plugged in without the user noticing. A good implementation of a particular DataServer would separate the parts specific to the persistence device and data types. This would allow for maximum reusability.

The DataServerManagement interface is used to manage the DataServer at runtime. Logging functions, performance statistics, and query management are some of the basic functions that this interface provides. The DataServerManagement interface should be provided along with the DataServer so that certain features can be used when a DataServer is online.

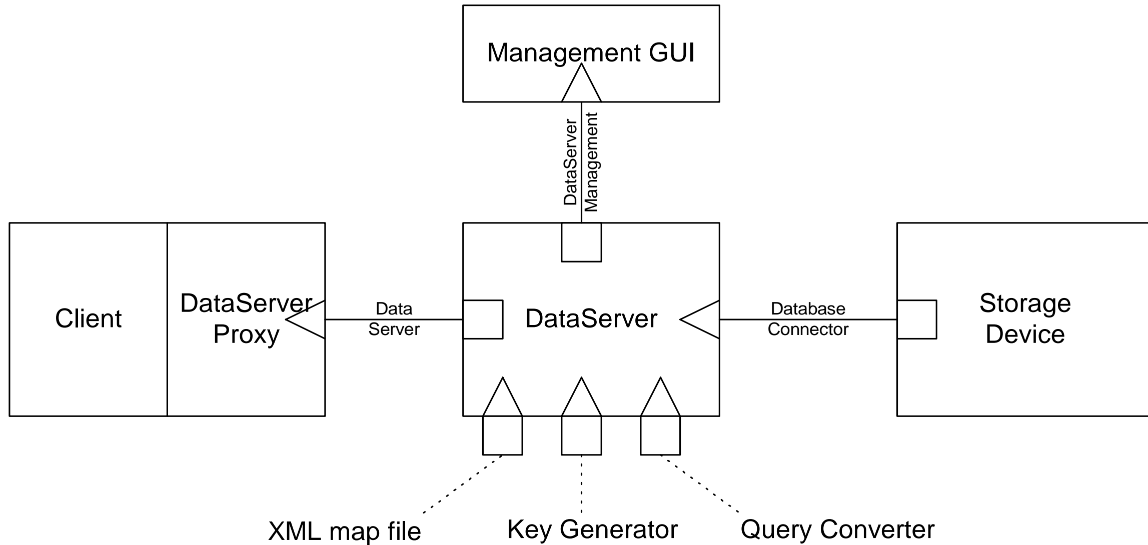

Figure 4: Data Services Interfaces

4.1 Object-Relational Mapping

The DataServer is an object based API that supports relational databases. Therefore mapping from an object model to a relational model is needed when the backend of a DataServer uses a relational database. DataServer developers can create these maps manually or use a commercially available OR-mapping tool.

4.1.1 OR-Mapping Tools

Object-Relational mapping tools are used to map an object model to a relational model. OR-mapping tools simplify the process of mapping objects by generating code that persists data in a relational model for a given database. OR-mapping tools have several other benefits also. Generating OR-mapping code by using OR-mapping tools can significantly cut development time. There is less code that needs to be generated manually when using these tools. Many of these tools also have special features that can be used to improve the performance and reliability.

4.1.2 OR-Mapping File

A good OR-mapping tool will separate the mapping from the code and place it in a non-compiled mapping file. This is significant for two reasons. First this means that code is written to interface to the OR-mapping tool's API can be reused for each data type. The code you write will be type independent. The tool should have an API that is type

independent and all the type dependent information should be contained in the non-compiled mapping file. This will make code generation easier for DataServers. The second benefit of having a mapping file is it can provide the information to objects other than the core DataServer as well. The mapping will not reside in compiled code but in a file that can be read on the fly. This is beneficial when using queries, as queries often need to know mappings. A standard format for these files needs to be used so that mapping information can be read by many different sources in the same way. OR-mapping tools use various formats to store their mappings. Mappings should to be converted into a standard XML format that can be used by any back end. The XML format needs to be developed and used as a standard for the OR-mapping industry.

4.2 Querying Language

One of the biggest issues with Data Services is that there is no universally accepted object querying language. The closest thing that comes to a standard for querying languages is SQL. However SQL does not provide the kind of functionality that many of the mapping tools or object databases would like to provide for their users. SQL is meant generally for use with relational databases and does not work well with object models. So a lot of the different OR-mapping tools and object databases use OQL or different proprietary query languages to do their queries. Data Services shall provide a query mechanism to be used with Data Services. A user will only have to deal with one type of query instead of learning several different types. This is especially useful since the type of query presented would depend on an underlying tool that can change from one instance of a Data Server to the next. Then the standard language will be converted by converter objects to the language that the implementing code uses.

4.2.1 Filters

DataServers use filters currently as their query mechanism. A filter is an object, which defines static queries that can be called on a particular DataServer. One or more attributes can define a particular query. For example, if the filter has an id attribute that is set to 22, then the query could return an object with an id of 22. Another example is an object has startTime and endTime attributes, which is meant to return all the times between their values. The behavior of a filter is determined in the filter converter object that resides on the DataServer for a particular data type. A converter determines exactly what the startTime and endTime attributes define in the query. Filters are not meant to be dynamic in nature. In fact, their static nature enforces control over the queries that are sent to the Data Server. There will not be any malicious query statements that are sent to the DataServer using filters. However, this can be limiting for a developer. A better object query language needs to be developed that will allow more flexibility to the queries.

4.3 Transactions

Transactions are an important part of persisting data because they allow changes to be rolled back if something goes wrong while in the middle of a transaction. If any part of a transaction fails, the entire transaction looks as if it never took place. Transactions in Data Services are dealt with in two ways. First it is possible to have a transaction for

every call to a DataServer method. So each insert, update, and remove call starts and commits a transaction. (Retrieves do not use transactions because they do not change data.) This type of transaction does not span over several calls to a DataServer, and is part of the back end implementation. The other type of transaction is a 2-phase transaction. This type is used in a distributed system like Data Services. It allows the user to start a transaction that makes multiple calls to various DataServers. If one of the calls fails, then each of the calls to various DataServers are rolled back. Some additions to the DataServer interface are needed to implement the 2-phase transactions. A transaction server could then perhaps be used to manage 2-phase transactions. Transactions will be addressed further in following releases.

4.3.1 Data Integrity vs. Performance

There is a major tradeoff in data integrity vs. performance when developing Data Servers. A Data Server can be set up to use transactions and locking to make sure that integrity of the data is strong. As mentioned before, when a transaction is used, all actions performed on the database during the transaction can be rolled back if something goes wrong with the system. A transaction uses locking to make sure that the data being manipulated in the transaction is not changed or misread in some way. Locking is good for data integrity but can be bad for performance. Using locking inhibits other threads outside a particular transaction from using data in the certain locations in the database. This means that they have to wait until the data in unlocked. Obviously if locking is turned off performance will drastically increase. It is important to pay close to how locking and transactions are configured in a DataServer.

4.4 Availability

In most cases, data is located on several DataServers. Several advantages are gained by distributing data over multiple DataServers. An advantage of each data type having their own DataServer is that each DataServer can be tuned to perform in different ways. Simple data types can use standard DataServer implementations, but complex data types have the flexibility to make adjustments that can help with performance and other areas. Another benefit of this is that there is no central point of failure. This means that if something occurs that breaks down a particular data type's DataServer, it does not bring down all the other DataServers. The DataServers are independent of each other. A distributed system also allows for better recovery. If one DataServer fails, then another of the same type can take its place or another can be started to replace it.

4.4.1 DataServer Discovery

Since there could be several DataServers on a network, there is always the possibility that there will be multiple DataServers of the same data type. When a DataServer is registered with Component Services, attributes should be supplied to make sure the appropriate DataServer is discovered. Some examples of attributes besides data type that could be used to identify DataServers are groups, database type, server type, and location on the network. The DataServerProxy should plug in an object that determines the appropriate DataServer to use based on these attributes.

4.4.2 Replication

Replication of data to other databases will not be supported directly in the DataServer interface. However, many databases provide tools that will take care of this in the back end. It would also be possible to create some sort of replicator on top of the DataServer that would send the same data to multiple DataServers.

4.4.3 Object Locality

The persistent objects that an application uses can be located remotely or locally. An application can use a copy of the objects retrieved from a persistence store or it can use a reference to those same objects. There are advantages to doing it either way. The major advantage of making a copy to the local application is speed. Since this is a distributed system, objects will be retrieved or accessed over a network. By making a copy of the objects to the local application, constant calls over the network are eliminated. An application won't have to go over the wire every time an attribute of an object is changed. The application can store an object once all attributes are changed and need to be persisted. However, by the time these changes need to be persisted, the data may become dirty. Dirty data is data that has changed since the last time it was accessed. This is where references to objects on a DataServer can be very useful. The DataServer can be setup to lock an object that is being used by an application. This is useful because if another application tries to access this data, it will not be allowed until the first application releases its lock. This means that the data integrity is intact for when other components use the objects.

4.5 Multiple Users

In most cases there will be multiple users making calls on the database. Because of this threads, connection pools, and locks must be considered in order to maintain acceptable performance and data integrity.

4.5.1 Threads

Multiple threads are needed because several users may be calling a DataServer at once and this allows the users to work concurrently as long as they are accessing different data. DataServers should use create a new thread inside each method call. Some OR-mapping tools have this capability and provide methods for doing this.

4.5.2 Connection Pools

Connection pools creates several connections to the database and stores them in a pool. Each thread pulls a connection out of the pool, makes calls to the database using the connection, and then returns the connection to the pool when it is through. Calls to the database should be quicker because retrieving a connection from the pool should be faster than creating a new connection. Some OR-mapping tools provide connection pools and should be used for convenience.

4.5.3 Locks

Locking is an another important issue to consider for systems with multiple users. As described above, locks are a mechanism for ensuring concurrency. There are two places where locks exist on a DataServer. The first place is in the database. Data being accessed by a user within the database is locked. This kind of lock is normally managed by the database. The second place is in the object cache of the DataServer. The cache is a pool of recently used objects stored on the DataServer. The number of calls to the database is reduced when using a cache because objects may be local. So when accessing objects in the cache, locks are used to ensure concurrency among the threads. Some OR-mapping tools provide this functionality and should be used for convenience.

4.6 Asynchronous vs. Synchronous

A synchronous DataServer call is a method, which returns before moving onto the next method. An asynchronous call will hand this call to a different thread to write data, and continue onto the next method without receiving the status of the method call. Synchronous calls are probably more common than asynchronous calls because of their transactional nature. When a synchronous call is made a confirmation is received which indicates whether or not a call has completed correctly, and the code can move on accordingly. In an asynchronous situation a user does not care if the call completes correctly or not. It moves on anyway. This is appropriate for instances where the data being written is not critical. Asynchronous calls can be used when a user wants to write a large amount of data and move on without waiting a long time to see if the call completes correctly.

4.7 Messaging Interface

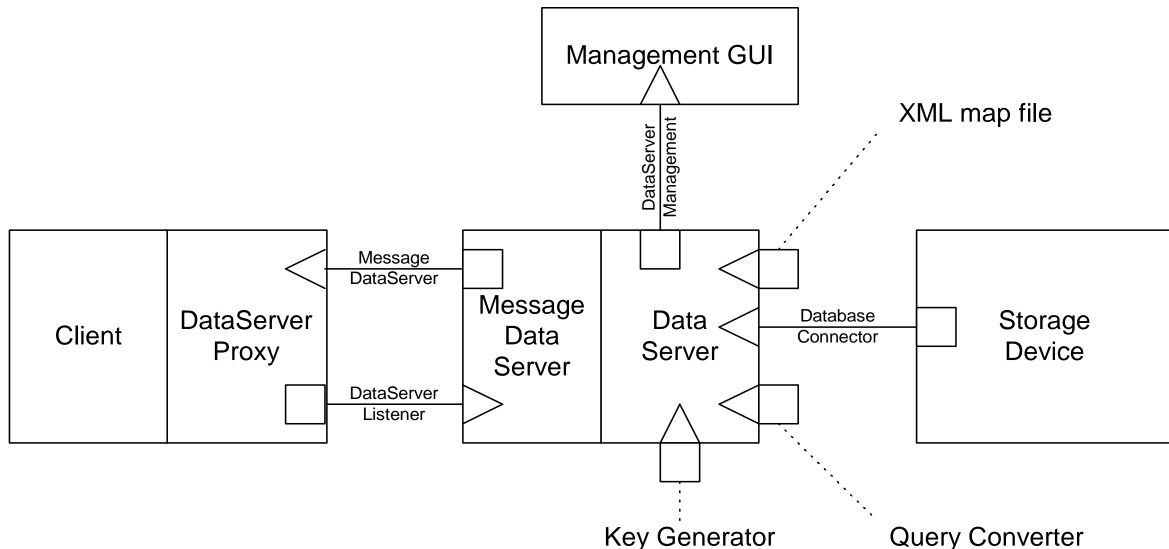

Figure 5: Message Based Data Server Interfaces

The messaging interface is used when a messaging service is used to connect to the DataServer. This interface is very similar to the standard DataServer interface. The main difference is that methods in this interface do not return objects. This is due to the nature of messaging services. Messages only go one way. Objects are returned using a listener interface implemented by the DataServerProxy. When using synchronous methods with the messaging interface, the DataServerProxy call must block until a call to the listener occurs. Also when implementing a messaging interface to the DataServer, it may be easiest to wrap a regular DataServer with the messaging interface.

4.8 Key Generation

Keys refer to primary keys used in relational databases. A primary key is one or more attributes that are used to identify an object. A key generator function is provided, for cases where the user does not supply the key. There are different ways to generate keys and data types do not always use the same key. So key generators are provided as plug-ins to supply the correct key for a data type. There is an interface for key generators that allow people to develop the appropriate key generator for their DataServer and plug it in. There will also be generic key generators that can be used as needed.

4.9 Management Interface

The management interface is used to configure the DataServer. This interface is used to turn on logging and debugging. This interface also provides information about the DataServer. Statistics can be turned on to see how well the DataServer is performing. Query converters, key generators, and other pieces of the DataServer may also be managed using this interface.

4.10 Configuration

Policies will be used to provide configuration information to the DataServer and DataServerProxy about how each will act. The policies are XML based as described in the Openwings Policy API.

4.10.1 DataServer Policies

Policies can be used to specify information that is used to make connections to the underlying database. It can provide user names, passwords, database locations, and other information used to make connections to a database. Policies can also be used to provide other configuration information such as number of database connections to be used, type of default query converter, type of default key generator, location of XML map file, and many other things.

4.10.2 DataServerProxy Policies

This policy may contain information pertaining to client performance statistics, logging, and identification.

5. Interfaces

The Data Services API has been designed to handle the use cases and architectural constraints described above. Data Services is divided into three tiers. Data Services provides two interfaces for the first two tiers. The DataServerProxy is the first tier and the DataServer is the middle tier. The DataServer interface is used to abstract the persistence of data. It provides methods for inserting, removing, updating, and retrieving specific data types. The DataServerProxy provides these same methods on a generic level. The proxy is used to find the correct DataServer and use it to persist data.

MessageDataServer and DataServerListener are interfaces that are used with message based connectors. MessageDataServer has the same functions as DataServer except nothing is returned using this interface. The DataServerListener interface is used to return data and exceptions from functions called on MessageDataServer.

There are a few more interfaces that are used in Data Services that are important. The key generator is used to generate keys for a particular data type, and the query converter converts filters into a query that can be used by the implementing code. The management interface is also very important as it is used to configure the DataServer.

Interface/Class	Role
DataServer	Implementer
DataServerProxy	User, Implementer
MessageDataServer	Implementer
DataServerListener	Implementer
DataServerManagement	Implementer, Administrator
DataServerStatistics	Implementer, Administrator
DataServerPolicy	Implementer
FilterConverter	Implementer
KeyGenerator	Implementer
AbortRequestException	User, Implementer
ConnectionException	User, Implementer
DatabaseException	User, Implementer
DataServerException	User, Implementer
DeadlockException	User, Implementer
DuplicateDataException	User, Implementer
FileSystemFullException	User, Implementer
FilterConverterException	User, Implementer
InvalidRequestException	User, Implementer
KeyAllocationException	User, Implementer
KeyGeneratorException	User, Implementer
MalformedQueryException	User, Implementer
ObjectNotFoundException	User, Implementer

Figure 6: Interface Roles

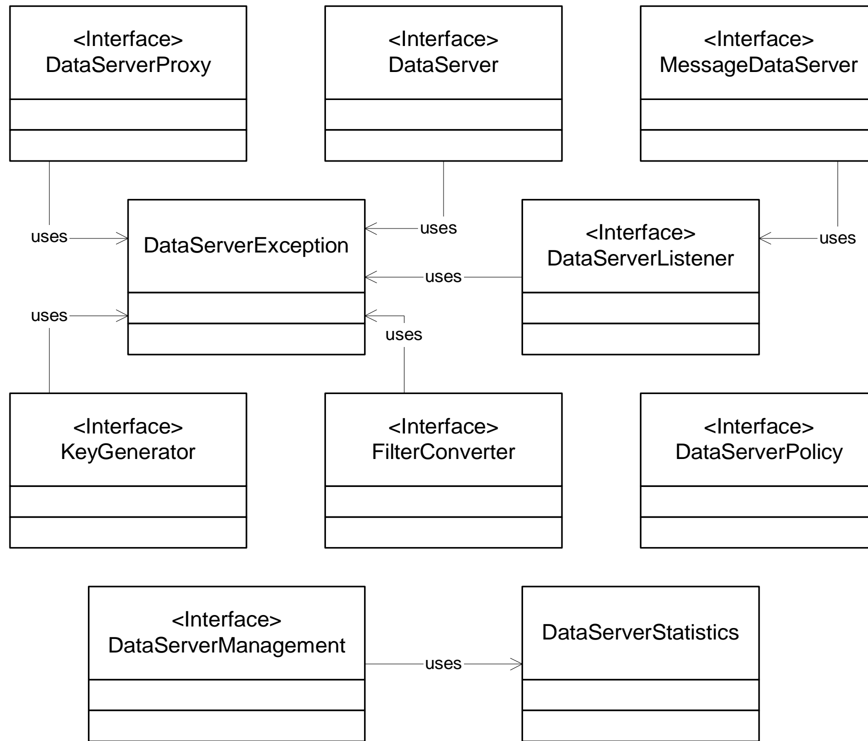

Figure 7: Class Diagram

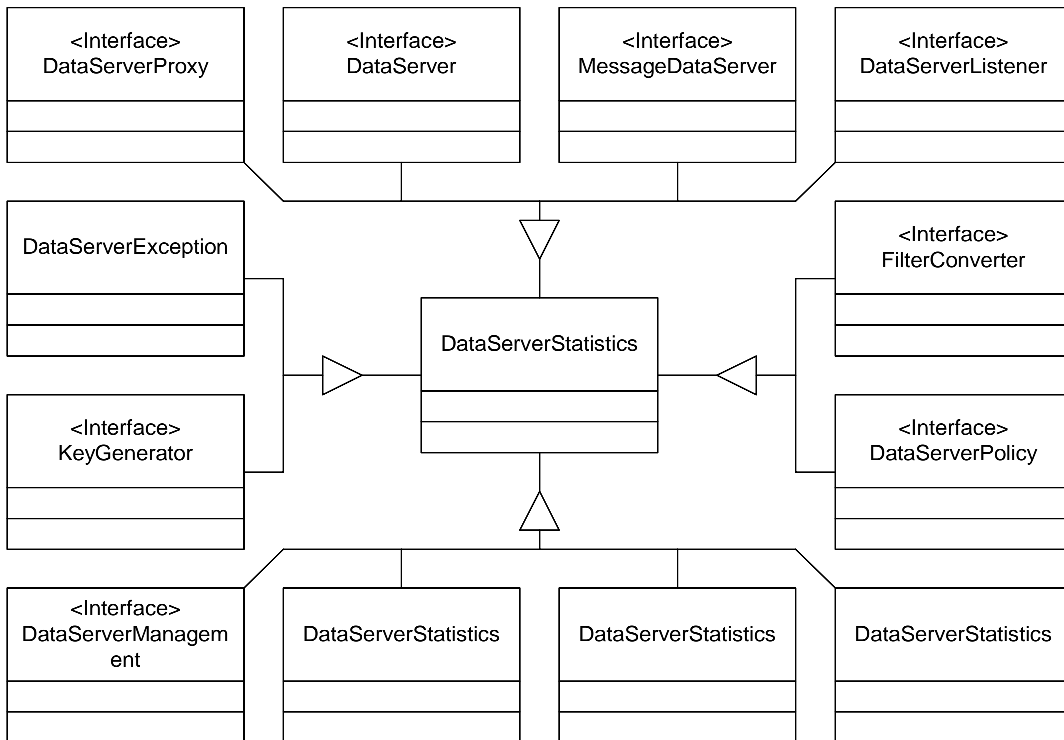

Figure 8: Exception Hierarchy

5.1 DataServerProxy

This is the first tier of Data Services. The DataServerProxy abstracts connecting to the middle-tier and logging.

```
public interface DataServerProxy
{
  Collection getKeys(Class datatype,
                     int numOfKeys)
                     throws DataServerException;

  void storeObject(Class datatype,
                   Object object,
                   int mode,
                   boolean waitForStatus)
                   throws DataServerException;

  void storeObjects(Class datatype,
                    Collection objects,
                    StorageMode mode,
                    boolean waitForStatus)
                    throws DataServerException;

  void removeObject(Class datatype,
                    Object object,
                    boolean waitForStatus)
                    throws DataServerException;

  void removeObjects(Class datatype,
                     Collection objects,
                     boolean waitForStatus)
                     throws DataServerException;

  void removeObjectsByFilter(Class datatype,
                             Object filter,
                             boolean waitForStatus)
                             throws DataServerException;

  void retrieveObjects(Class datatype,
                       Collection objects)
                       throws DataServerException;

  Collection retrieveObjectsByFilter(Class datatype,
                                     Object filter)
                                     throws DataServerException;

}
```

- `Collection` **`getKeys`**`(Class datatype,`
```
                  int numKeys)
                  throw DataServerException;
```
  This method returns unique keys that can be issued to objects that are going to be persisted. The `datatype` parameter is a Class that identifies which type of DataServer to use to get keys. The `numKeys` parameter determines how many keys to get. The method returns a `Collection` of keys. A `KeyAllocationException` is thrown if there are not enough free keys to be allocated for the call.

- `void` **`storeObject`**`(Class datatype,`
```
                Object object,
                int mode,
                boolean waitForStatus)
                throws DataServerException;
```
  This method inserts or updates an object into the database. The `datatype` parameter is a Class that identifies which type of DataServer to use to store the object. This parameter is used instead of inspecting the object, to allow data to be stored at multiple levels of inheritance. The `object` parameter is the object that is stored. The `mode` parameter determines how to store the object. There are three modes that decide how to insert or update. There is an INSERT mode that inserts the object only if the object is not in the database. There is an UPDATE mode that updates the object if it is in the database. Finally there is an INSERT_UPDATE mode that updates the object in the database if it is present and inserts it if it is not. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly. A `DuplicateDataException` is thrown when an object's key already exists in the database. A `FileSystemFullException` is thrown when the disk housing the database is full and data cannot be stored.

- `void` **`storeObjects`**`(Class datatype,`
```
                 Collection objects,
                 int mode,
                 boolean waitForStatus)
                 throws DataServerException;
```

  This method inserts or updates a collection of objects in the database. The `datatype` parameter is a Class that identifies which type of DataServer to use to store the objects. This parameter is used instead of inspecting the object, to allow data to be stored at multiple levels of inheritance. The `objects` parameter is the collection of

objects that is stored. The `mode` parameter determines how to store the object. There are three modes that decide how to insert or update. There is an INSERT mode that inserts the object only if the object is not in the database. There is an UPDATE mode that updates the object if it is in the database. Finally there is an INSERT_UPDATE mode that updates the object in the database if it is present and inserts it if it is not. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly. A `DuplicateDataException` is thrown when an object's key already exists in the database. A `FileSystemFullException` is thrown when the disk housing the database is full and data cannot be stored.

- ```
  void removeObject(Class datatype,
                    Object object,
                    boolean waitForStatus)
                    throws DataServerException;
  ```
  This method removes an object from the database based on a unique ID. The `datatype` parameter is a Class that identifies which type of DataServer to use to remove the object. This parameter is used instead of inspecting the object, to allow data to be stored at multiple levels of inheritance. The `object` parameter is the object that is removed. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly.

- ```
  void removeObjects(Class datatype,
                     Collection objects,
                     boolean waitForStatus)
                     throws DataServerException;
  ```
  This method removes a collection of objects from the database based on unique ID's. The `datatype` parameter is a Class that identifies which type of DataServer to use to remove the objects. This parameter is used instead of inspecting the object, to allow data to be stored at multiple levels of inheritance. The `objects` parameter is the collection of objects that is removed. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly.

- ```
  void removeObjectsByFilter(Class datatype,
                             Object filter,
                             boolean waitForStatus)
  ```
  ```

```
                                 throws DataServerException;
```

This method removes objects based on filter criterion. The `datatype` parameter is a Class that identifies which type of DataServer to use to remove the objects. The `filter` parameter is the object that determines which objects to remove. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly. A `MalformedQueryException` is thrown when the query formed by the filter is not correct.

- ```
  Collection retrieveObjects(Class datatype,
                             Collection objects)
                             throws DataServerException;
  ```

  This method retrieves a collection of objects from the database based on unique ID's. The `datatype` parameter is a Class that identifies which type of DataServer to use to retrieve the objects. This parameter is used instead of inspecting the object, to allow data to be stored at multiple levels of inheritance. The `objects` parameter is the collection of objects that is retrieved. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly. The method returns a `Collection` of the objects retrieved from the database. An `ObjectNotFoundException` is thrown if the object to be retrieved is not found in the database.

- ```
  Collection retrieveObjectsByFilter(Class datatype,
                                     Object filter)
                                     throws DataServerException,
                                            RemoteException;
  ```

  This method retrieves a collection of objects from the database based on filter criterion. The `datatype` parameter is a Class that identifies which type of DataServer to use to retrieve the objects. The `filter` parameter is the object that determines which objects to retrieve. The `waitForStatus` parameter determines whether the application should wait for status to be returned for a call or continue on without knowing if the call completed correctly. The method returns a `Collection` of the objects retrieved from the database. A `MalformedQueryException` is thrown when the query formed by the filter is not correct.

Some exceptions are common to all methods in the API. A `ConnectionException` is thrown if the underlying database cannot be found. An `InvalidRequestException` is thrown if a particular method is not valid for a particular DataServer. This may occur if an underlying database is read-only and an insert method is called. A `DatabaseException` is thrown if there is a general problem with the database. A `DeadlockException` is thrown when the method is chosen as a deadlock victim. An `AbortRequestException` is thrown when a method has aborted. This may happen for a request to get all the data in a really large table, as it would tie up the database. A `RemoteException` is thrown when an error related to the communications protocol occurs.

5.2 DataServer

This is the middle tier of Data Services. The DataServer abstracts storing and retrieving a specific type of data to and from a database.

```
public interface DataServer
{
  Collection getKeys(Integer numOfKeys)
                     throws DataServerException,
                           RemoteException;

  void insertObject(<DataType> object)
                    throws DataServerException,
                          RemoteException;

  void insertObjects(Collection objects)
                    throws DataServerException,
                          RemoteException;

  void updateObject(<DataType> object)
                    throws DataServerException,
                          RemoteException;

  void updateObjects(Collection objects)
                     throws DataServerException,
                           RemoteException;

  void removeObject(<DataType> object)
                    throws DataServerException,
                          RemoteException;

  void removeObjects(Collection objects)
                     throws DataServerException,
                           RemoteException;

  void removeObjectsByQuery(<Filter> filter)
                            throws DataServerException,
                                  RemoteException;

  Collection retrieveObjects(Collection objects)
                             throws DataServerException,
                                   RemoteException;

  Collection retrieveObjectsByQuery(<Filter> filter)
                                    throws DataServerException,
                                          RemoteException;
}
```

- ```
  Collection getKeys(Integer numKeys)
                     throws DataServerException,
                           RemoteException;
  ```
  This method returns unique keys that can be issued to objects that are going to be persisted. The `numKeys` parameter determines how many keys to get. The method returns a `Collection` of keys. A `KeyAllocationException` is thrown if there are not enough free keys to be allocated for the call.

- ```
  void insertObject(<DataType> object)
                    throws DataServerException,
                          RemoteException;
  ```

This method inserts an object of a particular type into the database. The `object` parameter is the object that is inserted. This object is of a particular data type, which is specified upon creation of the interface. A `DuplicateDataException` is thrown when an object's key already exists in the database. A `FileSystemFullException` is thrown when the disk housing the database is full and data cannot be stored.

- ```
  void insertObjects(Collection objects)
                     throws DataServerException,
                            RemoteException;
  ```
  This method inserts a collection of objects of a particular type into the database. The `objects` parameter is a collection of objects is inserted. A `DuplicateDataException` is thrown when an object's key already exists in the database. A `FileSystemFullException` is thrown when the disk housing the database is full and data cannot be stored.

- ```
  void updateObject(<DataType> object)
                    throws DataServerException,
                           RemoteException;
  ```
  This method updates an object of a particular type in the database. The `object` parameter is the object that is updated. This object is of a particular data type, which is specified upon creation of the interface. An `ObjectNotFoundException` is thrown when an object's key is not found in the database. A `FileSystemFullException` is thrown when the disk housing the database is full and data cannot be stored.

- ```
  void updateObjects(Collection objects)
                     throws DataServerException,
                            RemoteException;
  ```
  This method updates a collection of objects of a particular type in the database. The `object` parameter is the object that is updated. This object is of a particular data type, which is specified upon creation of the interface. An `ObjectNotFoundException` is thrown when an object's key is not found in the database. A `FileSystemFullException` is thrown when the disk housing the database is full and data cannot be stored.

- ```
  void removeObject(<DataType> object)
                    throws DataServerException,
                           RemoteException;
  ```
  This method removes an object of a particular type in the database. The `object` parameter is the object that is removed. This object is of a particular data type, which is specified upon creation of the interface.

- ```
  void removeObjects(Collection objects)
                     throws DataServerException,
                            RemoteException;
  ```
  This method removes a collection of objects of a particular type from the database. The `objects` parameter is the collection of objects that is removed.
  ```

- ```
  void removeObjectsByFilter(<Filter> filter)
                                 throws DataServerException,
                                        RemoteException;
  ```
  This method removes objects of a particular type from the database using filter criterion. The `filter` parameter is the object that determines which objects to remove. A `MalformedQueryException` is thrown when the query formed by the filter is not correct.

- ```
  Collection retrieveObjects(Collection objects)
                                 throws DataServerException,
                                        RemoteException;
  ```
  This method retrieves a collection of objects from the database based on unique ID's. The `objects` parameter is the collection of objects that is retrieved. The method returns a `Collection` of the objects retrieved from the database. An `ObjectNotFoundException` is thrown if the object to be retrieved is not found in the database.

- ```
  Collection retrieveObjectsByFilter(<Filter> filter)
                                         throws DataServerException,
                                                RemoteException;
  ```
  This method retrieves a collection of objects from the database based on filter criterion. The `filter` parameter is the object that determines which objects to retrieve. The method returns a `Collection` of the objects retrieved from the database. A `MalformedQueryException` is thrown when the query formed by the filter is not correct.

Some exceptions are common to all methods in the API. A `ConnectionException` is thrown if the underlying database cannot be found. An `InvalidRequestException` is thrown if a particular method is not valid for a particular DataServer. This may occur if an underlying database is read-only and an insert method is called. A `DatabaseException` is thrown if there is a general problem with the database. A `DeadlockException` is thrown when the method is chosen as a deadlock victim. An `AbortRequestException` is thrown when a method has aborted. This may happen for a request to get all the data in a really large table, as it would tie up the database. A `RemoteException` is thrown when an error related to the communications protocol occurs.
  ```

5.3 MessageDataServer

This is the middle tier of Data Services that used while using message based connectors. MessageDataServer is the same as DataServer except MessageDataServer uses DataServerCallback objects to return information.

```
public interface MessageDataServer
{
  void getKeys(Integer numKeys,
               DataServerListener listener)
               throws RemoteException;

  void insertObject(<DataType> object,
                    DataServerListener listener)
                    throws RemoteException;

  void insertObjects(Collection objects,
                     DataServerListener listener)
                     throws RemoteException;

  void updateObject(<DataType> object,
                    DataServerListener listener)
                    throws RemoteException;

  void updateObjects(Collection objects,
                     DataServerListener listener)
                     throws RemoteException;

  void removeObject(<DataType> object,
                    DataServerListener listener)
                    throws RemoteException;

  void removeObjects(Collection objects,
                     DataServerListener listener)
                     throws RemoteException;

  void removeObjectsByQuery(<Filter> filter,
                            DataServerListener listener)
                            throws RemoteException;

  void retrieveObjects(Collection objects,
                       DataServerListener listener)
                       throws RemoteException;

  void retrieveObjectsByQuery(<Filter> filter,
                              DataServerListener listener)
                              throws RemoteException;

}
```

- ```
  void getKeys(Integer numKeys,
               DataServerListener listener)
               throws RemoteException;
  ```

  This method returns unique keys that can be issued to objects that are going to be persisted. The `numKeys` parameter determines how many keys to get. The `listener` parameter is used to return a `Collection` of keys. The `listener parameter may`
  ```

also be used to throw a KeyAllocationException if there are not enough free keys to be allocated for the call.

- ```
  void insertObject(<DataType> object,
                    DataServerListener listener)
                    throws RemoteException;
  ```
  This method inserts an object of a particular type into the database. The object parameter is the object that is inserted. This object is of a particular data type, which is specified upon creation of the interface. The listener parameter is used to throw a DuplicateDataException when an object's key already exists in the database. The listener parameter also be used to throw a FileSystemFullException when the disk housing the database is full and data cannot be stored.

- ```
  void insertObjects(Collection objects,
                     DataServerListener listener)
                     throws RemoteException;
  ```
  This method inserts a collection of objects of a particular type into the database. The objects parameter is a collection of objects is inserted. The listener parameter is used to throw a DuplicateDataException when an object's key already exists in the database. The listener parameter may also be used to throw a FileSystemFullException when the disk housing the database is full and data cannot be stored.

- ```
  void updateObject(<DataType> object,
                    DataServerListener listener)
                    throws RemoteException;
  ```
  This method updates an object of a particular type in the database. The object parameter is the object that is updated. This object is of a particular data type, which is specified upon creation of the interface. The listener parameter is used to throw an ObjectNotFoundException when an object's key is not found in the database. The listener parameter may also be used to throw a FileSystemFullException when the disk housing the database is full and data cannot be stored.

- ```
  void updateObjects(Collection objects,
                     DataServerListener listener)
                     throws RemoteException;
  ```
  This method updates a collection of objects of a particular type in the database. The object parameter is the object that is updated. This object is of a particular data type, which is specified upon creation of the interface. The listener parameter is used to throw an ObjectNotFoundException when an object's key is not found in the database. The listener parameter may also be used to throw a FileSystemFullException when the disk housing the database is full and data cannot be stored.

- ```
  void removeObject(<DataType> object,
                    DataServerListener listener)
                    throws RemoteException;
  ```
  ```

This method removes an object of a particular type in the database. The `object` parameter is the object that is removed. This object is of a particular data type, which is specified upon creation of the interface.

- ```
  void removeObjects(Collection objects,
                     DataServerListener listener)
                     throws RemoteException;
  ```
  This method removes a collection of objects of a particular type from the database. The `objects` parameter is the collection of objects that is removed.

- ```
  void removeObjectsByFilter(<Filter> filter,
                             DataServerListener listener)
                             throws RemoteException;
  ```
  This method removes objects of a particular type from the database using filter criterion. The `filter` parameter is the object that determines which objects to remove. The `listener` parameter is used to throw a `MalformedQueryException` when the query formed by the filter is not correct.

- ```
  void retrieveObjects(Collection objects,
                       DataServerListener listener)
                       throws RemoteException;
  ```
  This method retrieves a collection of objects from the database based on unique ID's. The `objects` parameter is the collection of objects that is retrieved. The `listener` parameter is used to return a `Collection` of the objects retrieved from the database. The `listener` parameter is also used to throw an `ObjectNotFoundException` if the object being retrieved is not found in the database.

- ```
  void retrieveObjectsByFilter(<Filter> filter,
                               DataServerListener listener)
                               throws RemoteException;
  ```
  This method retrieves a collection of objects from the database based on filter criterion. The `filter` parameter is the object that determines which objects to retrieve. The `listener` parameter is used to return a `Collection` of the objects retrieved from the database. The `listener` parameter is also used to throw a `MalformedQueryException` when the query formed by the filter is not correct.

Some exceptions are common to all methods in the API and each method has a `listener` parameter that is used to throw exceptions. A `ConnectionException` is thrown if the underlying database cannot be found. An `InvalidRequestException` is thrown if a particular method is not valid for a particular DataServer. This may occur if an underlying database is read-only and an insert method is called. A `DatabaseException` is thrown if there is a general problem with the database. A `DeadlockException` is thrown when the method is chosen as a deadlock victim. An `AbortRequestException` is thrown when a method has aborted. This may happen for a request to get all the data in

a really large table, as it would tie up the database. A `RemoteException` is thrown when an error related to the communications protocol occurs.

5.4 DataServerListener

This interface is used while using message based connectors. It is used to return status and results for calls made on the MessageDataServer interface.

```
public interface MessageDataServerListener
{
  void postResults(Collection objects) throws RemoteException;

  void postStatus(DataServerException exception) throws RemoteException;
}
```

- `void postResults(Collection objects) throws RemoteException;`[REPEAT1]

  This method is used to pass data returned from the MessageDataServer. The `objects parameter` is the collection of objects being returned. A `RemoteException` is thrown when an error related to the communications protocol occurs.

- `void postStatus(DataServerException exception) throws RemoteException;`

  This method is used to return the status of a call that has executed on the MessageDataServer. The `exception parameter` is any exception returned in the event of an error. A `RemoteException` is thrown when an error related to the communications protocol occurs.

5.5 FilterConverter

This interface is used to convert filters into a query language used by the implementing DataServer code.

```
public interface <FilterConverter>
{
  <Query> convertTo<QueryLanguage>(Collection objects)
                                   throws DataServerException;

  <Query> convertTo<QueryLanguage>(<Filter> filter)
                                   throws DataServerException;
}
```

- `<Query> `**`convertTo<QueryLanguage>`**`(Collection objects) throws DataServerException;`
  This method converts the collection of objects into a query based on the objects' ids. The `objects` parameter is the collection of objects that is converted to a query. The function returns a `<Query>` object, which is used by the implementation to execute the query. A `MalformedQueryException` is thrown when the given collection of objects is not correct.

- `<Query> `**`convertTo<QueryLanguage>`**`(<Filter> filter) throws DataServerException;`
  This method converts the filter to the query language used by the underlying persistence API. The `filter` parameter is the filter that is converted. The filter is a specific type of filter associated with a data type. This filter type is specified when this interface is generated. The function returns a `<Query>` object, which is used by the implementation to execute the query. A `MalformedQueryException` is thrown when a given filter is not correct.

5.6 KeyGenerator

This interface is used to generate keys for objects.

```
public interface KeyGenerator
{
  Collection getKeys(int numKeys) throws DataServerException;
}
```

- `Collection` **`getKeys`**`(int numKeys) throws DataServerException;`
  This method returns a collection of keys. The `numKeys` parameter specifies the number of keys to return. This function then returns a `Collection` containing the keys. A `KeyAllocationException` is thrown if there are not enough free keys to be allocated for the application.

5.7 DataServerManagement

This interface is used to manage a DataServer. It is used to turn on or off logging and view statistics.

```
public interface DataServerManagement
{
  DataServerStatistics getStatistics() throws RemoteException;

  void setStatisticsLogging(boolean onOff) throws RemoteException;

  void setQueryLogging(boolean onOff) throws RemoteException;

  void setDebugLogging(boolean onOff) throws RemoteException;

  void setDataInLogging(boolean onOff) throws RemoteException;

  void setDataOutLogging(boolean onOff) throws RemoteException;

  boolean isStatisticsLoggingSet() throws RemoteException;

  boolean isQueryLoggingSet() throws RemoteException;

  boolean isDebugLoggingSet() throws RemoteException;

  boolean isDataInLoggingSet() throws RemoteException;

  boolean isDataOutLoggingSet() throws RemoteException;
}
```

- `DataServerStatistics` **`getStatistics`**`() throws RemoteException;`
  This method returns `DataServerStatistics`, which give performance numbers for the particular DataServer.

- `void` **`setStatisticsLogging`**`(boolean onOff) throws RemoteException;`
  This method turns statistics logging on or off. If statistics logging is turned on, the DataServer will keep track of statistics and place them in a DataServerStatistics object. This object can be retrieved using `getStatistics()`. The `onOff` parameter turns the statistics logging on if set to true and turns statistics logging off if set to false.

- `void` **`setQueryLogging`**`(boolean onOff) throws RemoteException;`
  This method turns query logging on or off. If query logging is turned on, the DataServer will print out queries to the screen before they are executed. The `onOff` parameter turns the query logging on if set to true and turns query logging off if set to false.

- `void` **`setDebugLogging`**`(boolean onOff) throws RemoteException;`
  This method turns debug logging on or off. If debug logging is turned on, the DataServer will print out stack traces and other debugging information to the screen.

The `onOff` parameter turns the debug logging on if set to true and turns debug logging off if set to false.

- `void` **`setDataInLogging`**`(boolean onOff) throws RemoteException;`
  This method turns data in logging on or off. If data in logging is turned on, the DataServer will print the data coming into the DataServer to the screen. The `onOff` parameter turns the data in logging on if set to true and turns data in logging off if set to false.

- `void` **`setDataOutLogging`**`(boolean onOff) throws RemoteException;`
  This method turns data out logging on or off. If data out logging is turned on, the DataServer will print the data returned by DataServer to the screen before returning it. The `onOff` parameter turns the statistics logging on if set to true, and turns statistics logging off if set to false.

- `boolean` **`isStatisticsLoggingSet`**`() throws RemoteException;`
  This method returns a `boolean`, which is used to determine whether statistics logging is turned on or off.

- `boolean` **`isQueryLoggingSet`**`() throws RemoteException;`
  This method returns a `boolean`, which is used to determine whether query logging is turned on or off.

- `boolean` **`isDebugLoggingSet`**`() throws RemoteException;`
  This method returns a `boolean`, which is used to determine whether debug logging is turned on or off.

- `boolean` **`isDataInLoggingSet`**`() throws RemoteException;`
  This method returns a `boolean`, which is used to determine whether data in logging is turned on or off.

- `boolean` **`isDataOutLoggingSet`**`() throws RemoteException;`
  This method returns a `boolean`, which is used to determine whether data out logging is turned on or off.

Each method throws a `RemoteException` when an error related to the communications protocol occurs.

5.8 DataServerStatistics

This is a class, which contains performance information about a DataServer.

```
public class DataServerStatistics
{
  public void addInsertStats(long numObjs, long time);

  public long getNumObjectsInserted();

  public long getTotalInsertTime();

  public void addUpdateStats(long numObjs, long time);

  public long getNumObjectsUpdated();

  public long getTotalUpdateTime();

  public void addRemoveStats(long numObjs, long time);

  public long getNumObjectsRemoved();

  public long getTotalRemoveTime();

  public void addRetrieveStats(long numObjs, long time);

  public long getNumObjectsRetrieved();

  public long getTotalRetrieveTime();

  public void clearStats();
}
```

- `public void` **`addInsertStats`**`(long numObjs, long time);`
  This method adds to the current statistics, the number of objects inserted and the time to insert the objects. The `numObjs` parameter is the number of objects inserted. The `time` parameter is the amount of time it took to insert the objects.

- `public long` **`getNumberObjectsInserted`**`();`
  This method returns a `long`, which is the number of objects inserted.

- `public long` **`getTotalInsertTime`**`();`
  This method returns a `long`, which is the total time in milliseconds it took to insert.

- `public void` **`addUpdateStats`**`(long numObjs, long time);`
  This method adds to the current statistics, the number of objects updated and the time to update the objects. The `numObjs` parameter is the number of objects updated. The `time` parameter is the amount of time it took to update the objects.

- `public long` **`getNumberObjectsUpdated`**`();`
  This method returns a `long`, which is the number of objects inserted.

- `public long` **`getTotalUpdateTime`**`();`
  This method returns a `long`, which is the total time in milliseconds it took to update.

- `public void` **`addRemoveStats`**`(long numObjs, long time);`
  This method adds to the current statistics, the number of objects removed and the time to remove the objects. The `numObjs` parameter is the number of objects removed. The `time` parameter is the amount of time it took to remove the objects.

- `public long` **`getNumberObjectsRemoved`**`();`
  This method returns a `long`, which is the number of objects removed.

- `public long` **`getTotalRemoveTime`**`();`
  This method returns a `long`, which is the total time in milliseconds it took to remove.

- `public void` **`addRetrieveStats`**`(long numObjs, long time);`
  This method adds to the current statistics, the number of objects retrieved and the time to retrieve the objects. The `numObjs` parameter is the number of objects retrieved. The `time` parameter is the amount of time it took to retrieve the objects.

- `public long` **`getNumberObjectsRetrieved`**`();`
  This method returns a `long`, which is the number of objects retrieved.

- `public long` **`getTotalRetrieveTime`**`();`
  This method returns a `long`, which is the total time in milliseconds it took to retrieve.

- `public void` **`clearStats`**`();`
  This method sets all the statistics back to zero.

5.9 DataServerPolicy

This interface is used to configure a DataServer.

```
public interface DataServerPolicy extends Policy
{
  public void setMapFile(String mapFile);

  public String getMapFile();

  public void setUserName(String userName);

  public String getUserName();

  public void setPassword(String password);

  public String getPassword();

  public void setKey(String key);

  public String getKey();

  public void setKeyGenerator(String keyGenerator);

  public String getKeyGenerator();

  public void setFilterConverter(String filterConverter);

  public String getFilterConverter();

  public void setDatabase(String database);

  public String getDatabase();

  public void setIDTable(String IDTable);

  public String getIDTable();
}
```

- `public void` **`setMapFile`**`(String mapFile);`
  This method has a `mapFile` parameter, which is used to set the location of the map file.

- `public String` **`getMapFile`**`();`
  This method returns a `String`, which is the location of the map file.

- `public void` **`setUserName`**`(String userName);`
  This method has a `userName` parameter, which is used to set the user name used to log into the database.

- `public String` **`getUserName`**`();`
  This method returns a `String`, which is the user name used to log into the database.

- `public void setPassword(String password);`
  This method has a `password` parameter, which is used to set the password used to log into the database.

- `public String getPassword();`
  This method returns a `String`, which is the password used to log into the database.

- `public void setKey(String key);`
  This method has a `key` parameter, which is used to set the authorization key for implementing software.

- `public String getKey();`
  This method returns a `String`, which is the authorization key for implementing software.

- `public void setKeyGenerator(String keyGenerator);`
  This method has a `keyGenerator` parameter, which is used to set the class used to generate keys.

- `public String getKeyGenerator();`
  This method returns a `String`, which represents the class used to generate keys.

- `public void setFilterConverter(String filterConverter);`
  This method has a `filterConverter` parameter, which is used to set the class used to convert filters to query languages used by the implementation.

- `public String getFilterConverter();`
  This method returns a `String`, which represents the class used to convert filters to query languages used by the implemenation.

- `public void setDatabase(String database);`
  This method has a `database` parameter, which is used to set the database to use.

- `public String getDatabase();`
  This method returns a `String`, which represents the database to use.

- `public void setIDTable(String IDTable);`
  This method has an `IDTable` parameter, which is used to set the ID table to use for generating keys.

- `public String getIDTable();`
  This method returns a `String`, which represents the ID table to use for generating keys.

5.10 DataServerException

This is a class is an exception, which is thrown when an error occurs in the DataServer. There are several classes that extend this class that address specific problems.

```
public class DataServerException extends Exception
{
  public DataServerException(Exception exception);

  public DataServerException(String message);

  public Exception getException();
}
```

- `public` **`DataServerException`**`(Exception exception);`
  This method is a constructor for `DataServerException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`DataServerException`**`(String message);`
  This method is a constructor for `DataServerException`. The `message` parameter sets the error message.

- `public Exception` **`getException`**`();`
  This method returns an `Exception` that is wrapped by this Exception.

5.11 AbortRequestException

This is a class is an exception, which is thrown when a request sent to the DataServer has been aborted.

```
public class AbortRequestException extends DataServerException
{
  public AbortRequestException(Exception exception);

  public AbortRequestException(String message);
}
```

- `public` **`AbortRequestException`**`(Exception e);`
  This method is a constructor for `AbortRequestException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`AbortRequestException`**`(String message);`
  This method is a constructor for `AbortRequestException`. The `message` parameter sets the error message.

5.12 ConnectionException

This is a class is an exception, which is thrown when there is a problem connecting to the underlying database.

```
public class ConnectionException extends DataServerException
{
  public ConnectionException(Exception exception);

  public ConnectionException(String message);
}
```

- `public` **`ConnectionException`**`(Exception e);`
  This method is a constructor for `ConnectionException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`ConnectionException`**`(String message);`
  This method is a constructor for `ConnectionException`. The `message` parameter sets the error message.

5.13 DatabaseException

This is a class is an exception, which is thrown when there is a general problem with a call to the underlying database.

```
public class DatabaseException extends DataServerException
{
  public DatabaseException(Exception exception);

  public DatabaseException(String message);
}
```

- `public` **`DatabaseException`**`(Exception e);`
  This method is a constructor for `DatabaseException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`DatabaseException`**`(String message);`
  This method is a constructor for `DatabaseException`. The `message` parameter sets the error message.

5.14 DeadlockException

This is a class is an exception, which is thrown when a deadlock has occurred.

```
public class DeadlockException extends DataServerException
{
  public DeadlockException(Exception exception);

  public DatabaseException(String message);
}
```

- `public` **`DeadlockException`**`(Exception e);`
  This method is a constructor for `DeadlockException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`DeadlockException`**`(String message);`
  This method is a constructor for `DeadlockException`. The `message` parameter sets the error message.

5.15 DuplicateDataException

This is a class is an exception, which is thrown while inserting an object containing a unique key which already exists in the database.

```
public class DuplicateDataException extends DataServerException
{
  public DuplicateDataException(Exception exception);

  public DuplicateDataException(String message);
}
```

- `public` **`DuplicateDataException`**`(Exception e);`
  This method is a constructor for `DuplicateDataException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`DuplicateDataException`**`(String message);`
  This method is a constructor for `DuplicateDataException`. The `message` parameter sets the error message.

5.16 FileSystemFullException

This is a class is an exception, which is thrown when the file system has no more space to store data.

```
public class FileSystemFullException extends DataServerException
{
  public FileSystemFullException(Exception exception);

  public FileSystemFullException(String message);
}
```

- `public` **`FileSystemFullException`**`(Exception e);`
  This method is a constructor for `FileSystemFullException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`FileSystemFullException`**`(String message);`
  This method is a constructor for `FileSystemFullException`. The `message` parameter sets the error message.

5.17 FilterConverterException

This is a class is an exception, which is thrown when there is a problem with the filter converter.

```
public class FilterConverterException extends DataServerException
{
  public FilterConverterException(Exception exception);

  public FilterConverterException(String message);
}
```

- `public` **`FilterConverterException`**`(Exception e);`
  This method is a constructor for `FilterConverterException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`FilterConverterException`**`(String message);`
  This method is a constructor for `FilterConverterException`. The `message` parameter sets the error message.

5.18 InvalidRequestException

This is a class is an exception, which is thrown to indicate the method called is not valid in that particular instance. For example, if a user calls insertObject on a read only system, this exception would be returned.

```
public class InvalidRequestException extends DataServerException
{
  public InvalidRequestException(Exception exception);

  public InvalidRequestException(String message);
}
```

- `public` **`InvalidRequestException`**`(Exception e);`
  This method is a constructor for `InvalidRequestException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`InvalidRequestException`**`(String message);`
  This method is a constructor for `InvalidRequestException`. The `message` parameter sets the error message.

5.19 KeyAllocationException

This is a class is an exception, which is thrown to indicate a problem generating a unique key.

```
public class KeyAllocationException extends DataServerException
{
  public KeyAllocationException(Exception exception);

  public KeyAllocationException(String message);
}
```

- `public` **`KeyAllocationException`**`(Exception e);`
  This method is a constructor for `KeyAllocationException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`KeyAllocationException`**`(String message);`
  This method is a constructor for `KeyAllocationException`. The `message` parameter sets the error message.

5.20 KeyGeneratorException

This is a class is an exception, which is thrown when there is a problem with the key generator.

```
public class KeyGeneratorException extends DataServerException
{
  public KeyGeneratorException(Exception exception);

  public KeyGeneratorException(String message);
}
```

- `public` **`KeyGeneratorException`**`(Exception e);`
  This method is a constructor for `KeyGeneratorException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`KeyGeneratorException`**`(String message);`
  This method is a constructor for `KeyGeneratorException`. The `message` parameter sets the error message.

5.21 MalformedQueryException

This is a class is an exception, which is thrown when a query sent to the DataServer has not been formed correctly.

```
public class MalformedQueryException extends DataServerException
{
  public MalformedQueryException(Exception exception);

  public MalformedQueryException(String message);
}
```

- `public` **`MalformedQueryException`**`(Exception e);`
  This method is a constructor for `MalformedQueryException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`MalformedQueryException`**`(String message);`
  This method is a constructor for `MalformedQueryException`. The `message` parameter sets the error message.

5.22 ObjectNotFoundException

This is a class is an exception, which is thrown when an object with a specific unique id is not found.

```
public class ObjectNotFoundException extends DataServerException
{
  public ObjectNotFoundException(Exception exception);

  public ObjectNotFoundException(String message);
}
```

- `public` **`ObjectNotFoundException`**`(Exception e);`
  This method is a constructor for `ObjectNotFoundException`. The `exception` parameter is an exception that is thrown by implementation that is wrapped by this exception.

- `public` **`ObjectNotFoundException`**`(String message);`
  This method is a constructor for `ObjectNotFoundException`. The `message` parameter sets the error message.

6. Tools

6.1 CodeGenerator

The CodeGenerator is an application that uses templates to create the type specific DataServers. This tool is very useful because most of the functionality inside a DataServer is not type specific, while the interface is type specific. So an implementer can use the same implementation for differing DataServers while still having type specific interfaces. The implementer can then make special changes to a specific DataServer if they are needed. Using templates should give the implementer of a DataServer at least a good portion, if not all of the DataServer code. The CodeGenerator should also be used to generate type specific interfaces, which besides the data types do not change.

6.1.1 Command Line Interface

Here is the command line interface to the CodeGenerator:

```
java [-cp classpath] com.mot.openwings.data.CodeGenerator file
```

Here is an example of a command that uses the interface:

```
java –cp mot_xml.jar:xml.jar:mot_data.jar
com.mot.openwings.data.CodeGenerator file:///d:/test/RedBlueDS.xml
```

6.1.2 XML and Template file formats

The CodeGenerator tag takes as a parameter the file to be generated, outputFile and the template file to use to generate the file, template. More than one CodeGenerator block can be used. Inside a CodeGenerator block are tags, which represent tags in the template. These tags will be replaced by the values specified.

Here is the format for the xml file that is used to generate code:

```
<ROOT>
<CodeGenerator template="filename" outputFile="filename">
       <tag1>value1</tag1>
       <tag2>value2</tag2>
        .
        .
        .
       <tagX>valueX</tagX>
</CodeGenerator>
<CodeGenerator template="filename" outputFile="filename">
        .
        .
        .
</CodeGenerator>
</ROOT>
```

Here is an example xml file that uses the format above:

```
<ROOT>
<CodeGenerator template="d:\templates\DataServerInterfaceTemplate.tmp”
outputFile="d:\source\com\mot\c4i\redblue\RedBlueDataServer.java">
      <package>com.mot.c4i.redblue</package>
      <interface>RedBlueDataServer</interface>
      <dataType>RedBlueForce</dataType>
      <filterType>RedBlueFilter</filterType>
</CodeGenerator>
<CodeGenerator template="d:\templates\TOPLinkKeyGeneratorTemplate.tmp"
outputFile="y:\source\com\mot\c4i\redblue\TLRedBlueKeyGenerator.java">
      <package>com.mot.c4i.redblue</package>
      <dataType>RedBlueForce</dataType>
      <keyImplementation>TLRedBlueKeyGenerator</keyImplementation>
</CodeGenerator>
</ROOT>
```

Here is the format of the template:

```
package <package>;

import java.util.Collection;
import java.rmi.RemoteException;
import net.openwings.data.*;

public interface <interface> //extends DataServerInterfaceTemplate
{
 public void insertObject(<dataType> object) throws RemoteException,
                                                    DataServerException;
   .
   .
   .
}
```

Here is what would result with using the xml file and template from above:

```
package com.mot.c4i.redblue;

import java.util.Collection;
import java.rmi.RemoteException;
import net.openwings.data.*;

public interface RedBlueDataServer //extends DataServerInterfaceTemplate
{
 public void insertObject(RedBlueForce object) throws RemoteException,
                                                    DataServerException;
   .
   .
   .
}
```

6.2 DataServerManager

The DataServerManager is used to configure and monitor DataServer's. The application looks for DataServerManagement interfaces published. It then uses these interfaces to configure logging and look at statistics.

6.2.1 Command Line Interface

Here is the command line interface to the CodeGenerator:

```
java [-cp classpath] com.mot.openwings.data.DataServerManager
```

Here is an example of a command that uses the interface:

```
java   -cp ow_data.jar;mot_data.jar;%COMPONENT_CLASSPATH%
       -Djava.security.policy=%SECURITY_POLICY%
       -Djava.security.manager=%SECURITY_MANAGER%
       -Dnet.jini.discovery.ttl=%JINI_DISCOVERY_TTL%
       -Djava.rmi.server.codebase="%LOCAL_CODEBASE% %COMPONENT_CODEBASE%"
       com.mot.openwings.data.DataServerManager
```

In the example above, variables with %'s around them are environment variables.

6.2.2 DataServerManager GUI

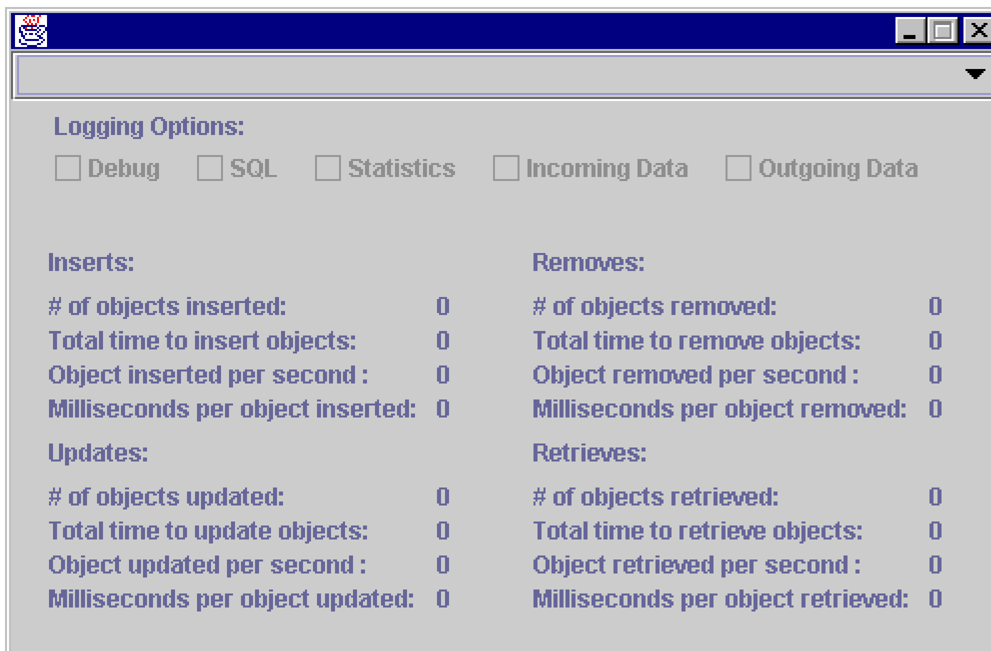

Figure 9: DataServerManager GUI

The drop box can be used to select a DataServer. The check boxes can be used to turn on the logging listed. If statistics is turned on the statistics for the DataServer selected will update every 15 seconds.

7. Examples

TBD

8. Compliance

The following rules are required for compliance to this specification:

#	Description
1	DataServers must pass the Data Service Validation Suite.
2	Compliant implementations must implement the interfaces specified in this document.

Figure 10: Compliance Table

9. References and Further Reading

Openwings Overview, http://www.openwings.org
Openwings Architecture White Paper, http://www.openwings.org
Openwings Component Services Specification, http://www.openwings.org
Openwings Policy Service Specification, http://www.openwings.org
Openwings Connector Services Specification, http://www.openwings.org
Openwings Security Specification, http://www.openwings.org
Openwings Naming Specification, http://www.openwings.org
Openwings Architecture Description Specification, http://www.openwings.org
Openwings Interface Definition Specification, http://www.openwings.org
Openwings Availability Specification, http://www.openwings.org
JDBC API and Enterprise Java Beans, http://www.java.sun.com, Sun Microsystems
ODMG Standard, http://www.odmg.org
TOPLink OR-Mapping Tool, http://www.objectpeople.com
Versant Object Database, http://www.versant.com
Sybase Relational Databases, http://www.sybase.com

[REPEAT1]Rose:COperation:MDLFilename=ebof.mdl,OperationID=374C2A350105