SAND2014-20568R

## US NDC Modernization

SAND-xxxx
Unclassified Unlimited Release
December 2014

# US NDC Modernization Iteration E1 Prototyping Report: Common Object Interface

Version 1.1

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.




SAND-xxxx
December 2014

# US NDC Modernization Iteration E1 Prototyping Report: Common Object Interface

Jennifer E. Lewis
Michael M. Hess

Version 1.11
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

## ABSTRACT

During the first iteration of the US NDC Modernization Elaboration phase (E1), the SNL US NDC modernization project team completed an initial survey of applicable COTS solutions, and established exploratory prototyping related to the *Common Object Interface (COI)* in support of system architecture definition. This report summarizes these activities and discusses planned follow-on work.

# REVISIONS

| Version | Date | Author/Team | Revision Description | Authorized by |
|---|---|---|---|---|
| 1.0 | 3/21/2014 | US NDC Modernization Team | Initial Release | M. Harris |
| 1.1 | 12/19/2014 | IDC Reengineering Team | IDC Release | M. Harris |
| | | | | |

# TABLE OF CONTENTS

# 1. OVERVIEW

The US NDC Modernization project statement of work identifies the definition of a modernized system architecture as a central project deliverable. As part of the architecture definition activity, the Sandia National Laboratories (SNL) project team has established an ongoing, software prototyping effort to support architecture trades and analyses, as well as selection of core software technologies.

During the first iteration of the US NDC Modernization Elaboration phase (E1), spanning Q1 - Q2 FY2014, the prototyping effort included initial COTS surveys and exploratory prototyping addressing three core elements of the system architecture:

1. The *Common Object Interface (COI)* provides the system and research tools with access to persistent data via an abstraction of the underlying storage solutions.

2. The *processing control framework* provides for the definition, configuration, execution and control of processing components within the system, supporting both automated processing and interactive analysis.

3. The *User Interface Framework (UIF)* provides a flexible platform for the definition of extensible graphical user interface (GUI) components & composition of GUI displays supporting users of the system and research tools.

This report summarizes the iteration E1 prototyping activities of the SNL project team *specific to the Common Object Interface*. E1 prototyping activities for the processing control framework and UI framework are described in separate reports.

# 2. SCHEDULE

This report summarizes the COI prototyping work completed during the three-month period from December 2013 to February 2014, based on the following schedule.

| Period | Activity |
|---|---|
| December 2013 | OSS/COTS survey |
| January – February 2014 | Initial Exploratory Prototyping |

# 3. MOTIVATION

Prototyping provides input critical in the definition of the system architecture, supporting selection of core software development languages and technologies, identification of architecture constraints & assumptions, and definition of high-level design patterns. In addition, the prototyping activity provides a foundation for development of the executable architecture deliverable.

# 4. COMMON OBJECT INTERFACE

## 4.1. Definition

The Common Object Interface (COI) is a core software mechanism providing access to persistent data within the system architecture. The COI provides an abstraction of the underlying data storage solutions, and includes the following major elements:

- An *application data model* encapsulating the physical representation of persistent data. All persistent data in the system will be accessed by the application software through the COI.
- An *Application Programming* Interface (API) providing Search/Create/Read/Update/Delete (SCRUD) functionality via the application data model for the system applications and research tools.

## 4.2. Design Goals

- Minimize dependencies between the system application/research tools and underlying data storage solutions.
- Decouple the application data model from the physical data model (e.g. DB schema).
- Provide a query language that is independent of the underlying data storage solution.
- Provide optimizations as needed to support system performance requirements – e.g. in-memory caching.

- Support the storage solutions defined for the modernized system.
- Support the application languages defined for the modernized system.

### 4.3. Constraints

- Performance: the COI must not slow down the Analyst's work nor introduce a bottleneck within automatic processing.
- Concurrency: the COI must support multiple, concurrent SCRUD operations against the underlying data storage solution(s).
- Interfaces with other frameworks: the COI must interface with the processing control framework as well as the GUI framework.
- Hardware: the solution must not be hardware dependent and must be able to run on a standalone or distributed system.
- COTS: Prefer Open Source Software (OSS) and other Commercial Off-The-Shelf (COTS) solutions to custom software development where available.
- Standards: Prefer solutions based on open standards wherever possible.

### 4.4. Iteration E1 Prototyping Activities

Iteration E1 prototyping activities focused on surveying COTS software solutions (principally open source software) addressing the requirements and constraints identified thus far for the COI. Candidate solutions were identified through online research into available COTS/open source projects and tools, and through discussions with other SNL project teams knowledgeable in COTS solutions for applications of similar scale.

Note that the survey results presented here are not exhaustive; they represent an initial effort constrained to the available E1 schedule and staffing resources. Identification and evaluation of candidate software solutions is intended to be an ongoing activity during the elaboration phase, as development of the architecture definition and executable architecture prototype progress.

Once these survey results were summarized and assessed, two COTS products, Hibernate and ODB, were selected for more thorough evaluation.

#### 4.4.1. Initial COTS Survey

A goal of the system prototyping effort is to leverage COTS products where possible. A range of software solutions exist that provide part of the functionality

needed by the COI, particularly within the ORM (Object Relational Mapping) domain.

ORM software attempts to overcome the object-relational impedance mismatch difficulties encountered when an object-oriented program is using data persisted in a relational database structure. This mismatch occurs when applications are using objects, which have both data and behavior, in conjunction with relational technologies focused on data storage and manipulation within a database [1, 5]. The lack of a direct mapping between these two technologies leads to difficulties when applications utilize both paradigms. While ORMs may not be able to handle all of these mapping related issues, they typically can handle 80-90% of them [2].

In addition, many of these products provide solutions for common database support needs including caching, transaction control, and concurrency management. While ORM software would not provide all of the functionality needed by the COI, it would provide a solid foundation that could be expanded to encompass system specific needs.

Currently, ORM solutions support multiple databases, but none of the surveyed ORM solutions support more than one language. We focused our survey on the solutions implemented in Java and C++.

See *Table 1. COI Survey Summary* in *4.6.Appendix A* for a listing of surveyed COTS solutions and summary of survey findings.

#### 4.4.1.1. Hibernate

Hibernate is the clear leader for ORM software implemented in Java. It has an engaged user community, is under active development (part of JBoss), and has extensive documentation available both via online communities and tutorials and in published literature.

Hibernate provides an abstraction of the underlying database schema through a framework for mapping an object-oriented domain model to a traditional relational database. These mappings can be specified either by using Java annotations or via XML. Because Hibernate supports reverse engineering, an initial mapping can be generated from an existing database schema that can then be modified to meet project needs moving forward.

Hibernate is a JPA (Java Persistence API) provider. This JPA compliance ensures adherence to this accepted standard and could ease future transition to other JPA providers, if needed.

Like most ORM software, Hibernate provides support for multiple database backend solutions. However, the only language Hibernate supports is Java. (cpphibernate purports to support a C++ interface to Hibernate, but it does not appear to have an active community.)

In addition to supporting mapping database objects to Java objects, Hibernate includes database transaction solutions such as transaction and concurrency control, batch processing, caching, locking, etc. Custom implementations of these solutions can dominate a project's development and maintenance schedule; receiving them as part of an ORM software package is an added advantage to using such a solution.

Hibernate provides its own query language, HQL (Hibernate Query Language). This allows users and applications with a standardized means of requesting data that Hibernate can convert into the correct "dialect" for the underlying database.

This rich feature set is what guided the COI prototyping team to explore Hibernate in more detail.

#### 4.4.1.2. Open JPA

OpenJPA is an ORM and JPA provider written in Java that supports multiple database backend solutions. Persistence metadata (mapping) is specified using Java 5 annotations, XML files, or both.

This project is part of the Apache Software Foundation, and appears to be under active development (latest software release: April 2013). However, OpenJPA's user community does not have the same level of engagement as Hibernate, and the number of related publications is considerably less (Open JPA is mentioned in sections in books as opposed to the entire focus of a book as is the case with Hibernate).

OpenJPA provides low-level access to the database through the use of SQL statements embedded in code. Given that this system modernization effort is attempting to move away from embedded SQL statements, the COI prototyping team did not select OpenJPA for further exploration as part of E1.

#### 4.4.1.3. Apache Cayenne

Apache Cayenne is an ORM that includes additional support for remote object persistence and/or object serialization. Cayenne is implemented in Java and supports multiple database backend solutions. Instead of using annotations or XML to manage the mapping from database data to Java objects, Cayenne uses a proprietary Cayenne Modeler. Cayenne also supports other database interaction features (e.g. querying, caching).

Apache Cayenne is under active development (latest release: February 2014), but it does not appear to have a substantial user base on par with Hibernate. Like OpenJPA, it does not have many publications devoted solely to its use, and its online documentation is unimpressive.

Because Apache Cayenne lacks the rich set of features found in Hibernate, the COI prototyping team opted not to explore Cayenne beyond the initial survey as part of E1.

#### 4.4.1.4. Apache Empire-DB

Another Apache ORM product is Empire-DB which is less of an ORM and more of a relationship database abstraction layer. This software uses dynamically-generated SQL as the cornerstone of its approach to modeling database entities in Java objects.

While this tactic can lead to highly efficient and optimized database interactions, it does not lend itself to a loose coupling between application code and the underlying database layer. This tight coupling, combined with this project's lack of active development, convinced the COI prototyping team not to explore this software beyond the initial survey as part of the E1 prototyping effort.

#### 4.4.1.5. Apache Torque

Apache Torque is an ORM written in Java. Instead of utilizing annotations or XML to define the mapping from database data to Java objects, Torque generates classes from an XML schema and DTD that describe the database schema. While this method allows Torque to support multiple database backend solutions, and avoids reflection, the documentation indicates that Torque performs best if no "exotic" features of the underlying database are used.

Torque's requirement that the domain model classes extend Torque specific classes introduces a tight coupling between Torque and the system within which it is used. This sort of coupling leads to difficulties if Torque must be replaced with a different ORM solution at a future time.

The intrusive nature of this software limits future development flexibility. The requirement of this tight coupling, combined with an apparent decline in interest based on internet trends, persuaded the COI prototyping team not to explore this software beyond the initial survey as part of E1.

#### 4.4.1.6. ODB

ODB, developed by Code Synthesis, is the clear leader for ORM software implemented in C++. It has an engaged user community, is under active

development, and has extensive documentation available online and in PDF format.

Used by companies such as Symantec, Intel, Lockheed Martin, et.al., ODB provides a C++ object to relational database mapping while requiring neither manually written code nor direct access to database tables, columns, or query language directly. These mappings are specified using ODB pragmas placed either inline in C++ header files or in separate header files making the object-relational mapping non-intrusive. ODB pragmas are simply a pragma-based language used to capture database-specific information about C++ types.

Like most ORM software, ODB provides support for multiple database backend solutions such as MySQL, SQLite, PostgreSQL, Oracle, and Microsoft SQL Server; and supports C++98/03 and C++11 standards. Software libraries, such as Boost and Qt, can be integrated allowing use of value types, containers, smart pointers, etc. However, the only language ODB supports is C++.

ODB supports both unidirectional (to-one, to-many) and bidirectional (one-to-one, one-to-many, and many-to-many) mappings between database tables and C++ objects. ODB also provides additional features such as transaction and concurrency control, caching of DB connections, prepared SQL statements, and buffers for improved performance. Custom implementation of these features, once again, provides an added advantage of using such a solution.

ODB provides use of its C++ integrated query expression and database-native SQL SELECT query expression allowing both users and applications a standardized means of requesting data from the underlying database(s).

This rich feature set is what guided the COI prototyping team to explore ODB in more detail.

#### 4.4.1.7. QxORM

QxORM is another ORM for the C++ language and supports multiple database backend solutions. Instead of using C++ pragmas to manage the mapping from database data to C++ objects, QxORM makes extensive use of C++ macros and templates.

QxORM is under active development (latest release: March 2013), but it does not appear to have as large a user base as ODB. Unlike ODB, QxORM's documentation is quite limited and its users are not listed on QxORM's website.

### 4.4.2. Exploratory Prototyping

For the exploratory prototyping phase of this work, the COI team delved more deeply into both Hibernate and ODB. The goal of this exploration was to learn more about these particular products, assess their utility for the system modernization effort, and gather information to aid in planning the path forward.

Hibernate and ODB are the clear leaders in ORM technology for Java and C++, respectively. They are under active development, have extensive documentation and engaged user communities. Both provide some of the capabilities needed within a COI. While all of the surveyed software provided multiple database backend support, no software was identified that provides support for more than one programming language.

For both Hibernate and ODB, the COI team wanted to assess the difficulty of mapping existing database tables to objects, particularly mapping multiple tables to a single object. This proved to be intuitive and straightforward using both Hibernate and ODB. For Hibernate, mapping using annotations were easier to use than XML mappings. For both explorations, data was mapped from classes to data in an Oracle database.

Based on the initial exploration, Hibernate and ODB were found to support many of the goals of the COI effort. Both decouple the application data model from the physical data model by providing classes that applications use to interact with the underlying data storage solution. While Hibernate provides a query language (HQL), ODB provides its `database` class' `query()` method, both of which are agnostic to the underlying database and can be used by both operational applications and research tools. Also, ODB offers integration with Boost C++ libraries in order to use their value types, containers, and smart pointers in persistent C++ classes. Hibernate and ODB purport to provide optimizations to support performance requirements (not investigated in E1), and support multiple storage solutions.

#### 4.4.2.1. Hibernate

##### 4.4.2.1.1. Installing Hibernate

Because Hibernate is written in Java, installation involved unzipping the Hibernate distribution zip file and including the Hibernate jar files on the Java classpath. The Hibernate software distribution included a variety of examples that were all configured to access an in-memory database. Thus, it was straightforward to verify the installation and begin to learn about basic Hibernate functionality.

#### 4.4.2.1.2. Connecting to the Oracle database

Hibernate manages the information needed to connect to the database via an xml configuration file (hibernate.cfg.xml). The tutorials included with Hibernate included this file; specifying the information needed to connect to the Oracle database involved copying and modifying the following entries in that file:

```
<!-- Database connection settings -->
<property name="connection.driver_class">oracle.jdbc.driver.OracleDriver
</property>
<property name="connection.url">jdbc:oracle:thin:@bambi:1521:kbdb</property>
<property name="connection.username">lewisje</property>
<property name="connection.password">******</property>

<!-- SQL dialect -->
<property name="dialect">org.hibernate.dialect.Oracle10gDialect</property>
```

Many other database configuration parameters can be set in this file, but these were the minimum needed to connect to the Oracle database.

#### 4.4.2.1.3. Mapping database data to Java objects

Hibernate supports two different ways to specify mappings between database data and Java objects: XML mappings and Java annotations.

##### 4.4.2.1.3.1. Mapping using XML

A benefit to XML mappings is that these mappings reside outside of the code for the Java object representing the database data. Given a small enough database schema change (e.g. renaming a column), the XML mappings could be updated and no code recompilation would be required. The downside is that the mappings can be difficult to generate, understanding those mappings requires an understanding of the associated Java objects, they must be kept in synch with the Java objects, and most changes to the underlying database schema will require changes to both the XML mappings and the Java.

To indicate that XML mappings are being used, information similar to the following must be included in the hibernate.cfg.xml file:

```
<!-- Mapping information -->
<mapping resource="usndc/coi/data/EventHypothesis.hbm.xml" />
```

The indicated EventHypothesis.hbm.xml file is where the mapping information resides.

#### 4.4.2.1.3.2. Mapping using Java annotations

Java annotations are a form of "syntactic metadata that can be added to Java source code" [6]. These annotations reside within the Java code for the Java classes that are mapped to the underlying database. This does require a recompile whenever changes to the database require changes to the mapping, but it produces clean, easy-to-understand code that only needs to be maintained in a single location.

To indicate that annotations are being used, information similar to the following must be included in the hibernate.cfg.xml file:

```
<!-- Mapping information -->
<mapping class="usndc.coi.hibernate.data.EventHypothesis"/>
```

The indicated class is where the mapping information resides.

#### 4.4.2.1.4. Mapping in practice

The primary goal of this phase of exploratory prototyping with Hibernate was to determine how to map multiple database tables to a single Java class. The Oracle database contained both an ORIGIN and ORIGERR table. Data from these tables was used to populate fields in an EventHypothesis class.

Below is an example of how this was accomplished using annotations:

```
@Entity
@Table(name = "origin")
@SecondaryTables(
    { @SecondaryTable(name = "origerr", pkJoinColumns =
    { @PrimaryKeyJoinColumn(name = "orid", referencedColumnName = "orid") }) })

public class EventHypothesis {
    @Id
    @Column(table = "origin", name = "orid")
    private long origin_id;
    @Column(table = "origin", name = "lat")
    private double latitude;
    @Column(table = "origerr", name = "smajax")
    private double semi_major_error_axis;
…
}
```

The Entity annotation indicates that this Java object will be persisted. The Table and SecondaryTables annotations indicate that this object will be populated with data from the origin table joined with the origerr table via the orid column. The

Id annotation indicates that the origin_id member variable is the id for this class, and the Column annotations are used to convey information about the columns in the database that will populate the corresponding data members.

This is a trivial example, but it illustrates the simplicity of mapping columns in database tables to elements in a Java class.

#### 4.4.2.1.5. Retrieving data

Now that a Java class exists to contain the data retrieved from the database, that data must be retrieved. Hibernate offers two ways to retrieve data from the database: querying via HQL (Hibernate Query Language) or building up Criteria objects. For this phase of the prototype exploration, HQL was used. This is how HQL is utilized within the Java code:

```
session.createQuery("from EventHypothesis where origin_id = 2719").list();
```

The SESSION variable above represents a Hibernate session, EVENTHYPOTHESIS is the name of a Java class, and ORIGIN_ID is the name of a variable within that class. A benefit of this HQL language is that class names and member variable names are specified, not database tables and columns, and it is fully object-oriented and understands notions like inheritance, polymorphism, and association. However, HQL has a SQL "feel" that is intuitive to users and developers accustomed to using SQL.

Hibernate uses the notion of SQL "dialects" in order to interface with different database implementations. By using HQL, the query code does not need to change whenever the underlying database changes since Hibernate handles the mapping between HQL and the appropriate dialect.

### 4.4.2.2. ODB

#### 4.4.2.2.1. Installing ODB

Because ODB is written in C++, installation involved untarring ODB's Common Core library, Oracle library, and ODB Compiler and building each one using its supplied Makefile. Installation also involved untarring Boost C++ source libraries to assess its seamless integration with ODB. The ODB software distribution included a variety of ODB examples highlighting database access, querying the database, using C++ containers comprised of database data, using inheritance, etc. Thus, it was straightforward verifying the installation and learning about basic ODB functionality once the supplied ODB examples were ultimately built and successfully ran against the Oracle database.

#### 4.4.2.2.2. Connecting to the Oracle database

ODB manages the information needed to connect to the database via a set of database parameters supplied at runtime on the command line (--user lewisje, etc.) or in a configuration file, such as:

```
<!-- Database connection settings -->
<property name="user">lewisje</property>
<property name=" password">******</property>
<property name=”service”>kbdb.world</property>
<property name=”host”>bambi.sandia.gov</property>
<property name=”port”>1521</property>
```

There are other available database parameters, but these were the minimum needed to connect to the Oracle database.

#### 4.4.2.2.3. Mapping database data to C++ objects

ODB makes use of its pragma-based language to capture database-specific information about C++ data types. The ODB compiler, not to be confused with the C++ compiler, uses these pragmas to automatically generate the C++ code that performs the conversion between persistent classes and their database representation. In the nominal case, a C++ class is manually created containing pragmas used to generate the C++ conversion code and database SQL TABLE definition. However, when a database SQL TABLE definition already exists, a C++ class is manually created, but must conform to the existing database representation. That is, each of the C++ class' attribute data types must conform to the SQL TABLE's attribute types. For reference, one must consult ODB conversion tables mapping RDBMS vendor-specific table attribute types to C++ data types.

[Note: The ODB compiler is a C++ compiler except that it produces C++ source code in lieu of assembly or machine code. As a result, the ODB compiler is capable of parsing any standard C++ code. ]

#### 4.4.2.2.4. Mapping using pragma language

ODB's pragma language has the following syntax:

```
#pragma db qualifier [specifier specifier …]
```

The `qualifier` tells the ODB compiler what kind of C++ construct this pragma describes. For instance, a pragma with the `object` qualifier describes the persistent object type. That is, it tells the ODB complier the C++ class it describes is a persistent class. The `specifier`, on the other hand, informs the ODB

compiler about a particular database-related property of the C++ declaration. For instance, the `member id` specifier tells the ODB compiler that this member contains the object's identifier.

ODB's pragma language provides other qualifiers and specifiers not included herein, but may be referenced within *C++ Object Persistence with ODB* manual.

#### 4.4.2.2.5. Mapping in practice

To demonstrate ODB's pragma language in our first example, Figure 1 – Origin Class exhibits a C++ persistent class, representing the `origin` database table containing two pragmas: 1) `#pragma db object`; and 2) `#pragma db id`.

```
// origin.hxx
//

#include <odb/core.hxx>

#pragma db object
class origin
{
   origin () {}

   #pragma db member id
   unsigned int orid;

   std::string algorithm;
   std::string auth;
   signed int commid;
   .
   .
};
```

Figure 1 – Origin Class

The above example can easily be extended in the event `origin` has relationship(s) with other persistent objects (i.e., unidirectional, bidirectional). For instance, as seen in Figure 2 – Origerr Class `origin` now has an association with an `origerr` C++ persistent class representing the `origerr` database table.

```
// origin.hxx
//

#include <odb/core.hxx>
```

```
class origerr;

#pragma db object
class origin
{
   origin () {}

   #pragma db member id
   unsigned int orid;

   std::string algorithm;
   std::string auth;
   signed int commid;
   .
   .
   #pragma db not_null
   shared_ptr<origerr> origin_origerr;
};
```

Figure 2 – Origerr Class

The only notable line in Figure 2 – Origerr Class is the `#pragma db not_null` which tells the ODB complier the `shared_ptr` always points to a valid object.

To compile the `origin.hxx` we created above and generate the support code for the Oracle database, we invoke the ODB compiler which produces three C++ files: origin-odb.hxx, origin-odb.ixx, and origin-odb.cxx. These files typically contain types and functions used to support the object-relational mapping and are not directly used. Rather, origin-odb.hxx is included in C++ files where one is performing database operations with classes from origin.hxx as well as compiling origin-odb.cxx and linking the resulting object file to your application.

To demonstrate ODB's pragma language in our second example, we create an ODB view which is a C++ class embodying one or more database tables which typically includes a subset of data members from each table. For instance, to extend our `origin` example seen in Figure 1, an `event_hypothesis` is created consisting of data members from both `origin` and `origerr` database tables as can be seen in Figure 3 – Event Hypothesis Class.

```
#pragma db view object(origin) object(origerr)
class event_hypothesis
{
   event_hypothesis () {}

   unsigned int orid;
   float lat;
   .
   .
   #pragma db column(origerr::commid)
   signed int commid;
   float smajax;
   .
   .
};
```

Figure 3 – Event Hypothesis Class

The complete syntax of the `db view object` pragma is shown below:

object(name [: join-condition])

The `name` is the qualified persistent class name that has been defined previously. The optional `join-condition` provides criteria used to associate a table with any of the previously associated tables. If left unspecified, the ODB compiler tries to come up with a join condition automatically.

Finally, the `db column` pragma is used to disambiguate data members with the same name.

#### 4.4.2.2.6. Retrieving data

ODB offers two means to retrieve data from the database using: 1) ODB C++ `database` class `query()` method; and 2) native SQL `SELECT` statements. For this phase of the prototype exploration, the `query()` method was used. This is how the `query()` method is utilized within the C++ code:

```
typedef odb::query<origin> query;
typedef odb::result<origin> result;
.
.
transaction t (db->begin());

result r (db->query<origin> (query::orid == 1224466));

t.commit();
```

The two `typedefs` create convenient aliases for two template instantiations used throughout the source code. The first is the query type for the `origin` objects and the second is the `result` type for that query.

A new `transaction` begins and the `query()` database method is invoked by providing a query expression `(query::orid == 1224466)` which limits the returned objects to one `origin` whose `orid` is `1224466`. The transaction is closed by calling commit.

A benefit of using the `database` class `query()` method is that class names and member variable names are specified, not database tables and columns. And it provides users and developers an intuitive member attribute - operator - expression notation.

### 4.5. Conclusions

This prototyping effort focused on surveying existing COTS/OSS software solutions that support the COI's need to meet the stated design goals. A range of database abstraction solutions were surveyed, and ORM software seems to be the best fit for the COI needs, specifically the Hibernate and ODB ORMs.

ORMs provide a decoupling of the application data model from the physical data model through a mapping mechanism. Because applications go through the COI to interact with the database, changes to the underlying database can be handled through the COI without affecting the applications. This minimizes the dependencies between the applications and the underlying database solution, which eases maintenance of the application code and provides freedom to optimize the database implementation without fear of breaking existing functionality.

ORMs also provide optimizations to support performance requirements associated with the retrieval and storage of data. This removes much of the overhead of maintaining these optimizations for each database solution.

Lastly, Hibernate and ODB provide a standardized query language that can be used by applications regardless of the underlying data storage implementation. This provides flexibility should the underlying data storage solution need to be modified in the future.

### 4.6. Follow-On Work

Given what's been learned in E1, these are some of the areas to be explored as part of the follow on work in the E2 iteration.

If possible, the E2 iteration should focus on a single ORM solution, to aid in more fully understanding the features provided by the chosen solution.

The focus of the COI prototyping effort for the E2 iteration within elaboration should continue to explore ways to support the COI component of the executable architecture deliverable. This includes examining how waveform data could be handled by the COI if an ORM is leveraged as well as how the COI will manage notifying subscribers of changes to data.

Investigating an ORM solution interfaces with more than one database is crucial to understanding how well the ORM can support multiple database vendors simultaneously. In addition, it could be beneficial to explore if it's possible to support more than one database schema at a time.

While an ORM solution might be implemented in a single language, it is worth investigating how an ORM could facilitate accessing data via candidate scripting languages (e.g. Python, Perl).

Performance assessments that compare an ORM solution to embedded SQL are needed. In addition, existing ORM support for other features of the COI (e.g. managing concurrent access, caching, transaction and connection management, etc.) merit further investigation.

Lastly, data provenance is a new addition to the modernized system that will need to be supported by the COI. Further investigation into solutions that support the persistence of provenance related data should be a part of future iteration efforts.

# REFERENCES

1. *The Object-Relational Impedance Mismatch*, Agile Data (http://www.agiledata.org/essays/impedanceMismatch.html)

2. *Orm Hate*, Martin Fowler (http://martinfowler.com/bliki/OrmHate.html)

3. *The Design of a Robust Persistence Layer for Relational Databases*, Scott W. Ambler (http://www.ambysoft.com/downloads/persistenceLayer.pdf)

4. *Encapsulating Database Access: An Agile 'Best Practice'*, Agile Data, *(*http://www.agiledata.org/essays/implementationStrategies.html)

5. *Object-relational impedance mismatch*, Wikipedia, http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

6. *Java annotation*, Wikipedia (http://en.wikipedia.org/wiki/Java_annotations)

This page intentionally left blank.

# APPENDIX A. SUMMARY OF INITIAL SURVEY

**Table 1. COI Survey Summary**

| Candidate Solution | Solution Type | URL | Summary Assessment |
|---|---|---|---|
| Hibernate | Java Object Relational Mapping (ORM) OSS | http://www.hibernate.org/orm | Advantages: Leading ORM candidate for Java. Hibernate Query Language (HQL) could provide both application and researcher level access to underlying COI objects. JPA provider.<br>Disadvantages: A dependence on HQL could introduce a tight coupling. |
| Open JPA | Java ORM OSS | http://openjpa.apache.org/ | Advantages: JPA provider.<br>Disadvantages: ORM features supported through embedded SQL. Not a prevalent software solution. |
| Apache Cayenne | Java ORM OSS | http://cayenne.apache.org/ | Advantages: Supports Remote Object Persistence<br>Disadvantages: CayenneModeler required for mapping. Not a prevalent software solution. |
| Apache Empire-DB | Java RDBMS Abstraction OSS | http://empire-db.apache.org/ | Advantages: Database interactions more easily optimized since interactions are at such a low level.<br>Disadvantages: Database abstraction layer (not an ORM). SQL-centric. Not a prevalent software solution. |
| Apache Torque | Java ORM OSS | http://db.apache.org/torque/torque-4.0/index.html | Advantages: Uses XML that describes the database schema, which avoids reliance on reflection.<br>Disadvantages: Requires that domain model extend Torque specific classes. Not a prevalent software solution. |
| ODB | C++ ORM OSS | http://codesynthesis.com/products/odb/ | Advantages: Leading ORM candidate for C++. Does not require manual entry of mapping code.<br>Disadvantages: Manufactured by Code Synthesis, located in South Africa. Does not provide C++ object to relational database mapping for existing DB tables. |
| QxORM | C++ ORM OSS | http://www.qxorm.com/ | Advantages: Supports object relational mapping with MySQL, SQLite, PostgreSQL, Oracle, and SQL Server databases.<br>Disadvantages: Market usage is unknown and documentation is limited. |

This page intentionally left blank.

This is the last page of the document.