Abstract

Java Object XML Mapping (JOXM) is a library that supports the automated persistence and querying of Java objects in a native XML database. The goal of this library is to provide a suitable alternative to standard object-relational mapping (ORM) tools, most notably Hibernate [1]. Unlike techniques such as XML-relational persistence, the storage mechanism provided by JOXM is transparent, allowing the developer to store, retrieve, and query typed Java objects as opposed to plain XML data.

Introduction

Hibernate [1] is one of the most popular ORM technologies for interacting with relational databases. Its popularity derives from its simplicity and abstraction over the interactions with the database. Hibernate relieves developers from writing SQL queries and needing to understand the details of converting data from an in-memory object model to the relational database model. One such difficulty is type conversion, mapping programming language types to SQL data types. Hibernate additionally provides its own querying language, the Hibernate Query Language (HQL), that abstracts the standard SQL querying mechanism so that querying is performed on objects rather than tables or relational data. Along with its other rich features, Hibernate allows developers to write their persistence code once without referring to a specific back-end database implementation.

Unlike relational databases, native XML databases are a relatively new and underdeveloped field. Much knowledge of their practicality, performance, and applicability in the real world remains undecided. A native XML database is one which defines a logical model for an XML document, uses XML as its underlying method of storage, and is not tied to any particular underlying storage model. There are three common approaches to handling XML data in a XML database application:

1. Place XML data directly into a relational database (e.g. blob fields)
2. Convert XML data model into a relational model (e.g. edge method)
3. Store data in a native XML database as indexed content

Strangely enough, neither one of these approaches seems to have more favor than the other in industry. The lack of sophisticated and automated tools for storing and processing XML data in databases seems to stifle its adoption in application development. The second approach has been taken by Hibernate in its implementation of XML-relational mapping. Despite Hibernate’s support for XML persistence, the following has been noted about its relatively new feature:

"[The] Java community now has a framework that provides an efficient and consistent method for effortless OR and XML persistence. With that in mind, understanding the Hibernate project’s mission is important. Even though the Hibernate 3 XML features are highly useful and attractive, they are not meant to replace the most popular XML marshalling or transformation frameworks. Despite a very comprehensive OR mapping solution, Hibernate is not expected to grow into a major XML manipulation framework (per Hibernate author Gavin King, TheServerSide Java Symposium 2005). For that reason, you should take the XML persistence features as helpful extensions to the already powerful Hibernate framework, which enable you to easily incorporate another contemporary data representation mechanism into your application. If you have to deal with intricate integration and transformation scenarios, however, look into XML-specific frameworks." [2]

As can be seen, Hibernate’s XML-relational mapping technology is not meant to manipulate or transform (update) XML. On the other hand, Hyperjaxb [3], like Hibernate, addresses the need to store XML data into a relational database. Hyperjaxb implements the Java Architecture for XML Binding (JAXB) [4] specification and provides a mechanism to marshal and unmarshal XML content in and out of a relational database. However, this technology relies on compile-time code generation and adherence to the JavaBean object model. Worse yet, like its counterpart in Hibernate, neither of these approaches allows the abstraction of storage model (XML) from the object model (Java objects).

Only one tool at the current time seems to implement the third approach, JaxMe [5]. JaxMe attempts to fill this purpose by implementing the JAXB specification and marshaling and unmarshaling to a XML database. However, this project has the following disadvantages discouraging its use:

- Unstable (currently on 0.5.2 release)
- Requires pre-known schema definitions to generate JAXB-enabled objects
- Built with JAXB in mind, XML databases were an afterthought
- Fails to abstract persistence layer from object model
- Very little support, documentation, and active development
- Sophisticated and complex API
- Does not support issuing XQuery or XPath queries
JaxMe is an open-source project that has yet to gain support or provide adequate documentation to support or advertise its use by developers. The latest update to the project was nearly a year ago, with their latest non-maintenance release being released two years ago. Additionally, this library was not originally created with a native XML database in mind, so support for XQuery \[6\] and XPath \[7\] has not been implemented, nor do they implement or make use of the XML:DB API.

The XML:DB API \[8\] is standard specification for providing a way of gaining access to data in an XML database. XML:DB can be viewed as being the XML equivalent to ODBC (for relational databases), and its use is promoted by the XML:DB Initiative. The XML:DB API specification is currently in a working draft and defines two levels of conformance, core level 0 and core level 1. The core level 0 is the base API which provides for the concepts of resources and services. Core level 1 consists of the specification for XPath querying services. Natively, the XML:DB API specification does not allow for querying typed objects. Rather, the specification provides return types of queries in the form of DOM, SAX, and text as well as binary content.

**Problem Statement**

A library sufficient to support the automated storage, retrieval and querying of typed Java objects in a native XML database has yet to be developed and documented.

**Implementation**

The Java Object XML Mapping (JOXM) library augments the previously described methods of handling XML data in an application by creating an automated persistence layer to persist Java objects directly to a native XML database. JOXM is an implementation of object-XML mapping (OXM) for the Java language. JOXM’s goal are the following:

- Concise connection, persistence, and querying APIs
- Connection to local databases through the XML:DB API
- Persistence API that abstracts XML data binding (marshaling and unmarshaling) and querying
- Persistence of any Java object into a native XML database
- Support for issuing XPath queries, returning results as typed Java objects

JOXM is a general purpose persistence library, with goals similar to those of Hibernate but placed in the context of native XML databases. JOXM provides a high-level abstraction of Java object persistence, with a "hands-off" approach in implementing a persistence layer. Knowledge of the connection protocol, storage format, and XML querying (XPath/XQuery) syntax is not required. As far as users of this library are concerned, they are saving Java objects to a database, which happens to use XML as its storage format.
Overview

The JOXM core provides the persistence, connection, and querying APIs that bind an application to the XML persistence model. Instead of the application needing to maintain intimate knowledge about the location or protocols for communicating with the database, it can communicate naively through the JOXM proxy. JOXM connects to a local/embedded database using the XML:DB API, ensuring that the implementation of the XML database need only to adhere to the XML:DB specification. This layer of abstraction allows the application to swap the back-end at any time. Besides support for connection management and transparent persistence, JOXM allows developers to issue XPath queries across its interface, with automated type conversion provided by default.

Detailed Design

The architecture of the JOXM library is composed of several modular components, as illustrated in Figure A. The JOXM library has one main package, edu.cp.joxm, composing two subpackages: edu.cp.joxm.annotations and edu.cp.joxm.aspects. The main package contains four classes designed to provide the core interfaces: (1) connection, (2) persistence and (3) querying.

Figure A: JOXM Design and Architecture

Connections are established with a database by the JOXM class. The sole purpose of this
class is to create sessions, which are stable connections to the database, or more specifically, resources (i.e. collections) in the database. A Session object is instantiated as a result of calling the createSession method of the JOXM class, documented below:

```java
/**
 * Creates a session to a database with the given name, located on the host, associated with the collection specified. The three parameters are used to create a URI for connecting to a database instance and collection in the following format: <dbName>://db/<host>/<collection>. If either the database or the collection to do not exists, they will be created automatically.
 * @param host the name of those host machine in which the database resides
 * @param dbName the name of the database
 * @param collection the path specifying the location of the collection, relative the collection root of the database
 * @return Session that is association with the specified collection of resources in the database
 */
public static Session createSession(String dbName, String host, String collection)
```

A Session object is not associated with a database, but more precisely, a collection of documents in the database. This means that more than one Session can be created to a single database instance concurrently, so long as they are associated with different collections (see Future Work for further discussions of enhancing this capability). The resulting Session object provides the users of this library the persistence and querying API, through the following methods (described by the Javadoc comments):

```java
/**
 * Persists the contents of the given object to the collection associated with this session (semantically equivalent to a relational insert)
 * @param obj the object to persist to the database
 */
public void save(Identifiable obj)

/**
 * Updates (replaces) the contents of the given object in collection associated with this session (semantically equivalent to a relational update). This call effectively synchronizes the state of the given object in memory with the state of the object in the database collection.
 * @param obj the object to update in the database
 */
public void update(Identifiable obj)

/**
 * Deletes (removes) the contents of the given object from the collection associated with this session (semantically equivalent to a relational delete).
 * @param obj the object to remove from the database
 */
```
public void delete(Identifiable obj)

/**
 * Deletes (removes) all of the content from the collection associated with
 * this session (semantically equivalent to a relational "delete * from
 * table")
 */
public void deleteAll()

/**
 * Execute the given XPath query and returns back the results as
 * typed objects with the type T
 * @param <T> runtime type of objects being queried (matches the 'type'
 * argument)
 * @param type the type of objects being queried
 * @param xpath the XPath query to execute
 * @return list of typed objects returned from the query
 */
public <T> List<T> executeXPathQuery(Class<T> type, String xpath)

/**
 * Closes the session and shuts down the connection to the database nicely
 */
public void close()

The combination of the save, update, delete and deleteAll method provide the core persistence API. The deleteAll method is provided for pure convenience, and is useful when the entire state of a collection should be removed. The persistence API works on an Identifiable, which is simply an interface that ensures objects can provide the persistence API identification information.

/**
 * Interface specifying that an object is identifiable by an ID
 * conforming to the 'name' production of the XML 1.0/1.1 specifications.
 */
public interface Identifiable {

/**
 * Returns a XML identifier that can identify this object
 * @return String XML identifier
 */
public String getXmlID();
}

The Identifiable interface is required by the persistence API. However, the JOXM library does not require the user application to manually implement this interface in all of its types. Using Aspect Oriented Programming techniques, the user must only mark the type with the @JOXMEntity. The Identifier aspect injects the Identifiable interface on all those annotated types, ensuring that they automatically implement that interface. The Identifiable interface is injected into classes that need to be persisted by means of Aspect Oriented Programming. The AOP in JOXM is facilitated by compile-time weaving provided by the AspectJ project [9]. The @JOXMEntity annotation is no
more than a marker used to identify types that may be used in the JOXM persistence API. Any class annotated by the JOXMEntity annotation are consequently advised by an aspect to inject the Identifiable interface. The Identifier aspect uses UIDs (unique identifiers) to identify the objects. UIDs are, for most practical concerns, unique across all virtual machines on a single host, as noted by the Java 1.5 Specification:

"An independently generated UID instance is unique over time with respect to the host it is generated on as long as the host requires more than one millisecond to reboot and its system clock is never set backward." [10]

The UIDs ensure that any object (i.e. XML element) placed in the database can be retrieved by id, rather than by location, which has benefits to both simplicity, querying, and performance (see XQuery Translations).

Although the Session is associated with a collection of documents, all operations invoked through the persistence API are conducted on a single document. Future implementations may expand the capability to specify more fine-grained control over the storage location (see Future Work). However, such functionality may violate the goal of abstracting the storage model from the user, as such specification would be tailored to the back-end implementation.

The Session makes use of two major components when persisting and retrieving objects to and from the database: the Marshaller and the database itself. This version of JOXM is based on the eXist native XML database, described in further detail in the following section. The Marshaller class provides all the functionality to marshal and unmarshal objects to and from the database. Using the XStream serialization library, discussed in a later section in further detail, the Marshaller component provides two core methods:

```java
/**
 * Marshalls the given object into an XML string
 * @param obj the object to marshall
 * @return String XML representation of the given object
 */
public static String toXML(Identifiable obj)

/**
 * Unmarshalls the given XML string into an object of
 * the given type
 * @param <T> the type of the object being unmarshalled to
 * @param type the type instance specifying the desired object's type
 * @param xml String XML representation of the unmarshalling object
 * @return typed instance of an object created by unmarshalling
 * the XML data into an object
 */
public static <T> T fromXML(Class<T> type, String xml)
```
The `toXML` method takes any `Identifiable` object and marshals it to a XML string. Conversely, the `fromXML` method takes a XML string and unmarshals it to a Java object. In the case of `fromXML`, context regarding the type of the object being unmarshaled is required, provided by the `Class` type argument. In order for the XStream library to be aware of how to marshal and unmarshal types, they must be registered at run-time dynamically. This is the purpose of the `registerType` method, detailed below:

```java
/**
 * Registers the given type with the marshaller so it can be
 * marshalled / unmarshalled with the {@link #toXML(Identifiable)} and
 * {@link #fromXML(Class, String)} methods respectively. This must
 * be called before objects of the given type are serialized.
 * @param type the class type to register
 */
public static void registerType(Class<?> type) throws Exception
```

Consequently, the application using the JOXM library must invoke the `registerType` method for all types that need to be marshaled or unmarshaled and stored in the database. Built-in types such as lists, maps, sets, primitives, numeric objects and strings need not be registered as they are handled by default. Only user defined types need be registered with the `Marshaler` class. A tool such as Spring Framework [16] could reduce the effort to register the types (see the Future Work section).

**eXist**

The JOXM library, in its current version, stores its data in an eXist native XML database [19]. eXist was chosen because it provides an open-source, Java-based, native XML database that integrates several popular XML technologies, including the XQuery and XUpdate languages. eXist also supports both and embedded and server mode, although only the prior is supported by the current version of JOXM. eXist conforms to the XML:DB API, using the Xindice [11] implementation, though it also offers a REST-style API to access the database through HTTP. eXist also provides the following advantageous features:

- Written in Java (portable), providing a seamless interface and driver for Java applications
- Some support for transaction management (enough to support crash recovery, read-only transactions bypass internal transaction management), though not yet full ACID compliant
- Uses an indexing scheme that is intended to maximize the efficiency of searching through an XML tree
- Support for XPath and XQuery updates (XUpdate)
XQuery Translations

The Session class uses XQuery, along with the Marshaller, to store and retrieve data from the database. This requires creating a translation from a logical operation, such as save, update or delete, to a database query. However, there does not currently exist a standard mechanism for dynamically modifying the contents of a document stored in a database through querying. The XQuery Update Facility (XQUF) [12] and XUpdate [13], both still in working drafts, have yet to be standardized. Since eXist was chosen as the back-end for the JOXM library and it currently supports XUpdate, XUpdate will be used by the persistence API to create data manipulation queries. However, this does not prohibit or exclude enhancements to support XQUF in the future, as there is an abstract layer between the database and JOXM, i.e. the Session class. Supporting XQUF in the future would simply require subclassing Session and overriding all the persistence API methods, a fairly simple task (see Future Work section).

The following list describes the translation logic used to generate XUpdate queries for each of the persistence API methods:

- **save:**
  
  update insert <xml> into //<object_type>[@id="<object_id>"]

- **update:**
  
  update replace //<object_type>[@id="<object_id>"] with <xml>

- **delete:**
  
  update delete //<object_type>[@id="<object_id>"]

- **deleteAll:**
  
  update delete //<object_type>

In each case, <xml> refers to the marshaled XML provided by the Marshaller class for the object being persisted. Each XUpdate query uses an XPath expression to determine the element under modification. In the case of save, update, and delete, the XPath query simply describes the path to the element that is being operated on, using the previously mentioned UID. The <object_type> is the simple name (i.e. class name) of the object, which matches the name of the element by default when marshaled and unmarshaled by the Marshaller. The deleteAll method is provided as a convenience to remove all objects from a collection, most useful when the user wants to clear an entire collection.
XStream

XStream [14] is a simple Java library that can be used to serialize objects to XML and back again. Unlike the JAXB interface which requires objects to be JavaBeans, XStream allows for the marshaling and unmarshaling of any Java object. XStream relies heavily upon Java annotations to provide the metadata in order to influence and configure the conversion to and from XML. Such annotations include hints as to aliases of fields (alternate names), structure (whether attributes or elements are used), and how collections are stored. By default, the conversion to and from XML is accomplished through the use of reflection, meaning that private and internal data will be serialized, not just its public interface (as is the case in JavaBeans). Consequently, XStream is a light-weight library that requires little configuration or customization to use it "out of the box". In the context of JOXM, XStream is used by the Marshaller class, which acts as an proxy between XStream and the Session class. XStream is used for the purpose of serializing objects from Java to XML, and vice versa. Further details of how this operates in practice can be in [14].

Demonstration Application

In order to demonstrate and test the capabilities provided by the JOXM library, a demonstration application was created. The k5ncal [15] desktop calendar, an open-source Java calendar tool, was enhanced as the front end. The k5n Desktop Calendar (k5nCal) is an open source calendar application that is built upon Java Calendar Tools. It has similar features to other calendar application such as iCal, Microsoft Outlook, or Google Calendars. The first publicly released version of this open source application, under the GNU General Public License, was June 1, 2007. Since it was developed in Java, the application can run on any operating system that supports a Java Virtual Machine (JVM) with version 1.5 or higher. Other 3rd party libraries such as Joda-Time, Google RFC2445 and the Java CSV Library are bundled with k5nCal.

K5nCal provides basic calendar operations found in most popular calendar applications. The application visually displays a calendar in a month view (although other views can be added), as illustrated in Figure B of Appendix C. Users can scroll through the calendar to create different sets of calendar events so that events are grouped logically and visually (by color). You can select at any time which calendars are visible. Events are described by information such as: subject, date and time, location, status, categories, and notes. Once an event is created, you can edit the event, as illustrated in Figures D and E of Appendix C, as well as delete it entirely. The application also supports importing calendar information from an iCalendar or CSV files and exporting to ics (iCalendar), however, the current version fails to support recurrent events.

All of the calendars are currently saved in iCalendar RFC 2445 file format so that it can be restored back into the application. The iCalendar is a standard (RFC 2445) for calendar data exchange. This format allows users to easily send meeting request and tasks to other users through email. The specification for this format is the result of the Internet Engineering Task Force Calendaring and Scheduling Working Group.
The JOXM library will replace the current file persistence in favor of XML database persistence. Additionally, a significant omitted feature in the current version of k5nCal is searching. There is currently no interface to search for events in a set of calendars. Storing calendar and event data in an XML database will significantly ease the addition of such an feature. Consequently, the k5ncal application will be enhanced so users can define and run custom queries against the current data set in all or a particular calendar. The interface, illustrated in Figure G of Appendix C, provides the user the ability to choose and provide values for all fields of events in order to generate a query. The results of this query will be shown in a dialog, as shown in Figure H of Appendix C.

**Testing & Evaluation**

Testing was integrated into the development of the library via the use of test driven development. Unit testing, specifically JUnit [17], was employed to provide regression testing throughout the project cycle. The tests provide full coverage of the JOXM library code. Further exploratory testing of the library was achieved through the development of the demonstration application.

Aside from the known issues, documented following, the core persistence API function adheres to its specification and design. Due to the fact that the JOXM library is much like a proxy between the front-end application and the back-end database, evaluation characteristics such as performance are less meaningful. The primary indicator of performance in the library is the performance of the underlying database, including its query optimization, indexing techniques, and feature set. For instance, the lack of transaction management in most native XML databases skews any logical performance comparison to a fully ACID compliant relational database. For a discussion of the performance and scalability of eXist, see [18]. However, techniques such as the use of UIDs attempt to optimize performance as much as possible in JOXM's persistence interface.

**Known Issues**

At the current time, there are several limitations in the design of this library. First and foremost, it does not support remote database connections. Despite the lack of this feature, including support for it would not require significant future development. One reason that remote database connections were not implemented was the lack of support for transaction management by most, if not all, native XML database providers. Since no guarantee can be made in regards to the accuracy and atomicity of the data modifications, it was felt that including support for an environment (i.e. remote) to expose such limitations would be ill-advised, if not dangerous. The final limitation is in regards to the custom setup of the databases. The current implementation provides for global configurations for databases instances, including properties such as cache sizes, data file locations and connection pools. For example, the default location for the
embedded database files is in a directory call data in the current working directory. Obviously, future implementation would need to address this limitation to support custom configurations and custom database locations (e.g. user home directory).

There are two additional key issues present in the current version that hinder its ability to be used in a production-level system. The first regards cyclical object models. The current version of JOXM supports only hierarchical tree models. A cycle in the object model occurs when a node contains a link to one of its ancestors. Running a query through the executeXPathQuery method with cyclic references will cause any reference beyond the scope of the query to fail to deserialize (returning null for the out of scope reference). For instance, if object A references object B, which references object C, which consequently references object A, a cycle exists. When an application persists A in the database, the persistence will succeed with A referring to B, B referring to C, and C referring to A. However, if the application issues an XPath query such as //C, the objects of type C returned will not correctly reference object A, since the scope of the reference to A is outside the scope of the query's result (the node for C). A solution to the cyclic graph problem is the use of deferred (lazy) evaluation, as discussed in the Future Work section. The effect of such deferred evaluation is issuing another query to "re-link" the reference to A on its first use. This type of evaluation is the approach commonly taken by [1].

The second issue relates to the order of persistence calls. Taking the object model from the previous example, if the objects A, B, and C are created in memory, the order in which the application saves those objects is arbitrary. Consequently, it is feasible that the object C is the first to be saved via the save persistence method. Once C is saved, its XML representation will exist at the top level of the collection in which it is stored. If the application further makes a save call to B, which refers to C, a notable issue occurs. Since B contains a reference to C, when marshaled to XML, B contains the XML representation of C. However, we have noted that this representation is already stored as a node in the top-level collection. A solution to such a problem is to remove the previous instance of C before inserting B into the collection. The current version of JOXM makes no provisions if such an ordering occurs. Therefore, in order for the data to be correctly persisted in this example, A is the only object that should be saved, causing B and C to be nested accordingly.

**Future Work**

The following lists outline several features that may be implemented as efforts in future development in the library. Addressing the known issue previously discussed are vital to the continued development and maintenance of the library. Additionally, added customization ability regarding the database configuration and type registration is desired.

- Solution to cyclical problem (see above)
- Solution to order problem (see above)
- Custom database configuration (via generic XML)
- Automatic registration of types (using the Spring Framework)

Beyond the scope of the original project, the following is a list of prospective extensions that may be undertaken to increase the utility and quality of the library:

- Allow users to issue XQuery queries to the database, returning a String object as a result
- XPath interpreter for the Hibernate Query Language (HQL) used by [1]
- Annotation-based specifications and customizations
- Custom specification for deferred (lazy) or immediate (greedy) evaluation
- More control of the marshaling and unmarshaling contexts (such as storing inlining data as attributes)
- Transaction management (as a layer above the DBMS, if no suitable native XML DBMS supports ACID transactions)
- Use of Virtual Machine Identifiers (VMIDs) instead of UIDs (increasing the uniqueness of identification values)
- Provide support for the XQuery Update Facility (XQUF) (once specification is approved)

**Conclusion**

Despite the previously mentioned limitations, we feel that this library provides a useful library that may (with substantial work in the future) fill the needs of many real-world applications for data persistence. Although the goal is not to provide a full featured OXM library for Java, the implementation of and extension of an existing application to use the JOXM library demonstrates that object-XML mapping and XQuery are valid, useful, and realistic technologies that can be utilized today. JOXM provides developers a tool to ease the entry into XML database applications by hiding the details of native XML database persistence and exposing the functionality and benefits of XPath and XQuery. JOXM is designed around simplicity and ease of use, not performance or transaction management. Therefore, applications striving for high-performance, high-concurrency, transactional support, or those with "flat" object models should refrain from using JOXM. In these cases, a technology such as Hibernate, paired with a relational database management system, would be more suitable. JOXM addresses the needs of applications looking for a generic persistence layer supporting the XML:DB API, XPath, and XQuery standards, especially those in which the object model is hierarchically complex.
Appendix A: XQuery/XPath Use

1. Calendar retrieval:

```xml
//Calendar
```

2. Calendar creation (XUpdate insert):

```xml
update insert <Calendar>
  <name>Calendar</name>
  <selected>true</selected>
  <foreground>
    <red>255</red>
    <green>255</green>
    <blue>255</blue>
    <alpha>255</alpha>
  </foreground>
  <background>
    <red>40</red>
    <green>227</green>
    <blue>193</blue>
    <alpha>255</alpha>
  </background>
  <border class="java.awt.SystemColor">
    <value>7</value>
    <falpha>0.0</falpha>
  </border>
  <events class="list"/>
  <id>_39e706a46ad531be:-14c92001:1169e787660:-7fff</id>
</Calendar> into /root
```

3. Calendar update (with new event):

```xml
update replace /root/Calendar[id="_39e706a46ad531be:-2b157bde:1169e482a30:-7f5c"] with <Calendar>
  <name>Jimmy</name>
  <selected>true</selected>
  <foreground>
    <red>255</red>
    <green>255</green>
    <blue>255</blue>
    <alpha>255</alpha>
  </foreground>
  ...<events class="list">
    <Event>
      ...<summary>
        <name>SUMMARY</name>
        <value>Puma: Horizontal Prototype</value>
        <attributeList/>
      </summary>
      <description>
        <name>DESCRIPTION</name>
      </description>
    </Event>
  </events>
</Calendar>
```
4. Search for events in a calendar with the subject "info", occurring after 1 Nov. 2007 and "Confirmed" status:

```
//Calendar[id="_39e706a46ad531be:-2b157bde:1169e482a30:-7fa9"]//Event[contains(lower-case(summary/value), "info") and number(substring(startDate/value, 1, 8)) >="20071001" and status="2"]
```

**Appendix B: XML Data**

1. **conf.xml** - This file contains eXist-specific configuration the embedded database instance (abbreviated for length and clarity)
2. catalog.xml - This file contains the catalog used by eXist for resolving DTD and XML schema references
Appendix C: Screenshots

Figure B: k5ncal Main Application Window

Figure C: Add Calendar Dialog
Figure D: Event Creation Dialog
Figure E: Event Editing Dialog

Figure F: iCalendar Import Dialog
Figure G: Event Search Dialog
Appendix D: Biographical

Michael Huffman received his B.S. degree in computer science in 2006 from the California Polytechnic State University, San Luis Obispo. Currently obtaining a M.S. in computer science, his research interests include cognitive informatics, software complexity, behavioral driven development, and agile methods. He is employed as a senior software developer at CDM Technologies, Inc. in San Luis Obispo, CA.

Jimmy Hua received his B.S. degree in computer science from the California Polytechnic State University, San Luis Obispo in 2006 and is currently obtaining a M.S. in the same field. His research interests include test driven development, agile methods, human computation, and software project management.

Jong-Seo Lee received his B.E. degree in architectural engineering from the KOREA University, Seoul, Korea and he got additional B.E. degree in information & computer engineering from the Hongik University, Seoul, Korea. Currently he is obtaining a M.S.
in the computer science at the California Polytechnic State University, San Luis Obispo. His research interests include software engineering, computer networks, and database systems.

Adam Dukovich is a member of the blended B.S. and M.S program at the California Polytechnic State University, San Luis Obispo. His research interests include computer science pedagogy, database systems, and computational theory.

**Bibliography**

[15] "k5n.us: k5n Desktop Calendar (k5nCal)." 2007. k5n.us. <http://www.k5n.us>