LexEVS 6.0 Design Document - Detailed Design - Data Access Layer

Contents

Document Information

Author: Craig Stancl, Kevin Peterson
Email: Stancl.craig@mayo.edu, peterson.kevin@mayo.edu
Team: LexEVS

Contract: CBITT BOA Subcontract# 29XS223
Client: NCI CBIIT
National Institutes of Heath
US Department of Health and Human Services

Revision History

Version	Date	Description of Changes	Author
1.0	5/14/10	Initial Version Approved via Design Review	Team

Data Access Layer

Solution Architecture

LexEVS lacks a generalized Read/Write interface that can support Authoring and incremental Coding Scheme changes. Also, database access is not isolated. Database-specific code is intermixed with the Service Layer, as well as Extensions. To make LexEVS more able to handle Authoring, while at the same time providing a clean, centralized Database Access Layer, these are the requirements:

- Service Layer code should not call any JDBC code other than a Data Access Interface
- The Data Access Layer should allow connection pooling
- The Data Access Layer should expose 'Service' or 'Repository' Interfaces to the Service Layer.
- Transactions should be well defined
- Allow for Backwards Compatibility
- Inserts, Updates, Selects, and Deletes should cascade the Object hierarchy if desired
- Service-Exposed Interfaces should use the Castor-generated beans.
- Batch inserts should be supported
- Lucene access should be included in the Data Access Layer
- Registry accesss should be included in the Data Access Layer.
- Expose an Event-Driven framework to allow users to insert business rules to control access to the database
- In terms of System Resource responsibilities, the Data Access Layer should be responsible for:
  - Detecting the LexGrid Schema version
  - Detecting the Database Type (MySQL, Oracle... etc)
  - Producing appropriate implementations of the Database Access code given the above
  - Loading and initializing all schemas on install
  - Tracking loaded Coding Schemes and other resources
  - All database admin functions (remove Coding Scheme, index, compute transitivity, etc)

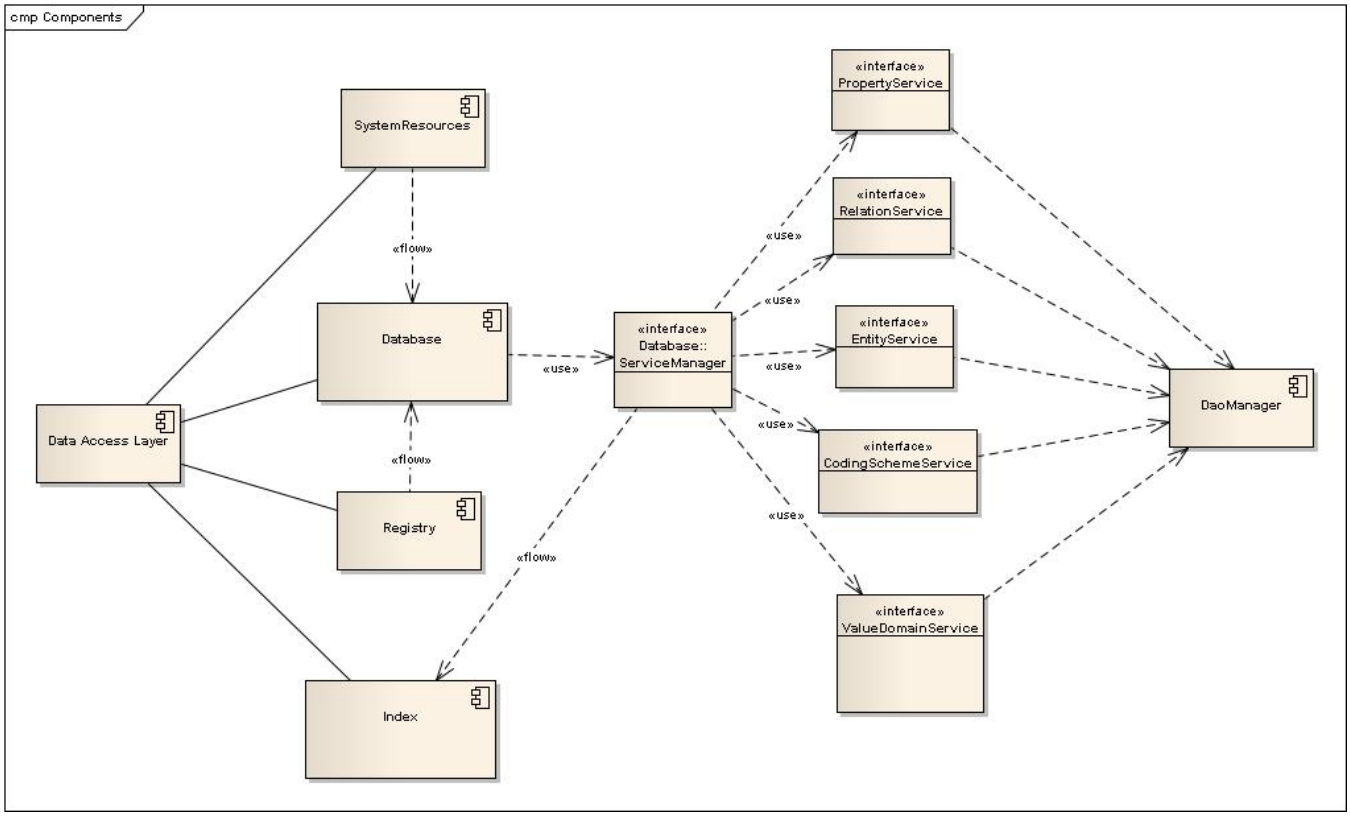
Structure

Main Components:

- Database
  - An access point for CRUD (Create, Read, Update, Delete) operations
  - Will contain a ServiceManager Interface, which will expose Service Layer consumable CRUD services.
  - Contains logic for LexGrid Schema Version detection and Database Type detection.
  - Serves as the Transaction Demarcation Layer
- System Resources
  - Provides various system-level services
- Registry
  - Maintains a list of available Resources (loaded Coding Schemes, Histories, etc)
- Index
  - Provides a Lucene-based indexing service

Sub Components:

- (Data Access Object) DAO Manager
  - Provides fine-grained CRUD access to the Database
  - Is NOT Transaction aware -- all transactions must be defined by the calling services
  - Does NOT cascade
    - This is done to allow the the Service to define exact transaction granularity
  - Event framework is implemented in the individual DAOs, not the Services



Domain Driven Design

The LexEVS Data Access Layer loosely follows the Domain Driven Design (DDD) Principals.

DDD Principals and the LexEVS Data Access Layer equivalent:

- **DDD Entities**
  - In DDD, an 'Entity' is an Object that has Identity and may have 'Versions'. In other words, if some part of an Entity changes, the identity of the Entity does not change, but it can be considered a different 'Version'
- **LexEVS Data Access Layer Entities**
  - In LexEVS, the idea of "having identity" and "having a version" are generally encompassed by Model Objects that implement the 'Versionable' Interface.

- **DDD Value Objects**
  - A Value Object does not maintain an identity, therefore, if its state changes, the Object itself is now something different.
  - Entities contain Value Objects
- **LexEVS Data Access Layer Value Objects**
  - All Model Objects in LexEVS that do NOT implement the 'Versionable' interface are considered Value Objects.
  - Considerations for LexEVS Value Objects:
    - Value Objects are NOT directly Updatable

        - Value Objects may only be Updated via a Update of the Containing Entity
        - Value Objects may only be Deleted via a Delete of the Containing Entity
        - Value Objects may be inserted independently given the ID of the Containing Entity
        - If a Value Object changes, its Containing Entity's Version will change
        - Examples: AssociationQualifiers, PropertyQualifiers, Sources, etc...

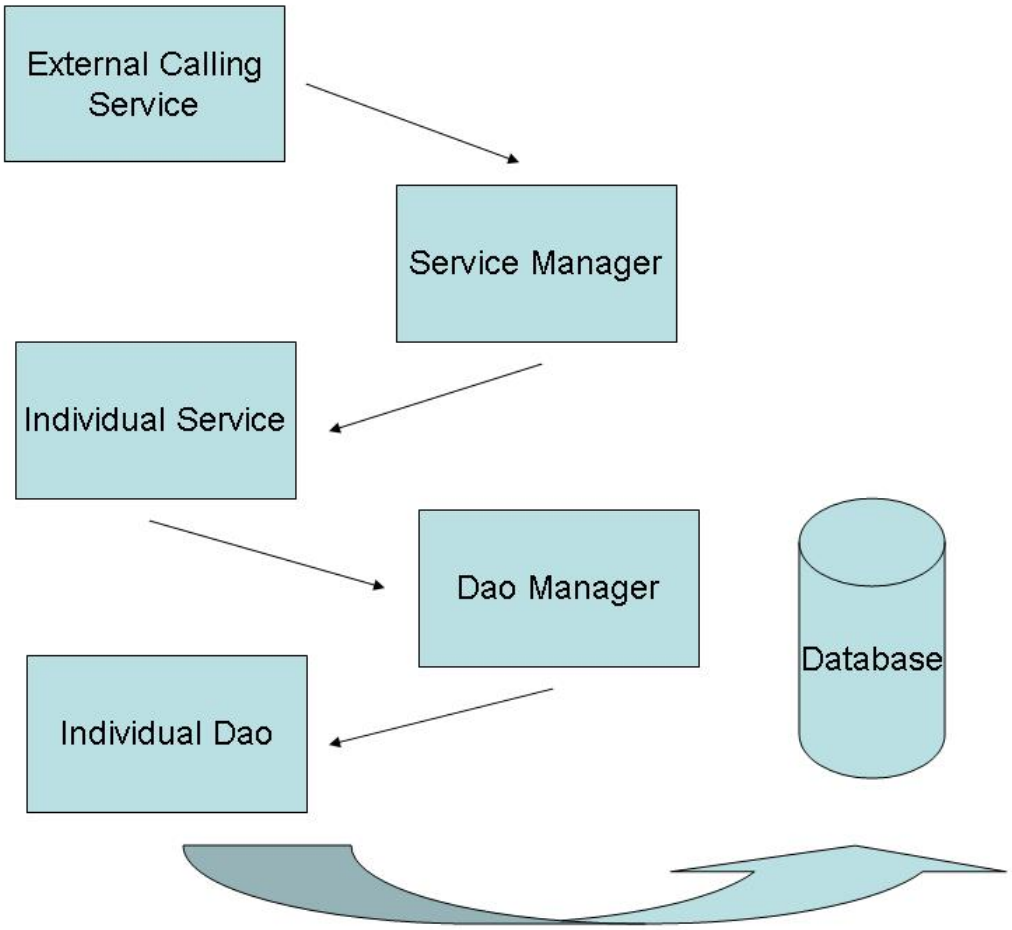
- **DDD Aggregate Roots**
    - A DDD Aggregate Root is a collection of Entities that share a common lifecycle.
    - All children are accessed through the top-level Object, or Root.
    - All CRUD operations are cascaded within the Aggregate Root boundary

- **LexEVS Data Access Layer Aggregate Roots**
    - LexEVS has a very heirarchical Object Model, meaning that most Entities are dependent upon the lifecycle of a parent. For instance, if a Coding Scheme is removed, all containing Concepts are removed. A Concept cannot exist in LexEVS without a Coding Scheme.
    - So how to define Aggregate Root boundaries? In LexEVS, almost everything that is an Entity can also be considered an Aggregate Root.

- **DDD Repository**
    - Repositories are for persisting Aggregate Roots. The aim is to apply Collection Semantics to Domain Objects. For instance, add(...), get (...), and so on.
- **LexEVS Repository**
    - LexEVS deviates from the idea of a 'Repository' for several reasons:
        - Finer grained inserts are needed by the loaders. For example, in some instances we may need to insert something independently that is not an Aggregate Root, like a PropertyQualifier.
        - Because we could in theory consider all of our Entities Aggregate Roots, we would have a proliferation of Repositories.

Service-Based CRUD and Component Interaction

General Component Flow:



External Calling Service

The Component, API, or Application that requires access to the database. The reference to the Service Manger is obtained through a Service Locator pattern, or by other means outside the scope of the Data Access Layer. The External calling service does not (and should not) use any JDBC specific code, including Transactional Demarcation.

;Service Manager

The Service Manager is the main entry point to all of the Data Access Services. The Service Manager provides a centralized reference for all of the available Database Access Services, allowing clients to select the Service or Services that they require.

Individual Service

Database Access is broken into groups of 'Services'. Each Service will be a logical grouping of functions loosely based on packages of the Object Model. A service serves these rols:

- Provides transaction boundaries
- Serves as a facade for the fine-grained DAO API.
- Is always implementation independent.

;DAO Manager

The DAO Manager serves the same purpose as the Service Manager -- to provide a central lookup point for all available DAOs in the system.

The DAO Manager is implementation independent, so it must follow certain rules:

- The DAO Manager will obtain references to individual DAOs through a factory, never directly. This decouples them from the actual implementation of the DAO, and allows underlying Database Type and LexGrid Schema Version checking mechanisms to be put in place
- There will be no implementation specific code in the DAO Manager.

Individual DAOs

Each individual DAO will be responsible for a logical set of CRUD database operations. DAOs themselves are expected to be implementation specific, MAY be LexGrid Schema Version specific and MAY be Database Type specific. So, for example, there may be one DAO for Oracle, one for DB2, and so on. The DAO is expected to provide enough information about itself for factories to be able to select the appropriate DAO for the DAO Manager.

DAOs may be implemented in any framework, as long as the adhere to the Interface contract of the DAO.

DAOs may NOT define Transaction boundaries within themselvs.

DAOs MUST fire the appropriate events as content is updated, inserted, or deleted.

Event Framework

The Data Access layer will support an Event-Based framework to allow External Calling Services an access point to monitor the changing state of the Database The Data Access Layer Event Framework will follow these guidelines:

- Any CRUD operation on a Versionable Entity will fire an Event
- External Services will be able to register an arbitrary number of listeners
- An Insert Operation will fire an event of this type:

Insert Event	**—**
Parameter	**Output**
Item to be inserted	A boolean indicating whether or not the insert should take place

Delete Event	**—**
Parameter	**Output**
Item to be deleted	A boolean indicating whether or not the delete should take place

Update Event	**—**
Parameter	**Output**
Item to be updated (before changes)	
Item to be updated (after changes)	A boolean indicating whether or not the update should take place

Implementation - Database

Ibatis

Apache Ibatis is a framework for decoupling SQL code from Java code. All SQL code is defined in XML files as opposed to in the Java classes themselves, making updates and edits to the SQL. Also, SQL code can be modified without recompiling the system.

Reasons for Ibatis:

- Allows direct access to the SQL code, allowing for complex, optimized queries
- Enables a clear distinction between Java code and SQL code
- Pre-mapped result sets and Row Mappers to convert from Database Row -> Domain Object

An example Ibatis XML document is shown below. Note that the SQL code, as well as the logic for assembling the results, is decoupled from the Java implementation:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sqlMap PUBLIC "-//iBATIS.com//DTD SQL MAP 2.0//EN"
    "http://www.ibatis.com/dtd/sql-map-2.dtd">
<sqlMap>

    <typeAlias alias="codingScheme" type="org.LexGrid.codingSchemes.CodingScheme" />
    <typeAlias alias="codingSchemeSummary" type="org.LexGrid.LexBIG.DataModel.Core.CodingSchemeSummary" />

    <resultMap id="codingSchemeResult" class="codingScheme">
        <result property="codingSchemeName" column="codingSchemeName" />
        <result property="codingSchemeURI" column="codingSchemeUri" />
        <result property="representsVersion" column="representsVersion" />
        <result property="formalName" column="formalName" />
        <result property="defaultLanguage" column="defaultLanguage" />
        <result property="approxNumConcepts" column="approxNumConcepts" />
        <result property="entityDescription.content" column="description" />
        <result property="copyright.content" column="copyright" />
        <result property="isActive" column="isActive" />
        <result property="entryState" resultMap="entryStateResult" />
    </resultMap>

    <resultMap id="codingSchemeSummaryResult" class="codingSchemeSummary">
        <result property="localName" column="codingSchemeName" />
        <result property="codingSchemeURI" column="codingSchemeUri" />
        <result property="representsVersion" column="representsVersion" />
        <result property="formalName" column="formalName" />
        <result property="codingSchemeDescription.content" column="description" />
    </resultMap>

    <select id="getCodingSchemeById" parameterClass="org.lexevs.dao.database.ibatis.parameter.PrefixedParameter" resultMap="codingSchemeResult">
        SELECT
            cs.codingSchemeGuid,
            cs.codingSchemeName,
            cs.codingSchemeUri,
            cs.representsVersion,
            cs.formalName,
            cs.defaultLanguage,
            cs.approxNumConcepts,
            cs.description,
            cs.copyright,
            cs.isActive,
            cs.owner,
            cs.status,
            cs.effectiveDate,
            cs.expirationDate,
            es.changeType,
            es.relativeOrder,
            es.revisionGuid,
            es.prevRevisionGuid,
            rev.revisionGuid
        FROM
            $prefix$codingScheme cs

        LEFT JOIN
            $prefix$entryState es
        ON
            cs.entryStateGuid = es.entryStateGuid
```

Ibatis vs. Hibernate

Hibernate is an Object Relational Mapping tool, Ibatis is not. Hibernate aims to abstract the developer from building SQL code, as all SQL is generated by the framework. Ibatis does not build or generate SQL code, but gives the user direct control of the SQL.

Hibernate assumes a certain database structure, as well as a specific Object Model structure to be able to function properly. There are several problems with using Hibernate with LexEVS.

- The LexEVS Object Model does not track the Id of an Object ('Id' meaning the unique database key)
- Associations in the Object Model are Uni-directional ONLY - even if they are Bi-directional in the database. This means that even though a certain Object may be dependent on the lifecycle of a parent Object, there is no direct refrence from Child -> Parent
- LexEVS must support Horizontal Partitioning of the Database based on Coding Scheme, so every Coding Scheme in LexEVS must be able to be partitioned into its own set of schema tables. This requires dynamically changing the table name at query-time to account for the different sets of tables. Hibernate does not handle this efficiently.

Ibatis vs. Spring JDBC

Spring JDBC is a JDBC framework that abstracts the developer from boilerplate JDBC code, such as managing connections, error handling, mapping rows to Objects, etc. It uses Template Patterns to provide commonly used JDBC functionality. Spring JDBC is powerful, flexible, and fast -- but it does not decouple the SQL code from the Java Code. Ibatis accomplishes much of the same thing as Spring JDBC -- they both allow dynamic SQL building and direct control of the SQL code.

Caching

Caching commonly accessed data is critical for performance, and the DAO Layer will have a built-in caching framework to accommodate this. The framework is annotation-based and could be generalized to other use-cases, not just the DAO layer.

Annotations

@Cacheable

The ***@Cacheable*** annotation marks a class as being eligible for method caching. Classes without this annotation will not considered by the Caching framework

This annotation accepts 2 parameters:

Parameter Name	Description	Required
cacheName	The unique name of the cache. This cache may be shared between classes, or each class may have its own unique cache.	YES
cacheSize	The number of cached elements to be held in memory. Once the limit is reached, the element that has be accessed will be removed	NO (Default = 50)

@CacheMethod

The ***@CacheMethod*** annotation is placed on an individual method to be cached. Method caching follows the following guidelines:

- The String representation of all parameters are combined to form the 'Key'. On subsequent calls to the method, the framework will compare the parameters to determine if the result can be resolved from the cache.

```
@CacheMethod
public CodingScheme getCodingSchemeById(String codingSchemeId)
```

In the example above, the String 'codingSchemeId' is evaluated as the cache 'key'. If future calls to the method have the some 'codingSchemeId' parameter - the method will not execute, and the result will be resolved from the cache.

Determining Cache Keys

NOTE: The cache key is determined by the 'toString()' representation of every parameter passed to the method.

@ClearCache

If a call to a method in a Cacheable class will invalidate the cache, the ***@ClearCache*** annotation may be used to reset the cache. An example use-case if this would be an 'update' method. The 'update' will change the underlying data in the data store, but the cache will still contain the old data. A ***@ClearCache*** annotation may be added to any 'update' method to reset the cache whenever the underlying data may have changed.