

# **Repositories**

## **Health Data Repository 3.8, Increment 12**

### **System Design Document, Volume 2**

**(Volume 2 of 2)**

**Version 0.3**



**March 2015**

## Revision History

Date	Version	Description	Author
09/08/2014	0.1	This document was developed in accordance with the updated ProPath template acquired on 02/10/2014.	
11/05/2014	0.2	Minor grammatical updates from Peer Review	
11/13/14	0.3	Signatures applied for this version of SDD	

## Table of Contents

6. Detailed Design.....	1
6.1 Hardware Detailed Design .....	1
6.2 Software Detailed Design.....	2
6.2.1 Conceptual Design .....	2
6.2.2 Specific Requirements .....	128
Attachment A. Reviews and Approval Signatures .....	140
Attachment B. Signature Verification.....	140

## Table of Figures

Figure 1 HDR General Support System (GSS).....	1
Figure 2 HDR 3.8 Production System .....	2
Figure 3 CDS Request Process Component.....	6
Figure 4 Template Cache Class Diagram .....	11
Figure 5 Filter Cache Class Diagram.....	12
Figure 6 Template and Filter Service Class Diagram .....	14
Figure 7 Template Validation Sequence Diagram.....	16
Figure 8 Filter Validation Sequence Diagram .....	16
Figure 9 Filter Manager Class Diagram.....	19
Figure 10 Filter Manager Sequence Flow .....	20
Figure 11 Template Manager Class Diagram .....	23
Figure 12 Template Manager Read request processing Sequence Diagram.....	24
Figure 13 Template Manager Write/Update/Delete request processing Sequence Diagram .....	24
Figure 14 Person Service Identity Handling in CDS Read Path Diagram.....	26
Figure 15 Person Service Identity Handling Class Diagram .....	28
Figure 16 Person Identity Handler Sequence Diagram.....	29
Figure 17 Person Identity Handler Sequence Diagram.....	30
Figure 18 Overview of SQL Based Filters Processing Flow .....	33
Figure 19 Sequence Diagram Class Interaction During an SQL Based Filter Method Call.....	35
Figure 20 SQL Based Filters Class Diagram .....	39
Figure 21 CDS Filter Class Diagram .....	41
Figure 22 Filter Query Sequence Diagram .....	43
Figure 23 VIM Aggregation System Architecture.....	45
Figure 24 VIM Aggregator Resolution Class Diagram .....	46
Figure 25 Response Sequencer Sequence Diagram.....	47
Figure 26 Response Sequencer Sequence Diagram.....	48
Figure 27 Sorting Read Response XML Based on Criteria Class Diagram .....	52
Figure 28 Sorting Read Response XML Based on Criteria Sequence Diagram.....	53
Figure 29 Transaction Manager Class Diagram .....	58
Figure 30 Transaction Manager Read request processing Sequence Diagram.....	59
Figure 31 Transaction Manager write/update/delete request processing Sequence Diagram .....	60

Figure 32 Persistence Locator Class Diagram .....	63
Figure 33 Persistence Locator Sequence Diagram .....	64
Figure 34 Site Specific Persistence Manager Factory Sequence Diagram .....	65
Figure 35 Persistence Manager Class Diagram .....	70
Figure 36 Persistence Manager Sequence Diagram.....	71
Figure 37 Persistence Manager Class Diagram .....	74
Figure 38 Rules Processor Sequence Diagram .....	75
Figure 39 Custom O/R Mapping Strategy FC 6.0 Component Class Diagram .....	78
Figure 40 Custom O/R Mapping Class Diagram.....	80
Figure 41 Custom O/R Mapping Sequence Diagram .....	85
Figure 42 Query Processor Sequence Diagram .....	87
Figure 43 Query Strategy Class Diagram .....	88
Figure 44 Query Association Class Diagram.....	90
Figure 45 Model Assembler Class Diagram.....	91
Figure 46 Create Query Work Sequence Diagram .....	92
Figure 47 Execute Query Work Sequence Diagram.....	93
Figure 48 Model Assembler Sequence Diagram .....	94
Figure 49 File Request Response Framework RefactorClass Diagram.....	96
Figure 50 File Concurrent Read Class Diagram .....	98
Figure 51 File Read Concurrent Work Manager Sequence Diagram .....	99
Figure 52 File Read Concurrent Work Manager Sequence Diagram .....	100
Figure 53 VIM Read Request Audit Overview .....	102
Figure 54 VIM Write Request Audit Overview .....	103
Figure 55 VIM Read Request Class Diagram.....	104
Figure 56 VIM Write Request Class Diagram.....	106
Figure 57 VIM Read Request Sequence Diagram.....	109
Figure 58 VIM Write Request Sequence Diagram .....	110
Figure 59 CDS Exception Handling Class Diagram.....	114
Figure 60 CDS Exception Handling Sequence Diagram .....	117
Figure 61 CDS Application Logging Overview .....	118
Figure 62 Application Logging Class Diagram .....	119
Figure 63 Application Logging Persistence Manager Class Diagram.....	121
Figure 64 Logger Persistence Sequence Diagram .....	123

Figure 65 Template Manager Class Diagram .....	124
--	-----

## Table of Tables

Table 1 Links to HDR Detailed Software Design Components .....	4
Table 2 Extend tables/Columns .....	129
Table 3 Stored Procedures and Functions.....	135

This is Volume 2 of 2 of the Health Data Repository (HDR) 3.8 System Design Document (SDD) which contains specific sections in regards to technical documentation. In the event there is a modification to these sections, it will be noted in the Revision History to the reviewer of the document.

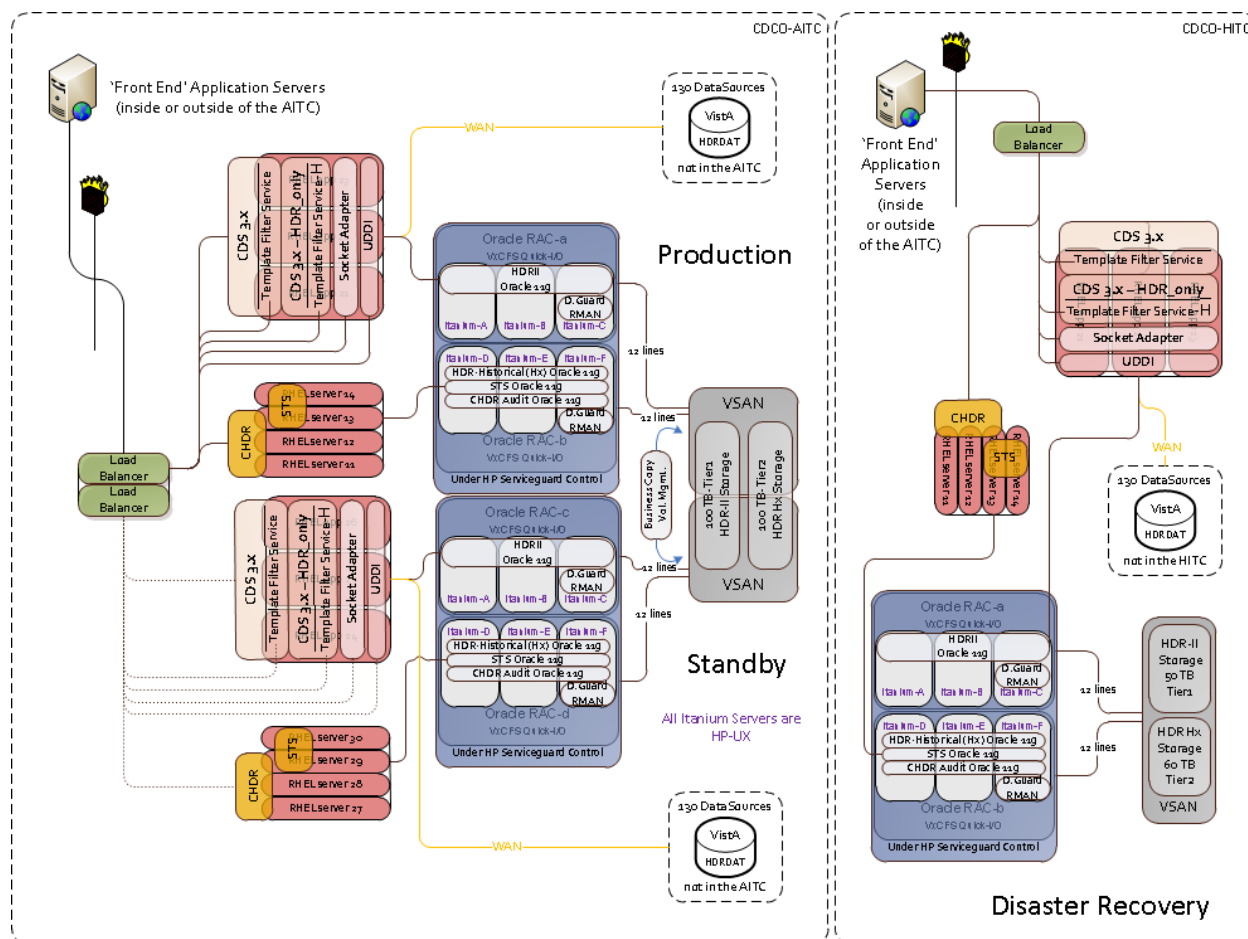
Volume 2 includes the following, as referenced in HDR 3.8 SDD Volume 1:

## 6. Detailed Design

### 6.1 Hardware Detailed Design

All production hardware is located in the AITC. Figure 1 illustrates a detailed production deployment environment showing an instance of the HDR 3.x system deployed to a WebLogic managed node within a cluster, and the cluster's relationship with the Oracle RAC and Cache cluster configurations.

**Figure 1 HDR General Support System (GSS)**



## 6.2 Software Detailed Design

The HDR software detailed design section describes the major components of the HDR 3.x system, including an overview, module design, processing, data structure, configurations, process flow diagrams and other diagrams as needed. HDR provides back end processing only and does not include user interfaces or interactions. The detailed software design begins with section 6.2.1.5 Request Response Processor.

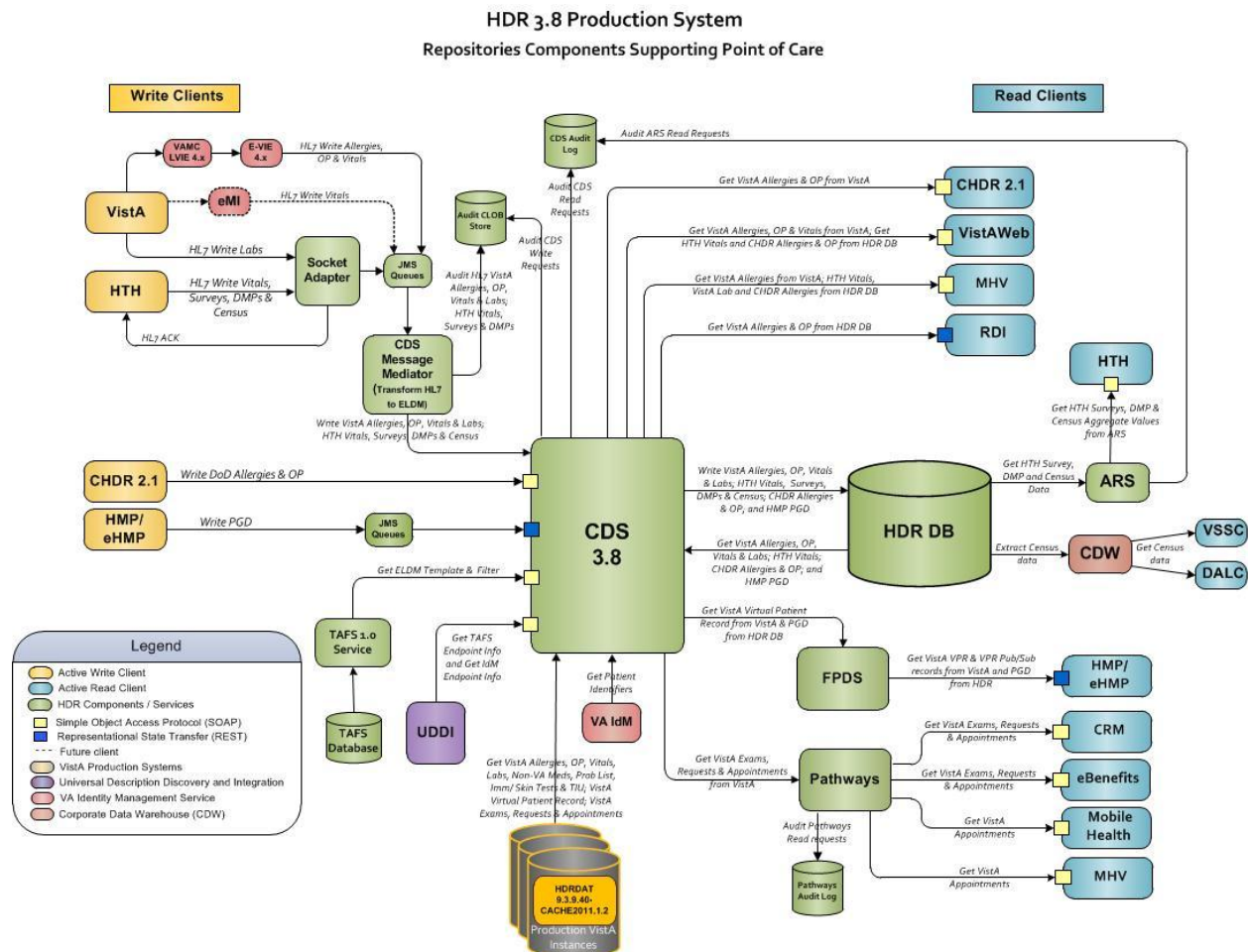
### 6.2.1 Conceptual Design

Refer to Table 1 below for a list of HDR detailed software design components and links to the specific sections contained in the SDD.

#### 6.2.1.1 Product Perspective

The HDR system is self-contained. The HDR 3.8 Data Flow Diagram, illustrated in Figure 2 below, represents the Write and Read client interfaces.

**Figure 2 HDR 3.8 Production System**





#### **6.2.1.1.1 User Interfaces**

Not applicable.

#### **6.2.1.1.2 Hardware Interfaces**

Not applicable.

#### **6.2.1.1.3 Software Interfaces**

Refer to Figure 1 above which illustrates the HDR 3.8 Data Flow Diagram representing Write and Read client interfaces.

#### **6.2.1.1.4 Communications Interfaces**

Not applicable.

#### **6.2.1.1.5 Memory Constraints**

Not applicable.

#### **6.2.1.1.6 Special Operations**

Not applicable.

#### **6.2.1.2 Product Features**

Product features are outlined in HDR 3.8 SDD, Volume 1, Section 1.

#### **6.2.1.3 User Characteristics**

Not applicable.

#### **6.2.1.4 Dependencies and Constraints**

Design Constraints are discussed in the HDR 3.8 SDD, Volume 1, Section 2.4.2.

**Table 1 Links to HDR Detailed Software Design Components**

Detailed HDR 3.x Software Component
<a href="#">Request Response Processor</a>
<a href="#">Database Repository</a>
<a href="#">Template and Filter Service Client</a>
<a href="#">Request Validator</a>
<a href="#">Filter Manager</a>
<a href="#">Template Manager</a>
<a href="#">Person Service Identity Handling</a>
<a href="#">SQL Based Filters</a>
<a href="#">VIM Filtering</a>
<a href="#">VIM Aggregation</a>
<a href="#">Response Sequencer</a>
<a href="#">Sorting Read Response XML Based on Criteria</a>
<a href="#">Persistence Subsystem</a>
<a href="#">Persistence Locator</a>
<a href="#">Persistence Manager</a>
<a href="#">Rules Processor</a>
<a href="#">Custom O/R Mapping Strategy</a>
<a href="#">Query Processor</a>
<a href="#">Concurrent Read and Persistence Locator</a>
<a href="#">Audit Logging</a>
<a href="#">Exception Handling</a>
<a href="#">Application Logging</a>
<a href="#">FPDS</a>

#### **6.2.1.5 Request Response Processor**

##### **6.2.1.1.1. Request Response Processor**

This section provides an overview of the request and response subsystem known as the CDS Request Handler component. Write/Update, and Read request templates are configured within the CDS 3.x application Spring context to be processed by the CDS application. As a Read request is received by the CDS 3.x application, the incoming request is validated against the TAFS and sent to the transaction processing component.

##### **6.2.1.1.2. Request Response Processor Module Design**

The significant component design logic exists within the Spring configuration for the TemplateRequestProcessor. The TemplateRequestProcessor is configured with an instance of the ClinicalDataServiceSynchronousInternal object, TemplateManager and FilterManger by the Spring context, and delegates create, update, read, delete incoming requests to the corresponding ClinicalDataServiceSynchronousInternal object after validating the incoming XML request. The incoming read request is validated using FilterManger. The incoming create, update and delete request XML requests are validated against Template Manager.

ClinicalDataServiceSynchronousInternal is injected with FilterFactory, FilterPatientResolver and TransactionManager to process the incoming request.

#### 6.2.1.1.3. Request Response Processing

Write request processing: When the write/update/delete request workflow reaches the createClinicalData method on the TemplateRequestProcessor object, the payload is validated and a call is made to the createClinicalData method on the ClinicalDataServiceSynchronousInternal passing in the incoming XML, template ID and request ID.

ClinicalDataServiceSynchronousInternal will process the incoming request by calling the corresponding business method on TransactionManger. See transaction manager section for more details.

Read request processing: When the read request workflow reaches the readClinicalData method on the TemplateRequestProcessor object, the payload is validated and a call is made to the readClinicalData method on the ClinicalDataServiceSynchronousInternal passing in the filter XML, filter ID, template ID and request ID. The filter content is processed by FilterPatientResolver and later the corresponding business method is called on the TransactionManager. See filter processing and transaction manager sections for more details.

#### 6.2.1.5.1 Request Response Processor Local Data Structures

The following details the TemplateRequestProcessor configuration:

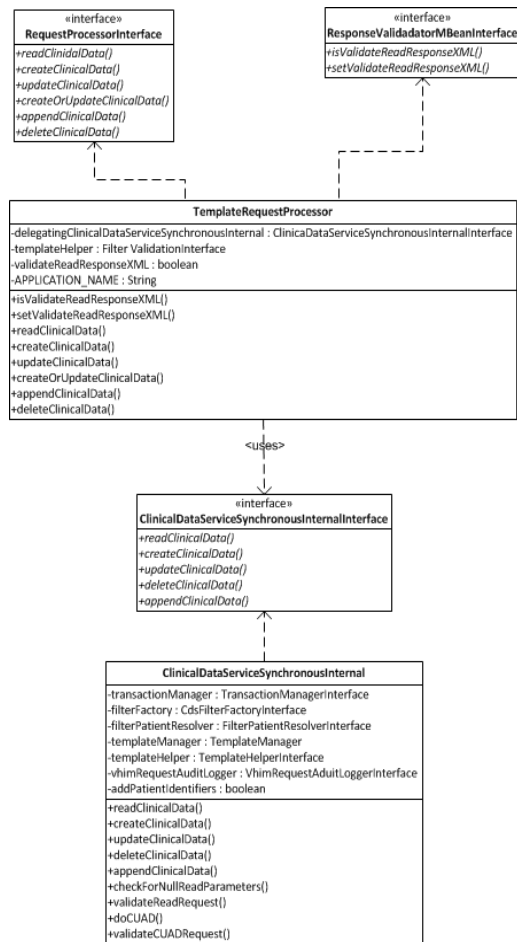
```
<bean id="requestProcessor" class="gov.va.med.cds.request.TemplateRequestProcessor">
    <property name="delegatingClinicalDataServiceSynchronousInternal"
ref="clinicalDataServiceSynchronousInternal" />
    <property name="templateHelper" ref="templateHelper" />
    <property name="filterManager" ref="filterManager" />
</bean>
```

The following details the ClinicalDataServiceSynchronousInternal configuration:

```
<bean id="clinicalDataServiceSynchronousInternal"
class="gov.va.med.cds.internal.ClinicalDataServiceSynchronousInternal">
    <property name="transactionManager" ref="transactionManager" />
    <property name="filterFactory" ref="filterFactory" />
    <property name="filterPatientResolver" ref="filterPatientResolver" />
    <property name="templateManager" ref="templateManager" />
    <property name="templateHelper" ref="templateHelper" />
    <property name="cdsAppName" ref="cdsAppName" />
    <property name="namespacesMap" ref="namespacesMap" />
    <property name="nestedElementWrapperUtil" ref="nestedElementWrapperUtil" />
    <property name="emptyElementListMap" ref="emptyElementListMap" />
</bean>
```

The following diagram specifies the HDR 3.x CDS Request processor Component functionality.

**Figure 3 CDS Request Process Component**



## 6.2.1.6 Database Repository

### 6.2.1.6.1 Database Repository Overview

This section provides an overview of the CDS Template and Filter Cache components. The cache is a component that transparently stores data so that future requests for that data can be served faster. The template and filter cache implementations initialize at application startup and expose in-memory cache mechanisms and management methods that store static VIM template and filter reference data derived from external services such as the Template and Filter Service (TAFS).

The concept and capability of ad hoc filters (Filter100) within the CDS framework provides a more flexible read interface to the system. In order to implement an ad hoc Filter100-based filter for a client, one must create the new Filter Meta Data for the filter, referencing Filter100.xsd as the filter schema, and load the few filter schema into the TAFS. Once the filter is hosted by the TAFS, configuration changes are applied to the CDS framework such that requests for the new template are processed by the criteria-based query strategy in the persistence subsystem. Once the template is available and appropriate configuration changes have been made, the client is

able to issue Filter100-based filters to the CDS framework that allow for filtering by patient or across patients by one or more date ranges, strings, or lists of strings. Filter parameters are specified using the optional query parameters elements defined in the Filter100 schema. Refer to the Filter100 XML schema definition document for specific information about how to specify filter parameters. The approach for providing flexible filtering of HDR data is done by using the naming conventions defined by the logical model to map filter parameters to query parameters that the database understands. In the event that a parameter is specified that is not part of the model, an error will be returned by the persistence subsystem of the CDS service. Currently, the ad hoc capabilities supported by the CDS Filter100 feature only supports filtering by entry point template elements. This is not obvious from the logical model itself, but is made apparent in the mapping files and entity-relationship diagrams (ERD) provided to the client. If a client tries to filter using a non-entry point model element, the query will fail and an error will be returned by the CDS persistence subsystem.

### **Template Cache**

Templates are XML Schema Documents (XSDs) created and validated by the CDS team in collaboration with client application teams. The template manager component manages and provides access to the template cache implementation. The templates define the content and semantics of clinical data mediated by CDS. The cache implementation is a realization of a CDS LRUHashMap object that wraps a Java LinkedHashMap object, keyed by the template ID, and is associated to a template meta data model object.

### **Filter Cache**

Filters are XML Schema Documents (XSDs) created and validated by the CDS team in collaboration with client application teams. The filter manager component manages and provides access to the filter cache implementation. The filter is created from the input XML document in the Request Processing layer and determines which records are returned to the client request. The cache implementation is a realization of a CDS LRUHashMap object that wraps a Java LinkedHashMap object, keyed by the filter ID, and is associated to a filter meta data model object.

#### **6.2.1.6.2 Database Repository Module Design**

Both the template and filter cache components implement a Least Recently Used (LRU) cache algorithm to manage how cache entries are discarded when new entries are added to the cache or the replacement algorithm is satisfied. The LRU algorithm will discard the least recently used entries first, which in this implementation is the object that has the oldest cache entry timestamp.

### **Template Cache**

An instance of the TemplateCache class represents the template cache implementation. The TemplateCache implements the TemplateCacheInterface interface initialized with an instance of a CDS LRUHashMap<K, V> object by the Spring container at application startup. The significant management method within the TemplateCache object is the loadSchemaIntoCache method. This method takes a TemplateMetaDataInterface object parameter and determines if the template schema associated to the meta data object exists within the cache. If it does not, it creates a TemplateCacheModelInterface object and inserts an entry into the cache.

The cache data container is implemented with the CDS `LRUHashMap<K, V>` class that extends a Java `LinkedHashMap` class. For the template cache, the `LRUHashMap<K, V>` object is defined as an instance of a `LRUHashMap<String, TemplateCacheModelInterface>` object. The `LRUHashMap<K, V>` is created with an initial entry size of 100 set within the Spring container. It has management methods to resize the map, and to remove the oldest entry in keeping with the LRU cache algorithm semantics. Entries within the cache are keyed by template ID. Each unique template ID key is associated to an implementation of the `TemplateCacheModelInterface` model, which keeps a template meta data realization of the `TemplateMetaDataInterface` class, and schema information in a `SchemaHelper` object for the template identified by the key.

The `SchemaHelper` holds a single template representation (metadata, compiled XML schema, and symbol map). The `TemplateCache`, using a layered initialization pattern, builds a cache of `SchemaHelpers` based on a configurable TAFS implementation; it provides access to the metadata, schemas (and validation logic), and symbol map through its implementation of the `MetaDataProviderInterface`, `SchemaSymbolMapProviderInterface`, and `TemplateValidatorInterface`.

### **Filter Cache**

The `FilterMemoryCache` object represents the filter cache implementation. The `FilterMemoryCache` implements the `FilterMemoryCacheInterface` interface and is initialized with an instance of the CDS `LRUHashMap<K, V>` object by the Spring container at application startup. The significant management method within the `FilterMemoryCache` object is the `loadFilterMetaDataIntoMemoryCache` method. This method takes a `FilterMetaDataInterface` object parameter and determines if the filter meta data object associated to the filter ID value exists within the cache. If it does not it inserts an entry into the cache.

The cache data container is implemented with the CDS `LRUHashMap<K, V>` class that extends a Java `LinkedHashMap` class. For the filter cache, the `LRUHashMap<K, V>` object is defined as an instance of a `LRUHashMap<String, FilterMetaDataInterface>` object. The `LRUHashMap<K, V>` is created with an initial entry size of 100 set within the Spring container. It has management methods to resize the map, and to remove the oldest entry in keeping with the Least Recently Used cache algorithm semantics. Entries within the cache are keyed by filter ID. Each unique filter ID key is associated to an implementation of the `FilterMetaDataInterface` object model, which keeps VIM version, and entry filter data and a `!String` representation of the filter schema.

#### **6.2.1.6.3 Database Repository Processing**

### **Template Cache**

For every domain-specific read request for clinical data, the CDS template manager component uses the associated template. On application startup, the template cache is populated from the TAFS service, which is the authoritative template data source. Because a template is static (a new template is created and the old one is retired) and will be used for many requests, performance is gained by caching these template instances. CDS supports these paradigms for populating this cache:

- Populate the cache with all active templates, retrieved from TAFS, when the application server container holding CDS starts up.

- When a request references a template not found in the cache, CDS retrieves the template from TAFS and adds it to the cache.

When the template manager component calls the `loadSchemaIntoCache` method on the `TemplateCache` object to load a template into cache, the `TemplateCache` object first retrieves the `schemaHelper` object associated with the template ID, then determines if the cache contains the template. If the template already exists, the method exits. If the template does not exist, a `TemplateCacheModelData` object is created and initialized with the `TemplateMetaDataInterface` object, `schemaHelper` and cache entry timestamp and entered into the cache. Likewise, when a template is requested from the cache, the template ID is given and used as the key to determine if the cache contains the template. The template is returned if it exists, and a null value if not. Refer to the template manager component for initialization details.

## Filter Cache

The filter cache supports the same paradigm as the template cache. On application startup, the filter cache is populated from the TAFS service, which is the authoritative filter data source. The filter manager component calls the `loadFilterMetaDataIntoMemoryCache` method on the `FilterMemoryCache` object to load a filter into the memory cache. The `FilterMemoryCache` object first determines if the filter is present in the cache by the filter ID. If the filter is in the cache, the method exits. If the filter is not in the cache, the `FilterMetaDataInterface` object is entered into the cache keyed by the filter ID. Refer to the filter manager component for initialization details.

### 6.2.1.6.4 Database Repository Local Data Structures

The following list displays the `TemplateCache` configuration:

```
<bean id="templateManager" class="gov.va.med.cds.template.TemplateManager">
  <constructor-arg index="0" ref="templateService" />
  <constructor-arg index="1" ref="templateCacheManager" />
  <constructor-arg index="2" ref="templateCache" />
</bean>

<bean id="templateService" class="gov.va.med.cds.template.TemplateServiceSimulator">
  <!-- If this property is missing, test templates won't be loaded -->
  <property name="testTemplates">
    <props>
      <!--
      If this prop is missing, then it will default to the local directory ("./")
      This can be either a path to a single file or a directory containing files
      with the suffix .meta. Each file contains information about where the
      template jar file is located and metadata about the template as NVP.
      -->
      <prop key="path">templatecache</prop>
      <!-- If not included, the default for this property is "false" -->
      <prop key="loadTestTemplates">true</prop>
    </props>
  </property>
</bean>
```

```

    </property>
</bean>

<bean id="templateCacheManager"
class="gov.va.med.cds.template.HibernateOracleTemplateCachePersistence">
    <property name="sessionFactory" ref="hdr2TfsSessionFactory"/>
</bean>

<bean id="templateCache" class="gov.va.med.cds.template.TemplateCache">
    <constructor-arg>
        <bean class="gov.va.med.cds.util.LRUHashMap">
            <!-- LRU cache size -->
            <constructor-arg index="0" value="100" />
        </bean>
    </constructor-arg>
</bean>

```

The following list displays the FilterMemoryCache configuration:

```

<bean id="filterManager" class="gov.va.med.cds.filter.FilterManager">
    <constructor-arg index="0" ref="filterService" />
    <constructor-arg index="1" ref="filterCachePersistence" />
    <constructor-arg index="2" ref="filterMemoryCache" />
</bean>

<bean id="filterCachePersistence"
class="gov.va.med.cds.filter.HibernateOracleFilterCachePersistence">
    <property name="sessionFactory" ref="hdr2TfsSessionFactory"/>
</bean>

<bean id="filterMemoryCache" class="gov.va.med.cds.filter.FilterMemoryCache">
    <constructor-arg index="0">
        <bean class="gov.va.med.cds.util.LRUHashMap">
            <!-- LRU cache size -->
            <constructor-arg index="0" value="100" />
        </bean>
    </constructor-arg>
</bean>

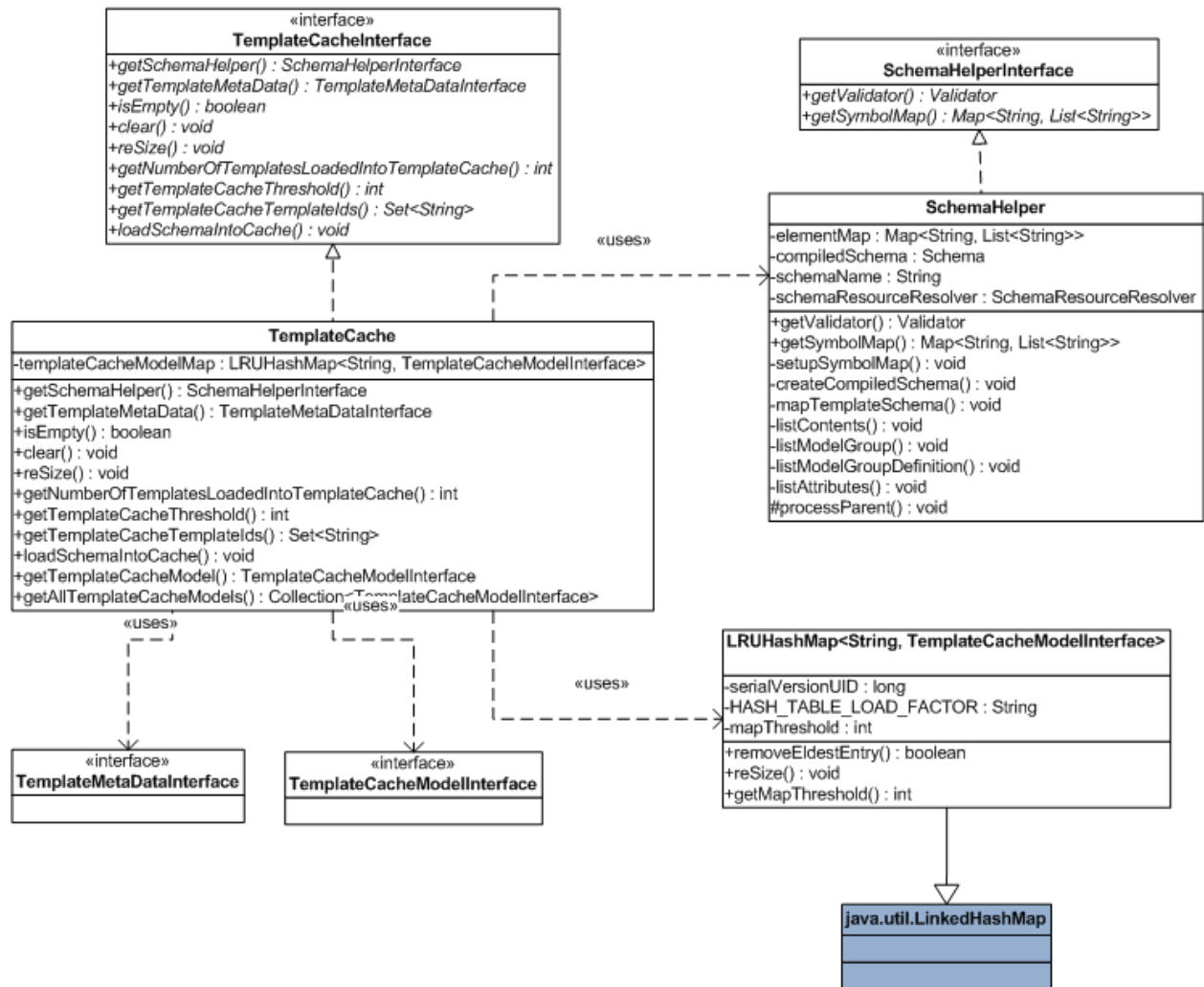
```

#### 6.2.1.6.5 Database Repository Module/Other Diagrams

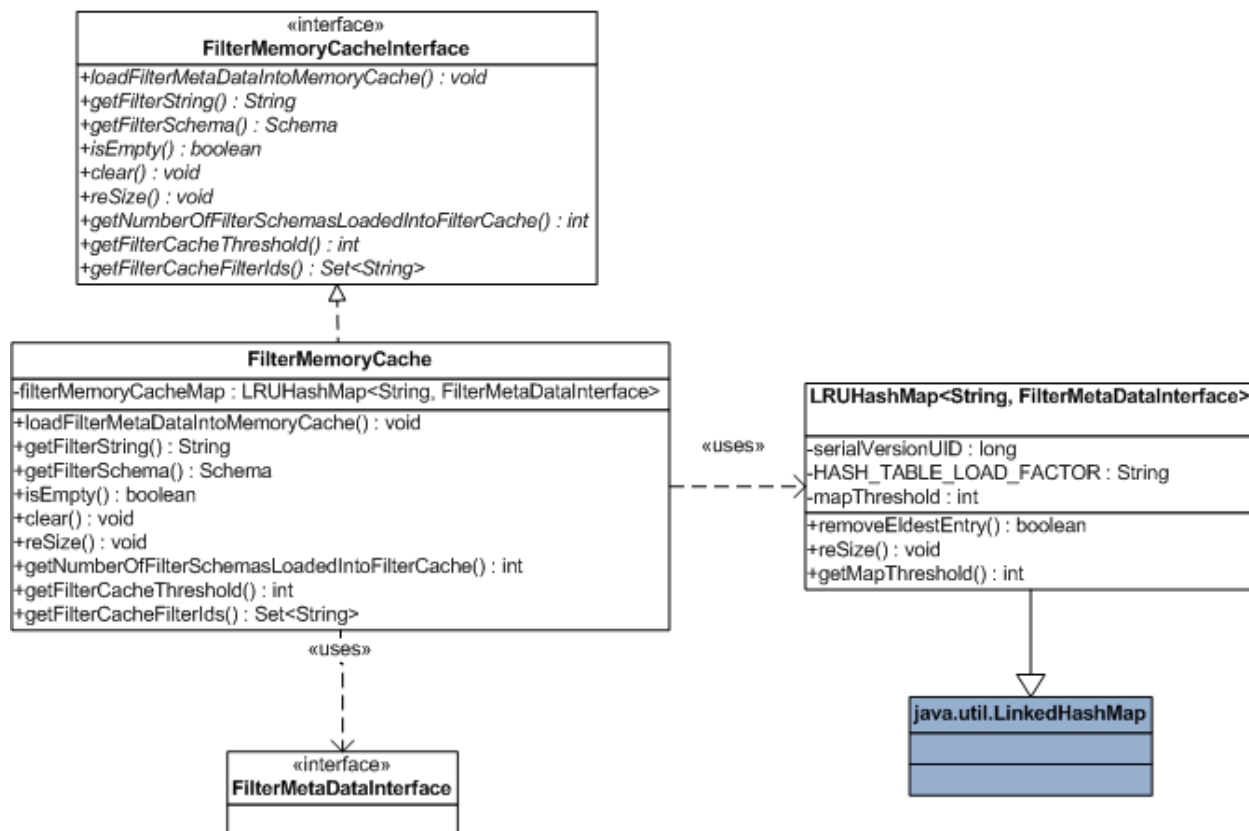
The following diagrams represent the template and filter cache module and its process flow:



**Figure 4 Template Cache Class Diagram**



**Figure 5 Filter Cache Class Diagram**



Refer to the template manager and filter manager process documentation for sequence diagram views.

## 6.2.1.7 Template and Filter Service Client

### 6.2.1.7.1 Template and Filter Service Client Overview

The CDS client API is defined by a small number of operations, one each for record read, create (which internally determines whether the operation is a record update), and 'isAlive' (confirms that the CDS service is available). The content of the parameters passed within the operations specifies the mediation of these operations. Each of these parameters is a string, either a single value or an XML document with defined content. This is true of the template and filter service client components.

The template and filter service client provide a service interface that enables clients to determine if the service is available, get active template and filter IDs, get template and filter metadata and get available supported VIM versions.

### 6.2.1.7.2 Template and Filter Service Client Module Design

Both the template and filter service clients follow a similar implementation model that is initialized by Spring configuration. The significant logic for both the template and filter client service component is implemented with the single WtfServiceManagement class (see below). The class implements the TemplateServiceInterface, FilterServiceInterface and

WtfServiceInterface interfaces which enable instances of the class to provide functionality for both template and filter service clients.

When the Spring container initializes the templateService and filterService beans for the client context, the implementation object for these beans is an instance of the WtfServiceManagement class. The 'tfsService' property on this class is initialized with an instance of a service delegate object, the TfsServiceFactory (see the Template and Filter Service Client Local Data Structures section below), which delegates client service requests to an instance of the TemplateFilterServiceDelegate which in turn implements the java.rmi.Remote interface.

#### **6.2.1.7.3      Template and Filter Service Client Processing**

On initialization, the Spring container creates instances of the WtfServiceManagement class for properties within the application context called 'filterService' and 'templateService'. When clients obtain instances of these properties and call methods as declared by the TemplateServiceInterface or the FilterServiceInterface, the logic is delegated to the WtfServiceManagement class. Depending on the method called on this object by the client, fulfillment of the request is passed to a service delegate object, the TfsServiceFactory object. Within the TfsServiceFactory object, delegation of the request to the application server is performed by the TemplateFilterServiceDelegate object that is initialized when the factory is created by the Spring context. The TemplateFilterServiceDelegate object is the CAIP defined TFS web service business delegate object.

#### **6.2.1.7.4      Template and Filter Service Client Local Data Structures**

The following local data structure displays the Template Service Client Context Configuration:

```
<bean id="templateService" class="gov.va.med.tfs.client.WtfServiceManagement">
  <property name="tfsService" ref="tfsServiceDelegate" />
</bean>
```

The following local data structure displays the Filter Service Client Context Configuration:

```
<bean id="filterService" class="gov.va.med.tfs.client.WtfServiceManagement">
  <property name="tfsService" ref="tfsServiceDelegate" />
</bean>
```

The following local data structure displays the TfsServiceFactory Context Configuration:

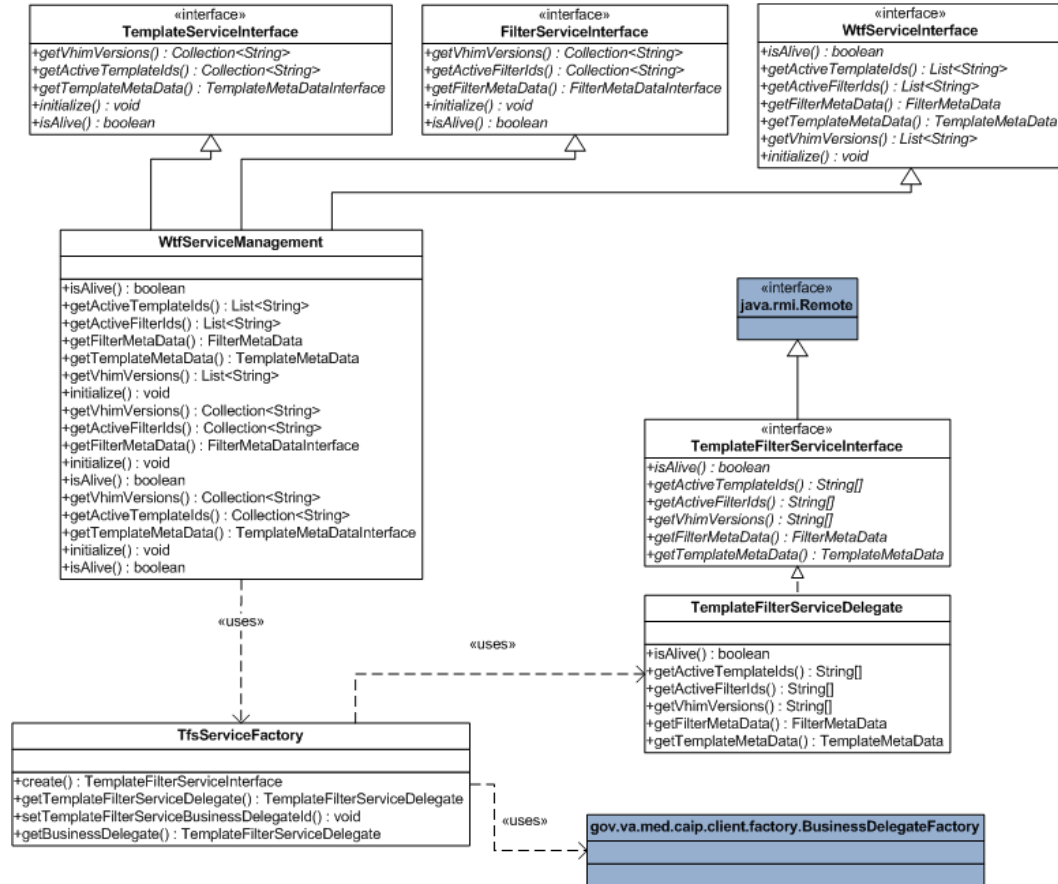
```
<bean id="tfsServiceDelegate" factory-bean="tfsServiceFactory" factory-method="create" lazy-
init="true">
  <constructor-arg name="tfsWebserviceTimeoutSecs" value="10" />
</bean>

<bean id="tfsServiceFactory" class="gov.va.med.tfs.client.TfsServiceFactory" lazy-init="true"/>
```

#### **6.2.1.7.5      Template and Filter Service Client Module/Other Diagrams**

The following figure represents the Template and Filter Service Class process flow:

**Figure 6 Template and Filter Service Class Diagram**



## 6.2.1.8 Request Validator

### 6.2.1.8.1 Request Validator Overview

This section provides an overview of the CDS Request Validation logic. “Request validation” refers to checking template and filter parameter objects for validity against parameters that are passed in as part of the Read request. In the case of Write request processing, the write templates are validated. This validation occurs before the request executed against a data source. The validation process also includes loading templates and filters into their respective caches if the cache objects have not been previously loaded

The template manager validates the template and confirms that the templateID passed in as part of the Read request refers to a valid read template. Within this process a TemplateMetaDataInterface object is retrieved from the template cache and validated for the request templateID, a validation exception is thrown if the templateID is not valid. For a further discussion on the template cache, see the template cache component documentation.

The filter manager validates filter objects. Like the template manager, the filter manager confirms that the filterID passed in as part of the Read request refers to a valid read filter. A FilterMemoryCacheInterface object confirms that the filterID is valid, and uses a schema object to validate the filter XML associated with the Read request. If the filterID is not valid, a validation exception is thrown. For a further discussion on the filter cache, see the filter cache component documentation.

#### **6.2.1.8.2 Request Validator Module Design**

Validation for the template and filter objects occurs within different request workflow objects. Template validation occurs within the Clinical Data Service layer by objects implementing the ClinicalDataServiceSynchronousInternalInterface interface, which in this case is the ClinicalDataServiceSynchronousInternal class. The class implements a readClinicalData method from the interface which ultimately delegates validation to a validateReadTemplateId method on the TemplateManager. Logic within the method validates the templateID and retrieves a TemplateMetaDataInterface object from the template cache.

Filter validation occurs within the Clinical Data Service layer in two parts. As with the template validation, filter validation that occurs within the ClinicalDataServiceSynchronousInternal object after the templateID is validated. This consists of checking that the collection of domain entry points contained in the filter are valid for the given templateID. Filter validation also occurs at the TemplateRequestProcessor layer. The TemplateRequestProcessor is initialized with an instance of the filter manager and delegates validation of the filterID and filter XML to the validateFilterXml method on the manager.

#### **6.2.1.8.3 Request Validator Template Processing**

Validation of the write template starts within the createOrUpdateClinicalData method of the ClinicalDataServiceSynchronousInternalInterface object.

Validation of the template starts within the readClinicalData method of the ClinicalDataServiceSynchronousInternalInterface object. After a cdsFilter object is created within the method, a private validateReadRequest method is called with templateId, requestId and cdsFilter parameters. The actual validation of the templateID is delegated to the validateReadTemplateId method on the TemplateManager instance. The manager attempts to obtain a TemplateMetaDataInterface object from the template cache. If the templateID is not valid, then a validation exception is thrown.

#### **6.2.1.8.4 Request Validator Filter Processing**

Validation of the filterID and filter XML starts within the TemplateRequestProcessor's readClinicalData method, which delegates the validation to the FilterManager's validateFilterXml method. Within this method, the FilterManager uses an instance of a FilterMemoryCacheInterface object to get the filter schema object associated with the passed-in filterID parameter. If the schema cannot be found by the filterID, a validation exception is thrown. The schema object is used to obtain a !Validator object which is used to validate the filter XML data. A validation exception is thrown if the XML data does not validate. The filter is also validated in the Clinical Data Service layer by the ClinicalDataServiceSynchronousInternal object's validateReadRequest method. Once the templateID has been validated, the read request logic continues by getting a collection of template entry points from the



filter XML document against an appropriate filter schema within the read request workflow. It implements management methods that initialize, clean, and resize the filter cache. For details on how the LRU filter cache component functions, please see the filter cache component documentation.

#### **6.2.1.9.2 Filter Manager Module Design**

The filter manager object is realized by the `FilterManager` class. It implements the interfaces `FilterManagerMBeanInterface`, and `FilterValidatorInterface`, which provide the methods to initialize and manage the behavior of the filter cache, and provide validation of the filter ID and filter XML to objects in the Read request workflow. The Spring container initializes the filter manager on context startup with instances of objects that implement the interfaces `FilterServiceInterface`, `FilterMemoryCacheInterface`, `FilterCachePersistenceInterface`. See the Filter Manager Local Data Structures section below for the Spring context configuration.

The `FilterServiceInterface` interface provides functionality to get VIM versions, active filter IDs and filter metadata from the filter repository which in this case is accessed via the TAFS service. The significant method that the `FilterServiceInterface` interface provides is the `getFilterMetaData` method. The `FilterManager` uses this method to retrieve filter metadata from the TAFS service to insert into the filter cache.

The `FilterMemoryCacheInterface` object implementation represents the filter cache. The filter cache contains in-memory filter metadata objects, and management methods to load, retrieve, resize and clear the cache. Please see the documentation for the filter cache component.

The `FilterCachePersistenceInterface` object implementation represents the persistence store for filter metadata objects. The `FilterManager` uses the `FilterCachePersistenceInterface` object to retrieve filter metadata from a persistence store, into the filter cache object.

#### **6.2.1.9.3 Filter Manager Processing**

During startup of the Spring context, the `FilterManager` is injected with instances of the `FilterServiceInterface`, `FilterMemoryCacheInterface`, `FilterCachePersistenceInterface` interfaces. Within the constructor of the `FilterManager`, an `initialize` method is called that starts the process to initialize and load the filter cache. The `FilterManager` calls the `getVhimVersions`, `getActiveFilterIds` and `getFilterMetaData` methods on the `FilterServiceInterface` object which initializes and loads the cache with filter metadata objects. The `FilterManager` then calls the `loadAllFilterSchemasIntoMemoryCache` method on the `FilterMemoryCacheInterface` object which loads the filter schemas into the cache, completing the cache initialization process.

Within the read request workflow the `RequestProcessorInterface` object implementation is the `TemplateRequestProcessor` which is injected with an instance of an initialized `FilterManager` object on Spring context startup. As the `readClinicalData` method is called on the `TemplateRequestProcessor`, the `validateFilterXml` method is called on `FilterManager` object to validate the filter ID and filter XML. If the filter ID is not found, a `ValidationException` is thrown. If the filter metadata has not previously been loaded into the filter cache, it is loaded and the filter metadata object is returned. Please see the request validator component for request validation information.

#### 6.2.1.9.4 Filter Manager Local Data Structures

The following local data structures specify the configuration of the filter manager:

```
<bean id="filterManager" class="gov.va.med.cds.filter.FilterManager">
  <constructor-arg index="0" ref="filterService" />
  <constructor-arg index="1" ref="filterCachePersistence" />
  <constructor-arg index="2" ref="filterMemoryCache" />
</bean>

<bean id="filterCachePersistence"
class="gov.va.med.cds.filter.HibernateOracleFilterCachePersistence">
  <property name="sessionFactory" ref="hdr2TfsSessionFactory"/>
</bean>

<bean id="filterMemoryCache" class="gov.va.med.cds.filter.FilterMemoryCache">
  <constructor-arg index="0">
    <bean class="gov.va.med.cds.util.LRUHashMap">
      <!-- LRU cache size -->
      <constructor-arg index="0" value="100" />
    </bean>
  </constructor-arg>
</bean>
```

#### 6.2.1.9.5 Filter Manager Module/Other Diagrams

The following diagram displays the filter manager class process flow:

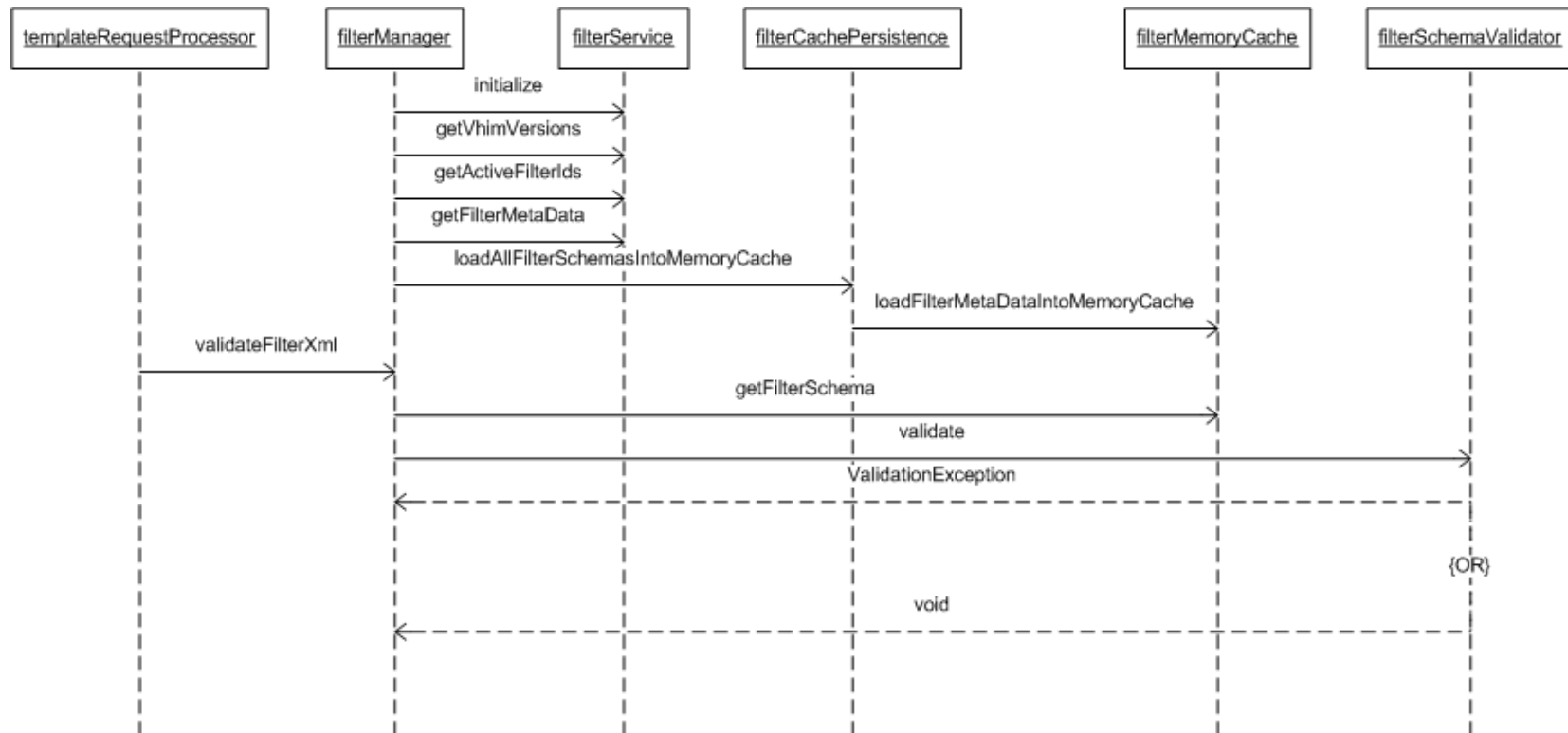


**Figure 9 Filter Manager Class Diagram**



The following figure displays the filter manager sequence:

**Figure 10 Filter Manager Sequence Flow**



## **6.2.1.10 Template Manager**

### **6.2.1.10.1 Template Manager Overview**

This section provides an overview of the CDS Template Manager component. The Template Manager coordinates initialization and management behavior of the template cache. It initializes the template service, which is an interface into the authoritative template source Template and Filter Service (TAFS). The Template Manager uses the service to get VIM versions, active template IDs, and template metadata for a given template ID. The Template Manager also validates a given template ID parameter against an appropriate template within the write/update/delete/read request workflow. For details on how the LRU template cache component functions, please see the template cache component documentation.

### **6.2.1.10.2 Template Manager Module Design**

The Template Manager object is realized by the TemplateManager class. It implements the interfaces TemplateMetaDataProviderInterface, RequestValidatorInterface, ResponseValidatorInterface, and TemplateManagerMBeanInterface. These interfaces provide the TemplateManager the method signatures to initialize and manage the behavior of the template cache, and to provide validation services to objects in the Read request workflow. When the Spring container initializes the application context, the TemplateManager is injected with implementations of the TemplateServiceInterface, TemplateCacheInterface, and TemplateCachePersistenceInterface interfaces.

The TemplateServiceInterface interface provides functionality to get VIM versions, active template IDs, and filter metadata for a given template ID from the template repository accessed via the TAFS service. The significant method that the TemplateServiceInterface interface provides is the getTemplateMetaData method. The TemplateManager uses this method to retrieve template metadata from the TAFS service to insert into the template cache.

The TemplateCacheInterface object implementation represents the template cache. The template cache contains in-memory template metadata objects, and the management methods to load, retrieve, resize and clear the cache. Please see the documentation for the template cache component.

The TemplateCachePersistenceInterface object implementation represents the persistence store for template metadata objects. The TemplateManager uses the TemplateCachePersistenceInterface object to retrieve template metadata from a persistence store, into the template cache.

### **6.2.1.10.3 Template Manager Processing**

During startup of the Spring context, the TemplateManager is injected with instances of the TemplateServiceInterface, TemplateCacheInterface, and TemplateCachePersistenceInterface interfaces. Within the constructor of the TemplateManager, an initialize method is called that starts the process to initialize and load the template cache. The TemplateManager uses the instances of the template service and template cache persistence to load the cache with template metadata for each template ID found in the TAFS service.

Within the Read request workflow started by the TemplateRequestProcessor object, the readClinicalData method is called on the ClinicalDataServiceSynchronousInternal object by an object implementing the ClinicalDataServiceSynchronousInternalInterface interface, which in this case is the ClinicalDataServiceSynchronousInternal object. Within the readClinicalData method in the ClinicalDataServiceSynchronousInternal object, the validateReadTemplateId method is called on the TemplateManager. The TemplateManger in turn, calls the getTemplateMetaData method on the TemplateCachePersistenceInterface interface implementation object. If the template ID is invalid a ValidationException is thrown. If the template metadata object has not been previously loaded into the template cache, it is loaded, and the template metadata object is returned. Please see the request validator component for request validation information.

#### 6.2.1.10.4 Template Manager Local Data Structures

```
<bean id="templateManager" class="gov.va.med.cds.template.TemplateManager">
  <constructor-arg index="0" ref="templateService" />
  <constructor-arg index="1" ref="templateCacheManager" />
  <constructor-arg index="2" ref="templateCache" />
</bean>

<bean id="templateService" class="gov.va.med.cds.template.TemplateServiceSimulator">
  <!-- If this property is missing, test templates won't be loaded -->
  <property name="testTemplates">
    <props>
      <!--
      If this prop is missing, then it will default to the local directory ("./")
      This can be either a path to a single file or a directory containing files
      with the suffix .meta. Each file contains information about where the
      template jar file is located and metadata about the template as NVP.
      -->
      <prop key="path">templatecache</prop>
      <!-- If not included, the default for this property is "false" -->
      <prop key="loadTestTemplates">true</prop>
    </props>
  </property>
</bean>

<bean id="templateCacheManager"
class="gov.va.med.cds.template.HibernateOracleTemplateCachePersistence">
  <property name="sessionFactory" ref="hdr2TfsSessionFactory"/>
</bean>

<bean id="templateCache" class="gov.va.med.cds.template.TemplateCache">
  <constructor-arg>
    <bean class="gov.va.med.cds.util.LRUHashMap">
```

```

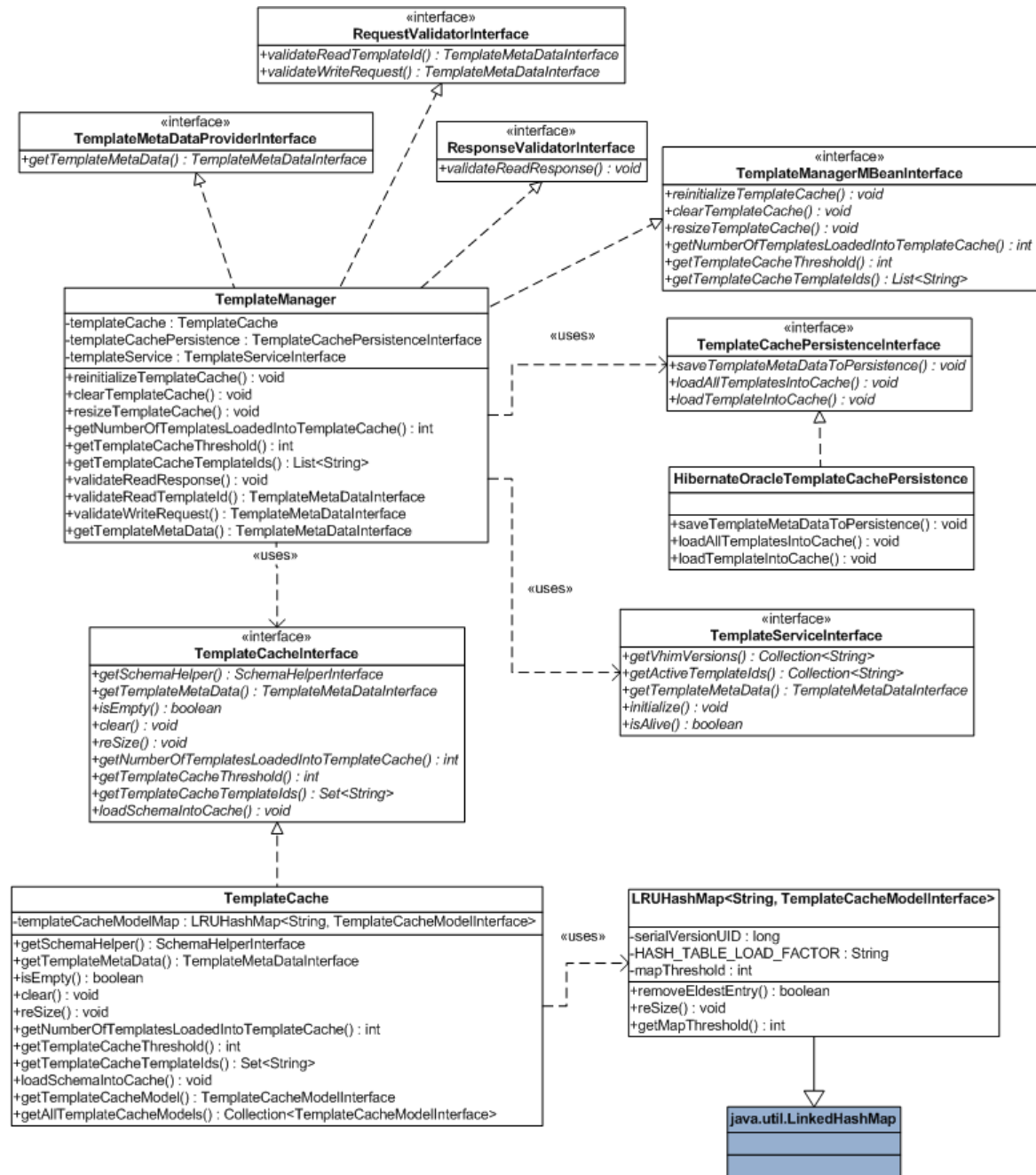
<!-- LRU cache size -->
<constructor-arg index="0" value="100" />
</bean>
</constructor-arg>
</bean>

```

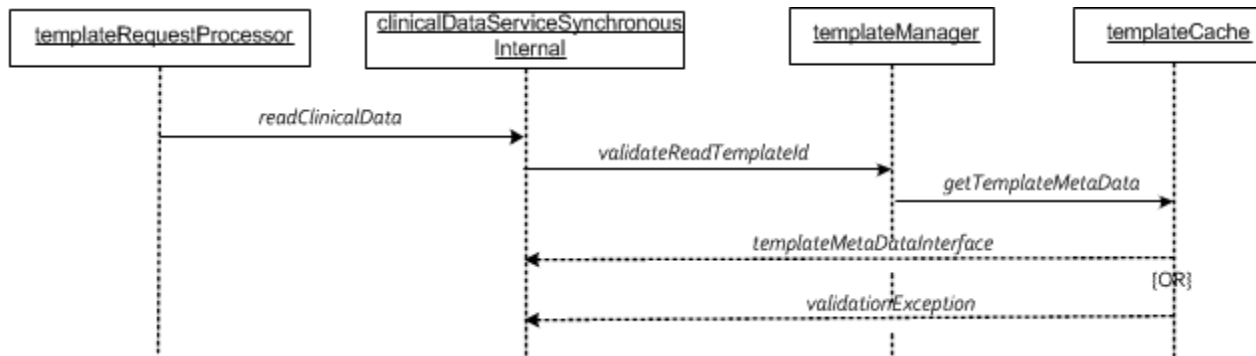
### 6.2.1.10.5 Template Manager Module/Other Diagrams

The following diagrams represent the Template Manager Class and Sequence Diagrams:

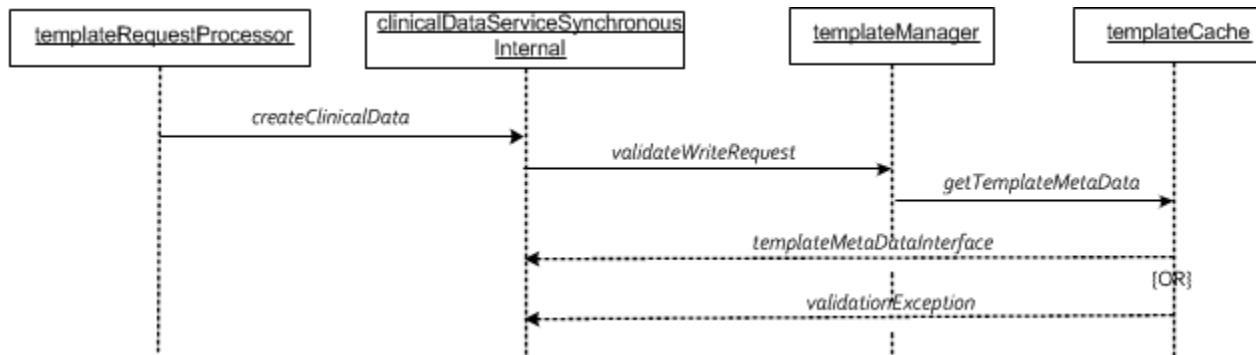
**Figure 11 Template Manager Class Diagram**



**Figure 12 Template Manager Read request processing Sequence Diagram**



**Figure 13 Template Manager Write/Update/Delete request processing Sequence Diagram**



### **6.2.1.11 Person Service Identity Handling**

This section is applicable Read request processing.

#### **6.2.1.11.1 Person Service Identity Handling Overview**

This section provides an overview of patient identity resolution that takes place during the processing of a CDS Read request. Read requests include patient identifiers for patients whose data is to be retrieved from various CDS data sources such as the HDR database and VistA systems. The various data sources associate clinical data for a patient with patient identifiers. A patient may have multiple patient identifiers and the Identity Management System (IdMS) provides services that enable its clients to obtain all corresponding identifiers given a single patient identifier. This capability enables CDS to return all clinical data for a patient given a single identifier in a request.

CDS clients may specify an unresolved patient identifier along with a list of excluded identifiers or may specify a list of resolved patient identifiers for each patient in a request through a request filter XML. CDS uses the 'GETCORRESPONDINGIDS' request on the Identity Management Web Service (IdM Service) in order to obtain all associated/corresponding identifiers for the given unresolved identifiers for each patient. In order to optimize performance, CDS restricts the number of calls to the IdM service to one call per unresolved identifier specified in the CDS filter. To facilitate this single call, once the corresponding identifiers have been obtained from the IdM service for an unresolved identifier, CDS modifies the request filter XML by replacing the unresolved identifiers with a list of resolved identifiers that do not match any specified excluded identifiers. Once all the unresolved identifiers have been replaced, the modified filter XML is returned and used in the downstream classes for performing clinical data retrieval based on the resolved list of identifiers for each patient. The rest of this section and its sub-sections detail the components and classes used to achieve the person service identity handling described above.

#### **6.2.1.11.2 Person Service Identity Handling Design**

When a Read request arrives at CDS's ClinicalDataServiceSynchronous EJB for processing, it consists of the response template details and the request filter XML with identifiers, dates and other filter parameters used for filtering data retrieved from various data sources. The EJB passes on the request to the DefaultClinicalDataServiceSynchronous POJO , for further processing. DefaultClinicalDataServiceSynchronous POJO delegates further processing to ClinicalDataServiceSynchronousInternal. ClinicalDataServiceSynchronousInternal will pass the filter to the FilterPatientResolver. It is here that any unresolved patient identifiers in the filter are resolved. The filter is updated with the resolved identifiers and passed to either a CDS database query or to a !VistAClinicalDataService for fulfilling the read request. If an exception is thrown from the FilterPatientResolver, all further processing stops and the exception is formatted into an error response, which is returned back to its caller.

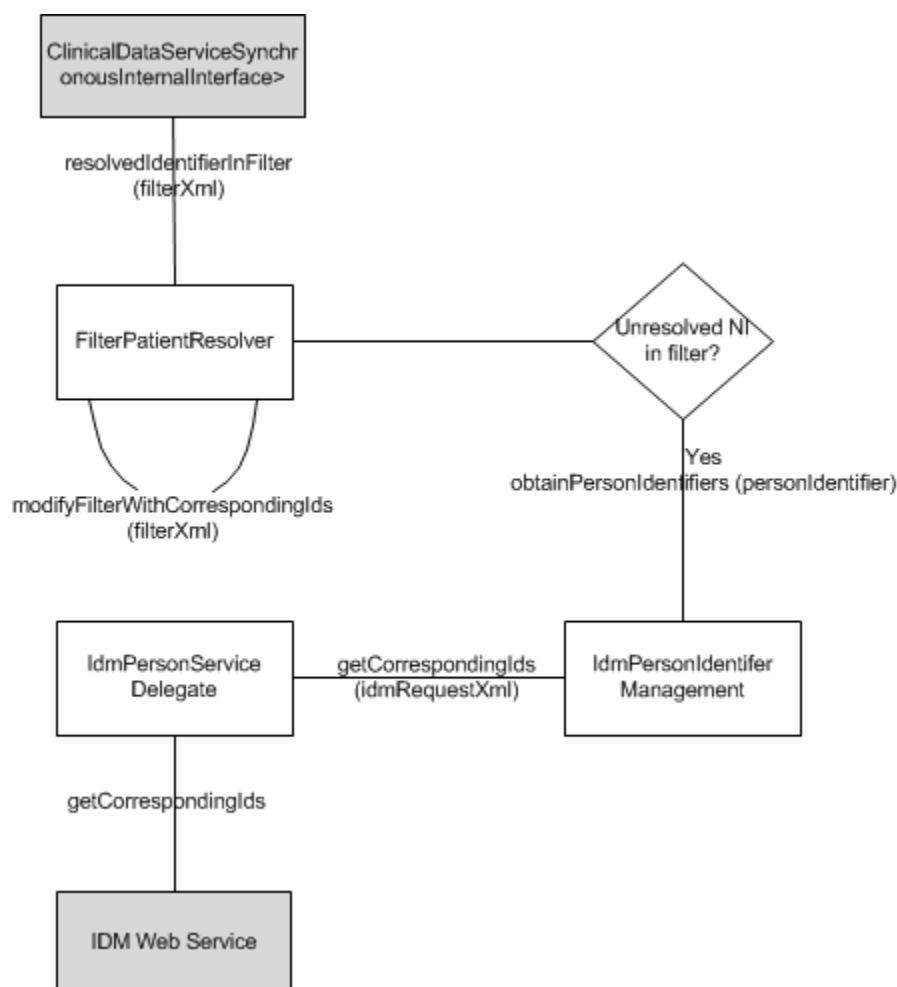
#### **6.2.1.11.3 Person Service Identity Handling Processing**

The FilterPatientResolver reads the request filter XML and determines if there are unresolved patient identifiers in the filter. If there are no unresolved patient identifiers, the FilterPatientResolver returns the unmodified original filter XML to the

ClinicalDataServiceSynchronousInternal class for further processing. Every unresolved patient identifier is assumed to be a national identifier and the IdM Web service is invoked to get the corresponding identifiers associated with that national identifier. Once the corresponding list of identifiers is returned from the IdM Web service, the FilterPatientResolver removes any excluded identifiers that were specified in the filter from the corresponding list of identifiers and modifies the filter by removing the unresolved identifier section for that patient and populating the resolved identifier section in the filter with the list of the corresponding identifiers obtained from the IdM Web service; excluding the specified excluded identifiers. Once all the unresolved identifiers have been resolved, the modified filter is returned to ClinicalDataServiceSynchronousInternal class for further read processing. If the IdM Web service is down or it cannot be connected to within a reasonable read duration or the IdM Web service returns an error response, an exception is thrown from the FilterPatientResolver with detailed error message identifying the reason for failure.

The following figure shows the overview of person service handling in the CDS Read path starting with the ClinicalDataServiceSynchronous class (the non-shaded boxes refer to the elements of this module):

**Figure 14 Person Service Identity Handling in CDS Read Path Diagram**





#### 6.2.1.11.4 Person Service Identity Handling Local Data Structures

The FilterPatientResolver reads and manipulates the 'patient' type in the CDS filter. The 'patient' type is defined as follows:

```
<xs:complexType name="patient">
  <xs:choice>
    <xs:element name="resolvedIdentifiers" type="filter:PersonIdentifier"
maxOccurs="unbounded"/>
    <xs:sequence>
      <xs:element name="NationalId" type="xs:string"/>
      <xs:element name="excludeIdentifiers" type="filter:ExcludeIdentifier"
minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

The 'NationalId' element in the above structure is taken to be the 'unresolved patient identifier' and is removed once the corresponding identifiers list minus the excluded identifiers are obtained and populated as 'resolvedIdentifiers' in the patient type.

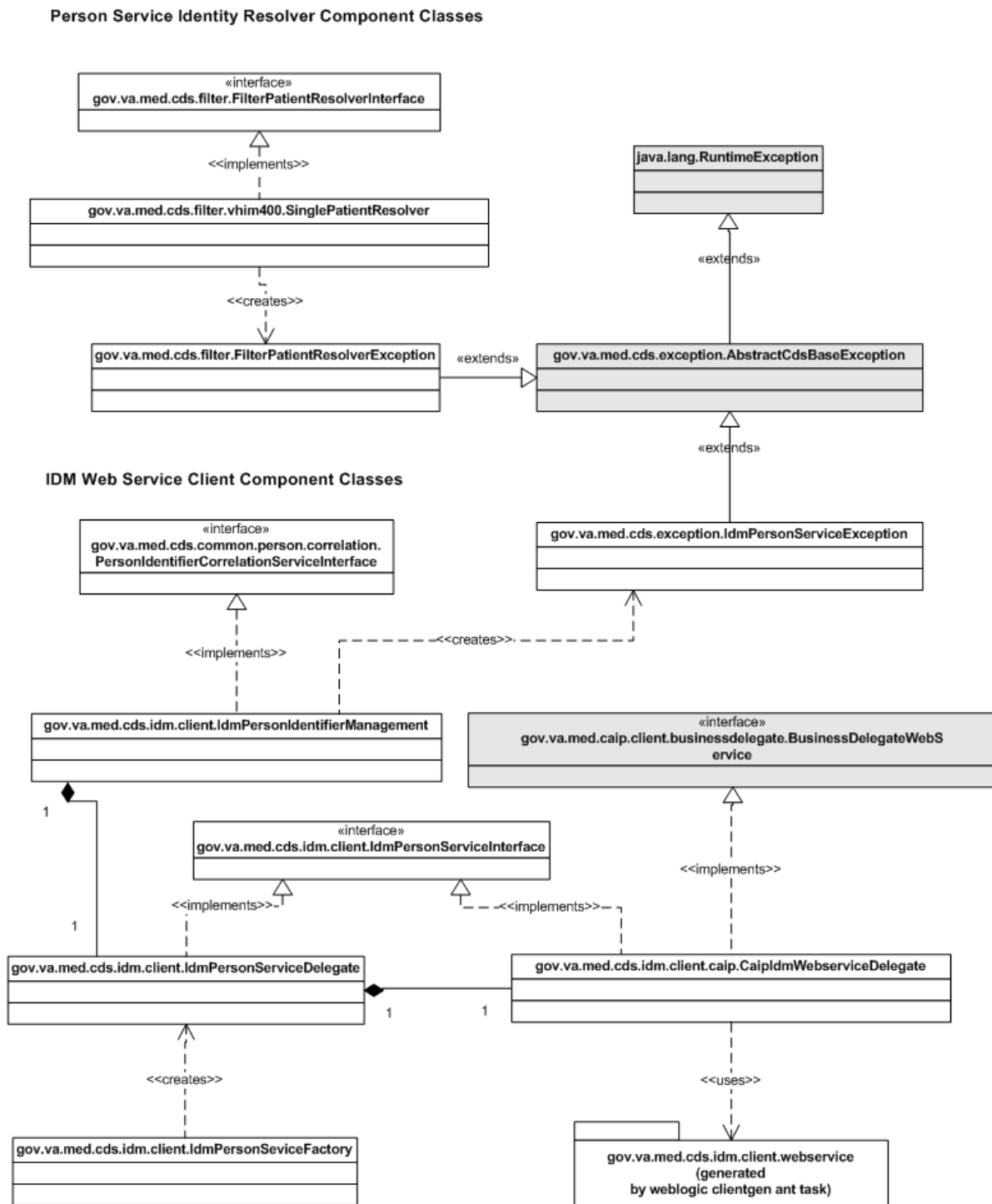
The other types used in the 'patient' type above are listed below for completeness:

```
<xs:complexType name="PersonIdentifier">
  <xs:sequence>
    <xs:element name="assigningAuthority" type="xs:string"/>
    <xs:element name="assigningFacility" type="xs:string"/>
    <xs:element name="identity" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ExcludeIdentifier">
  <xs:sequence>
    <xs:element name="assigningAuthority" type="xs:string"/>
    <xs:choice minOccurs="0">
      <xs:sequence>
        <xs:element name="assigningFacility" type="xs:string"/>
        <xs:choice minOccurs="0">
          <xs:element name="identity" type="xs:string"/>
        </xs:choice>
      </xs:sequence>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

### 6.2.1.11.5 Person Service Identity Handling Module/Other Diagrams

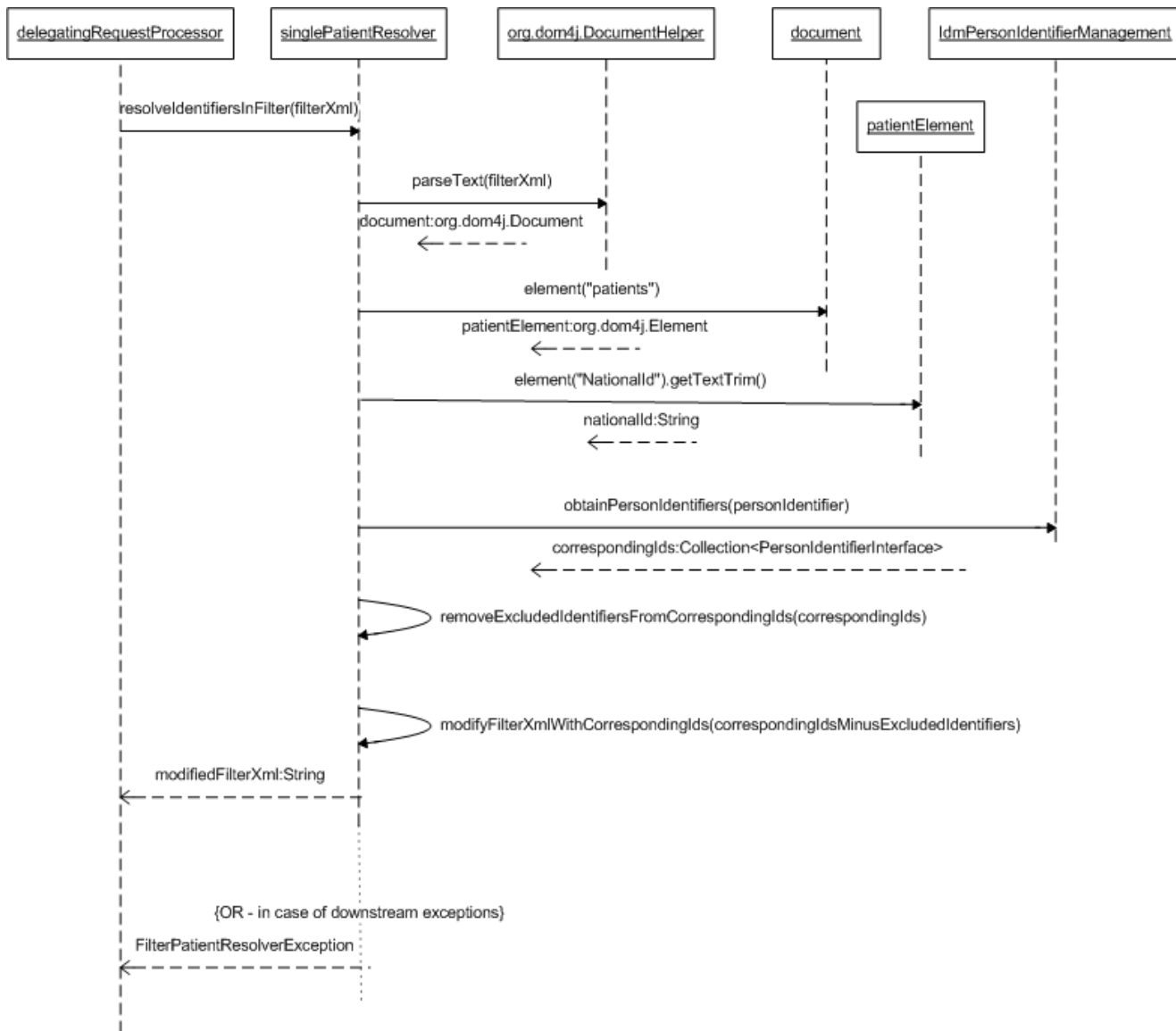
The following figures show the interfaces and classes that participate in person service identity handling:

**Figure 15 Person Service Identity Handling Class Diagram**



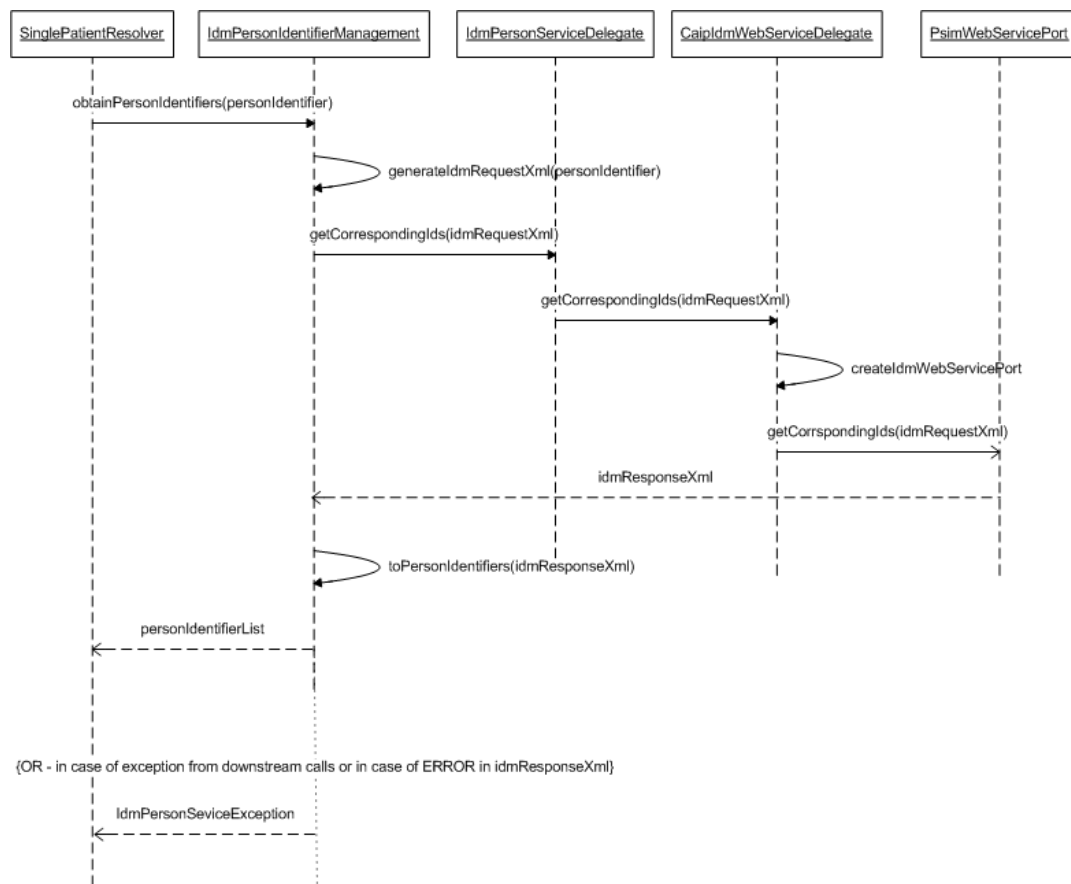
The following figure depicts the 'resolveIdentifiersInFilter' sequence diagram:

**Figure 16 Person Identity Handler Sequence Diagram**



The following figure depicts the 'obtainPersonIdentifiers' sequence diagram.

**Figure 17 Person Identity Handler Sequence Diagram**



## 6.2.1.12 SQL Based Filters

### 6.2.1.12.1 SQL Based Filters Overview

CDS clients use filter-based reads to get data stored in CDS data sources. The purpose of this module is to enrich the filter capability so clients can obtain specific data. This implies changing the current filter schema to optionally include query parameters specific to the entry points in the filter. The names of the parameters, as well as the values that may be specified, are based upon a standardized list provided by the Standards & Terminology Service (STS) team and are constrained by the functional requirements for each domain.

The SQL-based query filtering shall be initially introduced into the queries to VistA data sources. Once the success of VistA queries has been determined, queries to other CDS data sources shall also include SQL-based filtering.

### 6.2.1.12.2 SQL Based Filters Module Design

Filtering capability and its processing is currently part of CDS. The SQL-based filtering enhancements imply modifications to existing processes, files, classes and methods. The filter schemas are modified to include the additional query parameter values appropriate for the domain. The filter schemas are accessible from a map that is keyed on *filterId* that is unique to

each filter schema. The parameters of a CDS Read request are templateId, filterId, filter XML and requestId strings. When it is determined that the input values are not null or empty, the filterId is used to obtain the filter schema. The request filter XML is validated against that filter schema with the help of javax.xml.validation.Validator processor.

Note: The discussion of the Validator processor is beyond the scope of this design document. See documentation related to XML validation for further information.

Three new classes, FilterQueryMetadata, ParameterMap and QueryParameter are defined. These classes replace the current JAXB generated filter classes. Downstream classes that currently expect a JAXB created EntryFilter will be replaced with interfaces that expect the FilterQueryMetadata. The ParameterMap encapsulates the Java HashMap and provides methods for accessing the parameter names and parameter values based on the parameter names. The encapsulated HashMap holds a QueryParameter instance against each parameter name. The QueryParameter has the details of each parameter such as its name, value, and type. The ParameterMap method *getParameterValues(String parameterName)* returns a QueryParameter instance. The FilterQueryMetadata acts as a container for the ParameterMap. The FilterQueryMetadata replaces the JAXB generated EntryFilter instance encapsulated in the current CdsFilter object.

The additional parameter names and their values related to each filter are inserted into the persistence manager via Spring. The application accesses and uses this data when DefaultQueryWork runs its *buildQuery* method with session, query association name, and entry filter passed as parameters. The entry filter object has access to the map of the additional parameters. Together with required parameters, the additional parameters are used to dynamically build the query name.

```
// For each parameter, first required then additional the name of parameter will be added
// to the string to dynamically build a query name.

Object parameterValue = queryParameter.getValue();

queryName = String.format("%s.%s", queryName, filterParameterName);

// Get the named query from the session. throws Hibernate exception if the lookup fails.
Query query = session.getNamedQuery( queryName );
```

As result of extracting all the parameters to dynamically create a query name, a map is created that contains a complete list of all required and additional parameters as parsedname-value entries. If an expected parameter is missing, a Persistence exception is thrown. The next step is to set parameters for the named query using this map. Each named query takes a filter specific query parameter value. A query is created for each domain and can be located in the Hibernate mapping files. During the processing of the read request, the relevant named query is loaded by the DefaultQueryWork class for execution.

If the value of the query parameter is a type of List, the following method is called to set the value of the named parameter:

```
namedQuery.setParameterList(namedQueryParam, (Collection)queryParameter.getValue());
```

If the value of query parameter is not a List type, the following method is called to set the value of the named parameter:

```
namedQuery.setParameter(namedQueryParam, queryParameter.getValue());
```

The name of the named parameter (obtained from `query.getNamedParameters()` call) must match exactly with the name of the optional parameter listed under 'otherQueryParameters' element in filter XML.

For example, valid optional named parameters for the Text Integration Utility (TIU) domain that can be declared in the named queries for TIU read data (inside `hbm.xml` files) are: `documentClass` and `documentType`.

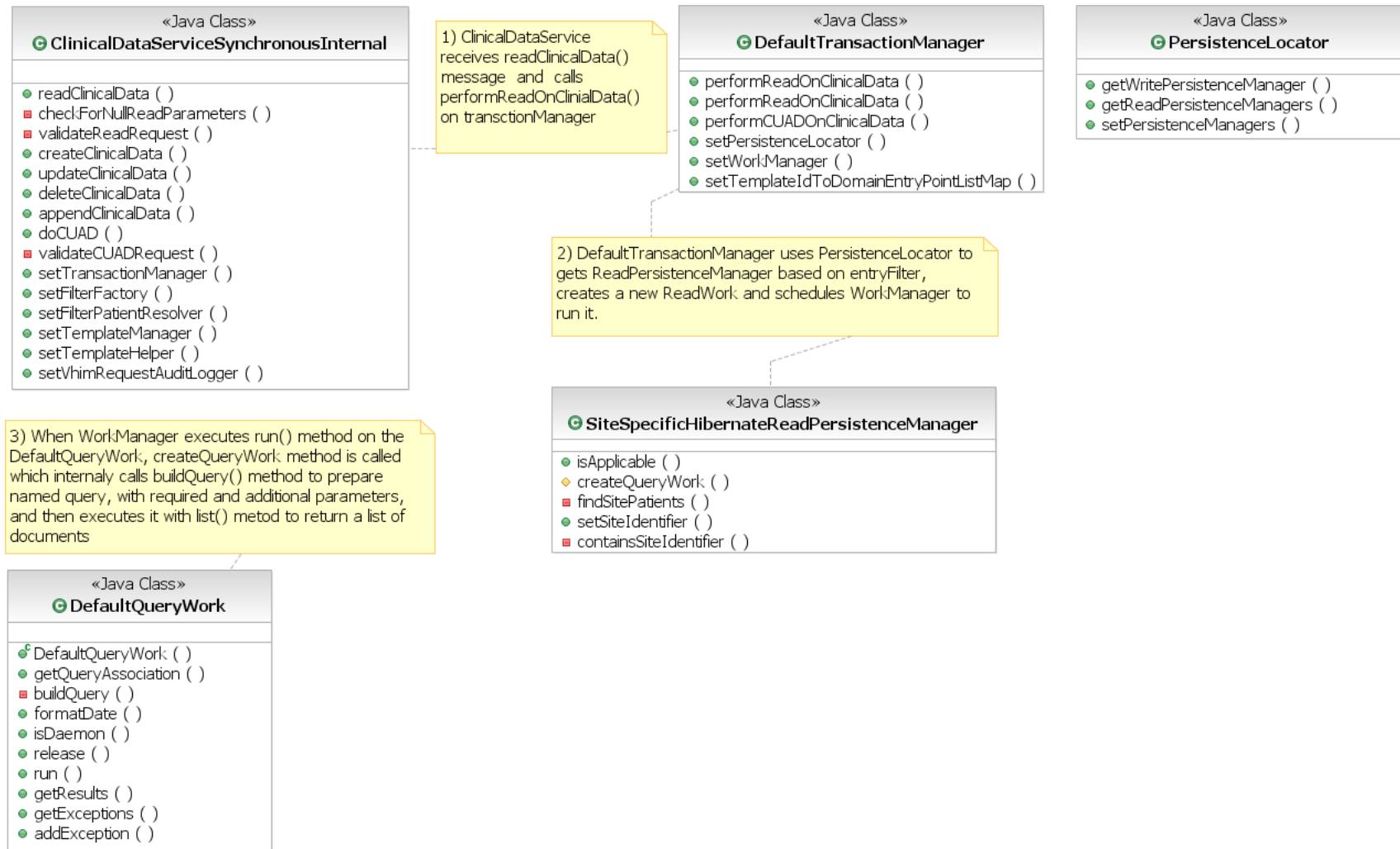
Spring injection is used to load the standardized list of parameter names and acceptable parameter values into a standardized map. The `CdsFilter` uses the Dom4j API and Java classes to populate the details of the `FilterQueryMetadata` and `ParameterMap` from the request XML filter.

#### **6.2.1.12.3 SQL Based Filters Processing**

The `ClinicalDataServiceSynchronous` class creates a CDS filter when processing a CDS Read request. The `FilterQueryMetaData`, the associated `ParameterMap`, and the `QueryParameter` instances are also created from the XML filter. The filter consists of resolved patient identifiers and entry filter detail information.

The Dom4j API and Java classes are used to parse and extract the filter details from the XML filter. The `DefaultQueryWork` object executes Hibernate queries based upon a query plan. The map of the parameter name-value pairs class is used to obtain parameter details about the query when it builds a query as shown in the following figure.

**Figure 18 Overview of SQL Based Filters Processing Flow**



A sequence of operations occurs when a CdsFilter is created by the VistAClinicalDataServiceSynchronous class during processing of a CDS read request.

The FilterQueryMetaData, the associated ParameterMap, and the QueryParameter instances are also created from the XML filter that consists of resolved patient identifiers and entry filter details.

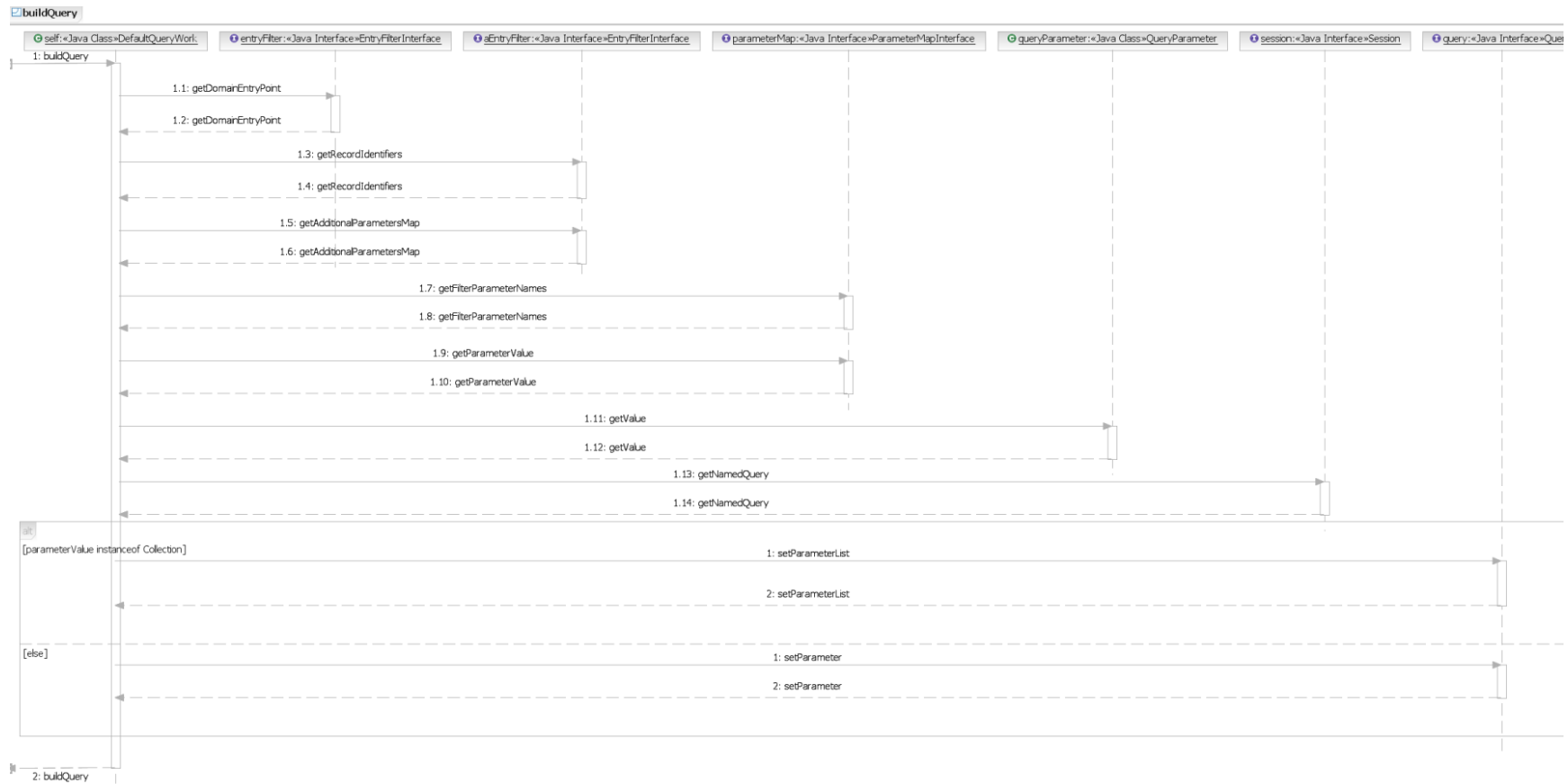
The Dom4j API and classes are used to parse and extract the filter details from the XML filter. The DefaultQueryWork object executes Hibernate queries based upon a query plan.

Parameter details about the query are created from a parameter name-value pairs class map.

The following figure depicts the sequence diagram class interaction during buildQuery() method calls.



**Figure 19 Sequence Diagram Class Interaction During an SQL Based Filter Method Call**



#### 6.2.1.12.4 SQL Based Filters Local Data Structures

The TIU filter schema 'TiuSinglePatientListDataFilter.xsd' is modified to include the optional parameters as follows:

```
<xs:complexType name="EntryFilter">
  <xs:sequence>
    <xs:element name="domainEntryPoint" type="filter:DomainEntryPoint"/>
    <xs:element name="startDate" type="filter:DateParameter" minOccurs="0"/>
    <xs:element name="endDate" type="filter:DateParameter" minOccurs="0"/>
    <xs:element name="otherQueryParameters" type="filter:OptionalParameters"
minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="queryName" type="xs:ID" use="required"/>
</xs:complexType>
```

The new 'AdditionalParameters' type in the TIU filter schema is specified below:

```
<xs:complexType name="AdditionalParameters">
  <xs:choice>
    <xs:element name="documentClass" type="filter:DocumentClassValues"/>
    <xs:element name="documentType" type="filter:DocumentTypeValues"
maxOccurs="unbounded"/>
  </xs:choice>
</xs:complexType>
<xs:simpleType name="DocumentClassValues">
  <xs:restriction base="xs:string">
    <!-- "PROGRESS NOTES" -->
    <xs:enumeration value="3"/>
    <!-- "DISCHARGE SUMMARY" -->
    <xs:enumeration value="244"/>
  </xs:restriction>
</xs:simpleType>
<xs:simpleType name="DocumentTypeValues">
  <xs:restriction base="xs:string">
    <!-- "ADVANCE DIRECTIVE" -->
    <xs:enumeration value="4696095"/>
    <!-- "CONSULT" -->
    <xs:enumeration value="4696107"/>
    <!-- "CRISIS NOTE" -->
    <xs:enumeration value="4697146"/>
    <!-- "DISCHARGE SUMMARY" -->
    <xs:enumeration value="4696112"/>
    <!-- "DO NOT RESUSCITATE" -->
    <xs:enumeration value="4696113"/>
    <!-- "LIVING WILL" -->
```

```

        <xs:enumeration value="4696119"/>
        <!--"NOTE"/-->
        <xs:enumeration value="4696120"/>
        <!--"RESCINDED ADVANCE DIRECTIVE"/-->
        <xs:enumeration value="4696125"/>
        <!--"RESCINDED DO NOT RESUSCITATE"/-->
        <xs:enumeration value="4696444"/>
    </xs:restriction>
</xs:simpleType>

```

Additionally, the entry filter type in the generic filter schema 'Filter100.xsd' now includes an 'otherQueryParameters' element of type 'OptionalParameters' as follows:

```

<xs:complexType name="EntryFilter">
    <xs:sequence>
        <xs:element name="domainEntryPoint" type="xs:string"/>
        <xs:element name="startDate" type="filter:DateParameter" minOccurs="0"/>
        <xs:element name="endDate" type="filter:DateParameter" minOccurs="0"/>
        <xs:element name="recordIdentifiers" type="filter:EntityIdentifier"
minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="XPathQuery" type="filter:XPathQuery" minOccurs="0"/>
        <xs:element name="otherQueryParameters" type="filter:OptionalParameters"
minOccurs="0"/>
    </xs:sequence>
    <xs:attribute name="queryName" type="xs:ID" use="required"/>
</xs:complexType>

```

The 'AdditionalParameters' type in the 'Filter100.xsd' is defined as follows:

```

<xs:complexType name="AdditionalParameters">
    <xs:sequence>
        <xs:element name="additionalParameter" type="filter:Parameter"
minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>

<xs:simpleType name="DateParameter">
    <xs:restriction base="xs:date"/>
    <!--xs:restriction base="xs:dateTime"/-->
    <!--xs:restriction base="xs:string">
        <xs:pattern value="((19\d{2})|(2\d{3}))((0[1-9])|(1[0-2]))((0[1-9])|([1-2])\d|([3[0-1]))"/>
    </xs:restriction-->
</xs:simpleType>

<xs:complexType name="Parameter">
    <xs:all>
        <xs:element name="value">
            <xs:simpleType>
                <xs:restriction base="xs:string"/>
            </xs:simpleType>
        </xs:element>
    </xs:all>
</xs:complexType>

```

```

        </xs:simpleType>
    </xs:element>
</xs:all>
<xs:attribute name="name" use="required"/>
<xs:attribute name="type"/>
</xs:complexType>
<xs:simpleType name="XPath">
    <xs:restriction base="xs:string"/>
</xs:simpleType>
<xs:simpleType name="FilterId">
    <xs:restriction base="xs:string"/>
</xs:simpleType>

```

The current Parameter type remains the same as the existing 'Parameter' definition as follows:

```

<xs:complexType name="Parameter">
    <xs:all>
        <xs:element name="value">
            <xs:simpleType>
                <xs:restriction base="xs:string"/>
            </xs:simpleType>
        </xs:element>
    </xs:all>
    <xs:attribute name="name" use="required"/>
    <xs:attribute name="type"/>
</xs:complexType>

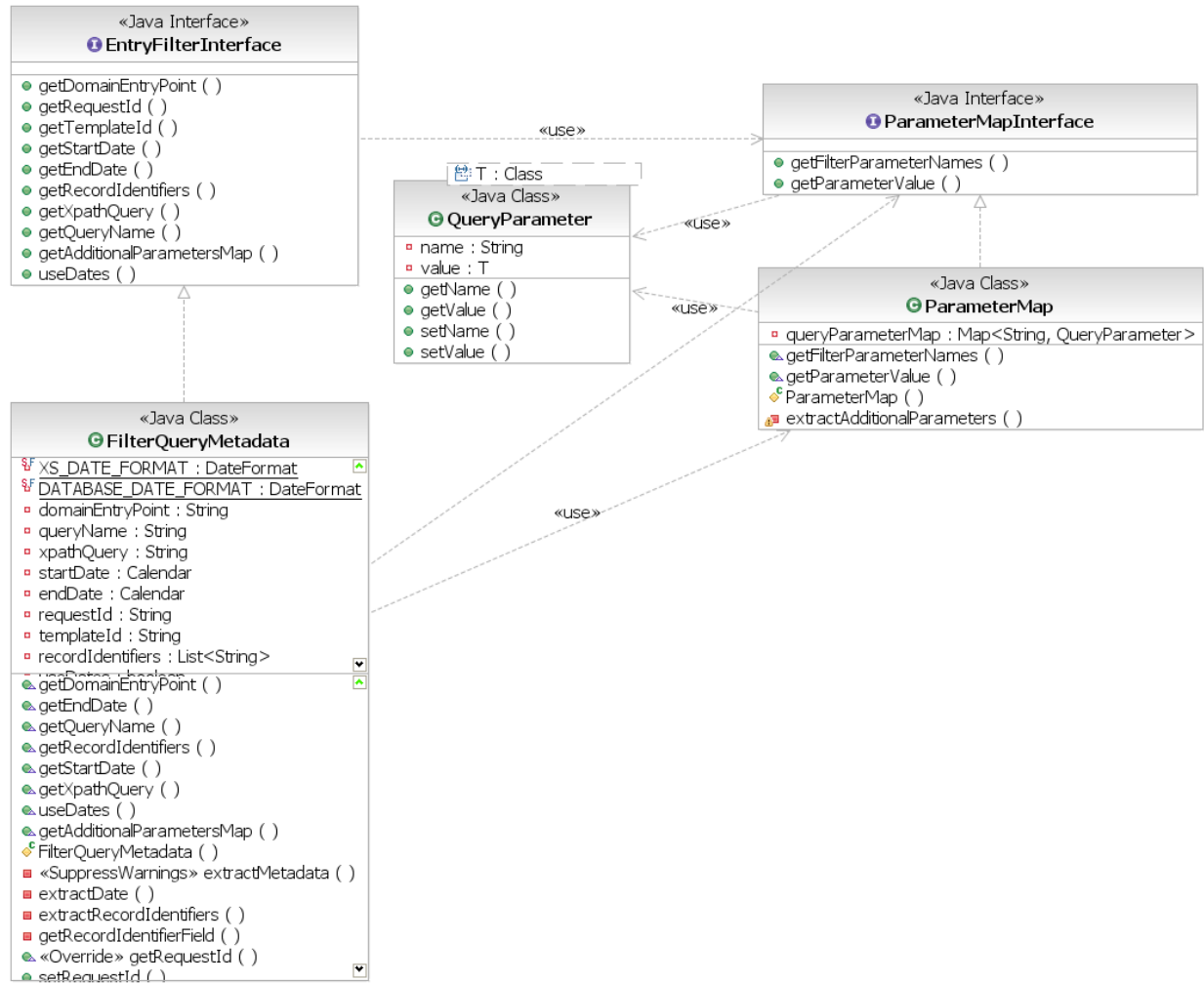
```

**NOTE:** The domain-specific filters specify the exact name of the optional parameter along with the possible set of values the parameter can take, whereas the generic filter permits string values to be specified, thus enabling the use of the schema for test purposes.

#### 6.2.1.12.5 SQL Based Filters Module/Other Diagrams

The following figure documents the class diagram including the interfaces that are introduced into the CDS system as part of the SQL-based filters module.

**Figure 20 SQL Based Filters Class Diagram**



### 6.2.1.13 VIM Filtering

#### 6.2.1.13.1 VIM Filtering Overview

Some CDS clients, such as VistAWeb, for example, specify additional restrictions on clinical data records. These clients might only be interested in a subset of data from the response. This behavior is implemented by adding an XPath query to a filter. The logical expression from an XPath query will be applied to the response XML as a post process.

#### 6.2.1.13.2 VIM Filtering Module Design

All VIM XML clients define criteria for data using filter ids in their Read requests. When the application processes a Read request, it matches filter id with a filter XML document. The information from the filter will be extracted and used to build an SQL query. When data for the Read request retrieved from the HDR database and converted to the VIM XML, the application will examine the filter and extract the XPath query for the XML. If the XPath query element is not empty, the XPath expression will be applied to the result XML document. As a result of this operation, collection of nodes defined by the query will be reduced based on the XPath

expression. If the XPath query is not a part of the filter, the VIM XML response will be left unchanged.

Use of XPathQuery element in the `VWALLERGYSinglePatientFilter.xsd` filter.

```
<xs:complexType name="EntryFilter">
  <xs:sequence>
    <xs:element name="domainEntryPoint" type="filter:DomainEntryPoint"/>
    <xs:element name="startDate" type="filter:DateParameter" minOccurs="0"/>
    <xs:element name="endDate" type="filter:DateParameter" minOccurs="0"/>
    <xs:element name="XPathQuery" type="filter:XPathQuery" />
  </xs:sequence>
  <xs:attribute name="queryName" type="xs:ID" use="required"/>
</xs:complexType>
<xs:complexType name="XPathQuery">
  <xs:sequence>
    <xs:element name="XPath" type="filter:XPath"/>
  </xs:sequence>
</xs:complexType>
```

Example of XPath element with expressions used in the `VWALLERGYSinglePatientFilter.xsd` filter.

```
<xs:simpleType name="XPath">
  <xs:restriction base="xs:string">
    <xs:enumeration value="allergyAssessments[status != 'E']"/>
    <xs:enumeration value="intoleranceConditions[status != 'E']"/>
  </xs:restriction>
</xs:simpleType>
```

### 6.2.1.13.3 VIM Filtering Processing

Read requests go through the chain of command all the way to `DefaultTransactionManager`, the method `performReadOnClinicalData` is called. The parameters for this method include template id, request id and filter - `CDSFilter` object. Each domain entry point in the filter is matched with a list of Read Managers. `DefaultTransactionManager` sets them up with criteria to retrieve data and delegates processing of the queries to the `WorkManager`, which handles concurrent execution. At the time when each clinical data document is returned and is ready to be added to list of results, `DefaultTransactionManager` calls `processEntryTypeQueries( document, XPath)`. The `processEntryTypeQueries()` method receives an XML document and an XPath query string in a format that is a valid XPath expression ("*vitalSignObservationEvents[status != 'E']*"). If the XPath expression is available and not null or an empty String, this expression is applied to the XML document to select matching nodes. A list of nodes (clinical observations) is collected as a result of this operation. This list contains clinical observations that the client wants. The other elements of this type will be removed from the document. If the document does not contain any elements that satisfy the criteria of the XPath expression, all clinical observation elements will be removed from the document.

#### 6.2.1.13.4 VIM Filtering Local Data Structures

CDSFilter class represents data from XML filter document. The document contains a collection of FilterQueryMetadata objects, each representing criteria information for building queries for each domain.

**Figure 21 CDS Filter Class Diagram**



#### **6.2.1.13.5 VIM Filtering Module/Other Diagrams**

This sequence diagram demonstrates steps to detach nodes not specified by the filter query from the clinical data document.

The XPath query expression is replaced with a wild card using the stringBetween method from the StringUtil Apache common library.

All nodes are selected by using the selectNodes() method using path with wild cards.

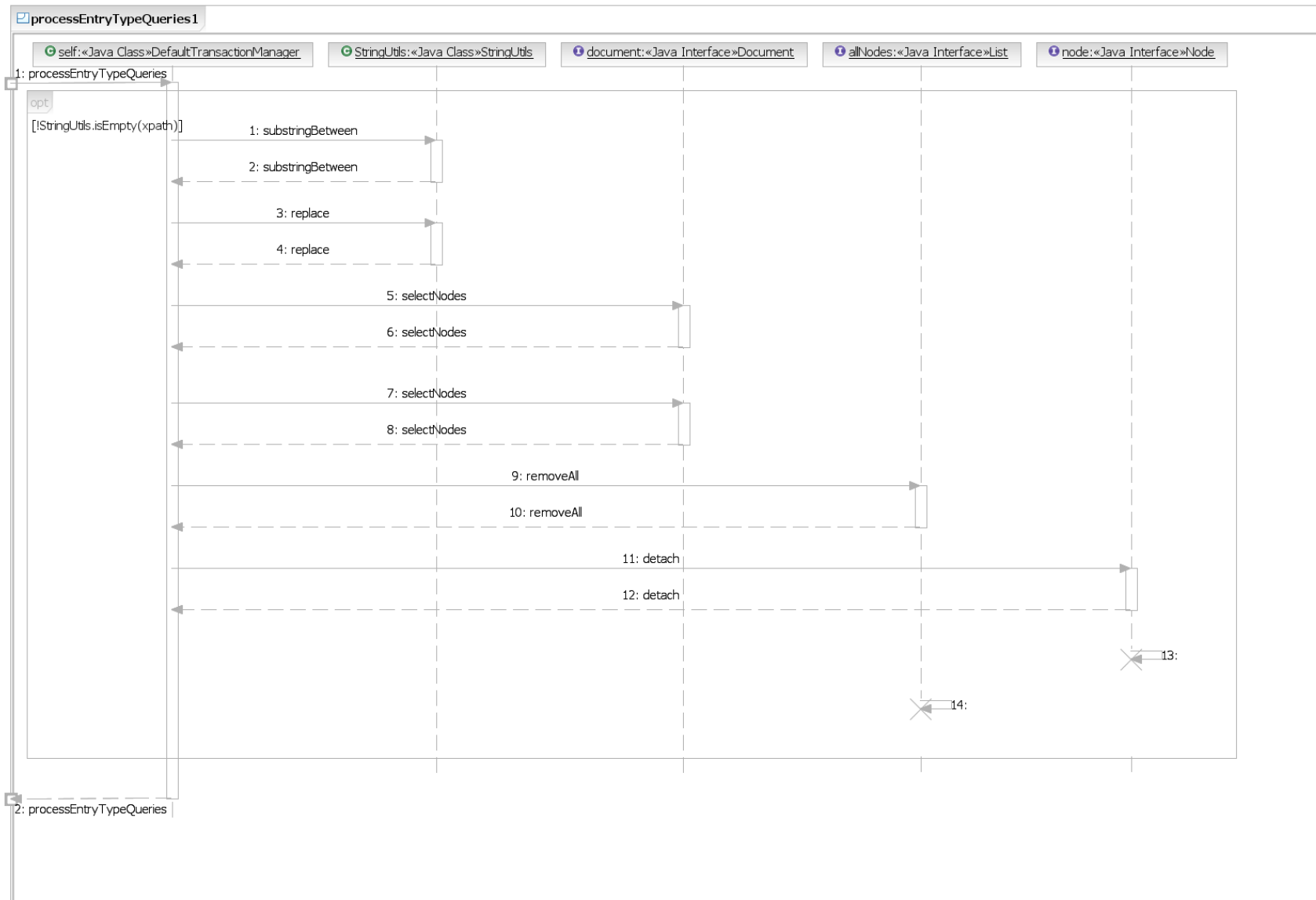
Nodes of interest are selected from the clinical document by using the selectNodes() method with XPath query from the filter.

The collection of nodes that we want to keep is removed from the collection of all nodes.

Nodes remaining in the collection are iterated over and detached.



**Figure 22 Filter Query Sequence Diagram**



#### **6.2.1.14 Response Generator**

The Response Generation component of CDS is responsible for creating a valid VIM XML response by combining the responses from all the queried systems.

Response Generation is comprised of Response Aggregation, Response Sequencing, and Response Organizing. Response Aggregation involves creating a single VIM response XML by combining VIM XMLs from all the VistA sites and CDS. Response Sequencer involves reordering the VIM response XML's elements to adhere to the respective XSD template. Response Organizing involves reordering the elements of the VIM response XML in the client requested manner – for example, ascending/descending order of the date field.

#### **6.2.1.15 VIM Aggregation**

##### **6.2.1.15.1 VIM Aggregation Overview**

CDS will receive a request from a client application for clinical data for a specific patient. CDS will manage communication/connection with multiple Data Sources that will return a formatted response in VIM XML. Each individual VIM response must be combined into a single VIM XML response that can be returned to a requesting client application. The VIM Aggregation component or the utility tool's design contained in this document will be used within CDS to merge or combine the clinical content of multiple VIM XML documents into a single aggregated VIM XML document.

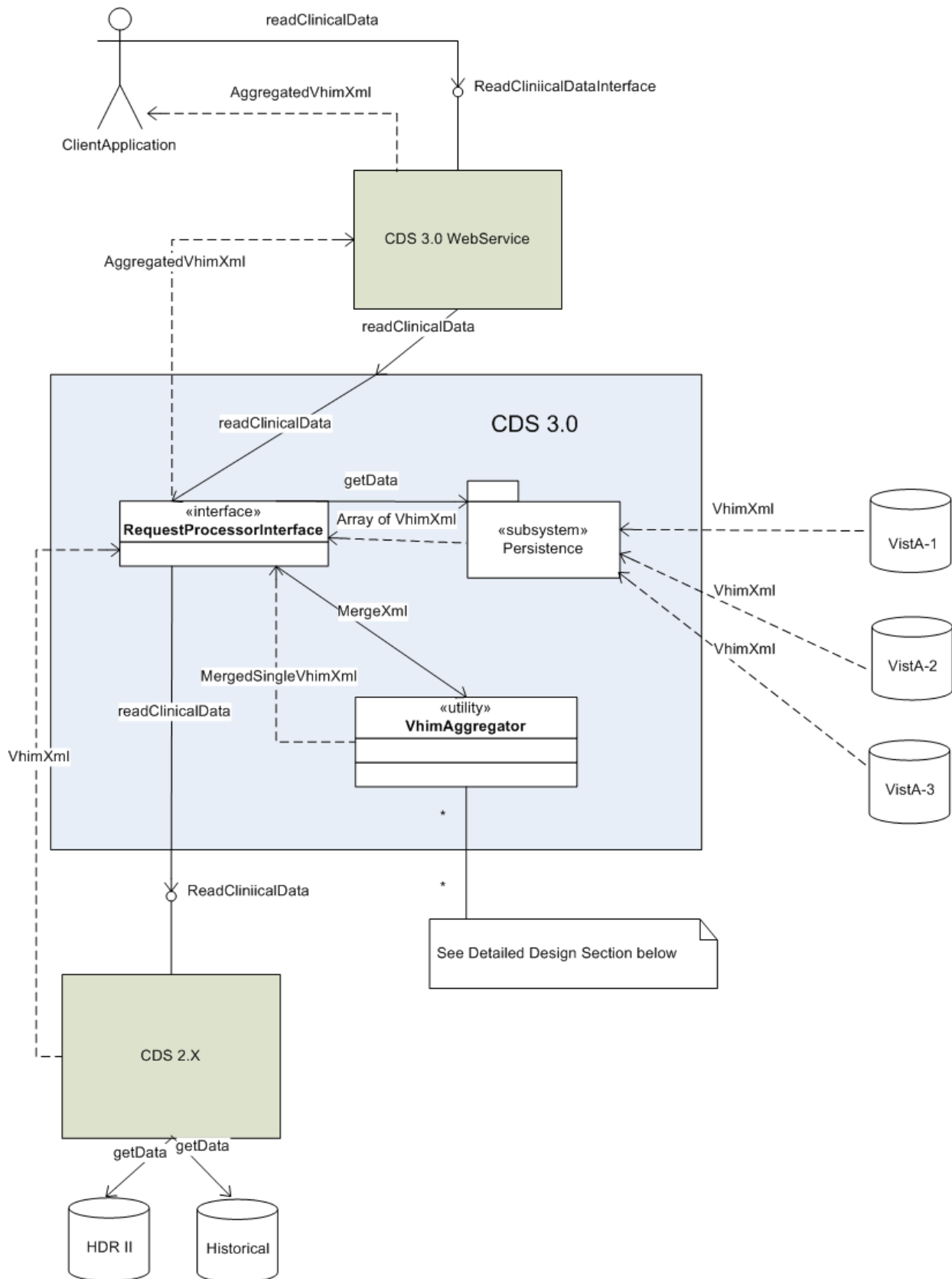
This document defines the design of the VIM Aggregation/Merge tool and, from high level perspective, defines how this tool fits into the architecture of CDS. This document describes from a high level how easily this tool can be fit into and used by the CDS application.

The scope of this documentation describes the design of the VIM Aggregation or Merge tool and identifies Unit Test Case scenarios, as well as illustrating how this tool fits into the CDS application structure. This documentation is not intended to define or design the CDS 3.x Architecture – rather, the CDS Architecture is represented in very basic terms for the purpose, as stated above, of showing how the VIM Aggregator is used within the application.

##### **6.2.1.15.2 VIM Aggregation Processing VIVEK**

CDS must provide the ability to obtain data from multiple sources, such as multiple VistA sites as well as from HDR. The VistA sites will return VIM XML in a non-tabular response when asked for clinical data. CDS must aggregate all VIM XML that it receives from multiple data sources into a single VIM XML and return the aggregated XML to the requesting client application. The design for VIM Aggregation detailed below will provide the aggregation or merging capabilities of CDS.

**Figure 23 VIM Aggregation System Architecture**



### 6.2.1.15.3 VIM Aggregation Component Design

CDS will make calls for data in a way that will avoid duplication of clinical data given to the VIM Aggregation utility. VistA sites store site specific data and calls to VistA sites return data for that site alone. HDR DB contains duplication of data across VistA sites for some clinical domains: Allergies, Pharmacy and Vitals for VA, CHDR and HTH patients. Clinical data from additional domains (starting with TIU) comes directly from the VistA sites. The data received from each data source is site specific and will also avoid obtaining duplicate clinical records for a specific patient. Because duplication is handled by CDS at a higher level, this simplifies the merging process within the VIM Aggregator.

The design for the Response Aggregator essentially consists of a ResponseAggregatorInterface. This interface uses only one method listed below:

```
public Document getAggregatedResult(List<Document> readResponses) throws  
ResponseAggregatorException;
```

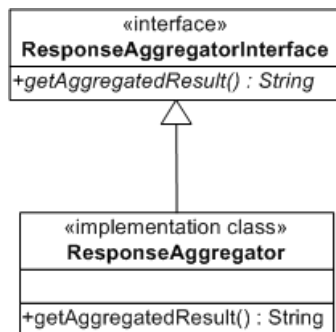
The return type for the above mentioned method is an org.dom4j.Document. This ensures an efficient design in CDS where parsing of String to Document and vice versa is eliminated. The classes implementing the ResponseAggregatorInterface are specific to a VIM version. Currently, only one class, ResponseAggregator, implements this interface and resides in the "gov.va.med.cds.response.vhim400" package indicating that this specific class is tied to VIM version 4.0.0.

The Aggregator will be given a list of VIM XML Documents (of type org.dom4j.Document). Since no duplicate clinical records will be contained within each VIM instance, the aggregation or merge logic merges or combines like clinical data objects and does not have to be concerned with stripping out duplicates.

#### Design Resolution

The design implemented in the ResponseAggregator class uses DOM4J API for accessing elements/nodes and manipulating them. This API approach was found to be substantially faster than using XPath for traversing the elements. Dom4J also had a slightly lower memory usage than some of the other solutions examined (See the Test Results section below.). The interface and class names were refactored to be less technology-specific and focus on the business use. Additionally, the use of the ResponseInterface was discarded.

Figure 24 VIM Aggregator Resolution Class Diagram



## 6.2.1.16 Response Sequencer

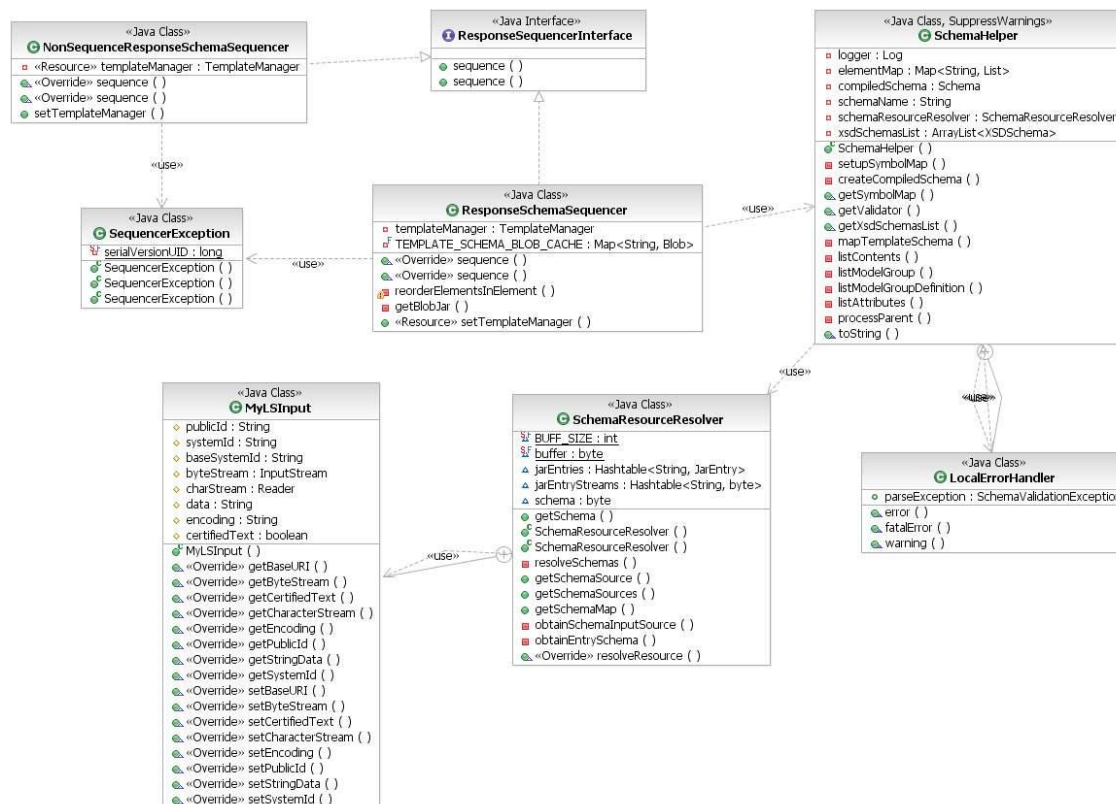
### 6.2.1.16.1 Response Sequencer Overview

The elements in the response XML from the Response Aggregator component are sometimes not in the sequence as defined in the corresponding XSD. The Response Sequencer component ensures the order of the elements in the VIM response adheres to the corresponding Response Template XSD specification. The Response Sequencer is used to reorder elements in the response XMLs of Allergies, Vitals, Pharmacy, and Lab domains. The response XML for TIU does not require sequencing.

### 6.2.1.16.2 Response Sequencer Module Design

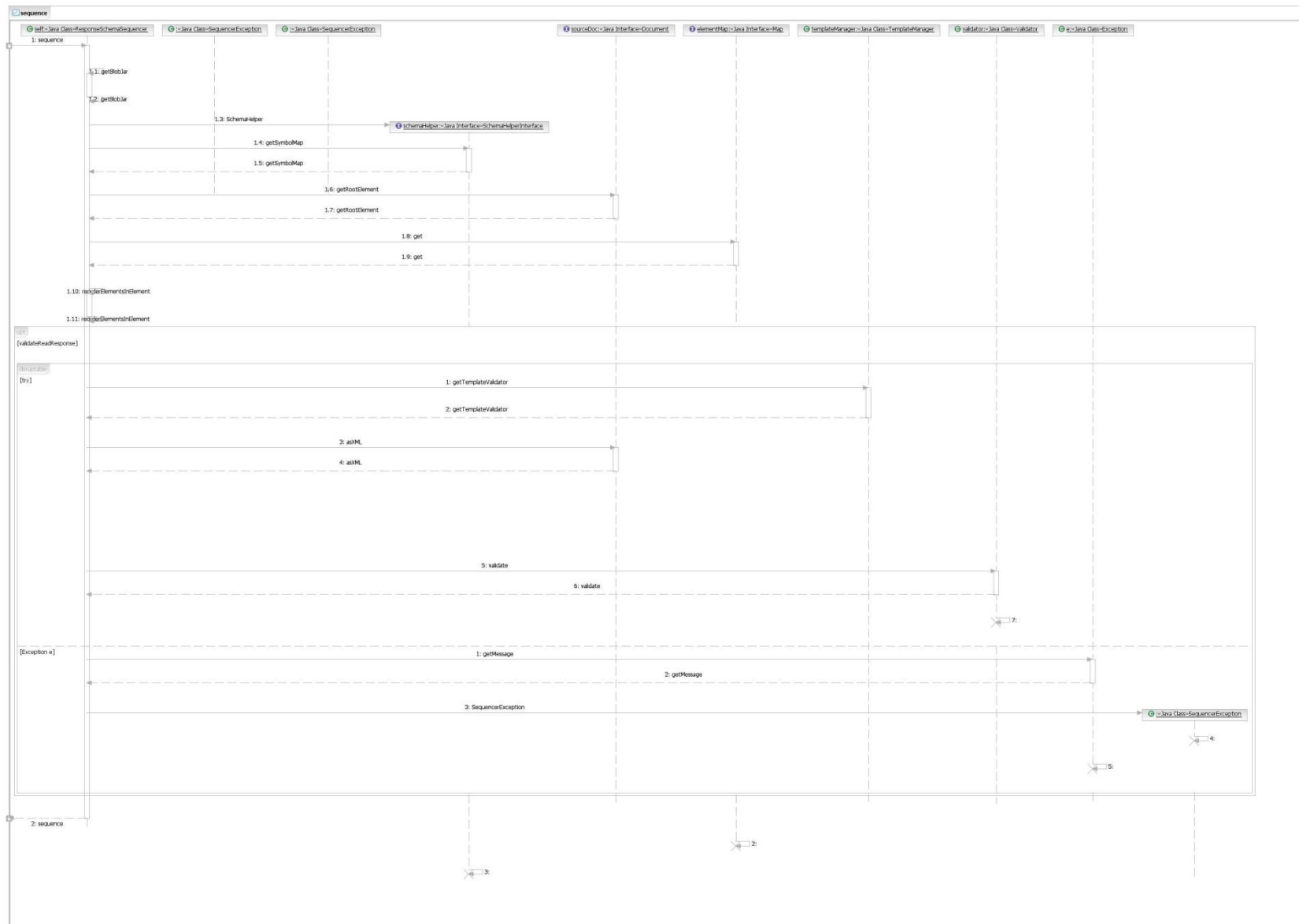
The ResponseSchemaSequencer reorders the elements in the VIM response XML based on the XSD information from SchemaHelper. The components of Response Sequencer are wired into the system in templateHelperContext.xml. The SchemaHelper uses SchemaResourceResolver to construct sequence information from XSD.

Figure 25 Response Sequencer Sequence Diagram



The sequence diagram below describes the steps involved in the reordering of elements in the VIM response XML. The TemplateRequestProcessor uses TemplateHelper to call ResponseSchemaSequencer.sequence with the aggregated VIM response XML. ResponseSchemaSequencer reorders the elements using the element ordering information in SchemaHelper.

**Figure 26 Response Sequencer Sequence Diagram**



## **6.2.1.17      Sorting Read Response XML Based on Criteria**

### **6.2.1.17.1      Sorting Read Response XML Overview**

CDS needs a capability to sort XML nodes in the response clinical document by criteria. The filter XML supports an XQuery sort expression snippet. The CDS sorts the final response document after data from various data sources is aggregated and sequenced.

### **6.2.1.17.2      Sorting Read Response XML Module Design**

#### **Sorting Criteria**

The information required to perform a sort on a document is listed below:

1. The elements in the document that need to be sorted for a given domain
2. The sorting key used for comparing each element
3. The order in which the elements will be sorted

#### **Sorting Technology**

XQuery and Saxon were selected as the best implementation technology to use for performing the sort. It provides more than adequate support to meet current and future requirements with excellent levels of performance and scalability. XQuery features a query language that provides many query features. The ‘order by’ feature will be used in this sort implementation. An example of the XQuery query used to sort TIU documents is noted below:

```
for $p in
//clinicalData:ClinicalData/patients/patient/clinicalDocumentEvents/clinicalDocumentEvent order
by $p /referenceDate/literal descending return $p
```

For example, this query sorts a TIU document containing multiple clinicalDocumentEvents as specified by the first XPath expression. It uses the referenceData as the values to compare in the sort as specified in the second XPath expression. The sort order direction is specified as ‘descending’. A list of sorted clinicalDocumentEvents will be returned.

#### **Sorting Code**

The code below demonstrates how to perform an XQuery query using Saxon’s Java implementation. A static query context and configuration are constructed. The data elements to sort and the sort expression are specified as Java variables and inserted into an XQuery query.

```
Configuration config = new Configuration();
StaticQueryContext staticContext = new StaticQueryContext( config );
XQueryExpression exp = staticContext.compileQuery( " for $p in " + pathToElements + " order by
$p" + criteria + " return $p" );
Next a dynamic query context is constructed. The document, uri, and configuration are set on the
context.
DynamicQueryContext dynamicContext = new DynamicQueryContext( config );
dynamicContext.setContextItem( new DocumentWrapper( document, uri, config ) );
```

Conversion between Dom4j and Saxon objects is done by using the DocumentWrapper? class, provided by the Saxon API. This class provides a wrapper around a Dom4j document, which

enables XQuery to work with a DOM document. The application next calls the evaluate method on the XQuery expression. The method returns a list of sorted dom4j nodes (DocumentWrapper manages the conversion internally).

```
List<Node> list = ( List<Node> )exp.evaluate( dynamicContext );
```

The nodes in the list are sorted in the desired order, but the elements in the XML document are not affected by this order. The application must modify the order of the nodes in the document, since XQuery cannot do this as efficiently. This is done by iterating over the collection of sorted nodes. Each node is detached from the document and added as a child to the current parent node as shown in the code below:

```
for ( Node node : nodes )
{
    Element parent = node.getParent();
    node.detach();
    parent.add(node);
}
```

The document has nodes sorted according to the XQuery query.

### **6.2.1.17.3      Sorting Read Response XML Processing**

CDS clients use filter XML to specify the information needed to query the data they require. Among other instructions, clients will be able to specify sorting criteria which will be used by CDS to arrange the data elements for the specified domain entry point in the reply XML that is returned to the client.

### **6.2.1.17.4      Sorting Read Response XML Local Data Structures**

This section contains description of classes used to sort Read response XML.

#### **Filter**

The TIU filter supports the order by feature of XQuery as noted below:

```
<orderBy>
    <xqueryExpression>/referenceDate/literal descending </xqueryExpression>
</orderBy>
```

The sort expression is specified in the filter XML as noted above. The XPath that selects the elements need to be sorted is also required by XQuery. This path is not passed in the filter XML, but rather it is injected and mapped to the domain entry point that is passed in the filter XML. This will limit the sort to be performed to only one set of elements. Support for querying on multiple sort keys is currently not required, but can be easily added in the future.

#### **Template Request Processor**

The TemplateRequestProcessor class processes the request to read data from multiple sources, aggregates all the responses into one document, applies logic to sequence the response document, and then checks for instructions in the filter request to organize/order the clinical data set elements in the response document that is returned to the client. This processing occurs as indicated by the code snippet below:



```

List<Document> readResponses = ClinicalDataServiceSynchronousInternal.readClinicalData(...);
responseDocument = templateHelper.getResponseAggregator( templateId ).aggregateResponses( ... );
responseDocument = templateHelper.getResponseSequencer( templateId ).sequence(... );
responseDocument = templateHelper.getResponseOrganizer( templateId ).organize( responseDocument,
filterXmlRequest);

```

## ResponseOrganizer

The ResponseOrganizer class checks, for the 'orderBy' element in the filter XML. If an element is found, the sort expression is extracted from the filter XML and the XQuerySortEvaluator strategy is called to perform the sort. Otherwise, the original unsorted document will be returned. For efficiency, do not create an instance of the !CDSFilter object using CdsFilterFactory, instead extract the two elements required from the filter by using the Dom4J API. The refactoring required to do this efficiently was determined to be out of scope.

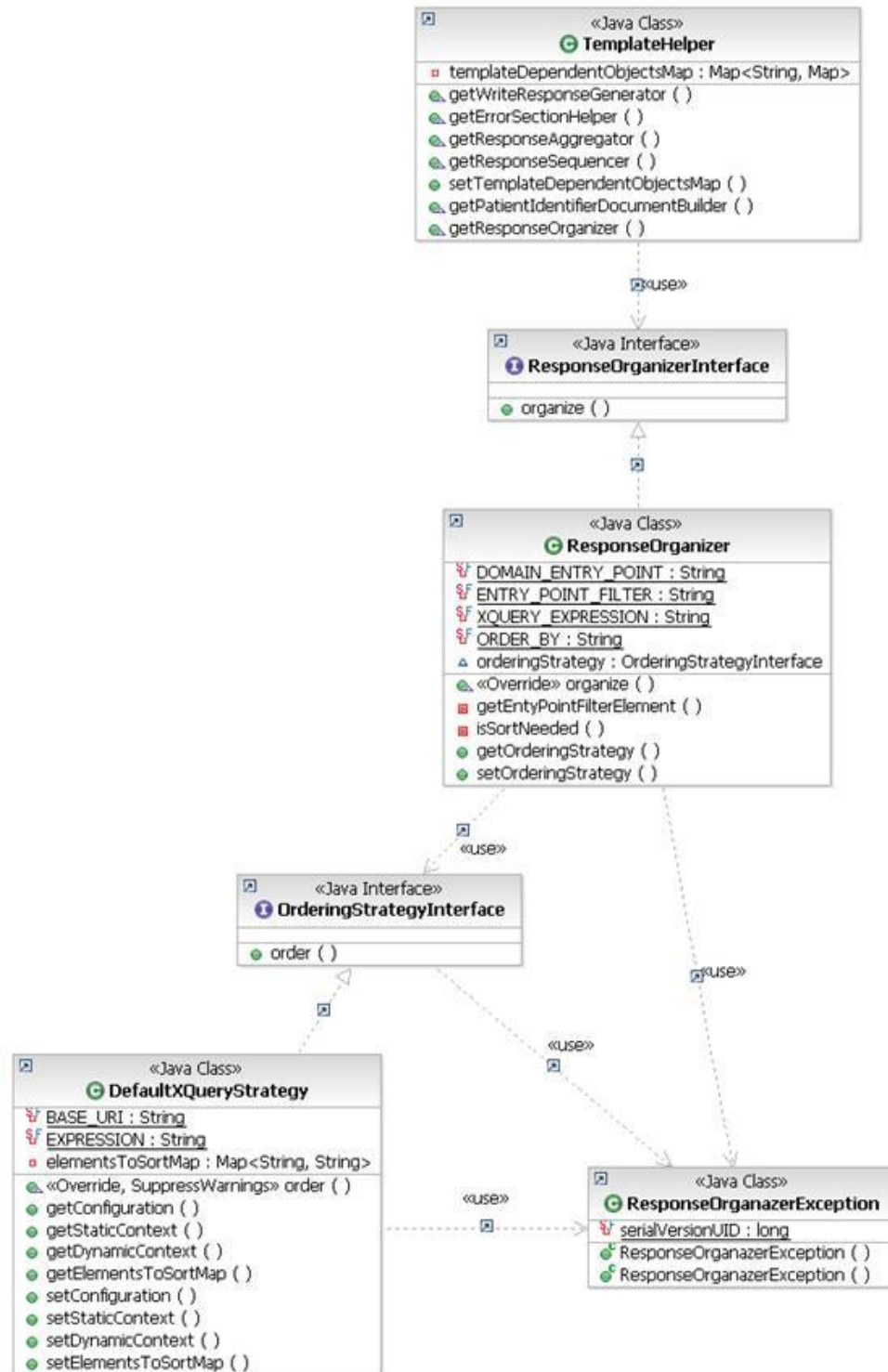
## XQuerySortEvaluator

The sort method of the XQueryEvaluator strategy takes an XML document, which has aggregated results from multiple sources, the domain entry point and the sorting criteria both from the filter XML as arguments. The exceptions thrown by sort will be handled in the usual manner by the ExceptionHandler. Document sort(Document document, String String domainEntryPoint, String criteria) throws XPathException, IOException, DocumentException. The sort method looks up the XPath used to select the elements that require sorting using domainEntryPoint as the key and sets the result to the pathToElements variable. The pathToElements and criteria variables are used to form the XQuery query. The XQuerySortEvaluator executes the code described in the XQuery section above and returned the sorted document.

### 6.2.1.17.5 Sorting Read Response XML Module/Other Diagrams

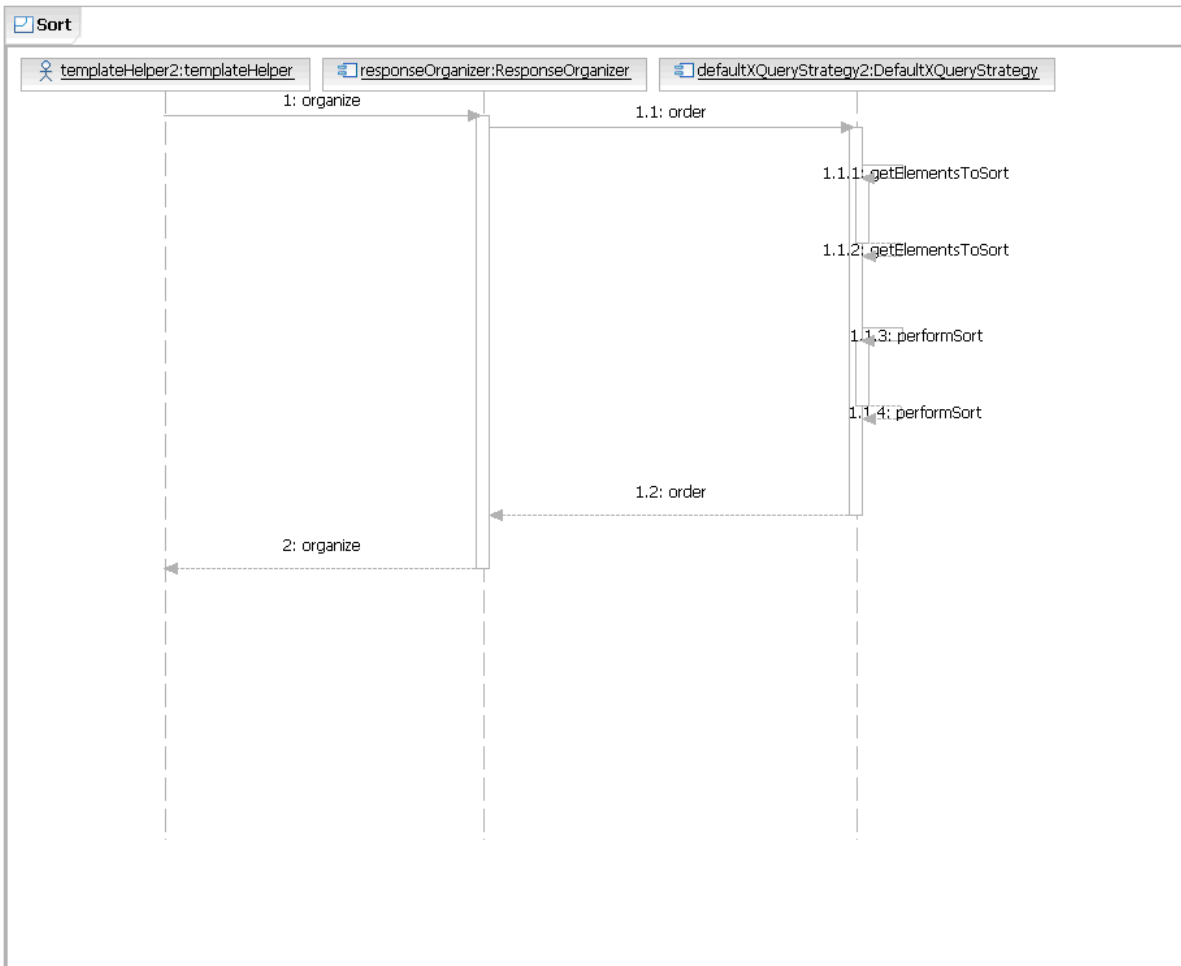
The following class diagram displays relationships and collaboration between classes used for sorting read response XML:

**Figure 27 Sorting Read Response XML Based on Criteria Class Diagram**



The following sequence diagram displays interaction between classes involved in sorting read response XML:

**Figure 28 Sorting Read Response XML Based on Criteria Sequence Diagram**



### 6.2.1.18 Persistence Subsystem

This section provides an overview of the persistence subsystem.

#### 6.2.1.18.1 Persistence Subsystem Transaction Manager Overview

. This component is responsible for managing the read, create, update and delete operations for a given clinical domain request. The component is called by an object that implements the `ClinicalDataServiceSynchronousInternalInterface` interface from within the Create/Read/Update/Delete (CRUD) request workflow. The `ClinicalDataServiceSynchronousInternal` object is injected with an instance of the transaction manager by the Spring context and continues the CRUD request workflow by calling the corresponding of the transaction manager.

The transaction manager CRUD methods use a persistence locator object injected from the Spring context. The component retrieves the write persistence manager from the persistence

locator that are associated to domain entry points which are obtained from the template id in the case of create/update/delete operations.

#### **6.2.1.18.2 Create request processing**

The create requests are sent to the write persistence manager. Write persistence manager returns a string value from hibernate session which indicates the successful insert of data in HDR. That value is treated as record identifier and is returned in the CDS response XML to the client. the client sends the update request along with the data that needs to be updated and the record identifier.

#### **6.2.1.18.3 Update request processing**

Client would send in the data to be updated and CDS framework will treat it as new record to be persisted in HDR. At the HDR database level, the older record is flagged inactive and the new record would be stored with status as active. Write persistence manager returns a string value from hibernate session which indicates the successful insert of data in HDR.

#### **6.2.1.18.4 Delete request processing**

The delete requests are sent to the write persistence manager. Write persistence manager supports the logical/physical delete of the requested data. In the case of logical delete, the client would send recordStatus value as 1. The write persistence manager updates the record so that the record status that is originally defaults to 0 at the time of the insertion of the record in the database is now updated to 0. In the case of physical delete request, wherein the record status is not updated, the write persistence manager deletes the record from the database.

---

**NOTE:** Note: Though CDS supports delete functionality, none of the clients use it.

---

#### **6.2.1.18.5 Read request processing**

The Transaction manager component retrieves a list of persistence managers from the locator that are associated to domain entry points which are obtained from the read method's request filter parameter. The transaction manager configures a new ReadWork object with the persistence manager, filter and person identifier information for each of the persistence managers found, and adds the ReadWork objects to a work queue for the actual lookup. Read results from the work performed by the persistence managers is formatted in a Document object and added to a List<Document> list for return to the read workflow.

#### **6.2.1.18.6 Persistence Subsystem Module Design**

The transaction manager component is an implementation (specification) of the TransactionManagerInterface interface object. This specification prescribes that a class provide implementation details for a performCUADOnClinicalData method. This method performs write/update/delete request based on the template id in a specific domain based on the domain entry point.

The transaction manager component is an implementation (specification) of the TransactionManagerInterface interface object. This specification prescribes that a class provide implementation details for a performReadOnClinicalData method, which takes template, request,

and filter parameters. This method performs the Read request against a given clinical domain as indicated by domain entry point filter criteria. The component is initialized with instances of the persistence locator, and work manager objects which are used to lookup the source databases and to schedule and perform the actual read requests. The component receives the read request results from the work manager and adds them into a List<Document> object to return to the calling component.

In the event that a read exception is returned by the work manager process, the component throws a PersistenceException indicating that the read request from the data sources has failed, and no read results are returned.

#### **6.2.1.18.7 Persistence Subsystem Transaction Manager Processing**

##### **Write request processing**

The transaction manager component is an implementation (specification) of the TransactionManagerInterface interface object. This specification prescribes that a class provide implementation details for a performCUADOnClinicalData method. This method performs write/update/delete request based on the template id in a specific domain based on the domain entry point. Transaction manager calls the persistence locator that returns the write persistence manager.

##### **Read request processing**

For read requests, the component is called by an instance of a ClinicalDataServiceSynchronousInternalInterface interface. The calling class executes the performReadOnClinicalData read method on the transaction manager component by passing in templateID, requestID, and filter parameters. The component returns to the ClinicalDataServiceSynchronousInternalInterface instance a List<Document> object containing the read results appropriate to the filter criteria. Within the read operation, the component uses the request filter content to obtain a collection of readable persistence managers from a persistence locator, which are appropriate to the domain entry points contained in the filter. As the collection of persistence managers is iterated over, a new ReadWork object is constructed with the persistence manager and filter information, and added to the work queue for actual request lookup. The persistence manager is configured with the appropriate Hibernate data source in which to perform the read on, and returns a Document object containing the read result to the ReadWork object. Each of the read requests is executed in a separate thread by the work manager, the results of which are stored in a ReadWork object.

When all the threads have completed, the collection of results is iterated over and examined for success or failure conditions. A successful read will result in a Document object that is added to the List<Document> return object. Read failures are reported back to the calling client in a PersistenceException being thrown.

#### **6.2.1.18.8 Persistence Subsystem Transaction Manager Local Data Structures**

The Transaction Manager component is managed within the Spring context with the following configuration:

```
<bean id="transactionManager" class="gov.va.med.cds.transaction.DefaultTransactionManager">
  <property name="persistenceLocator" ref="persistenceLocator" />
</bean>
```

```

<property name="workManager" ref="workManager" />
<property name="templateIdToDomainEntryPointListMap">
<map>
<!-- So that the transaction manager can resolve template ids to a list of domain entry points.
-->
<entry key="AllergyAssessmentCreate40020">
    <list>
        <value>AllergyAssessment</value>
    </list>
</entry>
<entry key="AllergyAssessmentCreateOrUpdate40060">
<list>
    <value>AllergyAssessment</value>
</list>
</entry>
<entry key="IntoleranceConditionCreate40020">
<list>
<value>IntoleranceCondition</value>
</list>
</entry>
<entry key="IntoleranceConditionCreateOrUpdate40060">
<list>
<value>IntoleranceCondition</value>
</list>
</entry>
<entry key="LabCreate40020">
<list>
<value>LabTestPromise</value>
</list>
</entry>
<entry key="LabCreateOrUpdate40060">
<list>
<value>LabTestPromise</value>
</list>
</entry>
<entry key="PharmacyCreate40020">
<list>
<value>OutpatientMedicationPromise</value>
</list>
</entry>
<entry key="PharmacyCreateOrUpdate40060">
<list>
<value>OutpatientMedicationPromise</value>

```

```

        </list>
      </entry>
      <entry key="VitalsignsCreate40020">
        <list>
          <value>VitalSignObservationEvent</value>
        </list>
      </entry>
      <entry key="VitalsignsCreateOrUpdate40060">
        <list>
          <value>VitalSignObservationEvent</value>
        </list>
      </entry>
    </map>
  </property>
</bean>

```

The transactionManager parameter is injected with configured instances of the persistence locator, and work manager objects. The templateIdToDomainEntryPointListMap map is used to map a template id to a domain entry point name and is used in transaction manager CRUD operations.

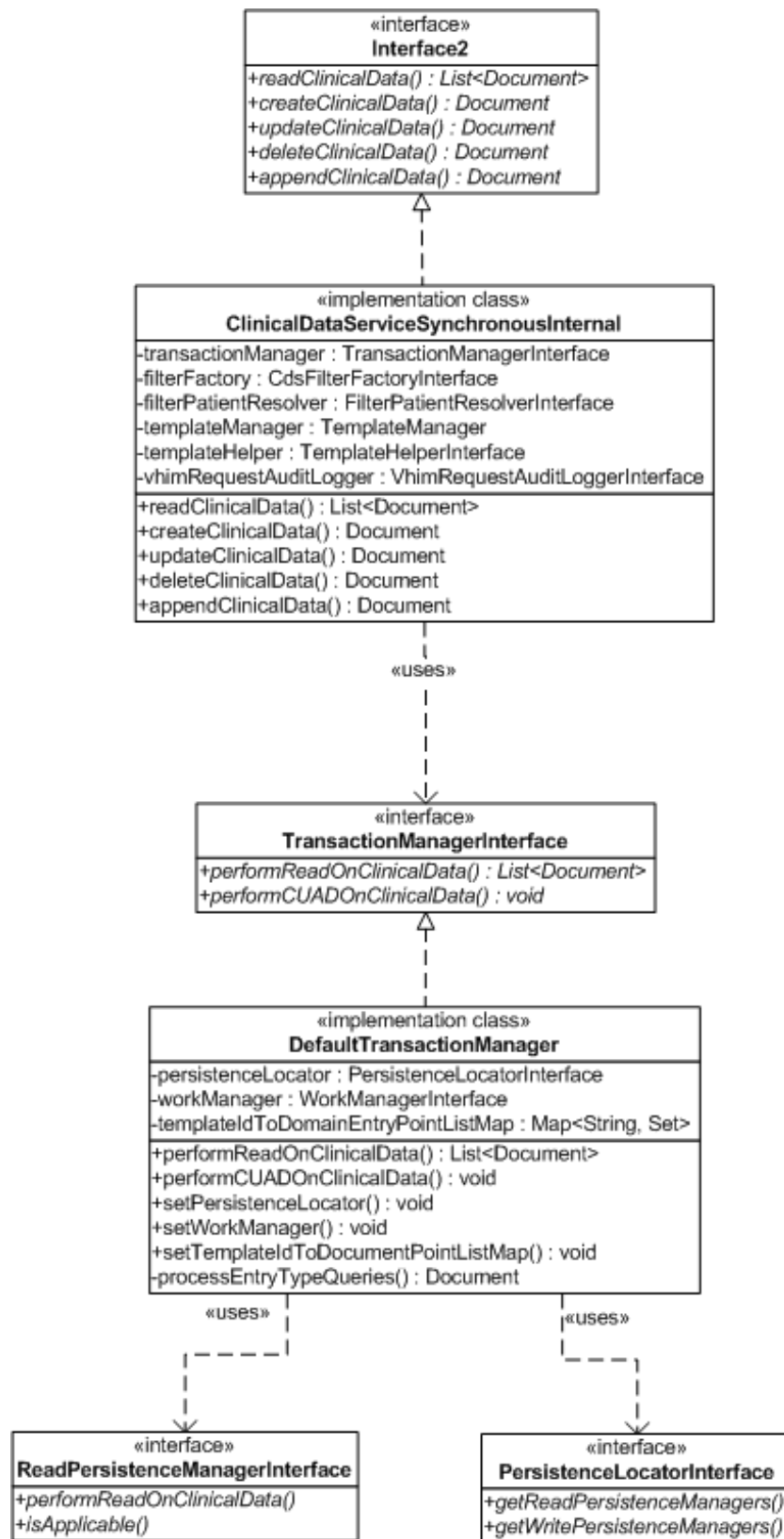
The CDS persistence framework supports paging result data for queries that run against a single clinical domain and data source. Additionally, support for counting the number of records that would be returned by a query has been added to the CDS persistence mechanisms.

Additionally, the CDS persistence framework supports optional query parameters in queries. This modification was implemented to decrease the number of queries required to support the clinical domain and combinations of query parameters. These changes do not impact query performance negatively.

#### **6.2.1.18.9 Persistence Subsystem Transaction Manager Module/Other Diagrams**

The following diagram depicts the static structure of the component. The concrete implementation of the TransactionManagerInterface is the DefaultTransactionManager:

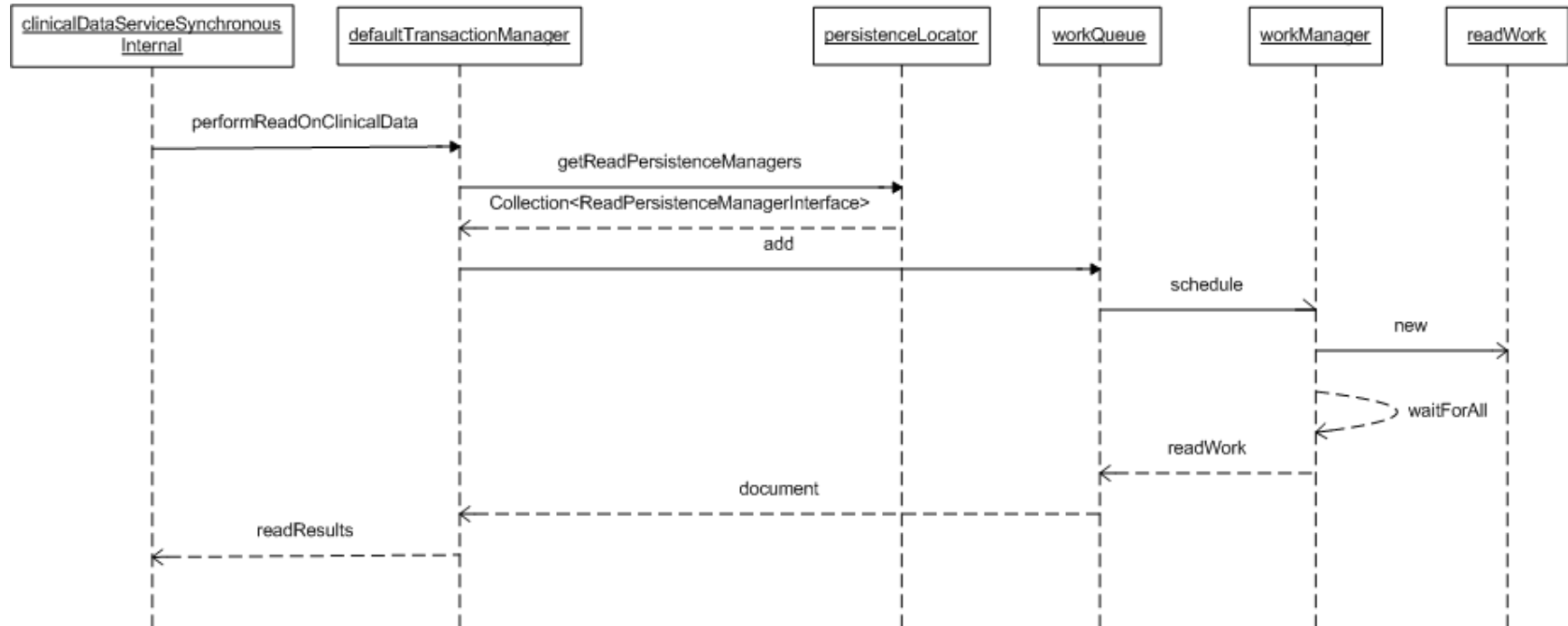
**Figure 29 Transaction Manager Class Diagram**



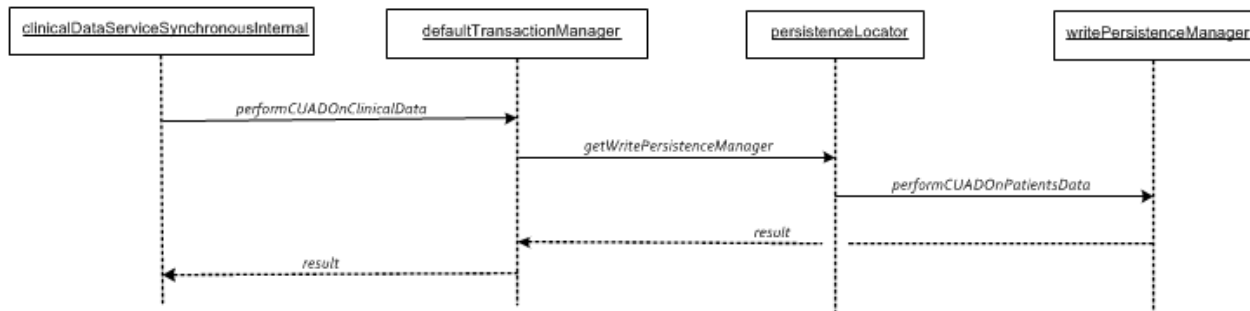


The following sequence diagram depicts the sequence of calls made from the clinicalDataServiceSynchronousInternal calling code.

**Figure 30 Transaction Manager Read request processing Sequence Diagram**



**Figure 31 Transaction Manager write/update/delete request processing Sequence Diagram**



## 6.2.1.19 Persistence Locator

### 6.2.1.19.1 Persistence Locator Overview

#### Write request

As part of the write request processing, CDS will manage communication/connection with HDR data source and store the data.

#### Read request

CDS will manage communication/connection with multiple VistA Data Sources and HDR data source where the data is available for the patient of interest for read request processing. CDS will look-up and load a data source information at runtime (ReadManagers) for all VistA systems where the data is available.

The Persistence Locator component will receive information about resolved patient identifiers from the IdM service using the "Identity Management Integration" component. It is important that the persistence locator component be supplied with the resolved identifiers in order to query VistA systems.

The subsystem will also continue to query the heterogeneous VistA data repositories for the request of clinical patient data and will then aggregate the clinical data retrieved from multiple repositories into a single data record. The subsystem has the ability to perform CRUD operations to the HDR database.

### 6.2.1.19.2 Persistence Locator Module Design

#### Write request processing

The incoming write/update/delete request is processed accordingly based on the corresponding operation type. The operation types are write, update and delete. The data is created, updated or deleted in HDR.

#### Read request processing

For every incoming read request for patient data, the IdM service will be used to resolve the patient IDs and to figure out all the VistA sites where data is available for the patient of interest. If the IdM service fails, a fatal error (as VIM XML) will be reported back to the client in the

response. For a successful patient lookup from the IdM service, the request will be delegated to the `ClinicalDataServiceSynchronousInternal` class.

The `ClinicalServiceSynchronousInternal` class performs the following steps to fetch data from the VistA sites:

- Create a `CDSFilter` from the incoming filter request XML. The `CDSFilter` contains information for the `domainEntryPoint`, patient IDs and local site IDs.
- Call the `performReadOnClinicalData` method of the `DefaultTransactionManager`, passing in the `CDSFilter` created above.

The `DefaultTransactionManager` delegates to the `PersistenceLocator` which will locate all the persistence managers that contain data relating to the `PersonIdentifiers` passed in the filter. These persistence managers will be connected to the VistA sites as applicable.

### **6.2.1.19.3 Persistence Locator Processing**

#### **Write request processing**

The incoming write/update/delete request is processed accordingly based on the corresponding operation type by `writerPersistenceManger` that has connection information of HDR. The operation types are write, update and delete. The data is created, updated or deleted in HDR.

#### **Read request processing**

The `DefaultTransactionManager` loops through all entry point filters passed in the `CDSFilter` object. For each entry point filter, the `PersistenceLocator` `getReadPersistenceManagers` method is called, passing in the current entry point filter and a list of `PatientIdentifier` objects. The `PersistenceLocator` will return all of the `ReadPersistenceManager` objects that contain data for the patients identified in the list of `PatientIdentifier` objects. Some or all of these persistence managers may connect to a VistA site.

Next, the `DefaultTransactionManager` loops through each `ReadPersistenceManager` and schedules a `WorkItem` job with the `commonj.work.WorkManager` so that it will fetch data concurrently. The `DefaultTransactionManager` waits for all the work jobs to complete before building a list of dom4j result documents that is returned to the caller.

The `PersistenceLocator` contains a list of pre-instantiated `ReadPersistenceManager` objects. The persistence manager implementation used for the VistA sites is the `SiteSpecificHibernateReadPersistenceManager`. The dynamic persistence managers are injected by Spring into the `PersistenceLocator` by a `SiteSpecificHibernateReadPersistenceManagerFactory`.

The `SiteSpecificHibernateReadPersistenceManagerFactory` dynamically instantiates a `ReadPersistenceManager` for each VistA JNDI data source. The VistA site data sources are obtained by delegating to the `JNDIDataSourceBindingLocator` in which a collection of `javax.naming.Bindings` are returned. For each binding, a `SiteSpecificHibernateReadPersistenceManager` object is instantiated after which the Spring injected static properties are assigned. The site identifier is determined from the name in the JNDI binding. This implies a naming convention discussed below.

A map of templateIds and HibernateSessionFactoryPropertyHolder objects is injected by Spring into the SiteSpecificHibernateReadPersistenceManagerFactory. A map of templateIds and org.hibernate.SessionFactory objects is created for each data source Binding. The properties needed for the org.hibernate.SessionFactory are obtained from the respective HibernateSessionFactoryPropertyHolder objects. Each map of org.hibernate.SessionFactory objects share the same data source instance.

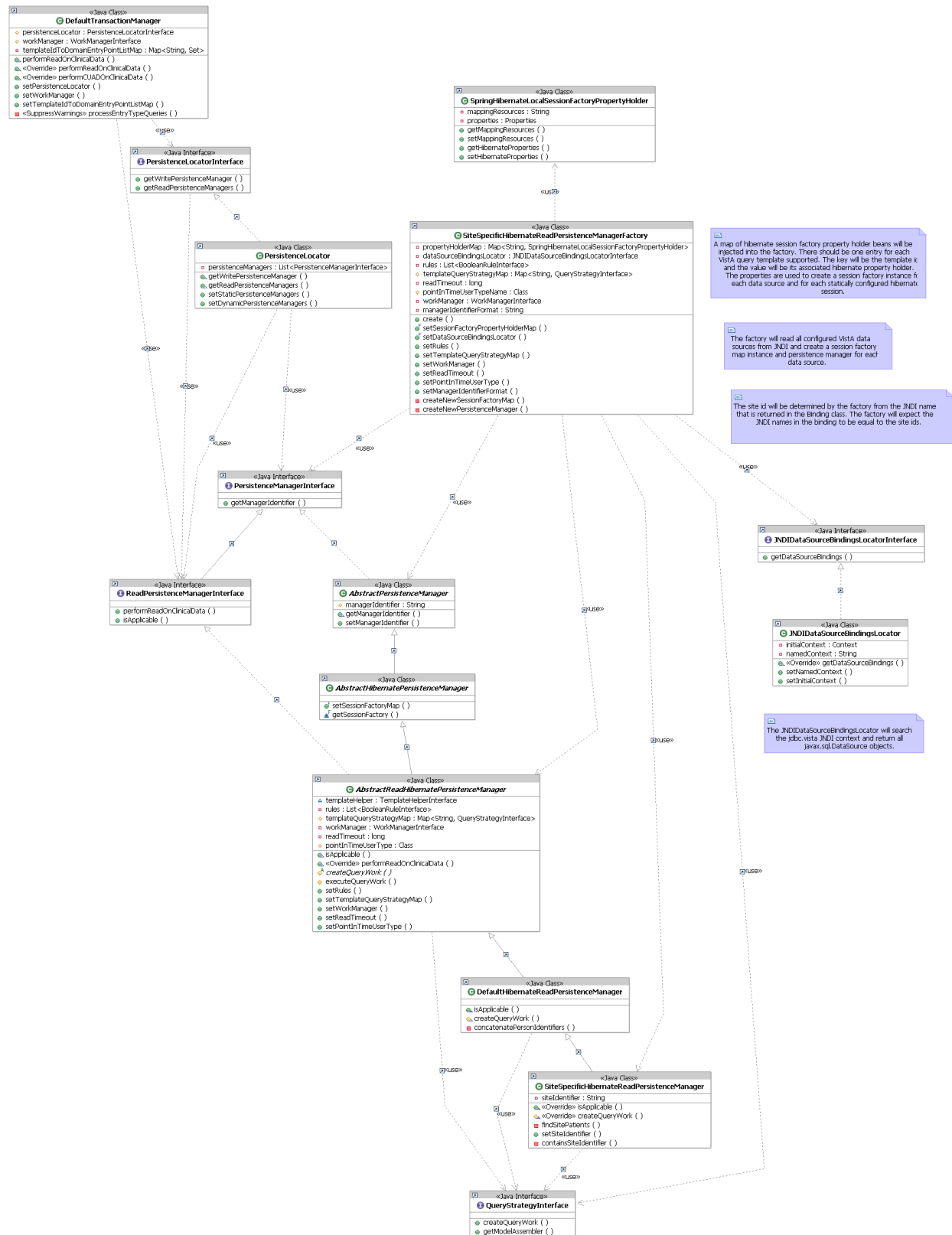
The data sources for all VistA systems (129) will need to be created and configured accurately inside of the WebLogic server. The JNDIDataSourceBindingsLocator getDataSourceBindings method performs a JNDI lookup of all data source bindings that conform to the following naming convention: 'vista<siteId>DataSource'. Where siteId is 3 digits. This method returns a Collection of Binding objects.

For testing purposes, a testing specific JNDIDataSourceBindingsTestLocator implementation shall support injection of basic data source (from Spring Framework) that will be used so that the test can be run outside an application server. These basic data sources will be pointing to test systems (currently two) to test the functionality.

#### **6.2.1.19.4 Persistence Locator Module/Other Diagrams**

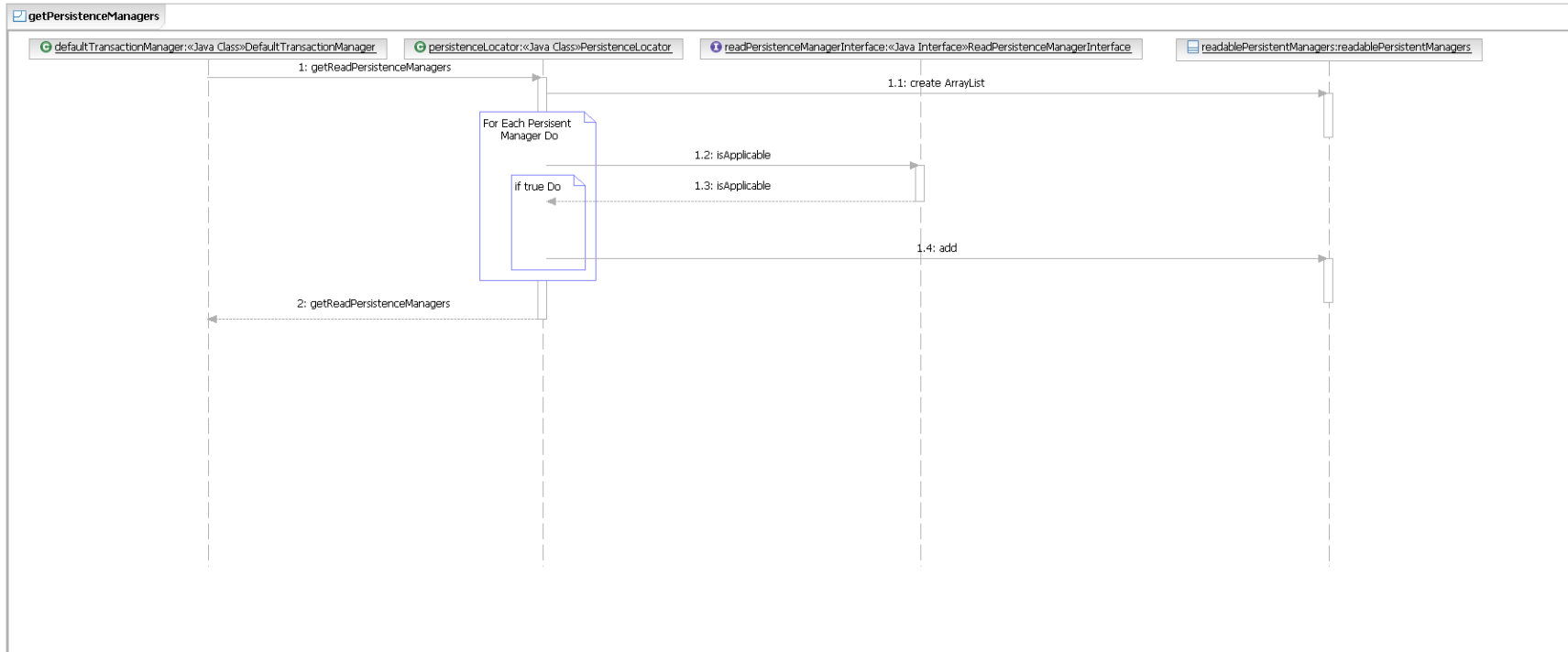
The classes involved and the relationships they have with one another are modeled in the class diagram pictured below:

Figure 32 Persistence Locator Class Diagram

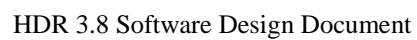


The sequence of calls required to retrieve the collection of persistence managers from the PersistenceLocator pictured below:

**Figure 33 Persistence Locator Sequence Diagram**



### Figure 34 Site Specific Persistence Manager Factory Sequence Diagram



## **6.2.1.20 Persistence Manager**

### **6.2.1.20.1 Persistence Manager Overview**

This section provides an overview of the CDS Persistence Manager component. This component is responsible for managing three significant processes in the write, read request workflow to write/read data to/from data sources configured within the Spring context. It is responsible for determining if a write/read persistence manager is applicable for a particular domain entry point, as well as managing the process to create a work object used in the query process, and to set up the work queue containing the threads that will perform the write/read operation on clinical data. The component is injected into the persistence locator object by the Spring framework and executed from within the transaction manager when it begins the query execution process.

#### **Write processing**

In the case of write request processing, the persistence framework allows create/update operations utilize a more flexible strategy for resolving data persisted in the HDR. The framework supports the capability only supported retrieving records by the HDR generated identity column (or primary key) value. The framework also supports usage of surrogate key (client generated) values. The `WriteableHibernatePersistenceManager` class is used for this purpose.

#### **Read processing**

In the case of read request processing, additional persistence capabilities in support of a more flexible filtering interface for clients to utilize have been incorporated into the HDR 3.5 CDS framework. These changes utilize Java introspection and Hibernate Criteria queries to dynamically build queries at runtime based on information passed in in the entry filter class. These new capabilities have been exposed by the CDS framework as a Query Strategy that maps filter request with specific identities to the Criteria based Query Work implementation, which does the work of processing the filter query request. This Query Work class supports building dynamic queries that can filter data by any combination of patient identity, date ranges, string values, and lists of string values. Currently, there is a restriction to only filtering results by columns contained within the entry point table. In the event that the filter criteria presented in the entry filter are not supported by the physical data model, an error will be returned to the client indicating the issue.

Additionally, query timeout capabilities supported by the HDR persistence framework were enhanced to exit from long running stored procedures. These long running stored procedures most often resulted in stuck threads that led to Cache system resources being tied up and made unavailable to client systems. By enhancing the timeout capabilities, stuck threads and Cache resource utilization have decreased.

In support of HDR VistA reads, functionality to pre-query and resolve Data File Number (DFN) to their corresponding row identifier can be configured and executed to speed up data reads.

### **6.2.1.20.2 Domain Entry Point Applicability**

Persistence Manager handles the rule processing for determining if the component is applicable for a given domain entry point. This logic is executed by the persistence locator when the



transaction manager calls for a list of persistence managers associated with an entry filter and list of person identifiers.

**NOTE:** Please refer to the documentation for the Rules processor component.

#### **6.2.1.20.3 Create Query Work Object**

The component manages the creation of a `List<QueryWorkInterface>` list containing `QueryWorkInterface` query work objects. The configuration of a query work object includes a Hibernate session that will be used for the query, as well as the HQL query for the domain model or association to be executed. The query work object extends the functioning interface, and is called by the work manager for query execution.

#### **6.2.1.20.4 Execute Read Query**

Once the query work `List<QueryWorkInterface>` list is created with instances of persistence manager components that are specific to a data source, the list is iterated over and each query work object is added to a work manager queue for scheduling. The work manager controls the pool of threads used to execute each query, and waits until all queries have been completed. The results of query execution are processed by the transaction manager component.

#### **6.2.1.20.5 Persistence Manager Design**

##### **Write processing**

`WriteableHibernatePersistenceManager` manages the create/update/delete operations using hibernate session which provides the HDR database connection.

##### **Read processing**

There are three concrete class implementations of the persistence manager component:

`DefaultHibernateReadPersistenceManager`

- `FailingReadableHibernatePersistenceManager`
- `SiteSpecificHibernateReadPersistenceManager`

The `FailingReadableHibernatePersistenceManager`, and the `SiteSpecificHibernateReadPersistenceManager` both extend the `DefaultHibernateReadPersistenceManager`.

Of these implementations, the `DefaultHibernateReadPersistenceManager` and `SiteSpecificHibernateReadPersistenceManager` play the significant role in the read request process for this component. Each of these concrete classes inherits a hierarchy of abstract base classes and interfaces that set up the Hibernate query environment, configures applicability rules, creates associations to delegation objects, and adds data source specific read functionality.

**NOTE:** See the Persistence Manager Applicability Rules Context Configuration data structure below.

The component is instantiated and configured when the persistence locator is injected with an instance of a `SiteSpecificHibernateReadPersistenceManagerFactory`.

**NOTE:** See the `PersistenceLocator` Context Configuration and `SiteSpecificHibernateReadPersistenceManagerFactory` Context Configuration data structures below.

The factory object creates persistence managers and Hibernate session factories that are bound to specific Vista data sources. It is important to note that each persistence manager instance, when instantiated is initialized to be applicable to a specific data source and list of domain entry point rules. The locator in turn provides the transaction manager the collection of persistence managers applicable to a particular entry filter.

The `DefaultHibernateReadPersistenceManager` object extends the `AbstractReadHibernatePersistenceManager` class, where it inherits the public method, `performReadOnClinicalData`. This method is the entrance point to the read request workflow for this component and is responsible for selecting the person identifiers that are applicable to this persistence manager and creating the query work objects. The `SiteSpecificHibernateReadPersistenceManager` delegates the call to this method during a read request to its base class, providing an implementation for the `createQueryWork` abstract method to create the query objects specific to the entry filter and patient identifiers for this read request.

#### **6.2.1.20.6 Persistence Manager Processing**

In the scope of this component, a read request is initiated by the transaction manager component. The transaction manager first calls the `getReadPersistenceManagers` on the persistence locator which returns a `Collection<ReadPersistenceManagerInterface>` object. This collection contains persistence managers that are applicable to the domain entry points in the entry filter, and the data sources appropriate to the person identifiers. For each of the persistence managers in the collection, a new `ReadWork` object is created, scheduled and added to a work queue. The work manager that oversees the queue starts the `ReadWork` threads and waits until all threads have finished.

The `ReadWork` threads which were created with the persistence managers returned by the locator, are the functioning objects started by the work manager. Their run methods call the `performReadOnClinicalData` methods on their persistence manager instances. For this component, the `performReadOnClinicalData` method is called on the `SiteSpecificHibernateReadPersistenceManager`, which inherits the method implementation from the `AbstractReadHibernatePersistenceManager` class. This method obtains a query strategy object appropriate to the template id and domain entry point in the entry filter, and creates a list of query work objects for the query strategy. The list is iterated over and each query work item is executed in its own thread by the work manager. Read results are passed back to the transaction manager as Document objects.

#### **6.2.1.20.7 Persistence Manager Local Data Structures**

##### **PersistenceLocator Context Configuration**

```
<bean id="persistenceLocator" class="gov.va.med.cds.persistence.PersistenceLocator">
```

```

<property name="staticPersistenceManagers">
<list>
<ref bean="hdr2Readable"/>
<ref bean="hdr2Writeable"/>
</list>
</property>
<property name="dynamicPersistenceManagers">
<bean factory-bean="vistaPersistenceManagerFactory" factory-method="create"/>
</property>
</bean>

```

## SiteSpecificHibernateReadPersistenceManagerFactory Context Configuration

```

<bean id="vistaPersistenceManagerFactory"
class="gov.va.med.cds.persistence.hibernate.SiteSpecificHibernateReadPersistenceManagerFactory">
<property name="sessionFactoryPropertyHolderMap" ref="vistaSessionFactoryMap"/>
<property name="dataSourceBindingsLocator" ref="vistaDataSourceBindingsLocator"/>
<property name="rules" ref="defaultVistaRules"/>
<property name="templateQueryStrategyMap" ref="vistaTemplateQueryStrategyMap"/>
<property name="pointInTimeUserType"
value="gov.va.med.cds.persistence.hibernate.common.VistaPointInTimeUserType"/>
<property name="managerIdentifierFormat" value="VISTA-{0}-R"/>
<property name="workManager" ref="workManager" />
</bean>

```

## Persistence Manager Applicability Rules Context Configuration

```

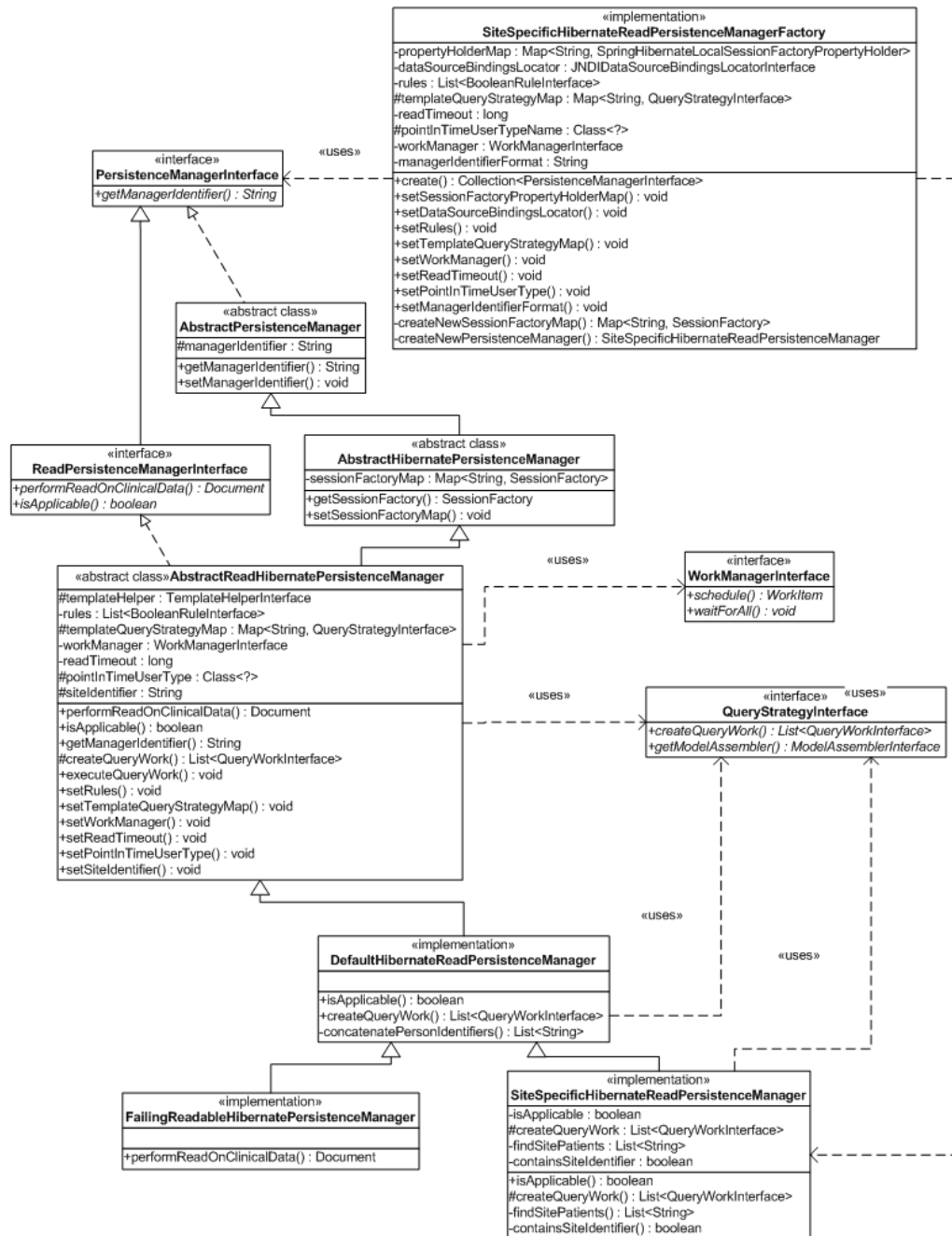
<util:list id="defaultVistaRules">
<bean class="gov.va.med.cds.rules.DomainEntryPointRule">
<property name="entryPoint" value="AllergyAssessment" />
</bean>
<bean class="gov.va.med.cds.rules.DomainEntryPointRule">
<property name="entryPoint" value="IntoleranceCondition" />
</bean>
<bean class="gov.va.med.cds.rules.DomainEntryPointRule">
<property name="entryPoint" value="ClinicalDocumentEvent" />
</bean>
<bean class="gov.va.med.cds.rules.DomainEntryPointRule">
<property name="entryPoint" value="VitalSignObservationEvent" />
</bean>
</util:list>

```

### 6.2.1.20.8 Persistence Manager Module/Other Diagrams

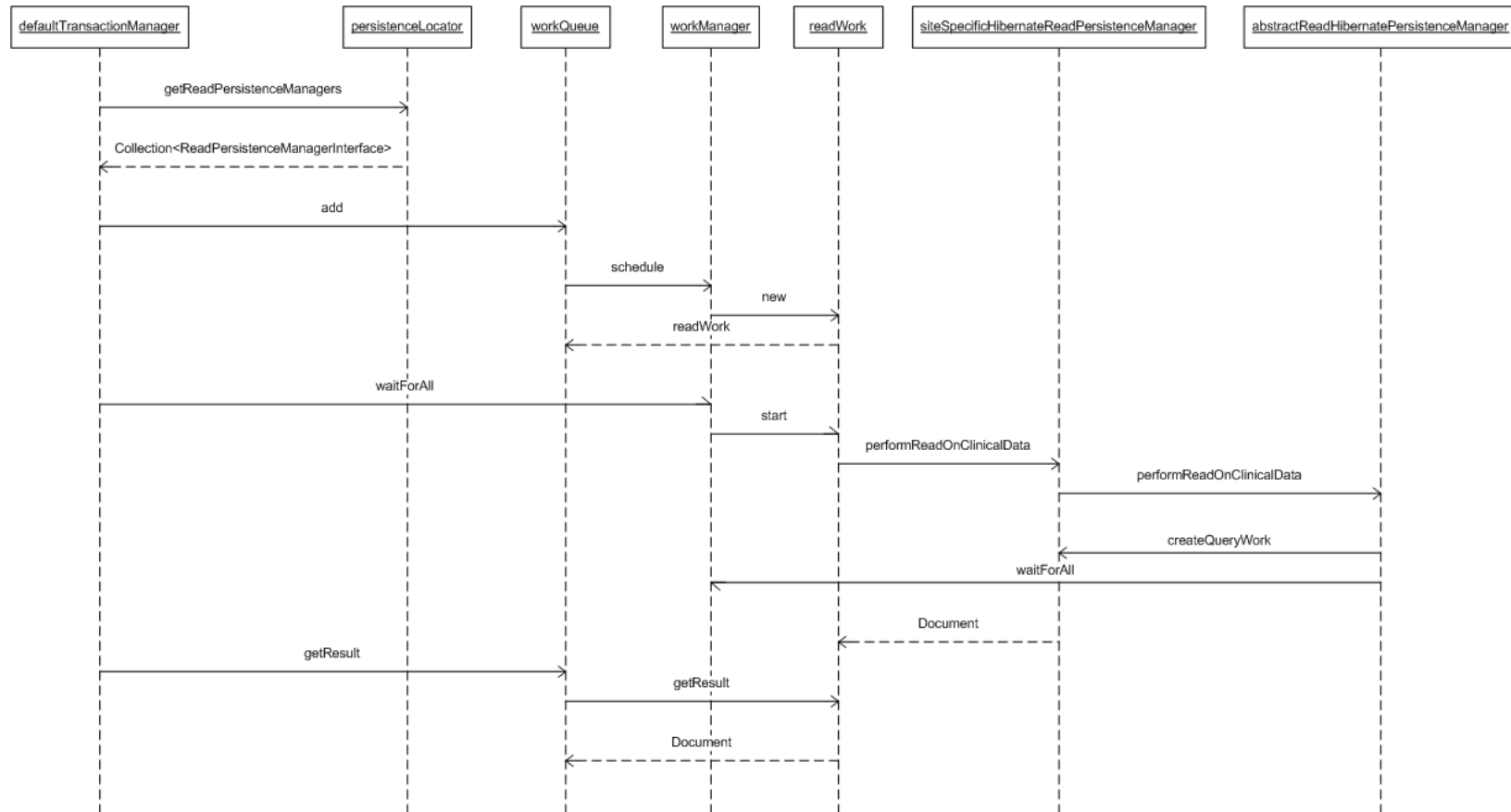
The following figure represents the static structure of the component. Concrete implementations of the DefaultHibernateReadPersistenceManager, FailingReadableHibernatePersistenceManager, and SiteSpecificHibernateReadPersistenceManager.

Figure 35 Persistence Manager Class Diagram



The following figure depicts the sequence of calls made from the clinicalDataServiceSynchronousInternal calling code.

**Figure 36 Persistence Manager Sequence Diagram**



### **6.2.1.21 Rules Processor**

This section is applicable for write/read request processing.

#### **6.2.1.21.1 Rules Processor Overview**

This section provides an overview of the domain entry point applicability rules processing logic within the persistence manager component for write/read request processing. The role of the component is to determine if a particular instance of a persistence manager can service a read request. Each persistence manager instance created by a `SiteSpecificHibernateReadPersistenceManagerFactory` object is initialized with a `List<BooleanRuleInterface>` list containing a set of applicability rules which are defined in the Spring context. Each individual rule is associated to a domain entry point, and is tested by the persistence locator to determine if a read persistence manager can participate in a read request for the domain entry point given in an entry filter.

#### **6.2.1.21.2 Rules Processor Module Design**

The initialization and configuration of the rules processing logic is performed by the Spring container. The list of rules is represented at runtime in a `List<BooleanRuleInterface>`, which is configured in Spring context at container initialization, as explained in section 6.2.1.21.4 below, and injected into an instance of a `SiteSpecificHibernateReadPersistenceManagerFactory`. A rule is an implementation of the `BooleanRuleInterface` interface, which in this case is a `DomainEntryPointRule` object. Each 'entryPoint' property of the `DomainEntryPointRule` object is configured with the value of a domain entry point name in the Spring configuration, which will be tested to match the name of a domain entry point in an entry filter within the read request process.

Spring injects an instance of the `SiteSpecificHibernateReadPersistenceManagerFactory` into the persistence locator during initialization. This factory object creates a persistence manager instance for each configured data source as explained in section 6.2.1.21.4 below, and each of these managers initializes with the list of rules to be used during the applicability test.

#### **6.2.1.21.3 Rules Processor Processing**

Within the read request workflow, when the transaction manager executes a read query, it obtains a `Collection<ReadPersistenceManagerInterface>` collection object containing persistence managers from the persistence locator. Before the collection is returned to the transaction manager, the persistence locator iterates over its list of all persistence managers to determine applicability by executing the 'isApplicable' method on the persistence manager. Within this method, the domain entry point name in the filter parameter is tested against the value in the 'entryFilter' property of the persistence manager. If the values equate, the persistence manager is able to service the read request and is added to the `Collection<ReadPersistenceManagerInterface>` object returned to the transaction manager.

#### **6.2.1.21.4 Rules Processor Local Data Structures**

The following configuration details the `PersistenceLocator` context:

```
<bean id="persistenceLocator" class="gov.va.med.cds.persistence.PersistenceLocator">
  <property name="staticPersistenceManagers">
```

```

<list>
  <ref bean="hdr2Readable"/>
  <ref bean="hdr2Writeable"/>
</list>
</property>
<property name="dynamicPersistenceManagers">
  <bean factory-bean="vistaPersistenceManagerFactory" factory-method="create"/>
</property>
</bean>

```

The following details the SiteSpecificHibernateReadPersistenceManagerFactory context configuration:

```

<bean id="vistaPersistenceManagerFactory"
class="gov.va.med.cds.persistence.hibernate.SiteSpecificHibernateReadPersistenceManagerFactory">
  <property name="sessionFactoryPropertyHolderMap" ref="vistaSessionFactoryMap"/>
  <property name="dataSourceBindingsLocator" ref="vistaDataSourceBindingsLocator"/>
  <property name="rules" ref="defaultVistaRules"/>
  <property name="templateQueryStrategyMap" ref="vistaTemplateQueryStrategyMap"/>
  <property name="pointInTimeUserType"
value="gov.va.med.cds.persistence.hibernate.common.VistaPointInTimeUserType"/>
  <property name="managerIdentifierFormat" value="VISTA-{0}-R"/>
  <property name="workManager" ref="workManager" />
</bean>

```

The following details the persistence manager applicability rules context configuration:

```

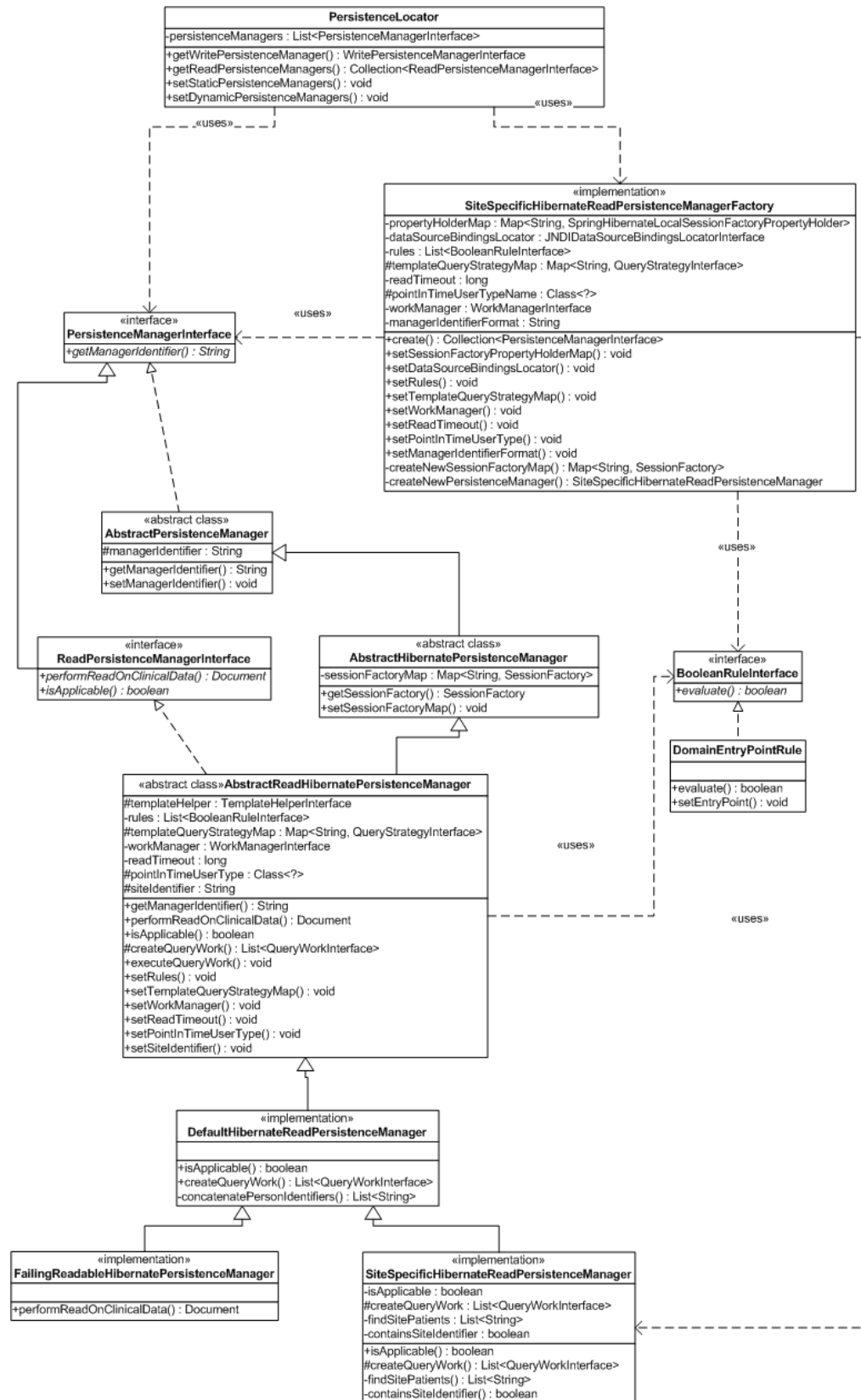
<util:list id="defaultVistaRules">
  <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
    <property name="entryPoint" value="AllergyAssessment" />
  </bean>
  <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
    <property name="entryPoint" value="IntoleranceCondition" />
  </bean>
  <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
    <property name="entryPoint" value="ClinicalDocumentEvent" />
  </bean>
  <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
    <property name="entryPoint" value="VitalSignObservationEvent" />
  </bean>
</util:list>

```

#### 6.2.1.21.5 Rules Processor Module/Other Diagrams

The following class diagram depicts the static structure of the component. It depicts the <<uses>> relationships between the persistence locator, the persistence factory object and the persistence manager:

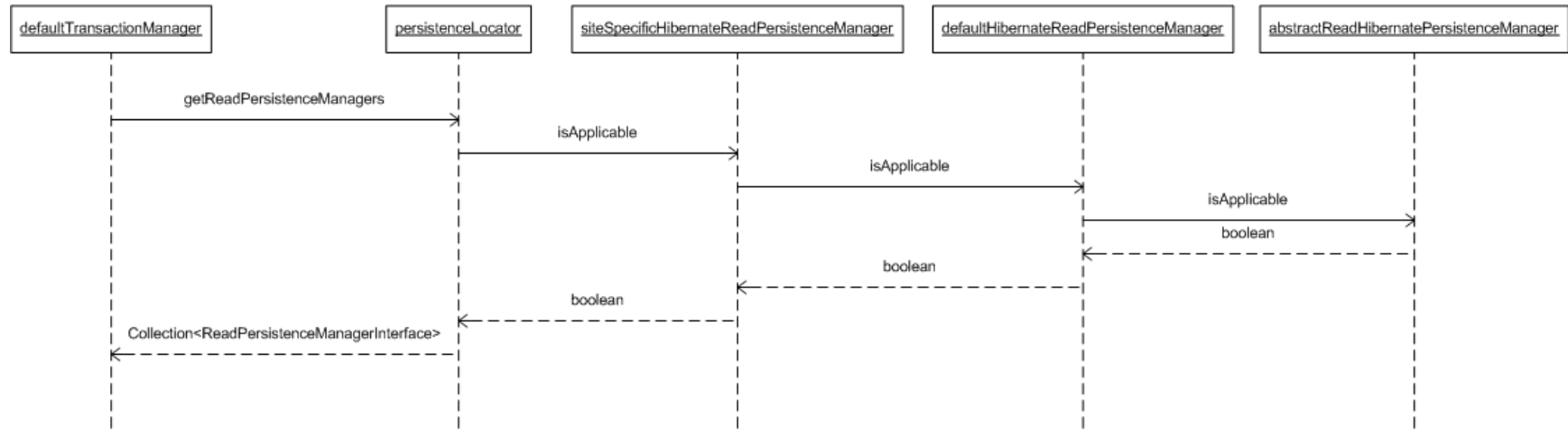
Figure 37 Persistence Manager Class Diagram





The following sequence diagram depicts the sequence of calls made from the defaultTransactionManager to the persistenceLocator, then to the read manager implementations. As can be seen, the read manager implementation SiteSpecificHibernateReadPersistenceManager code currently defers its call to 'isApplicable' to its base classes:

**Figure 38 Rules Processor Sequence Diagram**



### **6.2.1.22 Custom O/R Mapping Strategy**

This section is applicable for read request processing.

#### **6.2.1.22.1 Custom O/R Mapping Strategy Overview**

This section describes an object relational mapping strategy that creates concurrent Hibernate sessions to populate a model object defined by a given Hibernate hbm mapping file. The resultant model object should be created in a generic way, regardless of the model type as defined by the Hibernate hbm mapping file. The purpose of the strategy is to improve the efficiency of the Hibernate database query process that by default performs sequential select statements to populate a model object and its defined associations.

Subsequent to an hbm model definition having many-to-one associations, Hibernate can perform an excessive number of SQL queries, potentially as many as  $Kn+1$ , where K is the sequence (association) being computed and n is the number of objects returned. This means that for each object that Hibernate loads, it may execute one or more extra SQL queries serially to load associated objects. This can be demonstrated by viewing execution logs.

One well known solution to the  $Kn+1$  issue is to load associated objects in the initial query by using the "left join fetch" query construction, and configuring the fetch strategy as "fetch=join" on the association definition. This will necessitate left joins on all directly and indirectly associated objects, and can be shown to perform on models with one-to-one associations, and non-complex, eager loaded many-to-one associations. However, in cases with model objects that have several or complex many-to-one or many-to-many relationships, Hibernate can potentially generate an uncontrollable number of SQL queries (Cartesian joins), which cannot be resolved using the left join fetching syntax.

There are a number of solutions to improving performance for this scenario, possibly the most prevalent of which is to reduce the complexity of the many-to-one or many-to-many relationships by decreasing the number of fields returned by the query. This depends on business requirements and is not always possible.

The solution described here features a lightweight framework that implements the well-known ReadPersistenceManagerInterface, and extends the AbstractReadHibernatePersistenceManager contract for performing reads of patient data against the HDR II and site specific VistA Caché databases. The framework will create a new Hibernate session for each patient model, and a new session for each of the model's included associations as defined by the model's hbm mapping file. Each session will execute a named query defined within the hbm mapping file. The framework will execute the sessions concurrently, and assemble the query results into the final parent model using the Hibernate metadata and XML Mapping APIs. The results returned out of the framework will be an org.dom4j.Document compliant to the ClinicalData schema, as dictated by the ReadPersistenceManagerInterface contract.

#### **6.2.1.22.2 Custom O/R Mapping Strategy Module Design**

This strategy describes a component that exists within the CDS Business Capability BC 1.3 – TIU Read architecture (BC 1.3), and is designated as functional component "Custom O/R Mapping Strategy Using Hibernate FC 10.0" (FC 10.0). The following integration description refers to the BC 1.3 architecture.

In reference to the BC 1.3 diagram, the FC 10.0 component is called by the “Concurrent Data Source Call Processing FC 6.0” (FC 6.0) component. Please refer to the FC 6.0 design documentation for more detail on this component. A high level class diagram of the FC 6.0 component is shown below:

HDR 3.8 Software Design Document



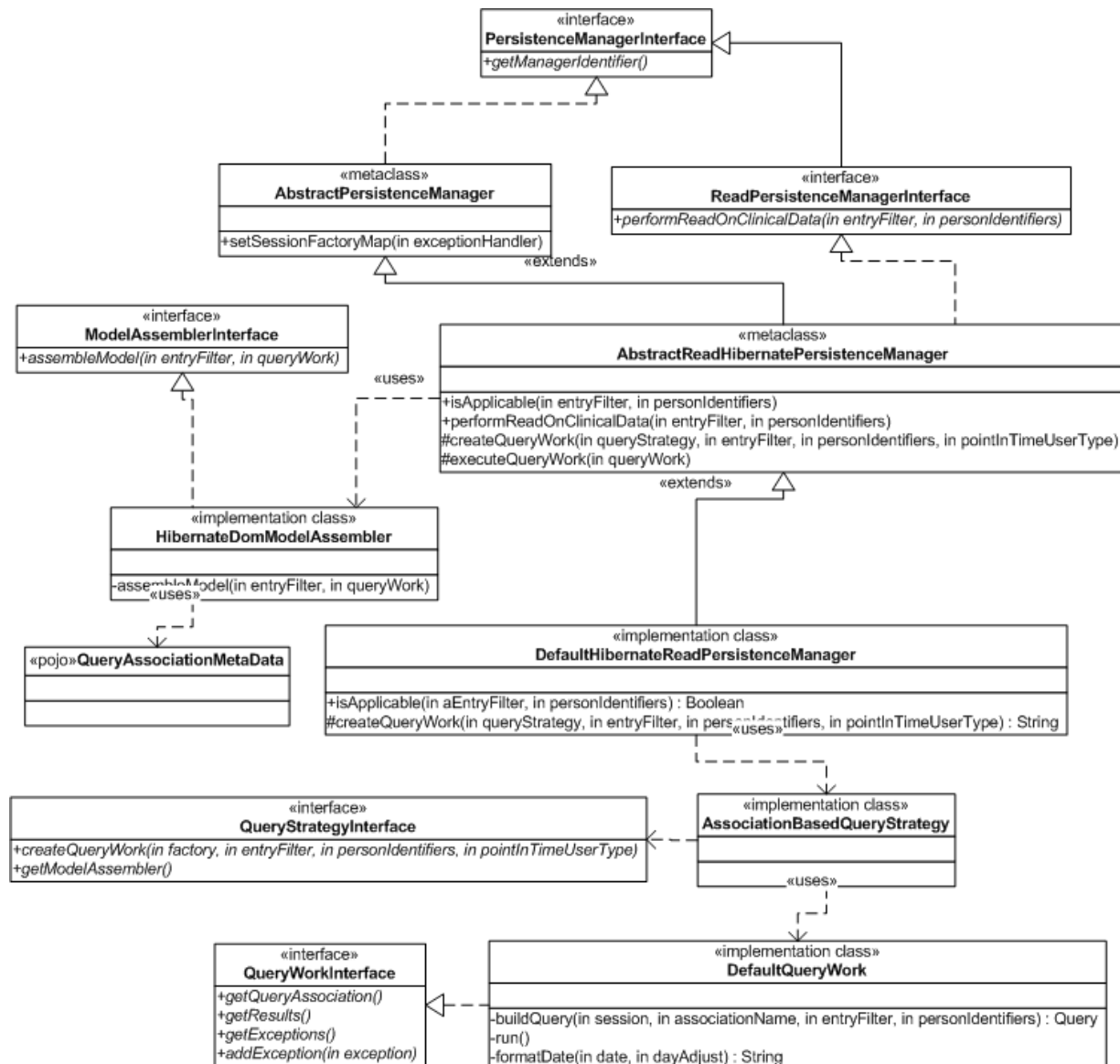
The integration of FC 10.0 into the BC 1.3 architecture is at the `ReadPersistenceManagerInterface` which extends the `PersistenceManagerInterface` plug point (shown above). As shown in the above diagram, “ReadWork?” objects are concurrently executing threads that instantiate instances of classes that implement a request contract, defined by the `ReadPersistenceManagerInterface` interface. A default implementation of the FC 10.0 `DefaultHibernateReadPersistenceManager` class implements this contract and can be executed by “VistARReadWork” objects (implementing the `ReadWork` Interface). The `ReadPersistenceManager` is described in more detail later. In addition, the contract also dictates that the processing results of the FC 10.0 component can be accepted by the BC 1.3 architecture, which in this case is an XML document that is compliant to the `ClinicalData` schema.

Integration calls made out of, or from the FC 10.0 component are directed to the “Managed JDBC Data Source FC 20.1” (FC 20.1) component. In reference to the BC 1.3 architecture diagram, the FC 10.0 component connects to the FC 20.1 component to gain access to patient data by executing named queries. In terms of the integration to the FC 20.1 component, the FC 10.0 framework is configured by a set of Spring context files that set up and manage JDBC connections to configured data sources. In this case, FC 10.0 is injected with a Hibernate session factory that is able to create connections to the appropriate data sources as represented by FC 20.1.

#### **6.2.1.22.3 Custom O/R Mapping Strategy Processing**

The following component design section refers to the class diagram below:

**Figure 40 Custom O/R Mapping Class Diagram**



The significant logic within the framework is performed by the **DefaultHibernateReadPersistenceManager** object which implements **ReadPersistenceManagerInterface**, the **DefaultQueryWork** object which implements **QueryWorkInterface** and the **HibernateDomModelAssembler** object which implements the **ModelAssemblerInterface**.

The bulk of the management logic within the framework is in the **DefaultHibernateReadPersistenceManager** object that extends the **AbstractReadHibernatePersistenceManager** object, which will have the Spring context configuration inject a map of query request strategies, template, and Hibernate session information upon initialization. A request strategy is contained within the **QueryStrategyInterface** implementation, and is defined as a collection of named queries within an hbm file configured for a VIM read request template. Because the template is known at the

time the DefaultHibernateReadPersistenceManager object is called, the manager object can determine what request strategy to create and pass to the QueryStrategyInterface object.

The DefaultHibernateReadPersistenceManager will create a DefaultQueryWork object for the patient identified in the person identifier object. Error handling logic will determine if any of the required parameters are null, or have not been configured appropriately. Any errors that do occur will result in error information being returned in the exception section of the ClinicalData XML document. The DefaultQueryWork, with the assistance of the HibernateDomModelAssembler reassembles the associations and the model into a coherent model object. The model is then transformed into an XML document using the Hibernate XML Mapping API. The patient XML documents are assembled into the ClinicalData XML document and returned to the external BC 1.3 architecture calling component.

On initialization, the AssociationBasedQueryStrategy object is injected with a Hibernate session factory, configured to synchronize database data in a "dom4j" entity mode, and a suitable request strategy configuration applicable to the VIM read request template. The design of this framework will only support the "dom4j" entity mode, which manipulates domain objects within a XML tree. Other entity modes such as "POJO" and "MAP" are available in the Hibernate API. The AssociationBasedQueryStrategy is responsible for creating a new Hibernate session for each of the named queries defined within a request strategy configuration. A collection of the queries to be executed is passed to the DefaultQueryWork object which then executes the queries in concurrent threads, and passes back any results in the form of a List of dom4j Element objects. Subsequent to obtaining query results, the DefaultHibernateReadPersistenceManager uses the HibernateDomModelAssembler object to assemble the list of Element objects into a coherent model object.

The Hibernate XML Mapping API enables client code to use persistent XML data (dom4j XML trees) in similar way as POJO objects. Domain model objects represented in XML trees can be synchronized to a database, much like a POJO counterpart, using Hibernates basic operations: persist(), saveOrUpdate(), delete() and replicate(). Named queries are supported and return results in dom4j Element objects. Configuration and representation of domain model objects is performed within the familiar HBM mapping file. It should be noted that at the time of this writing that the Hibernate XML Mapping API is an experimental feature of Hibernate 3.0, and is currently under active development. Please see the Hibernate reference documentation at:

<http://docs.jboss.org/hibernate/stable/core/reference/en/html/xml.html>

#### **6.2.1.22.4 Custom O/R Mapping Strategy Local Data Structures**

The input contract defined by the ReadPersistenceManagerInterface interface specifies that the framework take the read request entry filter, and the person identifier object, both of which are required parameters. Of particular significance to the framework is the entry filter. This input is used in conjunction with the Spring context configuration to determine the appropriate request strategy, or set of named queries, to execute for the patient identified in the patient identifier parameter.

The framework's injected Spring context configuration is at the heart of the FC 10.0 component, and dictates the operational capabilities of the implementation classes of both the ReadPersistenceManagerInterface and QueryStrategyInterface objects. The context is responsible for defining a request strategy for a given VIM read request template, and further

defining what named queries comprise a given request strategy. It is also responsible for making a Hibernate session factory and an exception handler available to the framework.

On initialization, the `ReadPersistenceManagerInterface` implementation class is injected with the following Spring configuration that defines a request strategy for a filter entry point. The class can be configured with any number of “requestStrategies”. The entry point property is mapped to the entry point filter as displayed in the following configuration:

```
<bean id="hdr2Readable"
class="gov.va.med.cds.persistence.hibernate.DefaultHibernateReadPersistenceManager">
    <property name="managerIdentifier" value="HDRII-R"/>
    <property name="sessionFactoryMap" ref="hdr2SessionFactoryMap"/>
    <property name="rules">
    <list>
        <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
            <property name="entryPoint" value="IntoleranceCondition"/>
        </bean>
        <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
            <property name="entryPoint" value="AllergyAssessment"/>
        </bean>
        <bean class="gov.va.med.cds.rules.DomainEntryPointRule">
            <property name="entryPoint" value="VitalSignObservationEvent"/>
        </bean>
    </list>
    </property>
</bean>
```

The session factory map contains definitions for all applicable schemas as in the following configuration:

```
<util:map id="hdr2SessionFactoryMap">
    <entry key="AllergiesRead40010" value-ref="hdr2SessionFactory"/>
    <entry key="AllergyAssessmentCreate40020" value-ref="hdr2SessionFactory" />
    <entry key="AllergyAssessmentCreateOrUpdate40060" value-ref="hdr2SessionFactory" />
    <entry key="AllergyAssessmentDelete40030" value-ref="hdr2SessionFactory" />
    <entry key="AllergyAssessmentRead40010" value-ref="hdr2SessionFactory" />
    <entry key="AllergyAssessmentVdmBuilderStringRead40019" value-ref="hdr2SessionFactory" />
    <entry key="IntoleranceConditionCreate40020" value-ref="hdr2SessionFactory" />}}
.
.
.
</util:map>
```

Likewise, each session factory has applicable Hibernate mapping files as displayed in the following configuration:

```
<bean id="hdr2SessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="hdrDataSource" />
    <property name="mappingResources">
```



```

<list>
  <!-- Major component mapping files -->
  <!-- Allergy -->
  <value>gov/va/med/cds/persistence/hibernate/allergy/IntoleranceCondition.hbm.xml</value>

  <value>gov/va/med/cds/persistence/hibernate/allergy/AllergyPractitionerParticipationChartMarker.hbm.xml</value>

  <value>gov/va/med/cds/persistence/hibernate/allergy/AllergyPractitionerParticipationIdBandMarker.hbm.xml</value>
  <value>gov/va/med/cds/persistence/hibernate/allergy/Reaction.hbm.xml</value>
  <value>gov/va/med/cds/persistence/hibernate/allergy/DrugClass.hbm.xml</value>
  .
  .
  .
</bean>

```

Furthermore, on initialization the `AssociationBasedQueryStrategy` implementation class is injected with the following Spring configuration that defines what named queries and query metadata comprise a request strategy. A `<map>` entry exists in the `AssociationBasedQueryStrategy` implementation class configuration for each of the “requestStrategies” defined for the `DefaultHibernateReadPersistenceManager` implementation class. Each entry of the `<map>` refers to a named query. Each of the entries includes the configuration of a helper class called `QueryAssociationMetaData`, that contains metadata information necessary to define and execute a named query within an hbm mapping file. The fields within an `QueryAssociationMetaData` object are defined as:

- `associationName`: the name of the association or model
- `filterModelName`: the name of the entry filter

Each entry in the `queryPlanMap` property refers to child associations of the parent model to be executed in concurrent threads.

A configuration example for a parent and association model is shown below:

```

<util:map id="templateQueryStrategyMap">
  <entry key="TiuDocumentListRead40011">
    <bean class="gov.va.med.cds.persistence.AssociationBasedQueryStrategy">
      <property name="enableFiltering" value="true"/>
      <property name="modelAssembler">
        <bean class="gov.va.med.cds.persistence.hibernate.HibernateDomModelAssembler"/>
      </property>
      <property name="queryPlanMap">
        <map>
          <entry key="clinicalDocumentEvents">
            <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
              <property name="filterModelName" value="clinicalDocumentEvents"/>
              <property name="associationName" value="patient"/>
              <property name="key" value="id"/>
            </bean>
          </entry>
        </map>
      </property>
    </bean>
  </entry>
</util:map>

```

```

        </bean>
    </entry>
    <entry key="otherExpectedCosigners">
        <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
            <property name="associationName" value="clinicalDocumentEvents"/>
            <property name="key" value="id"/>
            <property name="parentKey" value="clinicalDocumentId"/>
        </bean>
    </entry>
    <entry key="otherCosigners">
        <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
            <property name="associationName" value="clinicalDocumentEvents"/>
            <property name="key" value="id"/>
            <property name="parentKey" value="clinicalDocumentId"/>
        </bean>
    </entry>
    <entry key="images">
        <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
            <property name="associationName" value="clinicalDocumentEvents"/>
            <property name="key" value="id"/>
            <property name="parentKey" value="clinicalDocumentId"/>
        </bean>
    </entry>
    <entry key="documentUpdateEvents">
        <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
            <property name="associationName" value="clinicalDocumentEvents"/>
            <property name="key" value="id"/>
            <property name="parentKey" value="clinicalDocumentId"/>
        </bean>
    </entry>
</map>
</property>
</bean>
</entry>
</util:map>

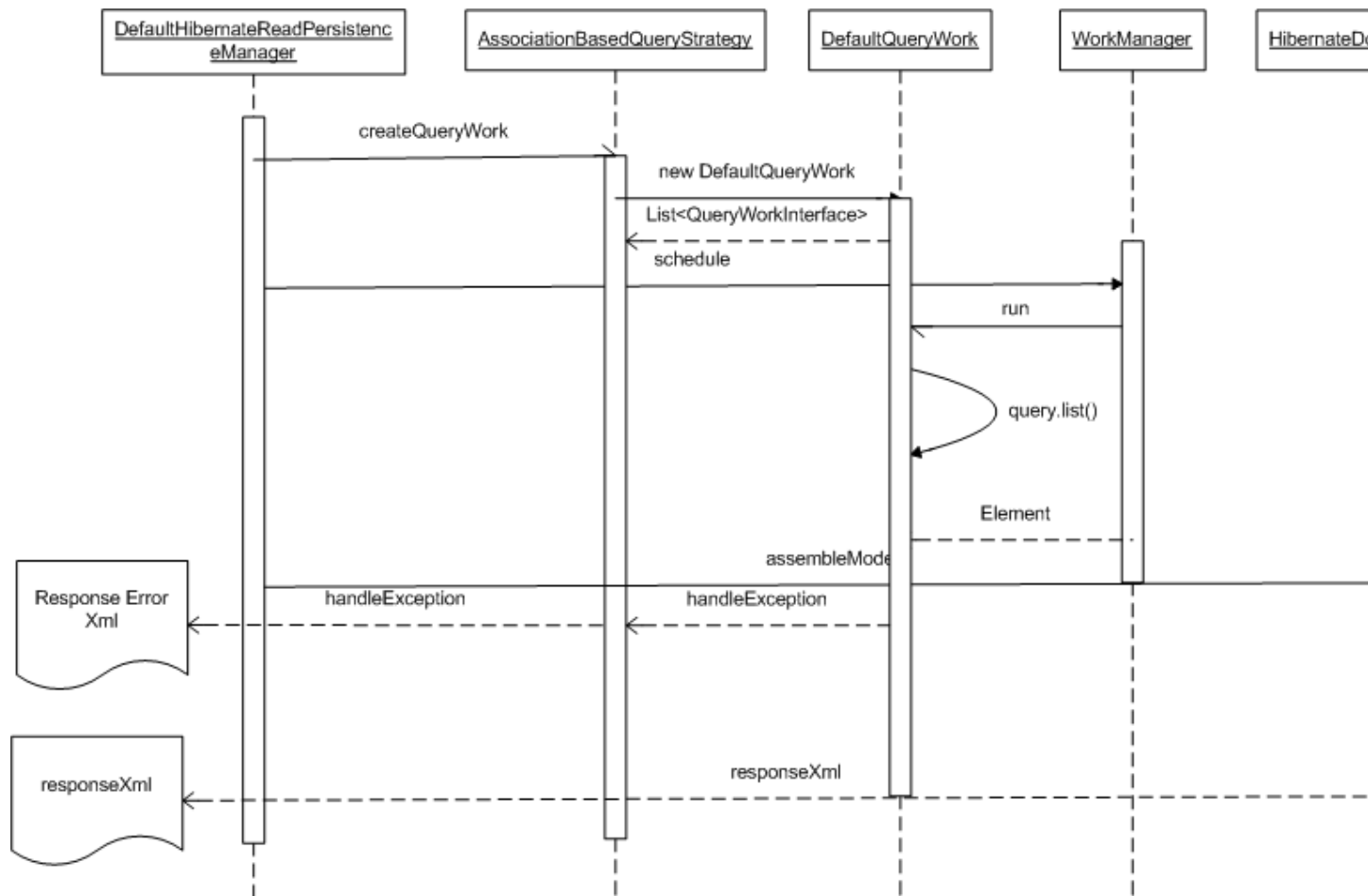
```

The output defined by the `ReadPersistenceManagerInterface` contract is a `org.dom4j.Document` object containing XML compliant to the `ClinicalData` schema, containing either patient data matching the filter, or exception data. The `QueryStrategyInterface` implementation class is responsible for capturing all exceptions generated by the framework, and creating an error XML response string that is included within the `ClinicalData` document.

#### 6.2.1.22.5 Custom O/R Mapping Strategy Module/Other Diagrams

The following component design discussion refers to the sequence diagram below:

**Figure 41 Custom O/R Mapping Sequence Diagram**



The sequence of message processing is initiated at the BC 1.3 architecture plug point represented by the `ReadPersistenceManagerInterface` object, which is itself a thread of execution within the architecture. The `ReadPersistenceManagerInterface` object calls the “`createQueryWork()`” method on the `AssociationBasedQueryStrategy` object, which creates a `DefaultQueryWork` object set for each of the associations defined by the Spring configuration, then calls the “`schedule()`” method on the `WorkManager` in order to execute the defined queries concurrently.

The `AssociationBasedQueryStrategy` creates a `DefaultQueryWork` object for each named query defined by the request strategy, and passes the object to a `WorkManager` which starts the query thread. The thread then executes the named query and terminates. The `AssociationBasedQueryStrategy` adds each named query result to an `associationModelMap`, and passed control back to the `ReadPersistenceManagerInterface` object. The manager in turn passes the map to the `HibernateDomModelAssembler`, which creates a `ClinicalData patient org.dom4j.Document` using the Hibernate metadata API. The document object is then returned to the `ReadPersistenceManagerInterface` object and out of the framework.

### **6.2.1.23 Query Processor**

This section is applicable for read request processing.

#### **6.2.1.23.1 Query Processor Overview**

This section provides an overview of the CDS Query Processor component and is applicable to read request processing. This component is responsible for executing the set of queries that retrieve the data used to populate the response model.

The following four sub-processes execute as part of query processing:

1. The Query Processor identifies the query strategy required to retrieve the data.
2. The Query Processor instantiates and initializes the query work associated with the query strategy in preparation for query execution.
3. The Query Processor executes each query work instance concurrently using the WebLogic Work Manager infrastructure of the WebLogic Server.
4. Once all query work has completed, the model assembler component assembles the response information into a single logical model (XML document). The Query Processor returns the fully assembled model to the persistence manager sub-system for further processing.

In the event that an exception occurs as part of the query processor process, a CDS specific persistence Exception is thrown to the persistence manager component for incorporation into the response to the client. Each of these sub-components or sub-processes will be further elaborated on in the following sections.

#### **6.2.1.23.2 Query Processor Module Design**

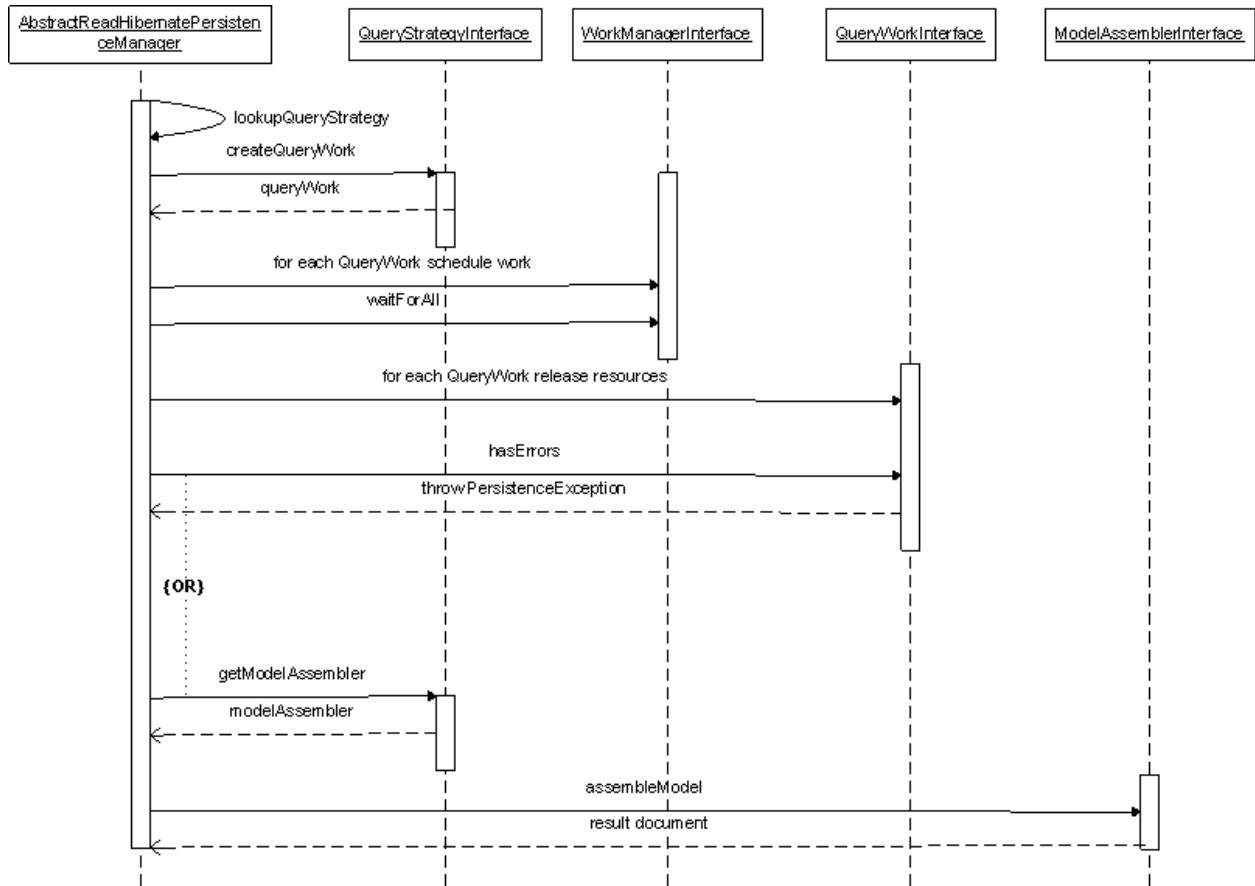
The Query Processor sub-system uses a strategy pattern to execute one or more queries against a single data source. This allows CDS to optimize reads of highly relational data across a wide area network. For instance, when CDS reads data stored across multiple tables with many relationships, the Cartesian Product effect can bloat the data, such as VistA data stored in the HDR database. This could impact performance if that bloated data gets packaged and sent across a WAN. By splitting a single join query into multiple concurrently executing queries, then reassembling the data at the server within the CDS service, the query result communication across the WAN can be optimized.

#### **6.2.1.23.3 Hibernate Read Persistence Manager Processing**

During read request processing by CDS, after one or more Hibernate read persistence managers have been identified and are ready for use, the Query Processor components are engaged to fulfill the request. The flow of events is described in the following numbered list and diagram:

1. Identify the query strategy to be used.
2. Create the query work using the identified query strategy.
3. Execute the query work using the Work Manager provided by the application server.
4. Assemble the model or XML document and return it to the caller.

**Figure 42 Query Processor Sequence Diagram**



For the read persistence manager to determine the query strategy to use to fulfill a particular read request, two pieces of information are needed: template ID and entry filter name. These two pieces of information are concatenated together with the hyphen character and mapped to the correct query strategy instance to use to fulfill the read request. The read persistence manager is initialized or injected with the map when the object is created by the Spring Framework. The following example shows the map data structure that the CDS Vista read persistence managers are injected with.

#### 6.2.1.23.4 Template Query Strategy Map – Mapping from Template ID and Entry Point name to Query Strategy

```

<util:map id="vistaTemplateQueryStrategyMap">
    <entry key="TiuDocumentListRead2-ClinicalDocumentEvent" value-
ref="multiQueryTiuDocumentReadListQueryStrategy" />
    <entry key="TiuDocumentDetailRead2-ClinicalDocumentEvent" value-
ref="singleQueryTiuDocumentDetailReadQueryStrategy" />
    <entry key="AllergiesRead40010-AllergyAssessment" value-
ref="singleQueryAllergyAssessmentQueryStrategy" />
    <entry key="AllergiesRead40010-IntoleranceCondition" value-
ref="singleQueryIntoleranceConditionStrategy" />
    <entry key="AllergyAssessmentRead40010-AllergyAssessment" value-
ref="singleQueryAllergyAssessmentQueryStrategy" />

```

```

        <entry key="IntoleranceConditionRead40010-IntoleranceCondition" value-
ref="singleQueryIntoleranceConditionStrategy" />

        <entry key="IntoleranceConditionReducedRead40011-IntoleranceCondition" value-
ref="singleQueryIntoleranceConditionStrategy" />

        <entry key="VWIntoleranceConditionRead40010-IntoleranceCondition" value-
ref="singleQueryIntoleranceConditionStrategy" />

        <entry key="MHVIntoleranceConditionRead40011-IntoleranceCondition" value-
ref="singleQueryIntoleranceConditionStrategy" />

        <entry key="VWAllergiesRead40010-IntoleranceCondition" value-
ref="singleQueryIntoleranceConditionStrategy" />

        <entry key="VWAllergiesRead40010-AllergyAssessment" value-
ref="singleQueryAllergyAssessmentQueryStrategy" />

        <entry key="VitalsignsRead40010-VitalSignObservationEvent" value-
ref="singleQueryVitalsignsQueryStrategy" />

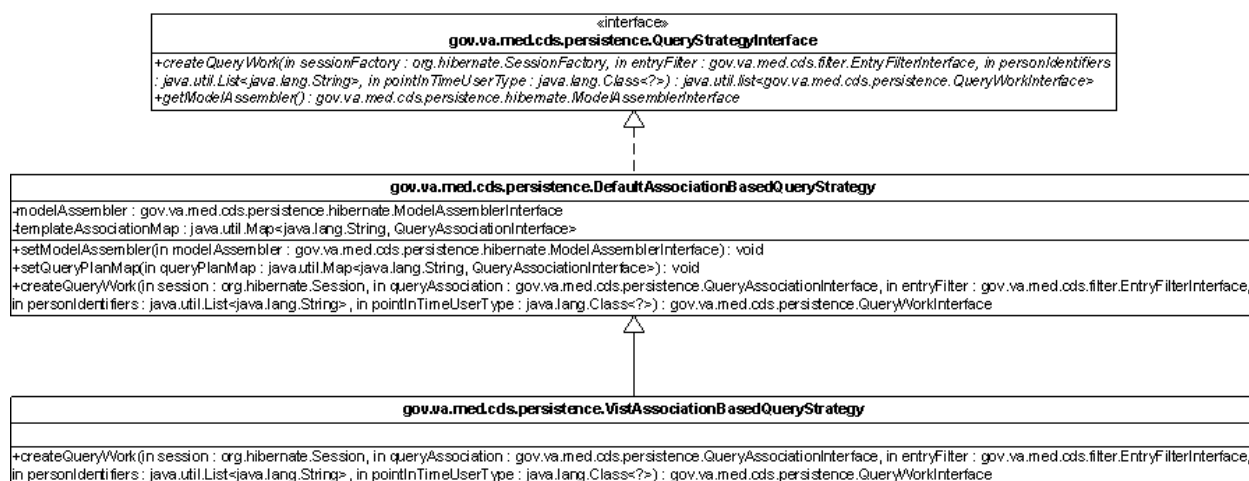
        <entry key="VWVitalsignsRead40010-VitalSignObservationEvent" value-
ref="singleQueryVitalsignsQueryStrategy" />
</util:map>

```

### 6.2.1.23.5 Interfaces, Classes, and Data Structures

The query strategy interfaces and classes are made up of one interface, QueryStrategyInterface, and two concrete classes DefaultAssociationBasedQueryStrategy and VistaAssociationBasedQueryStrategy (used for Vista reads) which is derived from the DefaultAssociationBasedQueryStrategy (used for HDR reads). The concrete classes include the text AssociationBased in their names because they are designed to require association mapping information as part of their configuration to help the model assemble components build the results returned by the multiple queries into a single logical model (XML document). The following figure displays the class diagram that describes the query strategy classes and interface with an example configuration of the TIU :

**Figure 43 Query Strategy Class Diagram**



### 6.2.1.23.6 Vista TIU Query Association Based Query Strategy Definition (Data Model)

This section lists the definition for the association based query.

```

<bean id="multiQueryTiuDocumentReadListQueryStrategy"
class="gov.va.med.cds.persistence.VistaAssociationBasedQueryStrategy">

    <property name="modelAssembler" ref="tiuModelAssembler" />

```

```

    <property name="queryPlanMap">
        <map>
            <entry key="clinicalDocumentEvent"> <!-- DomainEntryPoint -->
                <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
                    <property name="filterModelName"
value="clinicalDocumentEvent" />
                    <property name="associationParent"
value="clinicalDocumentEvents" />
                    <property name="associationName"
value="clinicalDocumentEvents" />
                    <property name="key" value="id" />
                </bean>
            </entry>
            <entry key="otherExpectedCosigners">
                <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
                    <property name="associationParent"
value="clinicalDocumentEvent" />
                    <property name="associationName"
value="otherExpectedCosigners" />
                    <property name="key" value="id" />
                    <property name="parentKey" value="tiuDocumentNumber" />
                    <property name="collapsible" value="true" />
                </bean>
            </entry>
            <entry key="otherCosigners">
                <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
                    <property name="associationParent"
value="clinicalDocumentEvent" />
                    <property name="associationName" value="otherCosigners" />
                    <property name="key" value="id" />
                    <property name="parentKey" value="tiuDocumentNumber" />
                    <property name="collapsible" value="true" />
                </bean>
            </entry>
            <entry key="images">
                <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">
                    <property name="associationParent"
value="clinicalDocumentEvent" />
                    <property name="associationName" value="images" />
                    <property name="key" value="id" />
                    <property name="parentKey" value="tiuDocumentNumber" />
                    <property name="collapsible" value="true" />
                </bean>
            </entry>
            <entry key="documentUpdateEvents">
                <bean class="gov.va.med.cds.persistence.QueryAssociationMetaData">

```

```

value="clinicalDocumentEvent" />
value="documentUpdateEvents" />

<property name="associationParent"
<property name="associationName"
<property name="key" value="id" />
<property name="parentKey" value="tiuDocumentName" />
<property name="collapsible" value="true" />

</bean>

</entry>

</map>

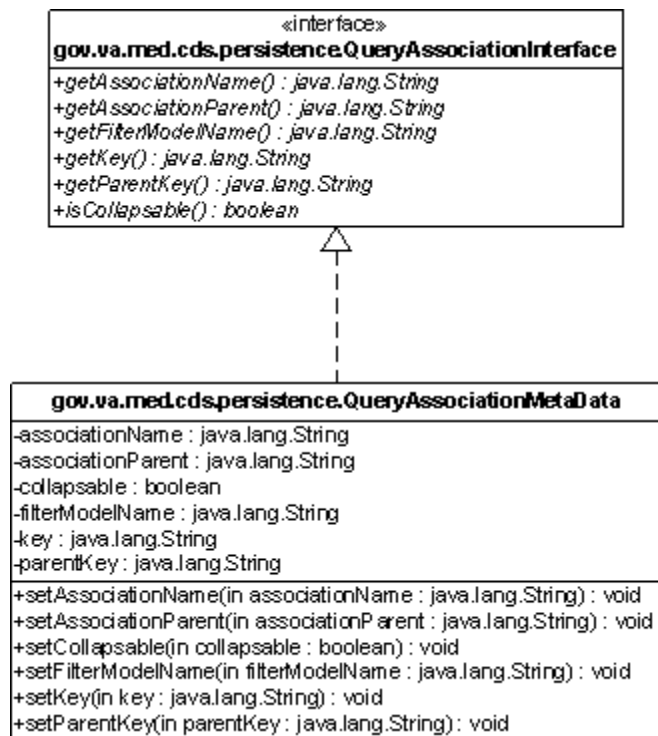
</property>

</bean>

```

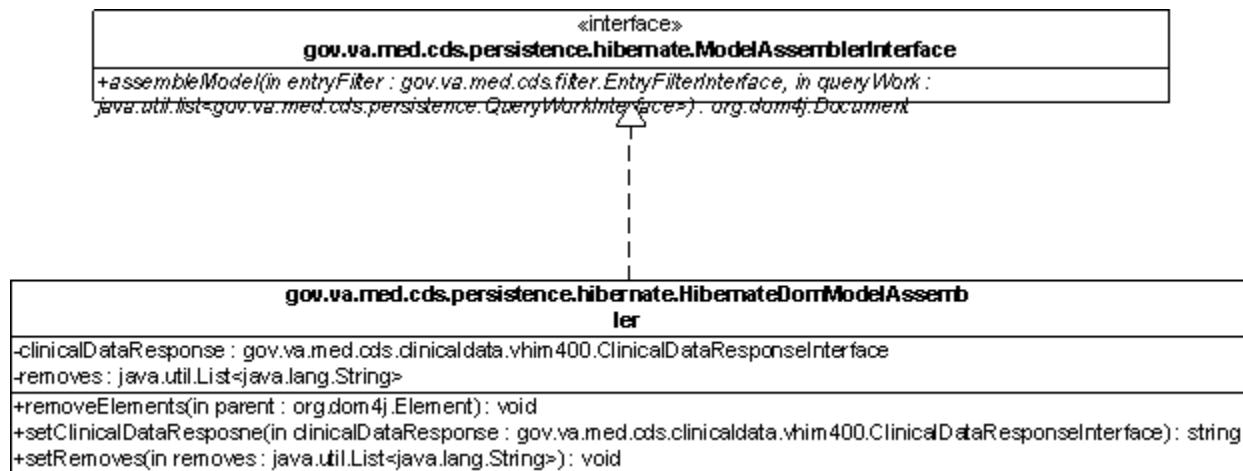
The query strategy implementation classes need two pieces of information in order to do their work: First produce a query plan (map) with the name for each entry to enable identification of the Hibernate named query to be used by the query work to read the data. The query association metadata helps the model assembler reconstruct a complete model after all queries complete execution. Second, select which model assembler to use to reconstruct the model. The following figure describes both of these Query Processor sub-system components:

**Figure 44 Query Association Class Diagram**





**Figure 45 Model Assembler Class Diagram**



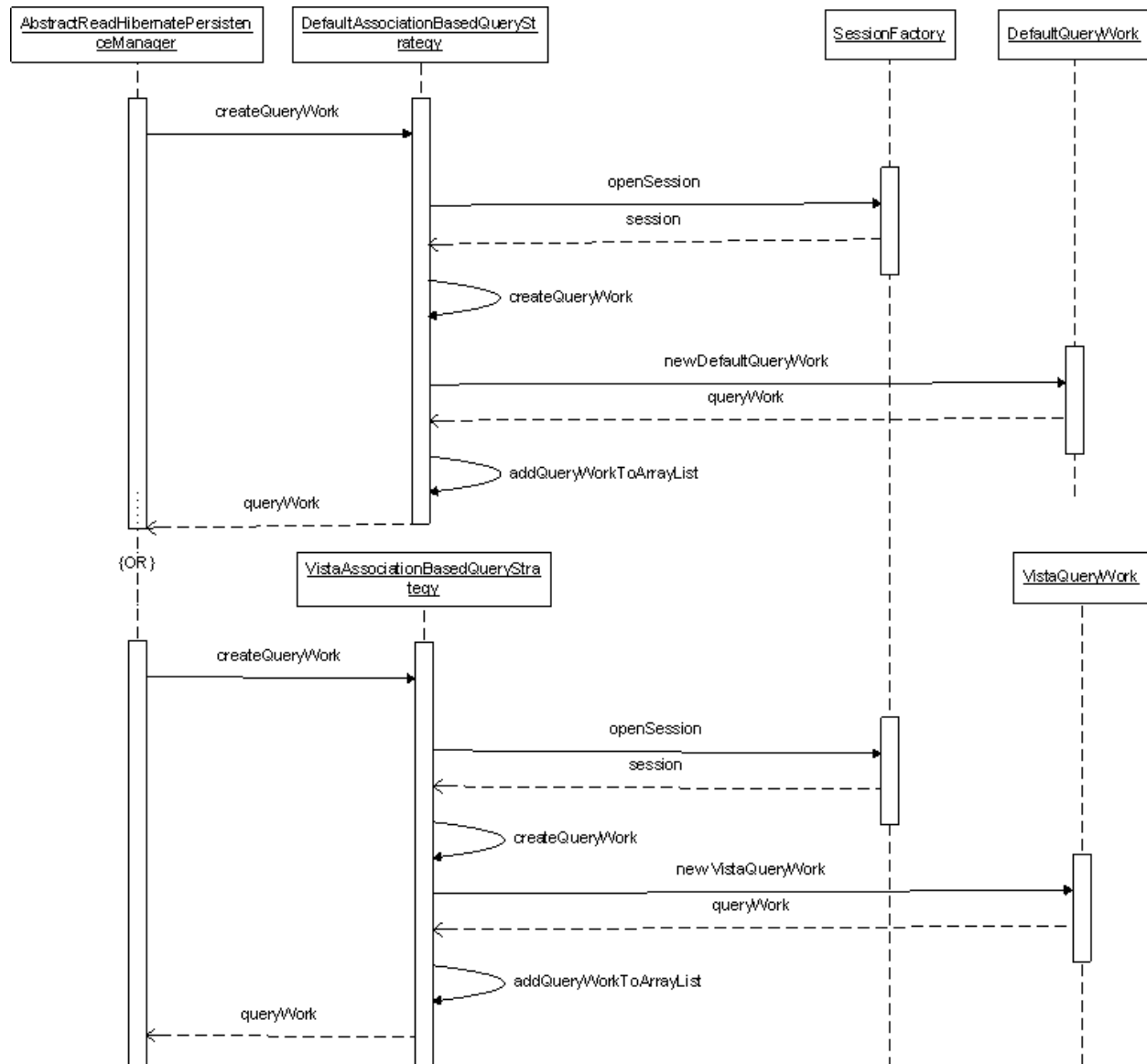
### Create Query Work Processing

The Hibernate read persistence manager identifies the query strategy to use for the read request. It then kicks-off the process of creating the set of query work necessary to fulfill the read request. If the query strategy associated with the template ID and entry point is a `DefaultAssociationBasedQueryStrategy`, `DefaultQueryWork` instances are created and used to fulfill the read request. If the query strategy associated with the template ID and entry point is a `VistaAssociationBasedQueryStrategy`, then `VistaQueryWork` instances are instantiated for use. The Hibernate read persistence manager uses the following steps to create query work:

1. The Hibernate Session Factory opens a new session to be used by the query work.
2. The Hibernate Session Factory instantiates the appropriate query work (either `DefaultQueryWork` or `VistaQueryWork`) depending upon the application configuration and initializes it with the Hibernate session, the query association, entry filter, person identifiers, and point in time user type necessary to fulfill the read operation.
3. The Hibernate Session Factory populates a list with the newly created query work.
4. The Hibernate Session Factory returns the list of query work instances to the client.

The query strategy takes the parameters and returns a newly created set of `QueryWork` instances, fully initialized and thread safe, ready to be used to fulfill the read request. The following figure below illustrates the sequence of events that transpire for both the default and Vista specific cases:

**Figure 46 Create Query Work Sequence Diagram**



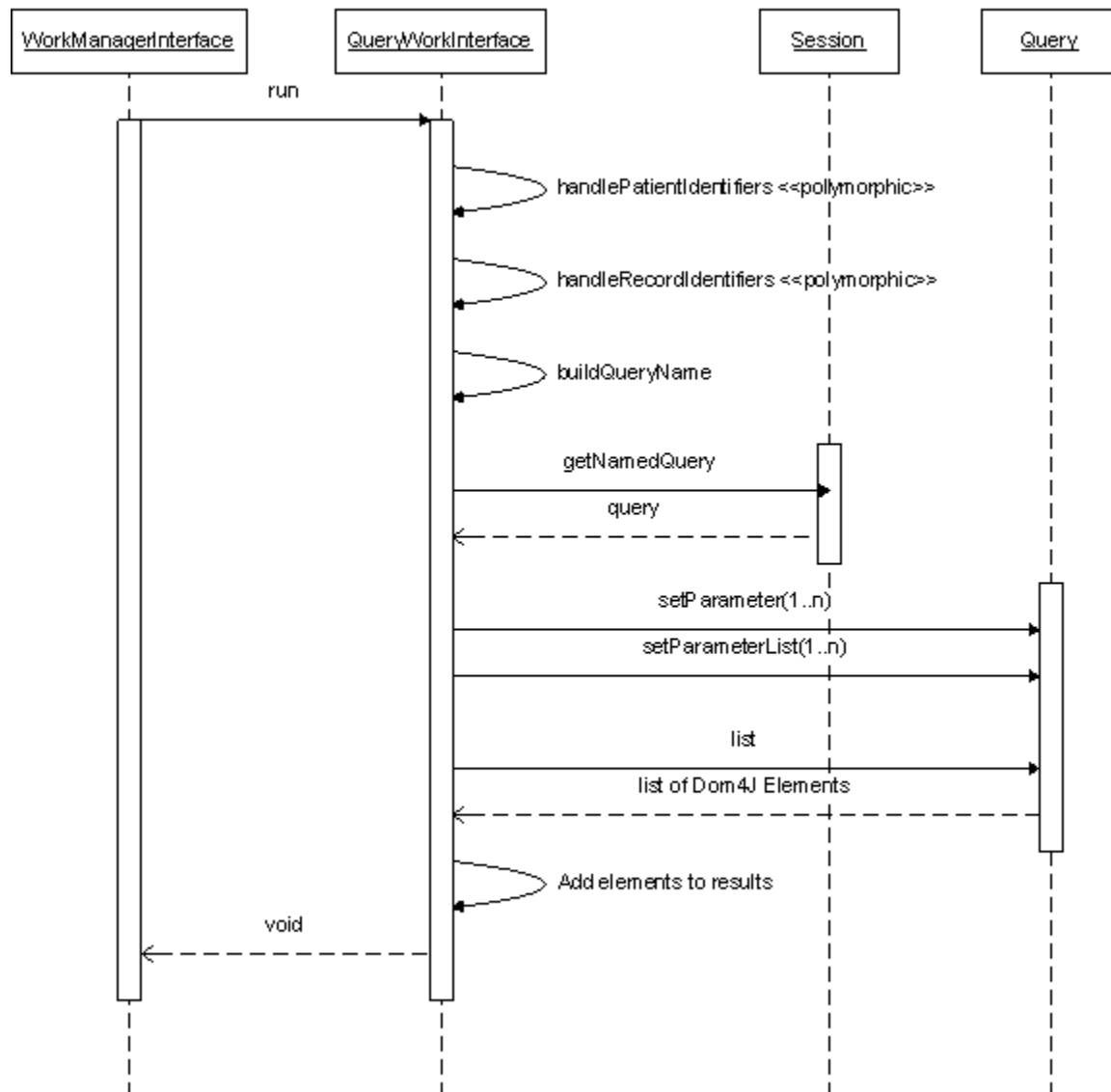
### Execute Query Work Processing

The Query Processor executes the work after identifying and initializing the query work with the necessary objects and data to fulfill its task. The Query Processor hands this work off to the work manager infrastructure of the application server to execute it concurrently using the pools of threads available for the work within the context of the enterprise application running on the application server. This work performs the following two steps:

1. Formats the patient identifiers parameter so that they can be used as query parameters
2. Formats the record identifiers parameter so that they

The following sequence diagram shows the flow of events, from the WorkManagers perspective:

**Figure 47 Execute Query Work Sequence Diagram**



#### 6.2.1.23.7 Model Assembler Processing

The Query Processor then assembles the complete logical model of the query results and returns them to the client.. Since the query for data can be broken up into multiple queries for performance reasons, the ModelAssembler classes in the Query Processing component go through the results of the queries and re-associate child records with parent records. . The ModelAssembler performs the following operations as part of preparing the query results for consumption by clients:

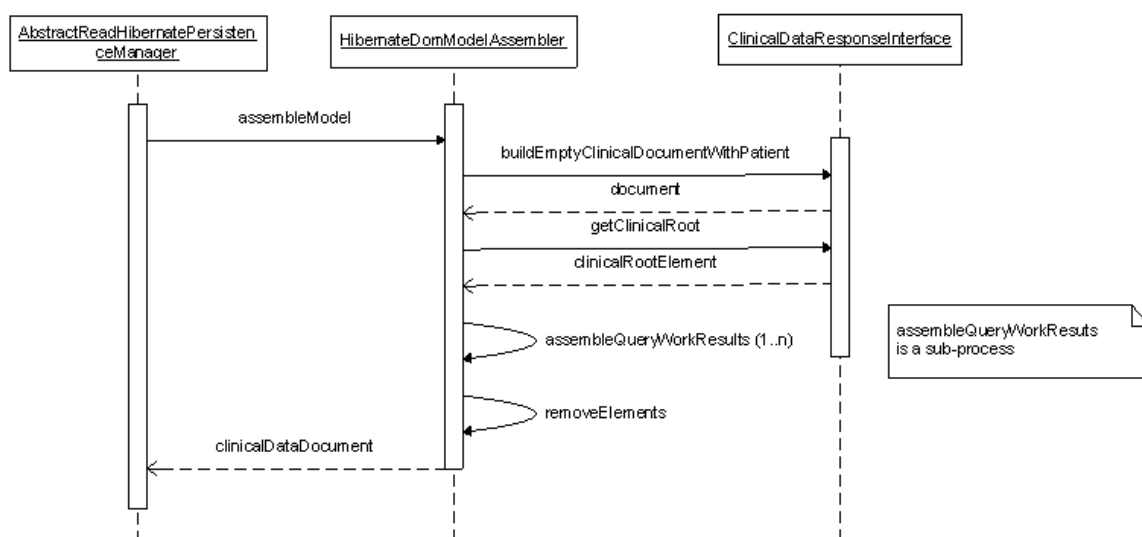
1. The ModelAssembler builds a version specific ClinicalData wrapper (XML) document to use as a container for the results.
2. The ModelAssembler gets the version specific clinical root element of the document to which all clinical results will be added.

3. The ModelAssembler iterates over the results, associating each of them with the parent document.
4. The ModelAssembler removes any elements from the result document required only by the model assembly process.
5. The ModelAssembler then performs a check to make sure that all elements from each QueryWork query were assembled. If any remain, an error is thrown indicating that something errored out.

The following sequence diagram illustrates this process:

**Figure 48 Model Assembler Sequence Diagram**

**Model Assembler  
Sequence Diagram**



#### 6.2.1.24 Concurrent Read and Persistence Locator

This section is applicable for read request processing.

##### 6.2.1.24.1 Concurrent Read and Persistence Locator Overview

CDS processes read data requests from client applications for a patient's clinical data. CDS delegates read requests pertaining to Outpatient Pharmacy and Lab to CDS 2.x, and handles Allergies, Vitals, and TIU read requests by fetching data concurrently from all relevant VistA systems (where data is available for the patient of interest) and from HDR II.

CDS will manage communication/connection with multiple VistA data sources using a Persistence Locator component (PersistenceLocatorInterface). The Persistence Locator component will look up appropriate Read Managers (classes implementing ReadPersistenceManagerInterface) that have information about connecting to different VistA systems. The Persistence Locator component will receive information about resolved patient identifiers from the IDM service using the "Identity Management Integration" component. The persistence locator component must be supplied with the resolved identifiers in order to locate the correct ReadPersistenceManagers for this call. Responses (in VIM XML format) from

multiple VistA systems will be aggregated so that the client application is provided with a single (standard) response that passes schema validation for a specific domain. All errors will be handled by CDS and clients will be notified of the same. Partial read responses will be supported by CDS.

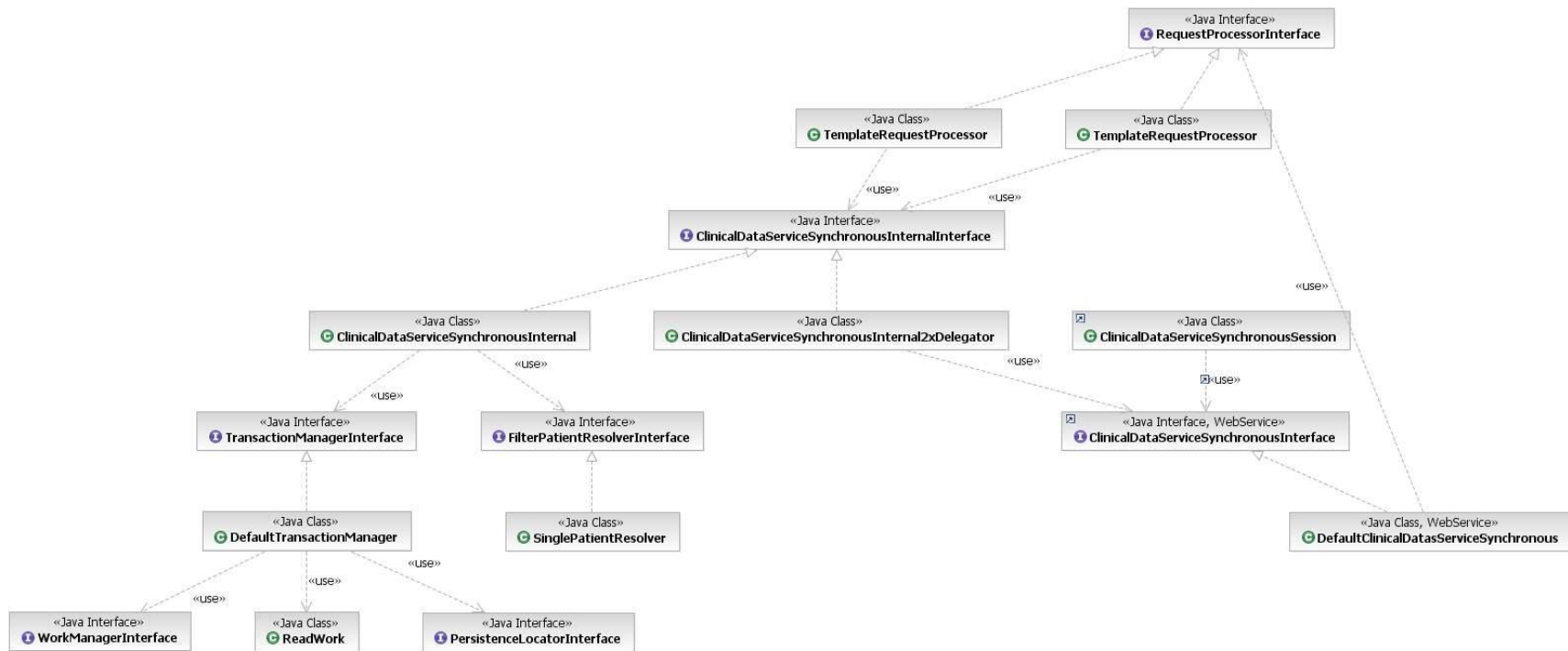
This document outlines the high level and low level design of the concurrent read strategy from VistA systems. This document also gives a brief overview of how this design aligns and fits into the architecture of CDS.

The scope of this document is limited to the architecture and design of concurrent read (from VistA systems) module. This document will identify unit test case scenarios as well as illustrate how this architecture can be fit into the CDS application structure.

#### 6.2.1.24.2 Concurrent Read and Persistence Locator Module Design

CDS must provide the ability to read data concurrently from multiple VistA systems (for the newer domains not currently supported by CDS 2.x). Data from VistA systems can be fetched by making concurrent queries to different VistA systems and then aggregating/assembling this data using an aggregator component. A high level class diagram for request response processing in CDS is shown below:

**Figure 49 File Request Response Framework RefactorClass Diagram**



In this diagram, the starting point for concurrent data processing starts at the TransactionManagerInterface level.

CDS must obtain information from the IDM service about resolved patient identifiers in all VistA systems where the patient of interest has any data available. CDS will then look up all relevant ReadPersistenceManagers at runtime using the PersistenceLocator component. This is done by the following method in the PersistenceLocatorInterface.

```
Collection<ReadPersistenceManagerInterface> getReadPersistenceManagers  
(EntryFilterInterface entryFilter, List<PersonIdentifierInterface> personIdentifiers );
```

A Spring Context File (transactionContext.xml) contains information about all ReadPersistenceManagers that are associated with the Persistence Locator. CDS then tries to connect and read data from VistA systems concurrently using the commonj.work.WorkManager. The read response (in the form of org.dom4j.Document) from each VistA system is then added to a list. This list is then provided to the response aggregator and sequencer components to aggregate and sequence all responses into a single consolidated response that is compliant with a VIM template for clients.

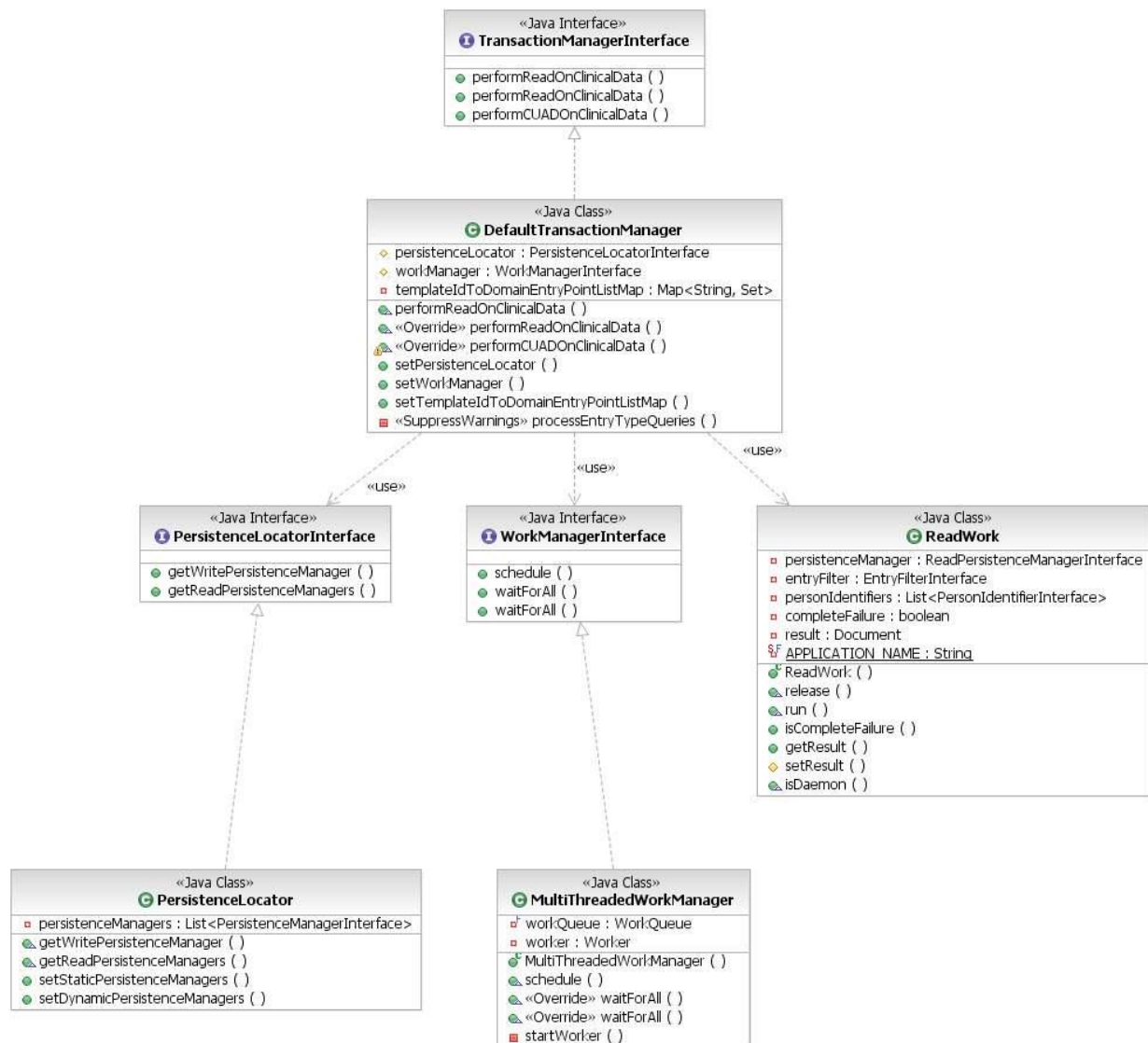
If the connection to a specific VistA system fails, a corresponding VIM XML will be generated by CDS with the error section detailing the cause of the error. If the IDM service fails, a fatal error will be returned back to the client. The concurrency and persistence locator modules only handles read requests. A high level diagram of the CDS architecture is shown below. This diagram is included to illustrate how concurrent read strategy will work into the overall structure of CDS.

#### **6.2.1.24.3 Concurrent Read and Persistence Locator Concurrent Processing**

This section is applicable for read request processing.

DefaultTransactionManager processes READ calls concurrently.

**Figure 50 File Concurrent Read Class Diagram**

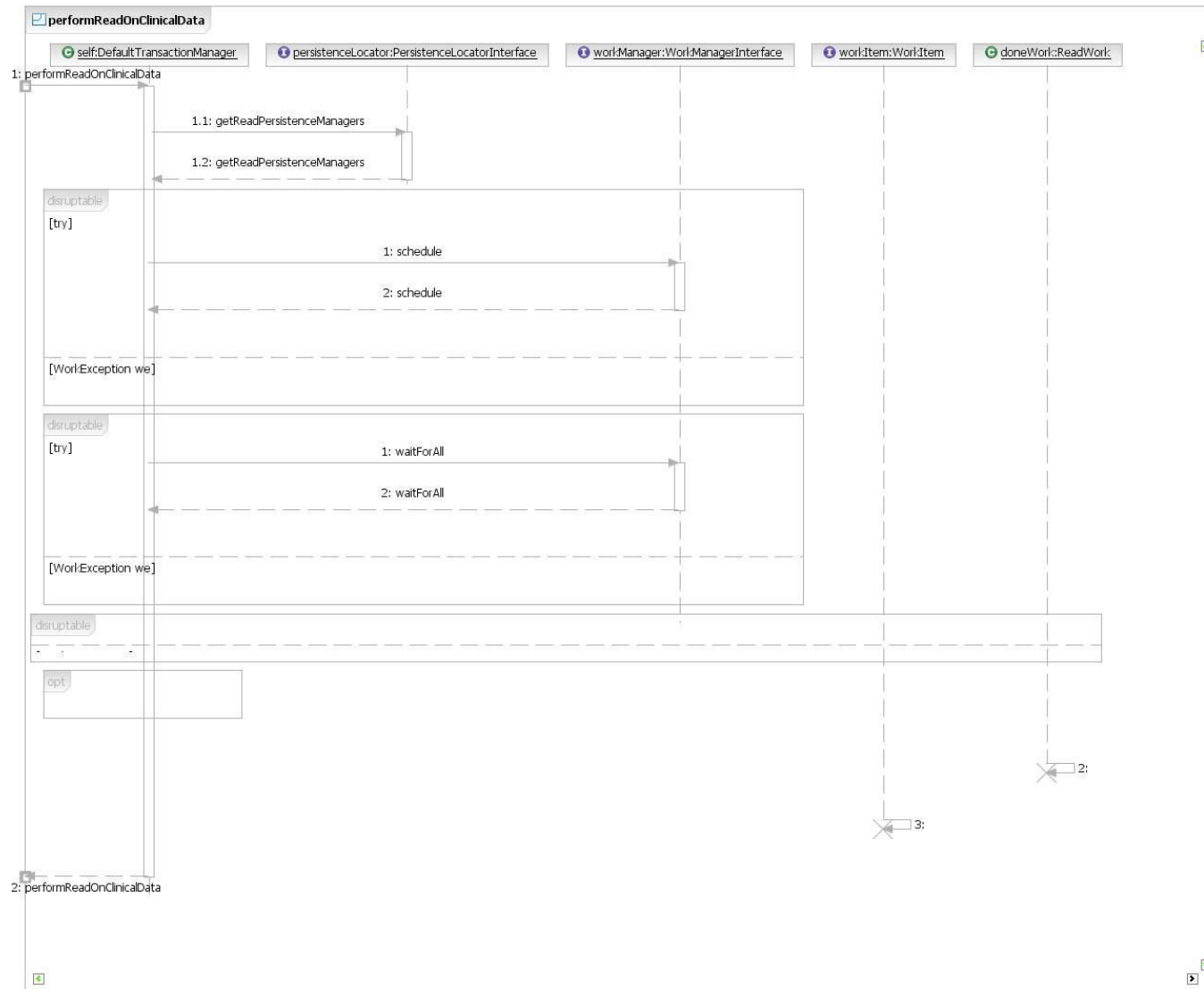


Read processing starts with the **DefaultTransactionManager** looking up a set of Domain Entry Points from the Cds Filter. The Persistence Manager to be used in the Read call is obtained from a Persistence Locator. Read calls are scheduled as **ReadWork** objects and a Read Persistence Manager, Filter, and Person Identifier are passed along during instantiation to be used to perform the actual DataSource connection and retrieval of patient clinical data. The **DefaultTransactionManager** will delegate concurrent processing of each **ReadWork** to a **WorkManager**. The **WorkManager** used is a Spring implementation class that extends the commonj framework. The **WorkManager** schedules **ReadWork** work objects to run concurrently within the commonj framework. The entire Read process waits for each scheduled process to complete before the set of Read results obtain from each site is passed up to the Response Generation area of the application where all results will be aggregated into a single response document.





**Figure 52 File Read Concurrent Work Manager Sequence Diagram**



### **6.2.1.25 Audit Logging**

#### **6.2.1.25.1 Audit Logging Overview**

This document outlines the high and low level design process of the CDS read and write request audit. This document also gives an overview of how this design aligns and fits into the current CDS architecture.

The scope of this document is limited to the architecture and design of the VIM read and write request audit module. This document also identifies the list of test scenarios that ensure that all requirements for the functional component have been accommodated for in the design and implementation.

CDS supports VIM structured read and write requests. The VIM clients send the read/write requests as XML documents structured in accordance with VIM templates. CDS returns VIM compliant messages to the clients carrying the response to the write transaction or the clinical data results for read requests as well as any errors related to either type of request. All the read and write request details are audited into HDR data base. The scope of CDS is limited to auditing of VIM read and write requests only.

#### **6.2.1.25.2 Audit Logging Module Design**

CDS audits both VIM read and write request details that are sent by CDS VIM clients.

CDS processes the VIM Allergy/Vitals/TIU read calls by fetching data from both, VistA sites and HDR; and the rest of the VIM Outpatient Medication Pharmacy/Lab read calls are redirected to CDS 2.1.x to fetch data from HDR and Hx data bases. All VIM read requests details are audited asynchronously in CDS.

If gathering of the clinical information fails for some reason, then an error is logged to the CDS Application Log, an error response is returned back to the client and also read request details are audited. If gathering of the clinical information succeeds, and for some reason auditing of the read request fails, then the read results are returned back to the client and read audit failure reason is logged to CDS Application Log.

CDS processes VIM Allergy, Vitals, VIM Outpatient Medication Pharmacy, Lab write requests. All VIM write requests are persisted into HDR data base. VIM write requests are audited synchronously in CDS 3.7.

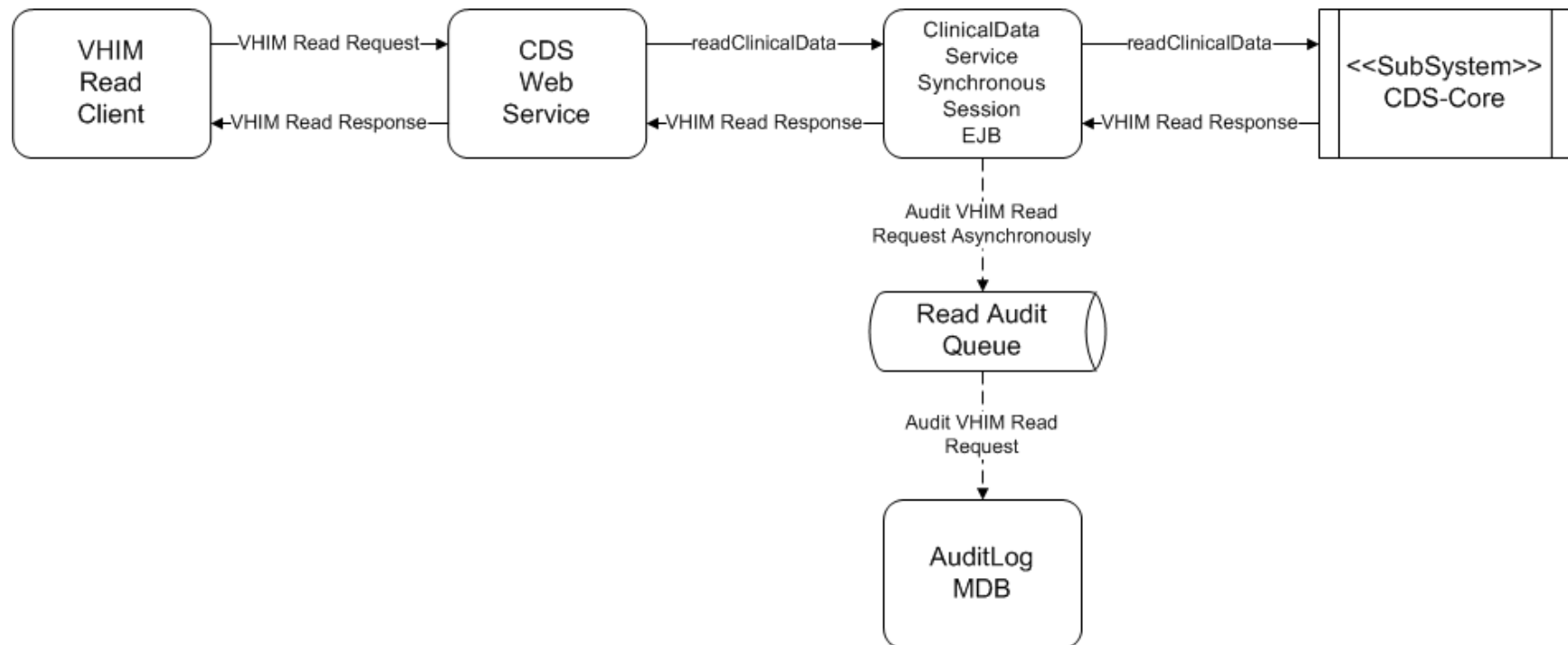
If errors occur while auditing the VIM write requests that results in the failure of the auditing logic in the Clinical Data Service Synchronous component, then the transaction fails and the remainder of the logic responsible for persisting the clinical information to the HDR data base is skipped (i.e. clinical information is not stored).

If VIM write request auditing logic completes successfully but the persistence of the clinical information to HDR fails, then the audit record remains and an error is logged to the CDS Application Log indicating the issue with the persistence of the clinical information in HDR.

The following figures diagram the VIM read and write requests.

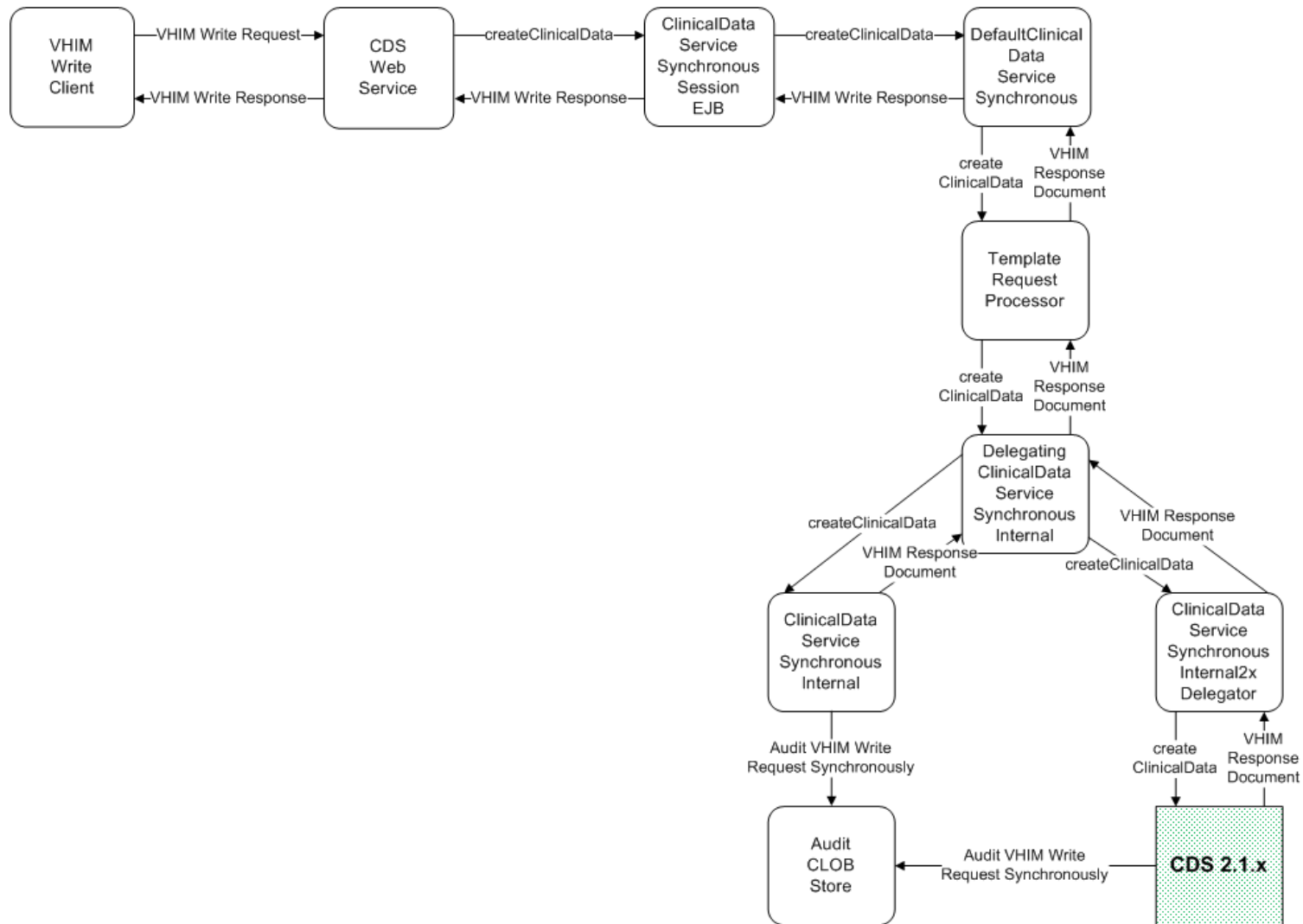
**Figure 53 VIM Read Request Audit Overview**

### VHIM Read Request Audit Overview



**Figure 54 VIM Write Request Audit Overview**

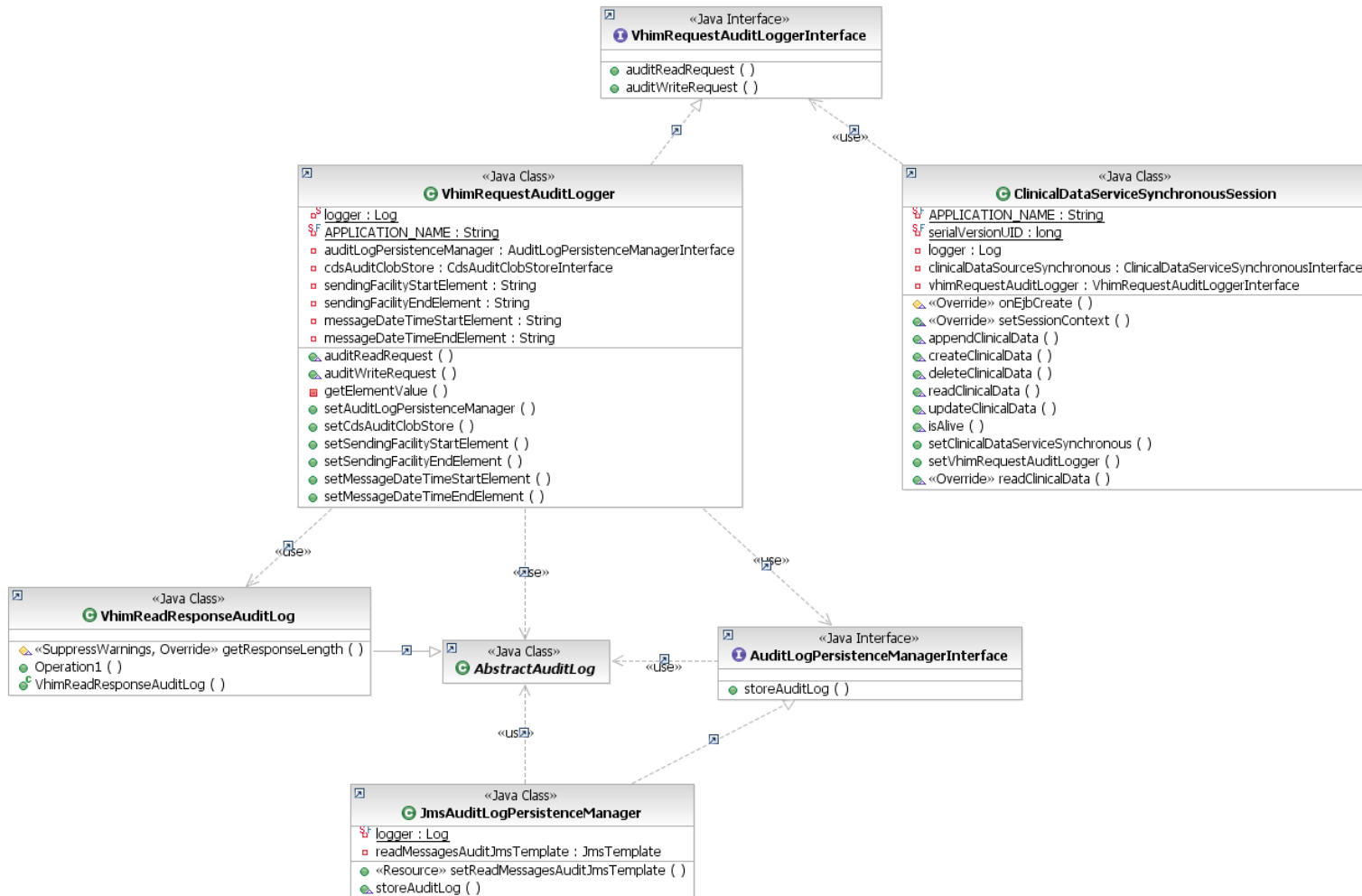
### VHIM Write Request Audit Overview



### 6.2.1.25.3 Audit Logging Processing

The following class diagram refers to the auditing of VIM read request:

Figure 55 VIM Read Request Class Diagram



A new POJO has been introduced `VhimRequestAuditLogger` (VIM Audit POJO) to audit VIM read request details.

`VhimRequestAuditLogger` uses `JmsAuditLogPersistenceManager` to audit the read request details asynchronously into `CDS_AUDIT_LOG_3` table of `LOGGER` schema in HDRII data base. Pathways data is persisted in `PATHWAYS_AUDUIT_LOG` table.

`ClinicalDataServiceSynchronousSession` is a stateless session bean (CDS EJB) that is responsible for processing VIM read requests synchronously, which implements `ClinicalDataServiceInterface`. The CDS EJB uses the `DefaultClinicalDatasServiceSynchronous` POJO to validate and process incoming VIM read messages and the `VhimRequestAuditLogger` class to audit VIM read requests. Processing of VIM read request is out of scope for this sub-system.

If `DefaultClinicalDatasServiceSynchronous` POJO fails in gathering of clinical information, then an error is logged to `CDS_APPLICATION_LOG /PATHWAYS_APPLICATION_LOG` table, and an error response is returned back to the client and also `JmsAuditLogPersistenceManager` audits the VIM read request details asynchronously into `CDS_AUDIT_LOG_3/PATHWAYS_AUDUIT_LOG` table.

If `JmsAuditLogPersistenceManager` fails to audit the VIM read request and gathered the clinical information successfully from `DefaultClinicalDatasServiceSynchronous` POJO, then the read results are returned back to the client and read audit failure reason is logged into `CDS_APPLICATION_LOG` table.

The following class diagram refers to the auditing of VIM write request:

Figure 56 VIM Write Request Class Diagram





A new POJO has been introduced `VhimRequestAuditLogger` (VIM Audit POJO) to audit VIM write request details.

`ClinicalDataServiceSynchronousSession` is a stateless session bean (CDS EJB) that is responsible for processing VIM write requests asynchronously, which implements `ClinicalDataServiceInterface`. The CDS EJB uses the `DefaultClinicalDatasServiceSynchronous` POJO to validate and process VIM write request. Processing of VIM read request is out of scope for this sub-system.

`ClinicalDataServiceSynchronousInternal` class audits the VIM write request using `VhimRequestAuditLogger`.

`VhimRequestAuditLogger` uses `PersistAuditStore` to audit the VIM write request details synchronously into `AUDITCLOBSTR` table of `MONITOR` schema in `HDRII` data base.

If `VhimRequestAuditLogger` POJO fails to audit the VIM write request, then an error is logged to `CDSAPPLICATIONLOG` table, and an error response is returned back and VIM write request is pushed into the error queue.

If `VhimRequestAuditLogger` POJO audits the VIM write request successfully, and `DefaultClinicalDatasServiceSynchronous` POJO fails to persist the write request into `HDRII` data base, then also an error response is returned back and VIM write request is pushed into the error queue.

#### 6.2.1.25.4 Audit Logging Local Data Structures

The `VhimRequestAuditLogger` class gets added by Spring framework with the audit log managers of read and write requests:

```
<bean name="vhimRequestAuditLogger" class="gov.va.med.cds.audit.VhimRequestAuditLogger">
    <property name="auditLogPersistenceManager" ref="auditLogPersistenceManager" />
    <property name="cdsAuditClobStore" ref="cdsAuditClobStore" />
</bean>
```

`JmsAuditLogPersistenceManager` class Spring framework configuration:

```
<bean id="jmsConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jms/cds/LoggerConnectionFactory"/>
</bean>
<bean id="readAuditQueue" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jms/cds/ReadAuditQueue"/>
</bean>
<bean id="readMessagesAuditJmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
    <property name="receiveTimeout" value="10000"/>
    <property name="defaultDestination" ref="readAuditQueue"/>
</bean>
<bean id="auditLogPersistenceManager"
class="gov.va.med.cds.audit.persistence.jms.JmsAuditLogPersistenceManager">
    <property name="readMessagesAuditJmsTemplate" ref="readMessagesAuditJmsTemplate"/>
```

```
</bean>
```

### HibernateAuditLogPersistenceManager class Spring framework configuration:

```
<bean id="auditLogSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="hdrDataSource"/>
    <property name="mappingResources">
        <list>
            <value>gov/va/med/cds/audit/persistence/hibernate/LegacyReadAuditLog.hbm.xml</value>
            <value>gov/va/med/cds/audit/persistence/hibernate/VhimReadAuditLog.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</prop>
            <prop key="hibernate.show_sql">false</prop>
        </props>
    </property>
</bean>

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="auditLogSessionFactory"/>
</bean>

<bean id="hibernateAuditLogPersistenceManager"
class="gov.va.med.cds.audit.persistence.hibernate.HibernateAuditLogPersistenceManager">
    <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>
```

### PersistAuditStore class Spring framework configuration:

```
<bean id="cdsAuditClobStore"
class="gov.va.med.cds.audit.persistence.hibernate.PersistAuditStore">
    <property name="hibernateTemplate" ref="auditLogHibernateTemplate" />
</bean>

<bean id="auditLogHibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="auditLogSessionFactory"/>
</bean>

<bean id="auditLogSessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="hdrDataSource" />
    <property name="mappingResources">
        <list>
            <value>gov/va/med/cds/audit/persistence/hibernate/CdsAuditClobStore.hbm.xml</value>
        </list>
    </property>
```

```

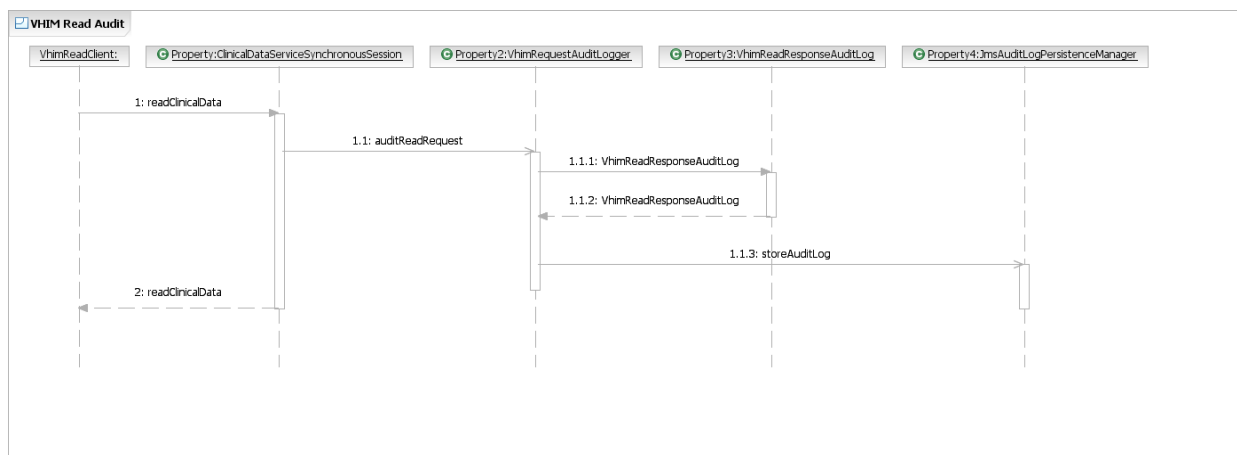
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.Oracle10gDialect</prop>
            <prop key="hibernate.show_sql">false</prop>
        </props>
    </property>
</bean>

```

### 6.2.1.25.5 Audit Logging Module/Other Diagrams

The following component design discussion refers to the VIM read request audit process sequence diagram:

**Figure 57 VIM Read Request Sequence Diagram**

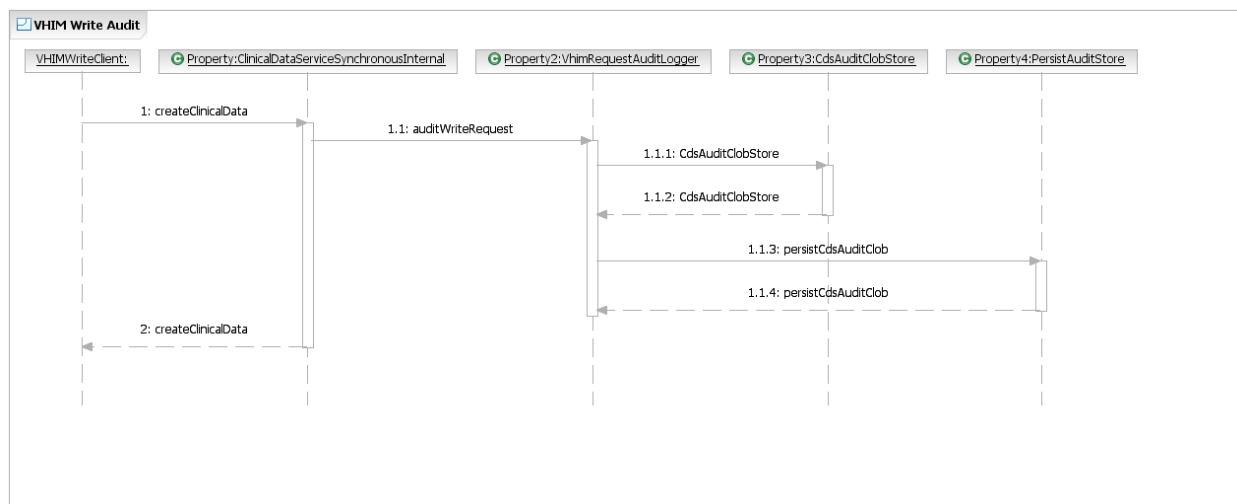


1. The ClinicalDataServiceSynchronousSession EJB invokes the readClinicalData() method on the DefaultClinicalDataServiceSynchronous POJO to perform the VIM read request; and will get back the read response XML.
2. ClinicalDataServiceSynchronousSession invokes auditReadRequest() method on VhimRequestAuditLogger POJO to audit the read request details.
3. VhimRequestAuditLogger POJO creates an instance of VhimReadResponseAuditLog and passes it on to JmsAuditLog persistenceManager to persist the read request details asynchronously into CDS\_AUDITLOG3/PATHWAYS\_AUDITLOG table of LOGGER schema in HDR data base.
4. If JmsAuditLogPersistenceManager fails to audit the VIM read request for some reason and gathers the clinical information successfully from DefaultClinicalDataServiceSynchronous POJO, then the read results are returned back to the client and the read audit failure reason is logged into CDS\_APPLICATIONLOG/PATHWAYS\_APPLICATIONLOG table.
5. If there is an error while gathering clinical data results from HDR:
  - a. An error response is returned back to the client

- b. Error details are logged into CDSAPPLICATIONLOG/  
PATHWAYSAPPLICATIONLOG table
5. VIM read request details are audited into  
CDSAUDITLOG 3/PATHWAYSAUDITLOG table
6. If the read result is returned successfully from DefaultClinicalDataServiceSynchronous  
POJO and there is an exception thrown by JmsAuditLogPersistenceManager while  
auditing the VIM read request
  - c. VIM read result is returned back to the client
  - d. VIM read audit failure reason is logged into CDSAPPLICATIONLOG/  
PATHWAYSAPPLICATIONLOG table
7. If there is no error while gathering clinical data results from HDR and the read result is  
returned successfully
  - e. VIM read response is returned back to the client

The following component design discussion refers to the VIM write request audit process  
sequence diagram:

**Figure 58 VIM Write Request Sequence Diagram**



8. The `ClinicalDataServiceSynchronousInternal` class invokes `createClinicalData()` method on `ClinicalDataServiceSynchronousInternal`.
9. The `ClinicalDataServiceSynchronousInternal` invokes `doCUAD()` method to process VIM write request.
10. The `ClinicalDataServiceSynchronousInternal` invokes `auditWriteRequest()` method on `VhimRequestAuditLogger` POJO to audit the VIM write request details.
11. `VhimRequestAuditLogger` POJO creates an instance of `CdsAuditClobStore` and passes it on to `PersistAuditStore` to persist the VIM write request details synchronously into `AUDITCLOBSTR` table of `MONITOR` schema in HDRII data base.
12. If the VIM write audit is not successful:
  - f. Error details are logged into `CDSAPPLICATIONLOG` table
  - g. No data is persisted to HDR
  - h. VIM write request is pushed into error queue
13. If the VIM write audit is successful, and there is an error while persisting the VIM write request into HDR data base
  - i. Error details are logged into `CDSAPPLICATIONLOG` table
  - j. VIM Write request is audited into `AUDITCLOBSTR`
  - k. VIM Write request is not persisted to clinical domain tables of HDR
14. If the VIM write audit is successful and there is no error while persisting VIM write request
  - l. VIM write response is returned back to the client.

## **6.2.1.26 Exception Handling**

### **6.2.1.26.1 Exception Handling Overview**

This section presents an overview of the exception handling of read and write requests processed within CDS. Exceptions are caught, logged, and handled. In some circumstances exceptions are thrown back up the chain until they are eventually handled and logged at a higher level – or in one location within a CDS application layer. The Exception framework is designed so that a set of Custom loggers can be pre-configured to log exceptions to any number of locations. The framework is also designed and configured to provide at least one default or 'Guaranteed' logger. Detailed information about an exception and metadata about the request used in constructing the XML response sent back to the client application.

#### **6.2.1.26.2 Exception Handling Module Design**

When exceptions occur during request processing they are generally passed to an exception handler. The `ExceptionHandler` class is a static utility that is configured to work with one or more loggers. The application can specify a set of custom loggers that can log errors in various ways from each other. CDS has defined a Default logger that will log exceptions asynchronously to the `CDSAPPLICATIONLOG / PATHWAYSAPPLICATIONLOG` table of the `LOGGER` schema in HDR II.

Exceptions are logged by the `DefaultLogger` using JMS. The `DefaultLogger` is a `Log4j` logger and is configured to use a JMS appender that retrieves a connection via a `WebLogic JMS ConnectionFactory`. The log message, containing information about the exception, is placed on a `WebLogic` queue. A J2EE Message Driven Bean (MDB) is registered to listen for messages deposited on the queue. Upon deposit of the log message, the bean will consume the message and persist it to the Oracle table and schema mentioned above. A more detailed discussion of the process or design of log message persistence will not be covered in this section.

The `ExceptionHandler` is also configured to use a VIM version specific `ErrorSectionHelper` that will construct an error section containing the error code and message of the exception being handled and return it within a `!Dom4j Document` and eventually returned to the client application.

#### **6.2.1.26.3 Exception Handling Processing**

There are several places within the CDS application where exceptions are caught, handled and logged:

The first is at the highest level, within the Request processing component — exceptions caught at this level may be CDS business exceptions, such as a `Validation` exception or they may be unknown and unexpected runtime exceptions that do not have a corresponding error code associated with them — except for the standard 'Unknown' error code.

The second level where exceptions may be caught, handled and logged is within a `Delegating` layer that determines which path the request will take, if it will be delegated to an external CDS 2.x or if it will be processed internally by CDS.

The third level that exceptions are handled is within the Transaction management layer where exceptions may happen while each `Persistent manager` invoked.

The final and lowest level exceptions are handled is within the Persistence layer — while accessing the `Hibernate Framework`.

Within each layer the component processing the request usually has the ability to categorize the exception and wrap it within a CDS Business exception and throw the exception back up to be handled and logged in one location within the layer. In this way specific error codes can be given to known exceptions and more meaningful client specific messages constructed and returned to the client applications.

However, there may be times that runtime exceptions happen that are unexpected and basically catastrophic. These exceptions are eventually handled and logged, but they are not wrapped within a CDS Business exception but are thrown up to be handled at a higher level. At the highest level, an exception that was not previously wrapped within a CDS Business exception will be caught and wrapped in an 'UNKOWN' exception – a message for the exception will be constructed and information about the exception will be included in the XML response.

See the Repositories TIU, Allergies, Vitals Read from VistA Business Capability Interface Control Document –Appendix B – Data Elements , section B.6 Errors Messages by CDS for a list of errors and error codes.





At each level of processing mentioned above an exception will be caught and given to the ExceptionHandler to handle or process the exception. When the ExceptionHandler receives an exception it will first check if it has received a !Dom4j Document. If it has not, it will retrieve a ClinicalDataResponseInterface from the TemplateHelperInterface with which it was configured. The ClinicalDataResponseInterface will be used to construct an empty Clinical data document (a specific document structure containing high level elements). The templateId parameter, if received, indicates the response template or data content the client application wanted to receive from their request. If the ExceptionHandler has received a templateId parameter it will set the templateId element of the Clinical data document. If it did not receive a templateId it will set the templateId of the document to the default response templateId – ‘CdsResponse400’. The ExceptionHandler will do the same with the requestId parameter – if the requestId parameter is empty it will be set to – ‘Request Id Not Set’.

Next the ExceptionHandler will check if the type of exception is a CDS Business exception (AbstractCdsBaseException) and if not it will construct a wrapper for the exception with a specific ErrorCodeEnum that defines a readable client exception message that is return in the Clinical data document.

The exception’s message is then given to an ExceptionUtil object which will format or build a readable client message.

After a readable message is constructed the ExceptionHandler obtains a List of ExceptionLoggerInterfaces from the exception’s ErrorCodeEnum. The list of loggers is iterated over and their log method is called. The loggers are given the exception, the LoggerSeverity (obtained from the ErrorCodeEnum), the readable client message, the template Id, the requestId and the applicationName (which is always ‘CDS’).

There may or may not be custom loggers configured – currently CDS is not configured without Custom loggers. The ExceptionHandler will then obtain the DefaultLogger and call it’s log method which will log an exception message to HDR II asynchronously via JMS – discussed briefly above.

Next an ExceptionInfo object will be constructed which is basically a data object containing properties used in constructing the ErrorSection !Dom4j Element. The ExceptionInfo properties are the readable client message, the ErrorCodeEnum, the CdsException, the exception’s message and the LoggerSeverity.

After the ExceptionInfo is constructed the ErrorSectionHelperInterface is obtained from the TemplateHelperInterface the ExceptionHandler was configured with at startup. The ErrorSectionHelperInterface is asked to build an Error Section and is given the Clinical data document, the ExceptionInfo and the !requestId.

The ErrorSectionHelper will construct an Error Section element and populate it from the ExceptionInfo object and then set the Error Section element on the Clinical data document. An example of an Error Section element is contained in the ‘Local Data Structures’ section below.

#### **6.2.1.26.4 Exception Handling Local Data Structures**

The following is the structure of an ErrorSection as defined in the XSD:

```
<!-- ~~~~~ -->
<!-- Error <xsdGen> -->
```

```

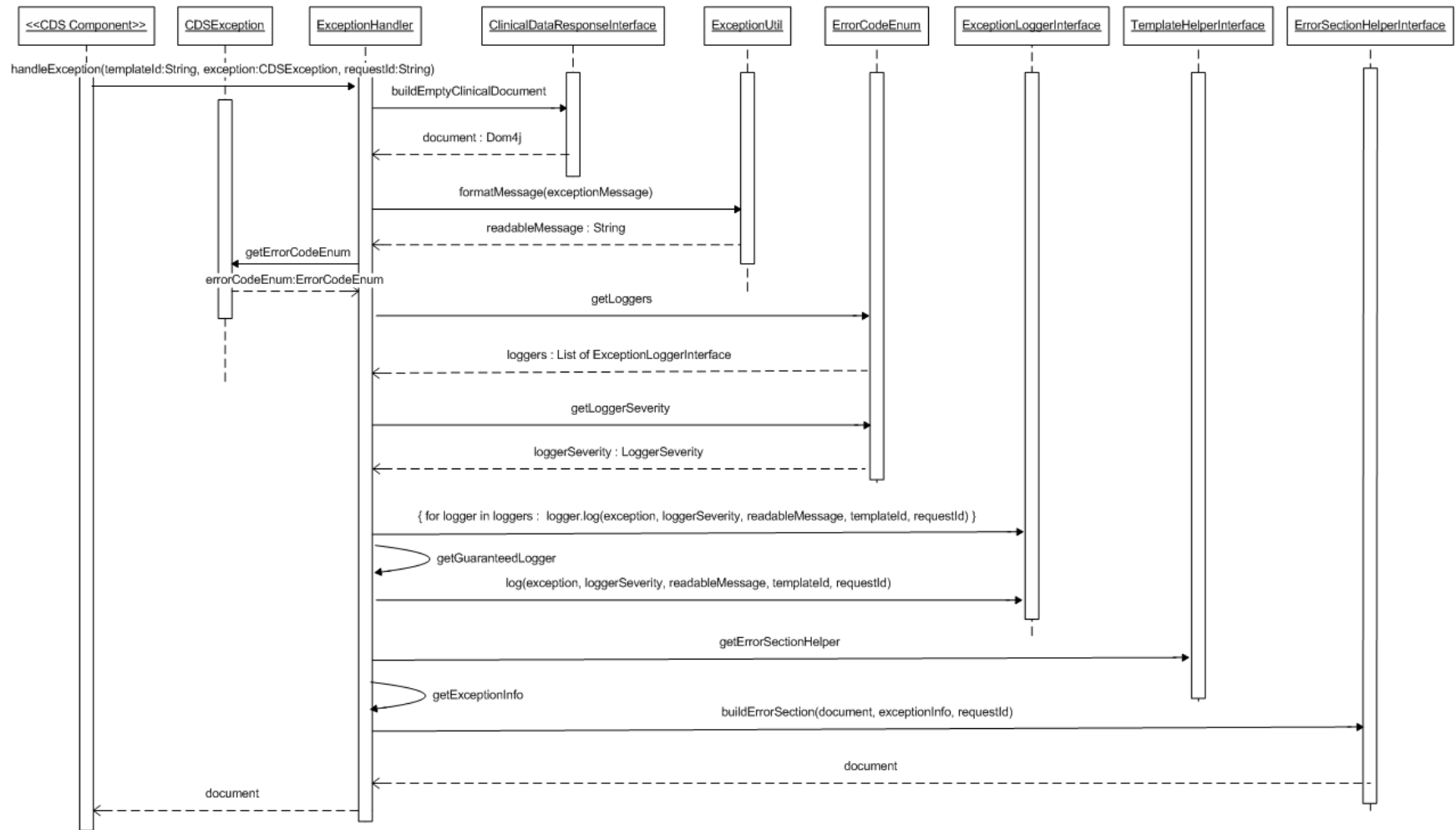
<!-- ~~~~~ -->
<xsd:complexType name="Error">
  <xsd:sequence>
    <xsd:element name="errorId" type="xsd:string" minOccurs="0"/>
    <xsd:element name="exception" type="xsd:string" minOccurs="0"/>
    <xsd:element name="exceptionMessage" type="xsd:string" minOccurs="0"/>
    <xsd:element name="errorCode" type="xsd:string" minOccurs="0"/>
    <xsd:element name="displayMessage" type="xsd:string" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<!-- ~~~~~ -->
<!-- ErrorSection <<xsdGen>> -->
<!-- ~~~~~ -->
<xsd:complexType name="ErrorSection">
  <xsd:sequence>
    <xsd:element name="errors" type="clinicaldata:Error" minOccurs="0"
maxOccurs="unbounded"/>
    <xsd:element name="fatalErrors" type="clinicaldata:Error" minOccurs="0"
maxOccurs="unbounded"/>
    <xsd:element name="warnings" type="clinicaldata:Error" minOccurs="0"
maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>

```

### 6.2.1.26.5 Exception Handling Module/Other Diagrams

The following is a Sequence Diagram – illustrating the processing steps in exception handling, discussed above.

**Figure 60 CDS Exception Handling Sequence Diagram**



## 6.2.1.27 Application Logging Subsystem

### 6.2.1.27.1 Application Logging Overview

The identification of errors and exception occurs throughout the CDS read/write request processing; all these exception messages are logged into the CDS application log. CDS application log stores the exception/error messages in HDR data base asynchronously.

When an exception occurs, CDS components push the exception/error messages into logger JMS queue (LoggerQueue). The LoggerQueue is consumed by the Logger Message Driven Bean and the exception/error messages are persisted into HDR data base asynchronously.

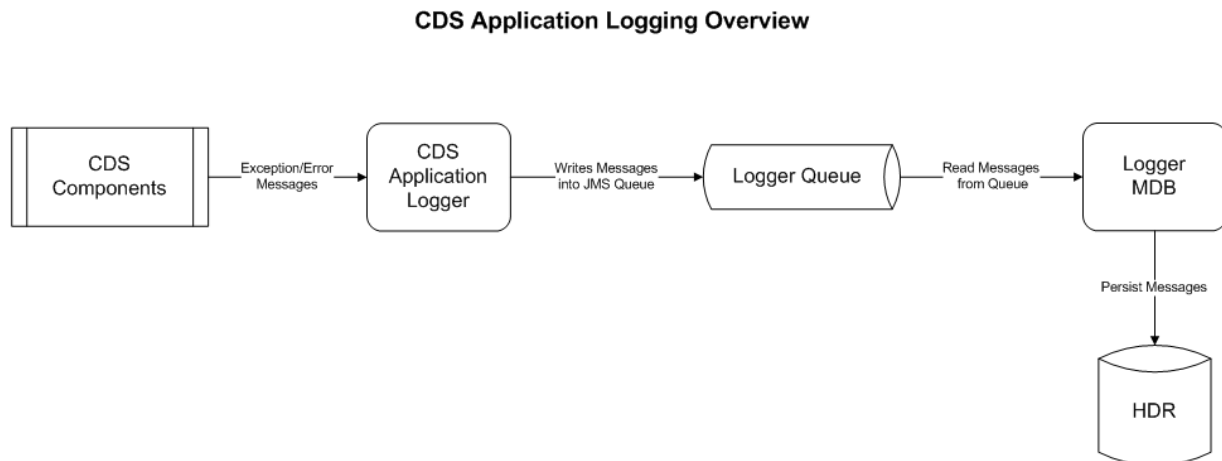
### 6.2.1.27.2 Application Logging Module Design

The CDS application uses Apache commons-logging with logging toolkit Log4J to log exception messages. CDS component classes use the Log interface that is intended to be both light-weight and an independent abstraction of logging toolkits like Log4J. CDSN 3.x uses ConsoleAppender and JMSAppender, which logs exception messages to the Weblogic .out file and writes messages to the LoggerQueue respectively. The cdsn-ejb component constitutes the log4j.xml, which includes the required appender and logger configurations.

An exception/error message details are pushed into the LoggerQueue by the CDS exception handler. The functionality of CDS exception handler is out of the scope of the current subsystem. Logger MDB, which is listening on LoggerQueue continues the CDS application exception/error message into the HDR data base using Hibernate framework.

The following figure diagrams the CDS application logging.

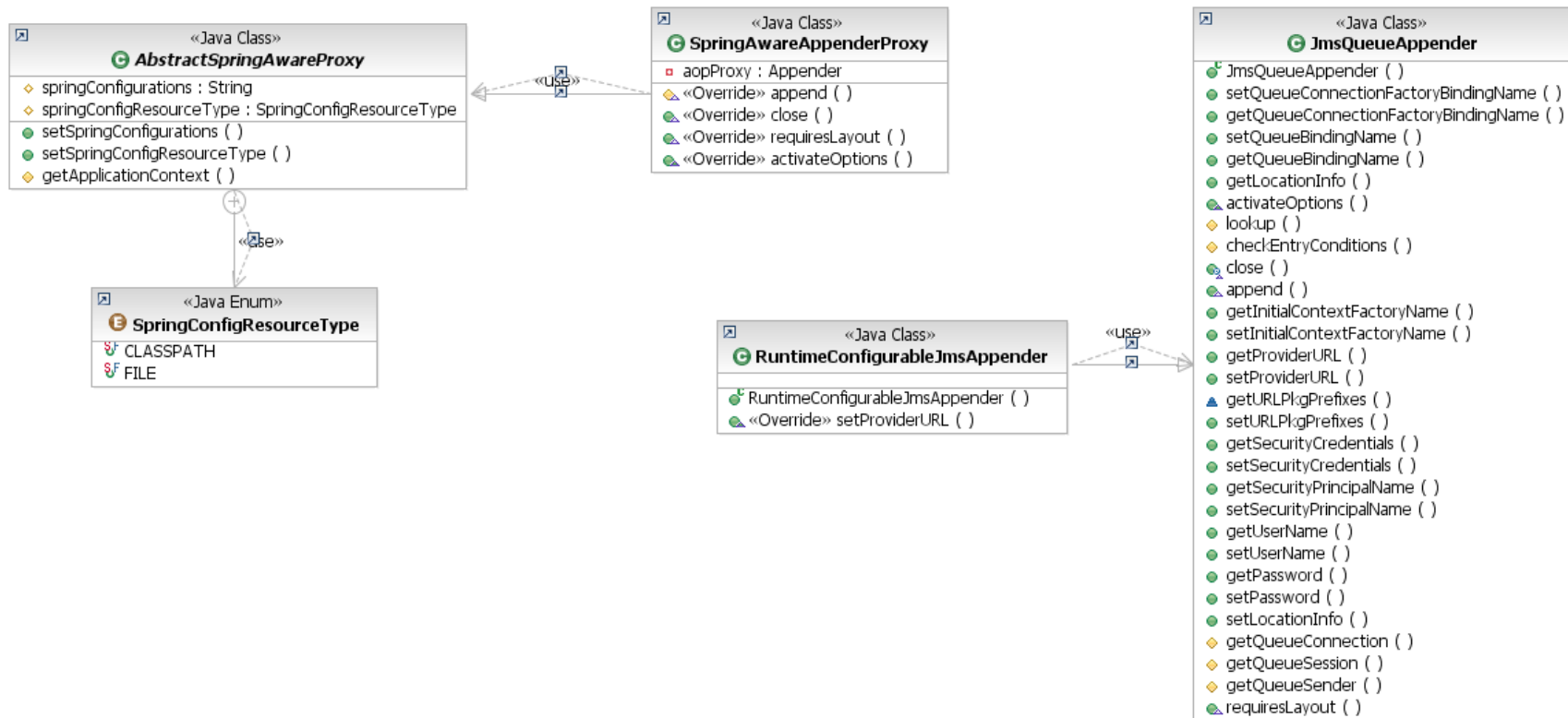
**Figure 61 CDS Application Logging Overview**



### 6.2.1.27.3 Application Logging Processing

The following class diagram refers to the Log4j JMS logger implementation by CDS application:

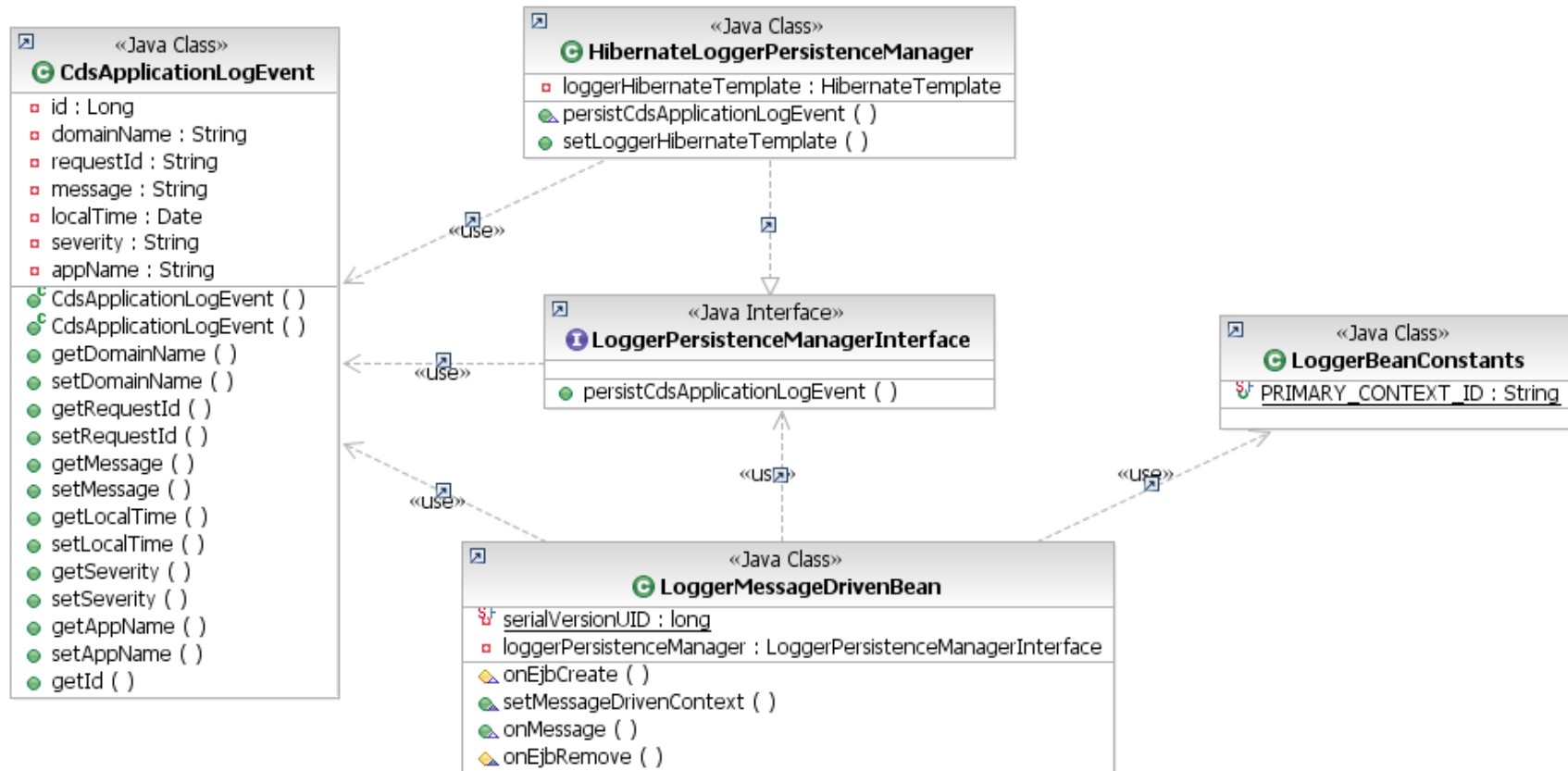
**Figure 62 Application Logging Class Diagram**



1. SpringAwareAppenderProxy is a proxy to an AOP proxy for JMS Appender. All calls upon this class are passed on to a Spring configured AOP proxy of a RuntimeConfigurableAppender.
2. Enum SpringConfigResourceType is used to specify how to load Spring configuration files; using either a FileSystem pattern (FileSystemXmlApplicationContext) or a ClassPath pattern (ClassPathXmlApplicationContext). SpringAwareAppenderProxy uses ClassPath pattern to load JMS logger Spring configuration files.
3. RuntimeConfigurableJmsAppender is a subclass of JmsQueueAppender, which accommodates the configuration of Weblogic's JNDI context and LoggerQueue connection details (like LoggerConnectionFactory, LoggerQueue).
4. JmsQueueAppender is a simple appender that publishes events to a JMS LoggerQueue. The events are serialized and transmitted as JMS message type ObjectMessage to LoggerQueue.

The following class diagram refers to the persistence of messages that are in LoggerQueue:

**Figure 63 Application Logging Persistence Manager Class Diagram**



1. CDS components push the exception/error messages into LoggerQueue using JmsQueueAppender. LoggerQueue is consumed by LoggerMessageDrivenBean.
2. LoggerMessageDrivenBean is a message driven bean that is responsible to process and persist the messages that are in LoggerQueue. LoggerMessageDrivenBean uses HibernateLoggerPersistenceManager to persist the LoggerQueue messages into HDR data base.
3. HibernateLoggerPersistenceManager uses HibernateTemplate to persist the messages that are in LoggerQueue into CDSAPPLICATIONLOG table of LOGGER schema in HDR data base.

#### 6.2.1.27.4 Application Logging Local Data Structures

JMS appender and logger are configured in log4j.xml:

```
<appender name="JMS" class="gov.va.med.cds.log4j.SpringAwareAppenderProxy">
    <param name="springConfigResourceType" value="CLASSPATH"/> <!-- May be FILE or CLASSPATH
only. -->
    <param name="springConfigurations" value="classpath:gov/va/med/cds/log4j/config/jms-app-
logging-context.xml"/>
</appender>

<logger name="gov.va.med.cds">
    <level value="ERROR" />
    <appender-ref ref="JMS" />
</logger>
```

#### RuntimeConfigurableJmsAppender Spring framework configuration:

```
<bean id="aop-proxy" class="gov.va.med.cds.log4j.RuntimeConfigurableJmsAppender">
    <property name="initialContextFactoryName"
value="weblogic.jndi.WLInitialContextFactory" />
    <!--
        This is the default value and can be over-ridden by the system
        property 'cds.jndi.provider.url'
    -->
    <property name="providerURL" value="t3://localhost:7000" />
    <!-- property name="providerURL" value="${cds.jndi.provider.url}" /-->
    <property name="queueConnectionFactoryBindingName"
value="jms/cds/LoggerConnectionFactory" />
    <property name="queueBindingName" value="jms/cds/LoggerQueue" />
</bean>

<bean id="appenderTransactionManager"
class="org.springframework.transaction.jta.WebLogicJtaTransactionManager" />
    <aop:config>
        <aop:pointcut id="appenderTxMethods" expression="execution(*
org.apache.log4j.Appender.*(..))" />
        <aop:advisor advice-ref="appenderTxAdvice" pointcut-ref="appenderTxMethods"
order="1" />
    </aop:config>
</bean>
```



```

</aop:config>

<tx:advice id="appenderTxAdvice" transaction-manager="appenderTransactionManager">
    <tx:attributes>
        <tx:method name="doAppend" propagation="REQUIRES_NEW"
isolation="READ_COMMITTED" />
    </tx:attributes>
</tx:advice>
</beans>

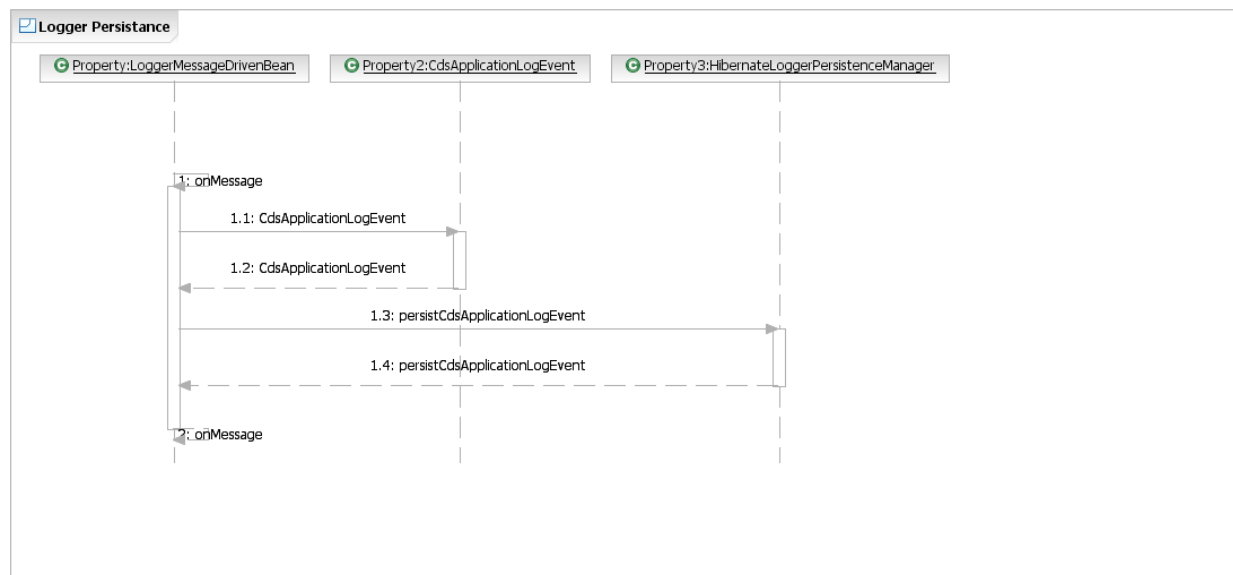
```

HibernateLoggerPersistenceManager class Spring framework configuration:

### 6.2.1.27.5 Application Logging Module/Other Diagrams

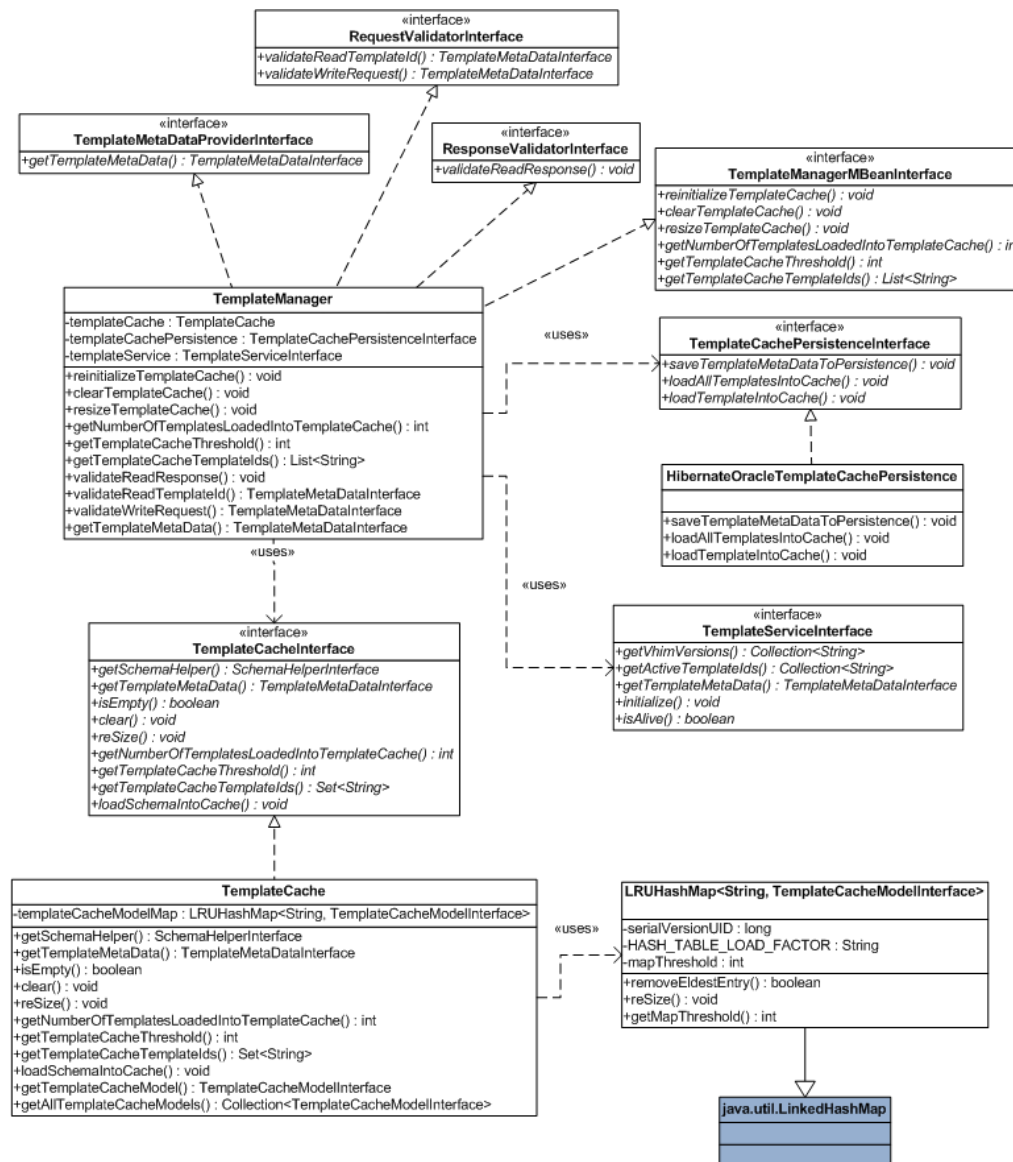
The following sequence diagram refers to the persistence process of LoggerQueue messages:

**Figure 64 Logger Persistence Sequence Diagram**



1. When an exception/error messages is pushed into LoggerQueue, LoggerMessageDrivenBean's onMessage() is notified.
2. onMessage() of LoggerMessageDrivenBean:
  - a. Type casts Message instance into LoggingEvent instance
  - b. creates an instance of CdsApplicationLogEvent using LoggingEvent instance and pass it onto HibernateLoggerPersistenceManager to persist the messages that are in LoggerQueue into CDSAPPLICATIONLOG table of LOGGER schema in HDR data base.
3. HibernateLoggerPersistenceManager uses HibernateTemplate to persist the messages into HDR data base.

**Figure 65 Template Manager Class Diagram**



### 6.2.1.27.6 HL7 messages processing

HL7 messages are converted into ELDM format XML payload by the Message mediator webservice which is a HDR component. Message mediator webservice acts as a client to CDS.

The HDR system components utilized by the HL7 message writes are the Socket Adapter for Laboratory HL7 writes and JMS Queues for asynchronous writes of Allergy, Outpatient Pharmacy, and Vitals from Vista via the Vista Interface Engine (VIE) infrastructure. Lab messages received by the Socket Adapter are forwarded to the Message Mediator through the Lab domain specific JMS message queue to await processing with all other HL7 messages. Once in the Message Mediator JMS queues, messages are delivered to the Message Mediator component for processing. The Message Mediator transforms the ER7-encoded HL7 messaging into XML-encoded HL7, which is then transformed by eXtensible Stylesheet Language (XSL)

Transformations (XSLT) into VIM messages that are sent to the CDS framework for persistence in HDR.

For additional details concerning the Socket Façade and the Message Mediator processing, see the *Socket Façade Software Design Document* and the *Message Mediator Transformation and Processing Design Document* respectively.

#### **6.2.1.28 FPDS**

##### **6.2.1.28.1 FPDS Overview**

FPDS webservice component is a HDR component that acts as a thin web-service adapter layer to CDS. FPDS is able to transform Representational State Transfer (REST) write/update/read requests into filter Read requests that can be processed by CDS and underlying CDS framework. Additionally, the FPDS will have the ability to transform results produced by CDS into JavaScript Object Notation (JSON) prior to sending the data back to its clients. The FPDS modules can return data domain response in JSON or in XML format. FPDS is deployed as an independent ear separate from CDS ear.

##### **6.2.1.28.2 FPDS Module Design**

The FPDS webservice is a CXF library based RESTful implementation. It acts as a client to CDS component. FPDS serves the clients that require the payload format in JSON.

FPDS supports Read operations for HMP data from VistA and Create/Update, Read operations for HMP PGD data from HDR. The URL comprises mandatory and optional query parameters. The URL and optional query parameters are parsed by an XSL or dom4j library in the FPDS webservice to generate a filter XML. The FPDS webservice then looks up the CDS service's EJB using JNDI lookup and calls the existing CDS Read webservice to make the read call. The patient lookup information is an Integration Control Number (ICN), whereby the ICN is correlated to its respective DFNs by CDS calling the VA Identity Management (IdM) webservice. The CDS framework queries the HMP Generic Stored Procedures at all the sites the patient was seen (indicated by the DFNs) and aggregates the response. The FPDS service has the option to return the response as a JSON document, or a ELDM compliant XML document.

##### **6.2.1.28.3 FPDS Processing**

FPDS processes the HMP/eHMP VPR Read requests by calling CDS that has access to the HMP/eHMP domain data via Cache Generic Stored Procedures. FPDS processes the HMP PGD write (asynch/synch), Read and Update requests by calling CDS that has access to the HMP domain data in HDR database. FPDS calls the remote stateless session bean of CDS.

The calling system queries HMP data through FPDS. The query call contains mandatory elements which comprise the URL, and optional elements which comprise the query parameters.

The URL and optional query parameters are parsed by an XSL or dom4j library in the FPDS webservice to generate a filter XML. The FPDS webservice then looks up the CDS service's EJB using JNDI lookup and calls the existing CDS read webservice to make the read call. The patient lookup information is an ICN, whereby the ICN is correlated to its respective DFNs by CDS calling the VA Identity Management (IdM) webservice. FPDS does a UDDI lookup for IdM service. The CDS framework queries the HMP Generic Stored Procedures at all the sites the

patient was seen (indicated by the DFNs) and aggregates the response. The FPDS service has the option to return the response as a JSON document, or wrap a XML around the JSON document.

FPDS also supports Create/Update (synchronous and asynchronous) and Read operations on the non-clinical PGD data stored in HDR.

FPDS maintains a MDB that processes the PGD JSON centric messages and wraps them in a standard PGD VIM template and then sends as a synchronous request to the CDS framework. The FPDS MDB will deposit an ACK response on the response queue in the case of successful transaction processing. FPDS MDB will deposit the error response in case of any error during the processing of the client request on the response queue. Client processes the messages from the response queue.

#### 6.2.1.28.4 FPDS Local Data Structures

The following details the FPDS configuration:

```
<bean id="routingXsltOutInterceptor"
class="gov.va.med.repositories.fpbs.interceptor.RoutingXsltOutInterceptor">
    <property name="outInterceptors">
        <map>
            <entry key="json"
value="templates/response/GenericObservationRead1-json.xml"/>
        </map>
    </property>
</bean>

<bean id="routingNonVAMedXsltOutInterceptor"
class="gov.va.med.repositories.fpbs.interceptor.RoutingXsltOutInterceptor">
    <property name="outInterceptors">
        <map>
            <entry key="json" value="templates/response/NonVaMedicationsRead3-
json.xml"/>
        </map>
    </property>
</bean>

<jaxrs:server id="fpds" address="/">
    <jaxrs:serviceBeans>
        <bean
class="gov.va.med.repositories.fpbs.webservice.FederatingPatientRecordRestService">
            <property name="clinicalDataService" ref="cdsStatelessSession" />
            <property name="requestValidator">
                <bean
class="gov.va.med.repositories.fpbs.validator.DefaultRequestValidator" />
            </property>
```

```

        <property name="filterBuilder">
            <bean
                class="gov.va.med.repositories.fpds.filter.HmpReadRequestXsltFilterBuilder">
                    <property name="filterTransformerTemplate"
value="templates/filter/HmpFilter100.xsl" />
                </bean>
            </property>
            <property name="applicationName" value="FPDS" />
            <property name="errorTemplateDocument"
value="templates/response/HmpGenericObservationRead1.xml" />
        </bean>
    </jaxrs:serviceBeans>
    <jaxrs:outInterceptors>
        <ref bean="routingXsltOutInterceptor"/>
    </jaxrs:outInterceptors>
</jaxrs:server>

<jaxrs:server id="fpds-nonvameds" address="/nonvamed">
    <jaxrs:serviceBeans>
        <bean
            class="gov.va.med.repositories.fpds.webservice.FederatingPatientRecordRestService">
                <property name="clinicalDataService" ref="cdsStatelessSession" />
                <property name="requestValidator">
                    <bean
                        class="gov.va.med.repositories.fpds.validator.DefaultRequestValidator" />
                </property>
                <property name="filterBuilder">
                    <bean
                        class="gov.va.med.repositories.fpds.filter.GenericReadRequestXsltFilterBuilder">
                            <property name="filterTransformerTemplate"
value="templates/filter/GenericFilter100.xsl" />
                        </bean>
                    </property>
                    <property name="templateBuilder">
                        <bean
                            class="gov.va.med.repositories.fpds.template.RequestXsltTemplateBuilder">
                                <property name="templateXsltMap"
ref="templateXsltMap" />
                            </bean>
                        </property>
                        <property name="applicationName" value="FPDS-NONVAMEDS" />
                    </bean>
                </property>
            </bean>
        </jaxrs:serviceBeans>
    </jaxrs:server>

```

```

        <property name="errorTemplateDocument"
value="templates/response/HmpGenericObservationRead1.xml" />
    </bean>
</jaxrs:serviceBeans>
<jaxrs:outInterceptors>
    <ref bean="routingNonVAMedXsltOutInterceptor"/>
</jaxrs:outInterceptors>
</jaxrs:server>

<util:map id="templateXsltMap">
    <entry key="NonVAMedicationCreateOrUpdate3"
value="templates/request/NonVAMedicationCreateOrUpdate3.xsl" />
    <entry key="NonVAMedicationDelete1"
value="templates/request/NonVAMedicationDelete1.xsl" />
</util:map>

<jee:remote-slsb id="cdsStatelessSession"
    jndi-name="ejb/remote/gov/va/med/cds/ClinicalDataServiceSynchronousSession"
    business-interface="gov.va.med.cds.client.ClinicalDataServiceInterface" />
</beans>

```

## 6.2.2 Specific Requirements

### 6.2.2.1 Database Repository

Refer to Section 6.2.1.6 Database Repository.

### 6.2.2.2 System Features

Refer to HDR 3.8 SDD, Volume 1, Section 2.5.1 for an Overview of Significant Requirements.

### 6.2.2.3 Design Element

The core CDS framework interacts with HDRDAT for retrieving data from VistA. HDRDAT is a data access component deployed to each VistA instance that enables VistA files to be queried via SQL. A corresponding Cache SQL table (Cache Persistent Class) exists for every Fileman file/sub-file that is a part of the national package, with 7,000+ tables in all. The VA CASH system generates these classes/tables automatically by reading the data dictionary. New versions of the classes are regenerated by CASH to keep sync with VistA patch releases and pushed out to all VistA sites automatically. There are no changes to the design elements for HDR 3.8 Release.

There are customizations made to some of the tables to make the CDS software more efficient. The focus is to eliminate some of the high overhead joins by encapsulating data columns in other tables and exposing them as if they were in the same table. These customizations are done by “Extending” (inheritance) to another class/table (Child Table). The child table (same name as the parent table with EXT appended) inherits all the columns from the parent class while exposing the customized columns in child table. The following table lists the extended tables and the customized columns.

**Table 2 Extend tables/Columns**

Cache Class Name	SQL Table Name	SQL Column	Data Type
ADVERSEREACTIONASSESSMENTExtended	ADVERSEREACTIONASSESSMENT12086EXT	AlternateProcedureCodeExt	%Library.String
		OrderedTestCodeExt	%Library.String
		OrderedTestTextExt	%Library.String
		AuthorNE	%Library.String
		DnsIp	%Library.String
		DomainNameX	%Library.String
		ErrorEntererNE	%Library.String
		HospitalLocationAbbreviation	%Library.String
GMRVVITALMEASUREMENTExtended	GMRVVITALMEASUREMENT1205EXT	AlternateProcedureCodeExt	%Library.String
		OrderedTestCodeExt	%Library.String
		OrderedTestTextExt	%Library.String
		AuthorNE	%Library.String
		DnsIp	%Library.String
		DomainNameX	%Library.String
		ErrorEntererNE	%Library.String
		HospitalLocationAbbreviation	%Library.String
		HospitalLocationDivision	%Library.String
		HospitalLocationDivisionX	%Library.String
		HospitalLocationInstitutionX	%Library.String
		InternationalRate	%Library.String
		InternationalUOM	%Library.String
		NamespaceId	%Library.String
		PatientNE	%Library.String
		Status	%Library.String
		StopCode	%Library.String
		StopCodeNumberX	%Library.String
		UOM	%Library.String
		VitalTypeVuid	%Library.String
LABORATORYTESTExtended	LABORATORYTEST60EXT	AlternateProcedureCodeExt	%Library.String
		OrderedTestCodeExt	%Library.String
		OrderedTestTextExt	%Library.String
PATIENTALLERGIESExtended	PATIENTALLERGIES1208EXT	AllergyTypeText	%Library.String
		AuthorNE	%Library.String
		DefaultInstitution	%Library.String
		DnsIp	%Library.String
		DomainNameX	%Library.String
		ErrorEntererComments	%Library.String
		ErrorIdentifierNE	%Library.String

Cache Class Name	SQL Table Name	SQL Column	Data Type
		GmrAllergyAgentAlternateCode	%Library.String
		GmrAllergyAgentAlternCodSys	%Library.String
		GmrAllergyAgentAlternDispText	%Library.String
		GmrAllergyAgentCodingSystem	%Library.String
		GmrAllergyAgentDisplayText	%Library.String
		GmrAllergyAgentVuid	%Library.String
		MechanismText	%Library.String
		MechanismVuid	%Library.String
		NamespaceId	%Library.String
		ObservedHistoricalText	%Library.String
		ObservedHistoricalVuid	%Library.String
		OfficialVaName	%Library.String
		OriginatorComments	%Library.String
		PatientNE	%Library.String
		SeverityX	%Library.String
		Status	%Library.String
		VerifierComments	%Library.String
		VerifierNE	%Library.String
PRESCRIPTIONExtended	PRESCRIPTION52EXT	ABBREVIATION	%Library.String
		CMOPEVENTSTATUS	%Library.String
		CMOPID	%Library.String
		DISPENSEDUNIT	%Library.String
		DOSAGEFORM	%Library.String
		DOSAGENAME	%Library.String
		GENERICREQUESTIDENTIFIER	%Library.String
		MEDICATIONCODECODE	%Library.String
		MEDICATIONCODESYSTEM	%Library.String
		MEDICATIONCODETEXT	%Library.String
		NCPDPNUMBER	%Library.String
		OrderingInstitutionIdentity	%Library.String
		OrderingInstitutionName	%Library.String
		PharmacyPatientType	%Library.String
		PHARMACYREQUESTSTATUS	%Library.String
		PHARMACYREQUESTSTATUSID	%Library.String
		SIGX	%Library.String
		SITENUMBER	%Library.String
		VAPRODUCTNAME	%Library.String
PROBLEM0Extended	PROBLM09000011EXT	DiagnosisDescription	%Library.String
		DiagnosisInactiveDate	%Library.String
		DiagnosisInactiveFlag	%Library.String
		LocationCode	%Library.String



Cache Class Name	SQL Table Name	SQL Column	Data Type
		LocationDisplayText	%Library.String
		ServiceCodeAbbreviation	%Library.String
SUBACCESSIONAREAExtended	LABORATOXACCESSIO NAREA6011EXT	PerformingOrgCityExt	%Library.String
		PerformingOrgNameRepExt	%Library.String
		PerformingOrgPostalCodeExt	%Library.String
		PerformingOrgStateExt	%Library.String
		PerformingOrgStationNumberExt	%Library.String
		PerformingOrgStreet1Ext	%Library.String
		PerformingOrgStreet2Ext	%Library.String
SUBCHARTMARKEDExtended	PATXCHARTMARKE D120813EXT	ChartMarkerNE	%Library.String
SUBCHEMHEMTOXRIAS ERetcExtended	LABXCHEMHEMTO X6304EXT	ConcatenatedResultsExt	%Library.String
		ConcatenatedResultsManualExt	%Library.String
		NameRepresentationExt	%Library.String
		OrderingFacilityCityExt	%Library.String
		OrderingFacilityNameExt	%Library.String
		OrderingFacilityStateExt	%Library.String
		OrderingFacilityStreet1Ext	%Library.String
		OrderingFacilityStreet2Ext	%Library.String
		OrderingFacilityZipExt	%Library.String
		RequestingOrganizationCodeExt	%Library.String
		RequestingOrganizationTextExt	%Library.String
		SpecimenLediHL7Ext	%Library.String
		SpecimenSourceCdExt	%Library.String
		SpecimenSourceCodeExt	%Library.String
		StationNumberExt	%Library.String
SUBCOMMENTSLExtended	PATALLEXCOMMENT S120826EXT	AuthorNE	%Library.String
		CommentTypeText	%Library.String
		CommentTypeVuid	%Library.String
SUBDRUGCLASSESExtended	PATXDRUGCLASSES 120803EXT	DefaultInstitution	%Library.String
		DrugClassClassification	%Library.String
		DrugClassCode	%Library.String
		DrugClassVuid	%Library.String
SUBDRUGINGREDIENTS Extended	PATIXDRUGINGRDNT 120802EXT	DrugIngredientName	%Library.String
		DrugIngredientVuid	%Library.String
SUBIDBANDMARKEDExtended	PATIDBANDMARKE D120814EXT	IdBandMarkerNE	%Library.String
SUBNONVAMEDSExtended	PHARMPAXNONVAM EDS5505EXT	DocumentedByNE	%Library.String
		DomainNameX	%Library.String
		DosageFormCode	%Library.String

Cache Class Name	SQL Table Name	SQL Column	Data Type
		DosageFormDisplayText	%Library.String
		Dose	%Library.String
		MedicationCmopCode	%Library.String
		MedicationCode	%Library.String
		MedicationCodingSystem	%Library.String
		MedicationDisplayText	%Library.String
		PatientIdentity	%Library.String
		PatientNE	%Library.String
		RouteCode	%Library.String
		RouteDisplayText	%Library.String
		StatusQuery	%Library.String
		StatusXExt	%Library.String
		TradeNameDisplayText	%Library.String
		Units	%Library.String
SUBORDERCHECKS0Extended	PHARMXORDERCHC KS55051EXT	OverridingProviderNE	%Library.String
SUBPARTIALDATEExtended	PRESCRIPTIONPARTIAL L522EXT	NCPDPNUMBER	%Library.String
SUBQUALIFIERExtended	GMRVVITALXQUAL_ 120505EXT	QualifierVuid	%Library.String
SUBREACTIONSExtended	PATALLEXREACTION S12081EXT	ReactionAuthorComment	%Library.String
		ReactionAuthorNE	%Library.String
		ReactionVuid	%Library.String
SUBREASONENTEREDIN ERRORExtended	GMRVXREASONENTI_ 120506EXT	ReasonEnteredInErrorVuid	%Library.String
SUBREFILLExtended	PRESCRIPTIONREFILL_ 521EXT	NCPDPNUMBER	%Library.String
SUBSITESPECIMENExtended	LABORATORYXSITESP ECIMEN6001EXT	DeviceIdentifierExt	%Library.String
		DeviceIdentifierTextExt	%Library.String
		ObservationMethodCodeExt	%Library.String
		ObservationMethodTextExt	%Library.String
		ObservationUnitsCodeExt	%Library.String
		ObservationUnitsIdentifierExt	%Library.String
		ObservationUnitsTextExt	%Library.String
		TestIdentifierAlternateCodeExt	%Library.String
		TestIdentifierAlternateTextExt	%Library.String
		TestIdentifierOriginalTextExt	%Library.String
		TestIdentifierTextExt	%Library.String
TIUAUDITTRAILExtended	TIUAUDITTRAIL8925 5EXT	IDiscNoteActionVuid	%Library.String
		InterdisciplinaryNoteMarkerNE	%Library.String

Cache Class Name	SQL Table Name	SQL Column	Data Type
TIUDOCUMENTExtended	TIUDOCUMENT8925EXT	AmenderNE	%Library.String
		AmendSignatureBlockValue	%Library.String
		AmendSignatureTitleValue	%Library.String
		AttendingPhysicianNE	%Library.String
		AuthorDictatorNE	%Library.String
		ChartMarkedCosignedNE	%Library.String
		ChartMarkedSignedNE	%Library.String
		ClinicName	%Library.String
		CoSignatureBlockValue	%Library.String
		CoSignatureModeText	%Library.String
		CosignatureNeededVuid	%Library.String
		CoSignatureTitleValue	%Library.String
		CosignerNE	%Library.String
		DocumentClasses	%Library.String
		DomainNameX	%Library.String
		ExpectedCosignerNE	%Library.String
		ExpectedSignerNE	%Library.String
		LocalDocumentTitle	%Library.String
		NamespaceId	%Library.String
		PatientMovementDateTime	CASH.HDR.Date Time
		PatientNE	%Library.String
		PrintName	%Library.String
		ProcedureSummaryCodeVuid	%Library.String
		ServiceCS	%Library.String
		ServiceName	%Library.String
		ServiceVuid	%Library.String
		SignatureBlockValue	%Library.String
		SignatureModeText	%Library.String
		SignatureTitleValue	%Library.String
		SignerNE	%Library.String
		StandardTitleCS	%Library.String
		StandardTitleText	%Library.String
		StandardTitleVuid	%Library.String
		StatusVuid	%Library.String
		StdDocTypeText	%Library.String
		StdDocTypeVuid	%Library.String
		TreatingSpecialtyCS	%Library.String
		TreatingSpecialtyName	%Library.String
		TreatingSpecialtyVuid	%Library.String
		UrgencyVuid	%Library.String
		VisitDateTime	CASH.HDR.Date

Cache Class Name	SQL Table Name	SQL Column	Data Type
			Time
TIUEXTERNALDATA LINKExtended	TIUEXTDATA LINK892591EXT	ImageLinkCodeCS	%Library.String
TIUMULTIPLESIGNATUREExtended	TIUMULTIPLESIG89257EXT	OtherCosignersNE	%Library.String
		OtherExpectedCosignersNE	%Library.String
VIMMUNIZATIONExtended	VIMMUNIZATION900001011EXT	Diagnosis2AltDisplayText	%Library.String
		Diagnosis3AltDisplayText	%Library.String
		Diagnosis4AltDisplayText	%Library.String
		Diagnosis5AltDisplayText	%Library.String
		Diagnosis6AltDisplayText	%Library.String
		Diagnosis7AltDisplayText	%Library.String
		Diagnosis8AltDisplayText	%Library.String
		DiagnosisAltDisplayText	%Library.String
		ImmunizationShortName	%Library.String
		ServiceCategory	%Library.String
		VisitCheckOutDate	%Library.String
		VisitComments	%Library.String
		VisitDate	%Library.String
		VisitId	%Library.String
VSKINTESTExtended	VSKINTEST900001012EXT	CptCode	%Library.String
		CptCodeShortName	%Library.String
		Diagnosis2AltDisplayText	%Library.String
		Diagnosis3AltDisplayText	%Library.String
		Diagnosis4AltDisplayText	%Library.String
		Diagnosis5AltDisplayText	%Library.String
		Diagnosis6AltDisplayText	%Library.String
		Diagnosis7AltDisplayText	%Library.String
		Diagnosis8AltDisplayText	%Library.String
		DiagnosisAltDisplayText	%Library.String
		ServiceCategory	%Library.String
		VisitCheckOutDate	%Library.String
		VisitComments	%Library.String
		VisitDate	%Library.String
		VisitId	%Library.String
		VisitServiceCategoryAltCode	%Library.String

HDRDAT also includes stored procedures and custom functions that are used to obtain Vista data via Cache Object Script (COS) classes, methods, and routines. Table 20 represents the current Cache SQL Stored Procedures and SQL Functions.

**Table 3 Stored Procedures and Functions**

Name	Input Parameters O–Optional R–Required	Output Parameters	Description
HDSREPRPCOPRxBLOB Type: SQL Stored Procedure Schema: VISTA Production: Yes	1.requestID (O) 2. callName (O) 3. patientID (R) 4. fromExpDt (R) 5. toExpDt (R) 6. RxID (R) 7. CustomFlag (O) 8. DebugFlag (O)	1. requestID 2. siteID 3. callName 4. returnType 5. params 6. customFlag 7. returnError 8. Blob	This stored procedure returns patient prescription data based on expiration date. Just one record is returned with all prescription related data placed into the Blob field. Data tags are used to identify the different levels of data (as noted below): <Rec> - Retrieves the top level prescription Information (File #52) <Refill> - Retrieves Refill information (File 52.1) <Partial> Retrieves Partial refill information (File 52.2) <ActLog> Retrieves Activity Log information (File 52.3) <MedIns> retrieves Medication Instructions information (File 52.0113) <ExpandIns> Retrieves expanded instructions (File 52.0115) <PharmIns> Retrieves pharmacy instructions (File 52.02) <ProvComments> Retrieves provider comments (File 52.039) <Sig1> Retrieves Sig1 data (File 52.04)
HDSREPRPCMHVLabLOB Type: SQL Stored Procedure Schema: VISTA Production: No (CDS Testing)	1.requestID (O) 2. callName (O) 3. patientID (R) 4. fromSpecimenDt (R) 5. toSpecimenDt (R) 6. CustomFlag (O) 7. DebugFlag (O)	1. requestID 2. siteID 3. callName 4. returnType 5. params 6. customFlag 7. returnError 8. Blob	This stored procedure returns patient Lab Test/Result data from file 63.04 (type “CH”) based on Specimen date. The crux of the logic uses an API routine call, RPT^ORWRP. Just one record is returned with all lab related data placed into the Blob field. Data tags are used to identify the different levels of data (as noted below): <Rec> Retrieves the top level lab information (File 63.04). <Results> Retrieves the Result information from an API and pulls Lab information from File 60/60.01 <ResultsRaw> Retrieves the Result information in its raw form. Most of the data elements stored at this level are NOT documented in Fileman therefore this option provides the raw data <LabComment> Retrieves Interpretation (Word process Multiple) for a given test/specimen file 60.07 <Accession> Retrieves information that performs the test from file 60.11.



Name	Input Parameters O–Optional R–Required	Output Parameters	Description
HDSREPRPCExamsBLOB Type: SQL Stored Procedure Schema: VISTA Production: Yes	1.requestID (O) 2. callName (O) 3. patientID (R) 4. fromExamDt (O) 5. toExamDt (O) 6. Status (O) 7. CustomFlag (O) 8. DebugFlag (O)	1. requestID 2. siteID 3. callName 4. returnType 5. params 6. customFlag 7. returnError 8. Blob	This stored procedure returns 2507 Exam information (File 396.4) for a given patient. Records can be filtered by Exam date and Status code. All exam data is returned in the Blob field using a CR/LF to separate the exam records. Information from the following files are retrieved: 2507 Exam (File 396.4) – 2507 Exam Detail 2507 Exam (File 396.3) - Remarks Patient (File 2) – Address, Temporary Address, and next of Kin
HDSREPRPCApptsBLOB Type: SQL Stored Procedure Schema: VISTA Production: Yes	1.requestID (O) 2. callName (O) 3. patientID (R) 4. fromApptDt (O) 5. toApptDt (O) 6. CustomFlag (O) 7. DebugFlag (O)	1. requestID 2. siteID 3. callName 4. returnType 5. params 6. customFlag 7. returnError 8. Blob	This stored procedure returns patient appointment data based on appointment date. Just one record is returned with all appointment related data placed into the Blob field. A Vista API is used to pull the appointment, “SDAPI^SDAMA301”.  Patient detail is pulled from Patient File (File 2)  Appointment Location is pulled from Hospital Location file (File 44).
HDSREPGSPGSP Type: SQL Stored Procedure Schema: VISTA Production: NO (Scheduled for CDS version 3.6.2)	1. ProjectID 2. Security 3. RPCid 4. Arguments	1. Node 2. Data	This is a generic stored procedure that allows calls to RPC file (File 8994). Currently the only RPC being called is “VPR GET PATIENT DATA JSON”.  <b>Input Parameters:</b> <u>ProjectID</u> – Currently the only projectID setup is for CDS.  <u>SecurityID</u> In order to run the GSP proper security key/encryption value must be passed. The encryption contains an algorithm that expires after 10 seconds  <u>RPCid</u> – Pass in the Name or ID of the RPC TO RUN  <u>Arguments</u> – pass in arguments for the RPC, environment variables, and override parameters  <b>Output Parameters:</b> 3 Possible Columns are returned:  1. Node – This contains information

Name	Input Parameters O–Optional R–Required	Output Parameters	Description
			<p>about the array or global being returned by the RPC</p> <ol style="list-style-type: none"> <li>2. Data – This contains the data returned by the RPC</li> <li>3. Error – Returns an Error if the RPC fails to run or invalid parameters are passed.</li> </ol>
HDSREPOPRxUtil1getRxProfiles Type: SQL Stored Procedure Schema: VISTA_HDSI Production: Yes	1. PatientID (DFN)	2. OPRXID	Returns a recordset of all Prescription IDs (File 52) for a given patient ID.
HDSREPOPRxUtil1getRxBdyDate Type: SQL Stored Procedure Schema: VISTA_HDSI Production: Yes	<ol style="list-style-type: none"> <li>1. PatientID (DFN)</li> <li>2. fromDate</li> <li>3. toDate</li> </ol>	1. OPRXID	Returns a recordset of all Prescription IDs (File 52) for a given patient ID that were active for a given date range.
HDSREPLabUtil1getOrderDetail Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. PatientID (DFN)	1. String	String contains two data element delimited by a carot “^”: <ol style="list-style-type: none"> <li>1. OrderID from the Order file (File 100)</li> <li>2. Package Reference field from the Order File (NODE4)</li> </ol>
HDSREPLabUtil1getLabReference Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. PatientID (DFN)	1. String	String returns the Lab Reference ID for a given patient from the Patient File (File 2)
HDSREPSQLUtil1stationNumber Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	None	1. String	Returns Station number for a Vista system instance from the Kernel System Parameters File (File 8989.3)
HDSREPSQLUtil1ProcedureSummaryCodeVuid Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. External VUID	1. String	Returns the internal VUID from the external VUID from file XTID VUID for set of Codes (File 8985.1)
HDSREPSQLUtil1TreatingSpecialtyName Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. IEN - Facility Treating Specialty (File 45.7)	1. String	Returns Specialty Name from Specialty File (File 42.4)
HDSREPSQLUtil1Treating	1. IEN - Facility	1. String	Returns Specialty Vuid from Specialty File

Name	Input Parameters O–Optional R–Required	Output Parameters	Description
SpecialtyVuid Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	Treating Specialty (File 45.7)		(File 42.4)
HDSREPSQLUtil1_ServiceName Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. IEN - Service/Section (File 49)	1. String	Returns the Service/Section name From (File 49)
HDSREPSQLUtil1_VisitDateTime Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. IEN - Visit File (File 9000010)	1. String	Returns the Visit Date time from the Visit (File 9000010)
HDSREPSQLUtil1_KSPDomainName Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	None	1. String	Returns Domain Name for a VistA system instance from the Kernal System Parameters File (File 8989.3)
HDSREPSQLUtil1_SiteDNSIP Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	None	1. String	Returns the sites DNS or IP Address for a VistA system instance from the Kernal System Parameters File (File 8989.3)
HDSREPSQLUtil1_OfficialVAName Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	None	1. String	Returns the sites Official VA Name for a VistA system instance from the Kernal System Parameters File (File 8989.3).
HDSREPSQLUtil1_PatientName Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. PatientID (DFN)	1. String	Returns a delimited string of data from the Patient File (File 2) as: Identifier^Family^Given^Middle^Prefix^Suffix^Title
HDSREPSQLUtil1_MiscNameNE Type: SQL Function Return Type: String Schema: VISTA_HDSI Production: Yes	1. IEN (File 200)	1. String	Returns a delimited string of data from the New Persons File (File 200) as: Identifier^Family^Given^Middle^Prefix^Suffix^Title





## Attachment A.      Reviews and Approval Signatures

This section contains the reviews and approval signatures for the *HDR 3 8 System Design Document, Volume 2*.

Peer Review: 10/31/2014

The signatures below indicate agreement and acceptance of the declarations contained in this document.

//es// [redacted]	11/12/14
[redacted]	Date
Project Manager, HDR	

//es// [redacted]	11/6/14
[redacted]	Date
Program Manager, Home Telehealth	

//es// [redacted]	11/13/14
[redacted]	Date
Program Manager, HMP/eHMP PGD	

//es// [redacted]	11/13/14
[redacted]	Date
Director, Data Architecture Division (DA)	

//es// [redacted]	11/9/14
[redacted]	Date
Chief, Health Application Support	

//es// [redacted]	11/13/14
[redacted]	Date
IT Program Manager, Repositories, Integrated Project Team Chair	

## Attachment B. Signature Verification

The Signature Verification section is used to verify and document the electronic signatures, concurrence and approval of the *HDR 3.8 System Design Document, Volume 2*.

---

**Approve: Concurrence Signature requested for HDR 3.8 System Design Document (SDD) due COB Nov 12, 2014**

