caArray 2.0

Technical Guide

NATIONAL INSTITUTES OF HEALTH

DEPARTMENT OF HEALTH & HUMAN SERVICES • USA

NATIONAL® CANCER INSTITUTE

Center for Bioinformatics

January 26, 2008

CREDITS AND RESOURCES

caArray Development and Management Teams			
Development	***Quality Assurance***	***Documentation***	***Project and Product Management***
Eric Tavela[2]	Tom Boal[5]	Eric Tavela[2]	Mervi Heiskanen[1]
Bill Mason[2]	Ron Keene[5]	Brent Gendleman[2]	Anand Basu[1]
Todd Parnell[2]	Xiaopeng Bian[1]	Todd Parnell[2]	Brent Gendleman[2]
Dan Kokotov[2]		Paul Duvall[4]	Juli Klemm[1]
Rashmi Srinivasa[2]		Levent Gurses[4]	
Scott Miller[2]		Jill Hadfield[1]	
Systems and Application Support		***Training***	
Sriram Kalyanasundaram[3]		Don Swan[3]	
Andrea Johnson			
Paul Duvall[3]			
[1] National Cancer Institute Center for Bioinformatics (NCICB)		[2] 5AM Solutions	[3] Terrapin Systems
[4] Stelligent	[5]NARTech		

Contacts and Support	
NCICB Application Support	http://ncicbsupport.nci.nih.gov/sw/ Telephone: 301-451-4384 Toll free: 888-478-4423

TABLE OF CONTENTS

USING THE CAARRAY TECHNICAL GUIDE

This chapter contains an overview of the technical guide.

Topics in this chapter include:

- *Introduction to caArray* on this page
- *Purpose of this Manual* on this page
- *Recommended Reading* on page 2
- *Organization of the Manual* on page 2
- *Document Text Conventions* on page 2

Introduction to caArray

The *caArray Technical Guide* describes the aspects of caArray's design that are considered to be architecturally significant; that is, those elements and behaviors that are most fundamental for guiding the construction of caArray and for understanding caArray as a whole. Stakeholders who require a technical understanding of caArray are encouraged to start by reading this document, then reviewing the caArray UML model, and then by reviewing the source code. Please note that all diagrams represented in this document are taken from the caArray UML model; for more detail about the elements in these diagrams, consult the source model. See https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/analysis_and_design/models/caarray.EAP.

Purpose of this Manual

The *caArray Technical Guide* provides a comprehensive architectural overview of the caArray system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

Existing caArray documentation can be found on the caArray page of the NCICB website: **http://caarray.nci.nih.gov/documentation**. This guide does not duplicate documents found independently at that website, but contains ancillary technical documentation contributing to the successful utilization of caArray.

Note: Uniform Resource Locators (URLs) are used throughout the document to provide sources for more detail on a subject or product.

Definitions and Acronyms

- **DAO –** Data Access Object
- **EJB –** Enterprise JavaBeans
- **J2EE –** Java 2 Enterprise Edition
- **Java SE –** Java Standard Edition
- **JDK –** Java Development Kit
- **JPA –** Java Persistence API
- **JSP –** JavaServer Pages
- **MAGE-TAB –** Microarray Gene Expression Object Model
- **POJO –** Plain Old Java Object
- **RUP –** Rational Unified Process
- **UML –** Unified Modeling Language

Organization of the Manual

The *caArray Technical Guide* contains the following chapters:

Document Text Conventions

Table *1.1* illustrates how text conventions are represented in this guide. The various typefaces differentiate between regular text and menu commands, keyboard keys, toolbar buttons, dialog box options and text that you type.

Convention	*Description*	*Example*
Bold & Capitalized Command **Capitalized command > Capitalized command**	Indicates a Menu command Indicates Sequential Menu commands	**New Array Design**
TEXT IN SMALL CAPS	Keyboard key that you press	Press ENTER
TEXT IN SMALL CAPS + TEXT IN SMALL CAPS	Keyboard keys that you press simultaneously	Press SHIFT + CTRL and then release both.

Table 1.1 caArray Guide Text Conventions

Convention	***Description***	***Example***
`Monospace type`	Used for filenames, directory names, commands, file listings, and anything that would appear in a Java program, such as methods, variables, and classes.	`ExperimentData`
Boldface type	Options that you select in dialog boxes or drop-down menus. Buttons or icons that you click.	From the Experiment Details page, click **Generate MAGE-ML**.
Italics	Used to reference other documents, sections, figures, and tables.	*caArray User's Guide*
`Boldface monospace type`	Text that you type	In the New Subset text box, enter **`Array Manufacture Software.`**
Note:	Highlights a concept of particular interest	**Note:** This concept is used throughout the installation manual.
Warning!	Highlights information of which you should be particularly aware.	**Warning!** Deleting an object will permanently delete it from the database.
`{}`	Curly brackets are used for replaceable items.	Replace `{root directory}` with its proper value, such as `c:\caarray`

Table 1.1 caArray Guide Text Conventions (Continued)

CHAPTER 1

ARCHITECTURAL REPRESENTATION OF CAARRAY

Architectural Representation

The caArray architecture is represented in the *caArray Technical Guide* and in the UML design models as a set of views of the system from different but complementary perspectives. These views are:

- The **Use-Case View** – Describes the functional requirements of the system. See *Chapter 3, Use Case View,* on page 11.
- The **Logical View** – Describes the organization of the system design into subsystems, interfaces, and classes and how these elements collaborate to provide the functionality described in the use-case view. See *Chapter 4, Logical View,* on page 15.
- The **Process View** - Illustrates the process decomposition of the system, including the mapping of classes and subsystems on to processes and threads.
- The **Implementation View** – Describes the software components that realize the elements from the logical view and the dependencies between these components. *Chapter 5, Implementation View,* on page 41.
- The **Deployment View** – Describes how the processes are allocated to hardware and execution environments and the communication paths between hardware nodes. *Chapter 6, Deployment View,* on page 43.

This style of describing software architecture is the approach recommended by the Rational Unified Process and is based on Philippe Kruchten's work, "*The 4+1 view model of architecture*" (http://portal.acm.org/citation.cfm?id=625529) and is refined in the Rational Unified Process [RUP].

Architectural Goals and Constraints

The following factors are key considerations beyond the functional requirements that are guiding the design of caArray 2.0.

caBIG Silver Compliance

caArray must be implemented in such a way that it may be certified caBIG Silver compliant. While Silver compliance is the requirement, caArray will provide a grid interface in anticipation of a possible move to Gold level compliance when the criteria for Gold compliance are established.

.Remote API Usability

One of the major flaws in releases of caArray prior to 2.0 has been the requirement to use the MAGE-OM to access annotation and data. Navigation between key classes in the MAGE is inefficient, difficult to understand, and difficult to implement. The object API exposed by the new evolution of caArray is designed to be easily-understandable and navigable by remote clients, whether they access the API via the grid or using a Java programmatic interface.

High Performance Data Parsing, Storage and Retrieval

Given that data storage and retrieval is the principal functionality of caArray, array data parsing, storage, and retrieval performance is key to a successful design.

CHAPTER 2

EXTRACTING DATA USING AN API

This chapter provides directions for extracting data from the caArray database using a Java API. Topic in this chapter include:

Introduction on this page

Deploying the Extraction on page 8

Introduction

Using External Systems, data in the caArray database can be extracted programmatically using an API. The following external systems can communicate programmatically with caArray:

- geWorkBench
- caIntegrator
- WebGenome
- Grid client through a caGrid service

Most users of this feature are grid users. From any one of these systems, or any other external system that can access caArray, you can programmatically extract annotation and array content from a running instance of caArray. using an API. This chapter describes the process for doing that.

This chapter also includes the steps for extraction of data from caArray by a grid client through a caGRID service. caBIG Analysis Services is the primary external system targeted, and public and protected data should be available provided appropriate authorization is used. This last sentence needs clarification. I'll add these to the bulleted list above.

Deploying the Extraction

Once you deploy caArray, the Java API discussed in this chapter becomes available. The Grid API requires a distinct deployment process in order to become available. For more information, xxxx

Note: Executing the features described in this chapter must be performed programmatically. The functions described in this chapter cannot be performed in the caArray user interface. For extracting data through the user interface, see the *caArray 2.0 User's Guide*, Chapter 7.

To extract data using the API, follow these steps:

1. From one of the external systems in the bulleted list above, execute the login API to login to caArray. Where is the API described if they want to use it?

   Note: To do this, you must have a valid user account. If you want to access protected data in caArray, you must have the appropriate permissions to the private data.

   The External System connects to the remote API, providing the user credentials for access. caArray authenticates the provided credentials.

2. Follow the appropriate procedures in the External System to submit a search for objects of interest in caArray (experiments, samples, hybridizations, etc.). Are the procedures visible and clearly displayed in the External Systems? The search can include data files, such as local text data files (delimited).

   Note: Consult the user documentation for the External System for directions about using that system.

   The External System interrogates and navigates the object graph starting with one of the entities returned in the search, ultimately locating hybridization or other such data of interest.

3. The External System requests the details of the design elements for the array design associated with the hybridization. You zeroed in on hybridization here, but is that all the user is going to look for? How about substituting the word Experiment for hybridization or Experiment details, because then user gets ALL or selected information surrounding an Experiment.

4. The External System requests a subset of the hybridization data for further analysis.

Note: The maximum amount of data that can be returned in a single data retrieval response is 500 MB.

Service Interfaces

caArray Remote Service Interface: Java client applications use this interface to communicate with a running instance of caArray to execute all types of searches. In addition, they can call service methods on domain objects (access backend objects via domain objects packaged in the client.jar).

caArray Grid Service Interface: Non-Java or Java client applications use this interface to communicate with a running instance of caArray to execute all types of searches.

caArray Web Services Interface: Non-Java or Java client applications use this interface to communicate with a running instance of caArray via the web services API (SOAP). The clients can execute only simple search-by-example queries. [Note: Not sure if this interface is a requirement. Not explicitly specified in silver/gold compatibility guidelines, but it *is* present in caArray 1.x, and caCORE-SDK-generated applications provide it too.]

CHAPTER 3

USE CASE VIEW

The use cases represented in *Figure 3.1* contain the functionality that have the greatest impact on the design of the caArray architecture. In brief, the use cases described in this chapter require implementation of mechanisms for security, validation, file management, data storage and retrieval, and API design. Brief descriptions of each of these use cases are provided below as extracted from the model. For information on the complete use-case model see the caArray Use-Case Summary [CAARRAY_UCS]..

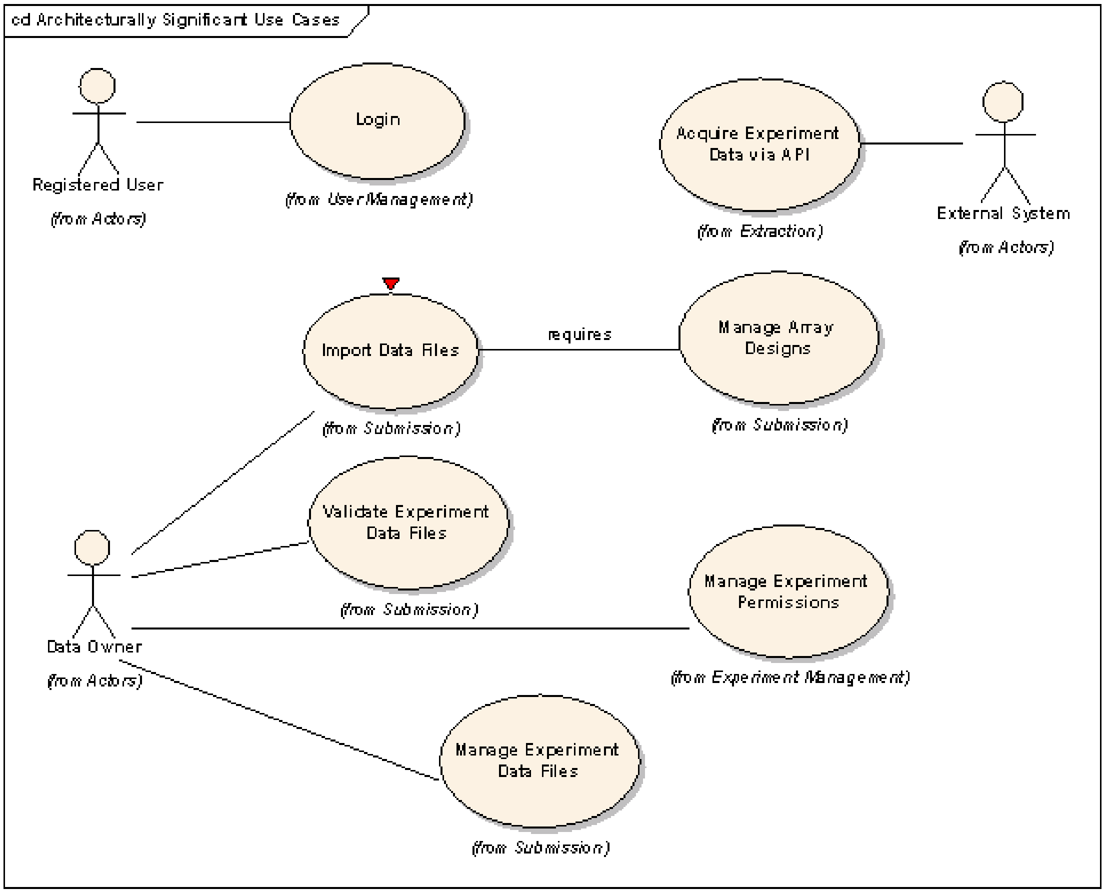


Figure 3.1 Use case summary

caArray Use Cases

Login

Initiated by any Registered User, the login use case allows for the validation of the authenticity and authority of the given user either against a networked (LDAP) set of users or a local set (database). As a result of a successful login, the registered user is presented with their home space and the set of operations they have been granted privileges to perform.

Manage Experiment Data Files

Initiated by a data owner, this use case enables the uploading of annotation and array content collectively and independently into a caArray project and then provides the ability to validate, import and/or delete the uploaded files and records each action taken. Due to the large file size of array data and to a lesser extent, the annotation, the transfer of the file from the client to the server may take minutes or even hours to complete. Therefore, the ability to offer to run the upload in the background (allowing the user to perform other functions inside caArray) is desirable.

Validate Experiment Data Files

Initiated by a data owner, this use case allows for the validation of file structure and content for annotation of array data, with a future intent to import the data into the project. The content validation is not to determine the scientific validity of the information; rather it is to ensure that the data files loaded comply with a pre-defined format for importing into the system. This use case will also be invoked when a data owner chooses to import non-validated data.

Import Experiment Data

Initiated by a data owner, this use case enables the import of annotation and array data from previously-uploaded files.

Manage Array Designs

Initiated by a curator, this use case allows for the uploading of array designs on an as-needed basis. No array designs will be pre-loaded.   This supports the flexibility for any particular organization to upload only the designs they use. This also reduces the overhead introduced by having to load unnecessary designs, which are often of significant size (100's of megabytes or more).

Manage Experiment Permissions

Initiated by a data owner, this use case allows for the promotion of an experiment's visibility. This action can apply to an entire project or to specific samples within the project. The basic visibility states of the project are: restricted, institution, group, public and collaborator. There is no restriction on changing a project's visibility, though it is not advisable once a publication has been published against the project.

Acquire Experiment Data via API

Initiated by an external system, this use case enables programmatic extraction of annotation and array content from caArray using an API. The use case also includes extraction of data from caArray by a grid client through a caGrid service. caBIG Analysis Services is the primary External System targeted and public and protected data should be available provided appropriate authorization is used. The use case may expand to include external systems such as GEO or Array Express or other systems that are local to the installer or NCICB that may have an interest in the data contained in caArray.

CHAPTER 4 LOGICAL VIEW

Overview

The design model (from which the logical view is taken) is the most significant model, requiring the most effort and containing the majority of the content. Accordingly, the description of the logical view of caArray's architecture receives the most attention here. This chapter of the *caArray Technical Guide* first describes the structural hierarchy of the system in layers, packages, and subsystems and then describes how these elements collaborate to provide the most architecturally significant functionality. *Figure 4.1* illustrates the top-level structural organization of caArray. The major dependencies between subsystems are represented as well, though it should be noted that some supporting dependencies have been elided to enhance readability of the diagram.

The only subsystems that are accessible to external systems are the subsystems represented in the Grid API layer and the Remote Java API layer. All subsystems implemented in the Application Logic and Business Logic layers are internal to the application and do not expose remote interfaces. The User Interface layer is accessible to web clients via HTTPS.

caArray is implemented as a J2EE 1.4 application built on top of Java SE 5 (JDK version 1.5.0_10) employing the following core J2EE technologies:

- SP 2.0
- Servlet 2.4
- JMS 1.1
- EJB 3.0.

Only EJB session and message-driven beans are employed; persistence is being managed directly with Hibernate 3.2 rather than EJB's persistence API (JPA). JPA provides no significant advantages over Hibernate at this point, and Hibernate provides additional extended functionality not included in JPA.

Clients of caArray can be characterized as either web UI clients or API clients.

Each of the subsystems shown in *Figure 4.1* is described briefly following the diagram. The functionality of each of these subsystems is described in the following section, *Architecturally Significant Design Elements*, and the context of their use is given in *caarraydb* on page 37 This 5.3 section (caarrayu db) is one very brief paragraph. Are you sure this is the correct reference? which documents the use-case realizations that employ these subsystems.

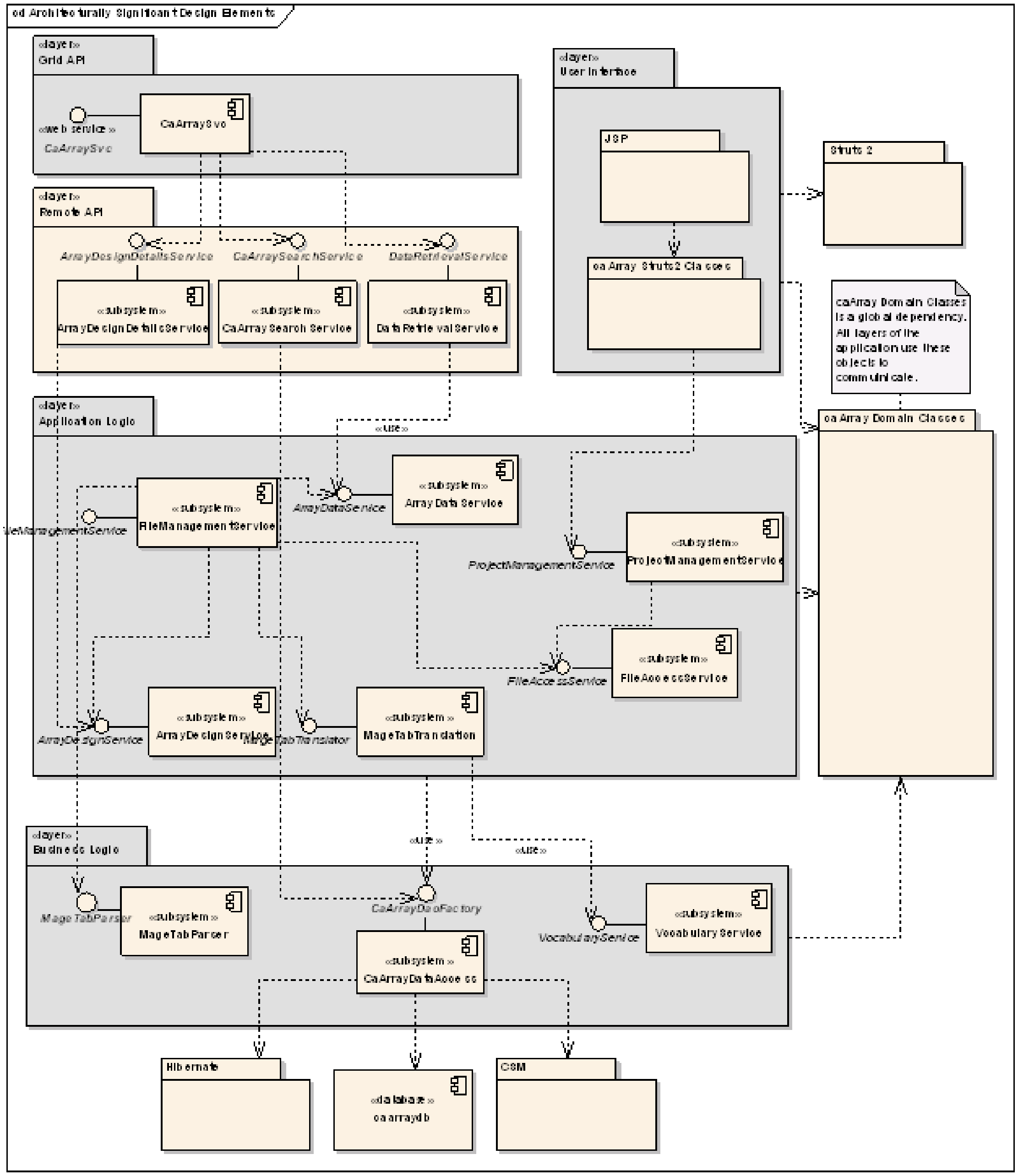


Figure 4.1 Architecturally Significant Design Elements

Architecturally Significant Design Elements

User Interface Layer

The caArray user interface is accessed as a standard web application via HTTPS. It is implemented as a J2EE web application employing Struts 2 as the Model-View-Controller implementation. This layer provides presentation, navigation and validation functionality only. Validation logic at this level is limited to standard form-based validation (for example, checking for appropriate field formats) and is implemented using Strut2 2 validation. Furthermore, a bridge from Struts 2 validation to the Hibernate Validator framework was implemented that allows the definition of these constraints to come straight from the data model. This ensures that the UI enforces the same constraints applied by the underlying storage mechanism. All application logic is implemented in the lower layers of caArray.

The pages presented to the web client use HTML and JavaScript only; no applets or other client-side component technologies are used. Many pages are dynamically updated based on user input without a complete page refresh using Ajax. This allows us to improve responsiveness, implement tabbed interfaces, and improve application usability. Ajax functionality is provided using the Prototype and Scriptaculous JavaScript libraries and the Ajaxtags tag library.

The User Interface layer also includes the login authentication classes CaArrayDBLoginModule and CaArrayDBLoginModule. These classes are used to integrate CSM authentication into the J2EE standard security model, allowing for both database- and LDAP-based authentication. The classes and their relationships to authentication classes from CSM and the Java security API are shown in *Figure 4.2*.

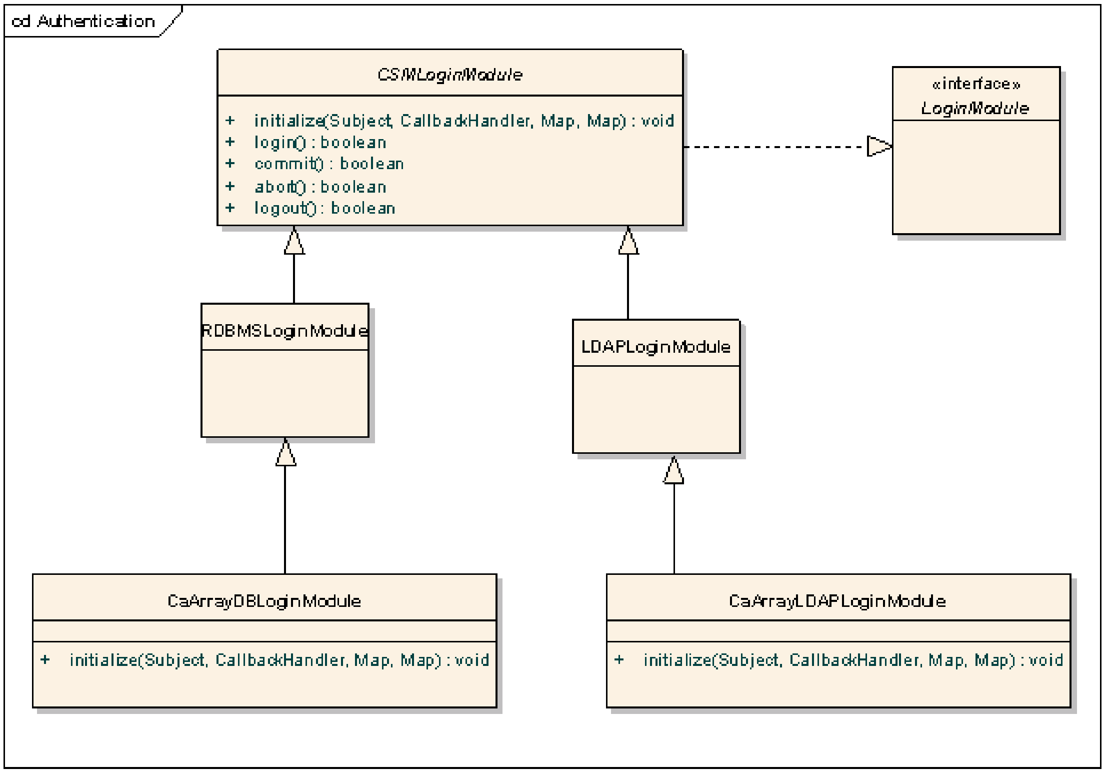


Figure 4.2 Authentication classes

Grid API

The caArray Grid API is a Grid 1.1 compliant data service with several analytical services. The service was created via the Introduce Toolkit, and then modified to improve performance and add additional features. The service connects, via JNDI and RMI, to a running instance of the caArray Remote Java API. The service connects to the web app at startup and uses the remote EJB API to service all requests received from the grid.

The grid service provides both the standard data query (CQLQuery) method and several analytic services. All data in caArray is available via the data service, but optimized data access is available via the analytic services. In particular, access to ArrayDesignDetails is best accomplished via the analytic services. This design choice was driven by the team's experience with caArray 1.x.

To perform CQL searches, the service uses the API method `List<Abstract-CaAraryObject> search(CQLQuery)` exposed by the CaArraySearchService EJB. After passing the CQLQuery to the EJB API, additional transformations are applied to generate a CQLQueryResults object for the grid client. The EJB search API performs the bulk of the work for grid clients. The search method accepts the CQLQuery object and returns matching objects from the domain model, ignoring any query modifiers in the original CQLQuery. caArray uses the CQL2HQL class provided by the core, which is immediately runnable in Hibernate.

Before any object or list of objects is returned, the server performs object graph cutting on the returned objects. This cutting prevents large, fully connected object graphs from being returned to clients and potentially overwhelming network, memory, or other resources. The graph cutting first initializes the root objects and all directly associated objects. Then, for each directly associated object, the associations from *those* objects to their directly associated objects are all set to null. As a result, remote clients, including the grid service itself, receive a limited set of data, and enough information about the dependent objects to continue to fill out the object graph to an arbitrary depth.

The grid service receives the list of matching domain objects from the search API and transforms those results into the CQLQueryResults expected by the grid client. To assist in this translation, caArray utilizes the CQLResultsCreationUtil from the SDK. Depending on query modifiers, the system either (1) translates whole objects, (2) translates [unique] specific properties, or (3) returns the count of objects in the list.

The Introduce generated components include all of the classes in *Figure 4.3***,** with the exception of CaArraySvcImpl and provide the standard marshalling and query functionality of a standard caGrid data service. Delegation to the Java Remote API is handled by the CaArraySvcImpl class, which wraps access to the EJB remote session beans that expose array annotation and data retrieval functionality.

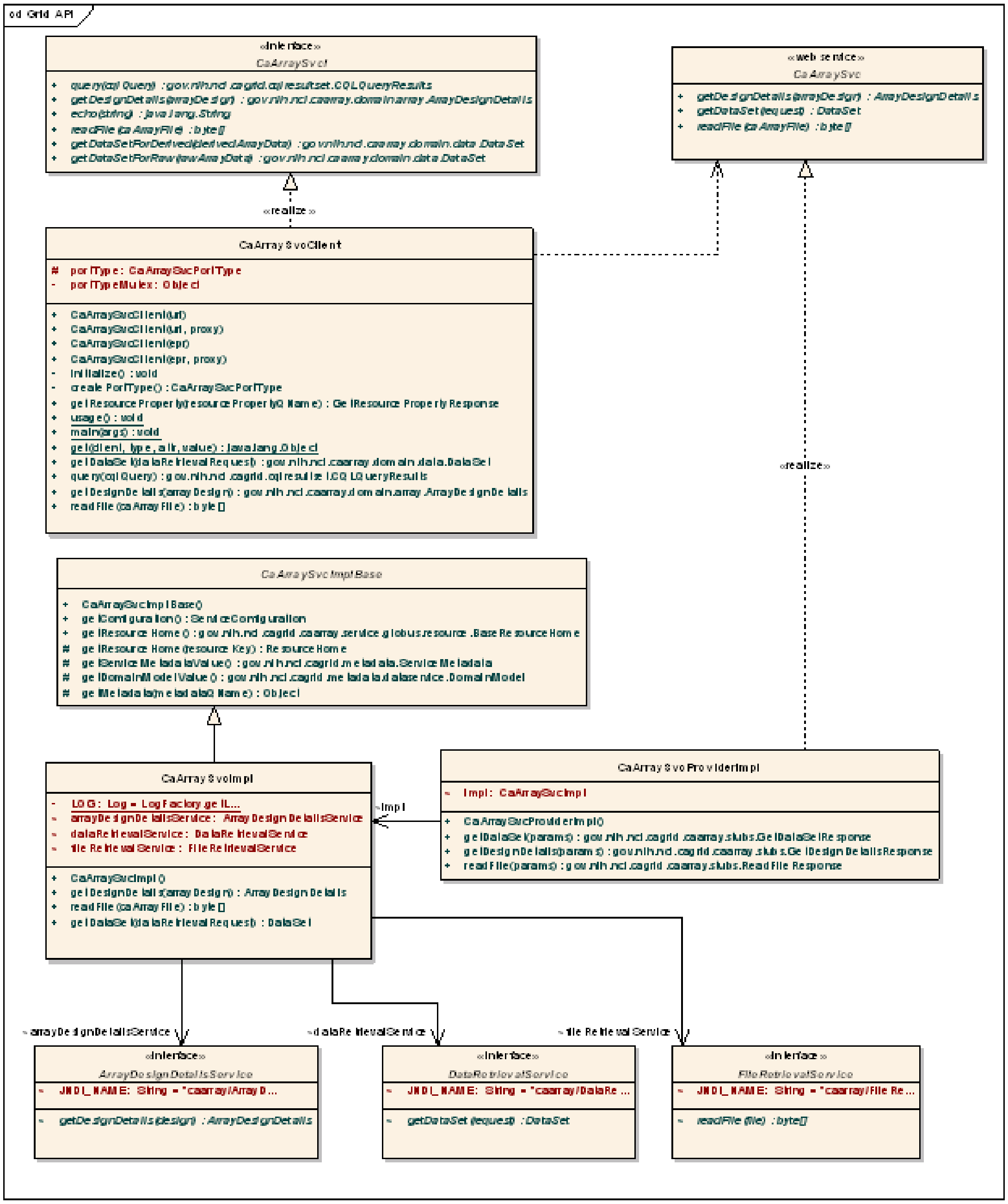

cd Grid API
«interface»
CaArraySvcI
+ query(cqlQuery) : gov.nih.nci.cagrid.cqlresultset.CQLQueryResults
+ getDesignDetails(arrayDesign) : gov.nih.nci.caarray.domain.array.ArrayDesignDetails
+ echo(string) : java.lang.String
+ readFile(caArrayFile) : byte[]
+ getDataSetForDerived(derivedArrayData) : gov.nih.nci.caarray.domain.data.DataSet
+ getDataSetForRaw(rawArrayData) : gov.nih.nci.caarray.domain.data.DataSet
«realize»
CaArraySvcClient
# portType : CaArraySvcPortType
- portTypeMutex : Object
+ CaArraySvcClient(url)
+ CaArraySvcClient(url, proxy)
+ CaArraySvcClient(epr)
+ CaArraySvcClient(epr, proxy)
- initialize() : void
- createPortType() : CaArraySvcPortType
+ getResourceProperty(resourcePropertyQName) : GetResourcePropertyResponse
+ usage() : void
+ main(args) : void
+ get(client, type, attr, value) : java.lang.Object
+ getDataSet(dataRetrievalRequest) : gov.nih.nci.caarray.domain.data.DataSet
+ query(cqlQuery) : gov.nih.nci.cagrid.cqlresultset.CQLQueryResults
+ getDesignDetails(arrayDesign) : gov.nih.nci.caarray.domain.array.ArrayDesignDetails
+ readFile(caArrayFile) : byte[]
«web service»
CaArraySvc
+ getDesignDetails(arrayDesign) : ArrayDesignDetails
+ getDataSet(request) : DataSet
+ readFile(caArrayFile) : byte[]
«realize»
CaArraySvcImplBase
+ CaArraySvcImplBase()
+ getConfiguration() : ServiceConfiguration
+ getResourceHome() : gov.nih.nci.cagrid.caarray.service.globus.resource.BaseResourceHome
# getResourceHome(resourceKey) : ResourceHome
# getServiceMetadataValue() : gov.nih.nci.cagrid.metadata.ServiceMetadata
# getDomainModelValue() : gov.nih.nci.cagrid.metadata.dataservice.DomainModel
# getMetadata(metadataQName) : Object
CaArraySvcImpl
- LOG : Log = LogFactory.getL...
~ arrayDesignDetailsService : ArrayDesignDetailsService
~ dataRetrievalService : DataRetrievalService
~ fileRetrievalService : FileRetrievalService
+ CaArraySvcImpl()
+ getDesignDetails(arrayDesign) : ArrayDesignDetails
+ readFile(caArrayFile) : byte[]
+ getDataSet(dataRetrievalRequest) : DataSet
~impl
CaArraySvcProviderImpl
~ impl : CaArraySvcImpl
+ CaArraySvcProviderImpl()
+ getDataSet(params) : gov.nih.nci.cagrid.caarray.stubs.GetDataSetResponse
+ getDesignDetails(params) : gov.nih.nci.cagrid.caarray.stubs.GetDesignDetailsResponse
+ readFile(params) : gov.nih.nci.cagrid.caarray.stubs.ReadFileResponse
~arrayDesignDetailsService
~dataRetrievalService
~fileRetrievalService
«interface»
ArrayDesignDetailsService
~ JNDI_NAME : String = "caarray/ArrayD...
~ getDesignDetails(design) : ArrayDesignDetails
«interface»
DataRetrievalService
~ JNDI_NAME : String = "caarray/DataRe...
~ getDataSet(request) : DataSet
«interface»
FileRetrievalService
~ JNDI_NAME : String = "caarray/FileRe...
~ readFile(file) : byte[]

Figure 4.3 CaArraySvc Grid API implementation

Remote Java API

The Remote Java API is implemented as a façade (the CaArrayServer class) representing a connection to caArray and a set of several stateless session EJBs with remote visibility. Clients instantiate a CaArrayServer instance, call the connect method and can then access the session EJB interfaces through accessor methods exposed by the CaArrayServer. These EJBs provide simplified, efficient access to caArray entities and data. Special consideration was given to the DataRetrievalService API to enable clients to retrieve only the data they require. Clients may select data for specific QuantitationTypes, Hybridizations, and AbstractDesignElements by configuring a DataRetrievalRequest object and passing it as an argument to the getDataSet() method. The remote interfaces and their exposed operations are shown in the class diagram provided in *Figure 4.4*.

As discussed above, each remote Java API method performs object graph cutting to minimize data transmissions. The DataRetrievalService's cutting is more sophisticated: instead of performing cutting at the child object level, all information about the DataSet is returned in a single request.

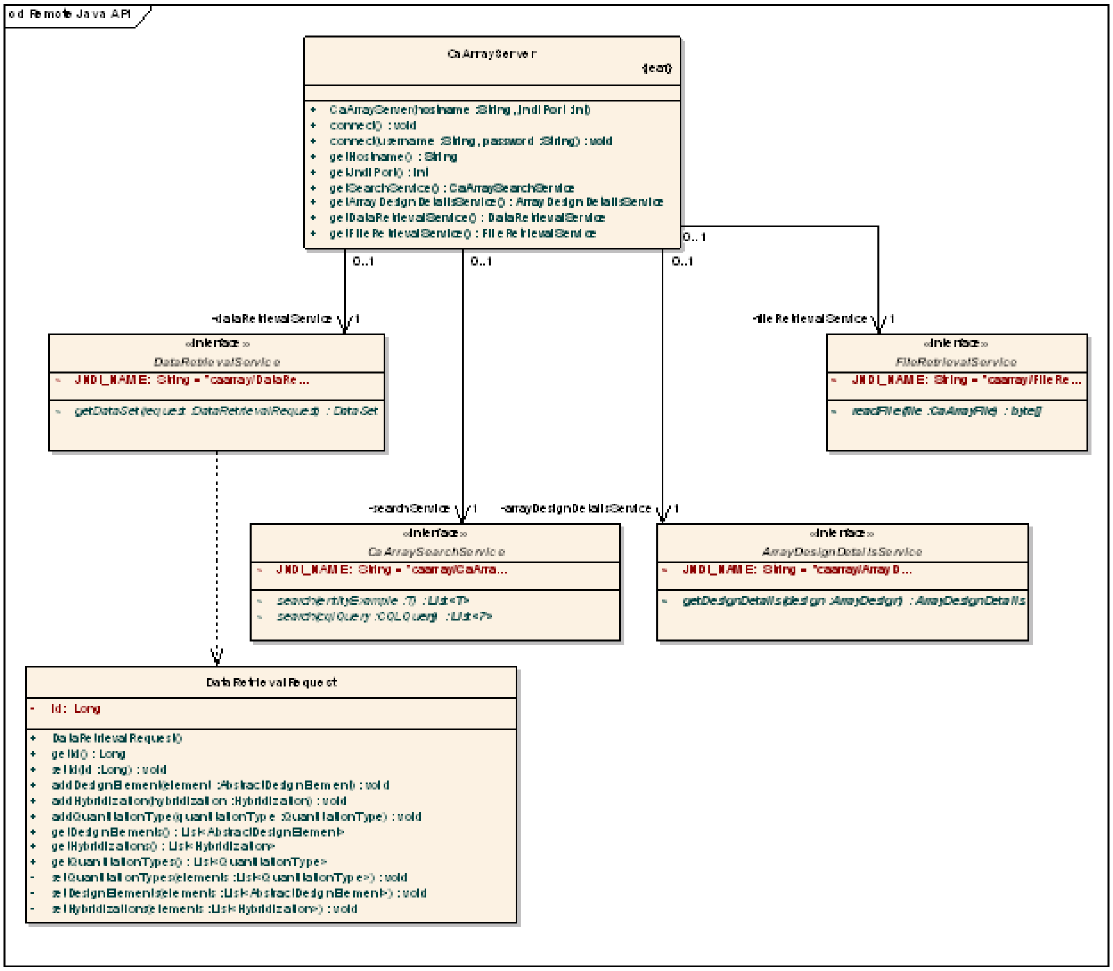

cd Remote Java API
CaArrayServer
{leaf}
+ CaArrayServer(hostname :String, jndiPort :int)
+ connect() : void
+ connect(username :String, password :String) : void
+ getHostname() : String
+ getJndiPort() : int
+ getSearchService() : CaArraySearchService
+ getArrayDesignDetailsService() : ArrayDesignDetailsService
+ getDataRetrievalService() : DataRetrievalService
+ getFileRetrievalService() : FileRetrievalService
0..1
0..1
0..1
0..1
-dataRetrievalService 1
-fileRetrievalService 1
«interface»
DataRetrievalService
~ JNDI_NAME: String = "caarray/DataRe...
~ getDataSet(request :DataRetrievalRequest) : DataSet
«interface»
FileRetrievalService
~ JNDI_NAME: String = "caarray/FileRe...
~ readFile(file :CaArrayFile) : byte[]
-searchService 1
-arrayDesignDetailsService 1
«interface»
CaArraySearchService
~ JNDI_NAME: String = "caarray/CaArra...
~ search(entityExample :T) : List<T>
~ search(cqlQuery :CQLQuery) : List&lt;?&gt;
«interface»
ArrayDesignDetailsService
~ JNDI_NAME: String = "caarray/ArrayD...
~ getDesignDetails(design :ArrayDesign) : ArrayDesignDetails
DataRetrievalRequest
- id : Long
+ DataRetrievalRequest()
+ getId() : Long
+ setId(id :Long) : void
+ addDesignElement(element :AbstractDesignElement) : void
+ addHybridization(hybridization :Hybridization) : void
+ addQuantitationType(quantitationType :QuantitationType) : void
+ getDesignElements() : List<AbstractDesignElement>
+ getHybridizations() : List<Hybridization>
+ getQuantitationTypes() : List<QuantitationType>
- setQuantitationTypes(elements :List<QuantitationType>) : void
- setDesignElements(elements :List<AbstractDesignElement>) : void
- setHybridizations(elements :List<Hybridization>) : void

Figure 4.4 caArray Remote Java API

caArray Domain Classes

This section describes the classes used to model the microarray experiment and data that caArray is designed to manage. Since these classes are employed by all of the caArray subsystems and also must be understood by remote caArray clients (for example, Java RMI clients[1]). Classes that represent important data constructs are given detailed description here.

[1] Grid clients do not need the domain classes, since our domain model is registered in caDSR. However, grid clients that do use our domain classes can make user of our Castor XML mapping classes.

The underlying business object model is implemented as a set of POJOs that model the domain of microarray experiments and data. Whereas earlier versions of caArray used MAGE-OM 1.1 as the basis for the underlying object and data model, the new caArray implementation is based on a completely revised, simplified object model. Although MAGE-OM is a published standard, there are significant disadvantages in using it as an underlying object model; it is complicated to understand, its complexity makes data storage inefficient, its structure does not permit useful object graph navigations, and many common relationships cannot be stored when complete experiment annotation is not available. For these reasons, we have chosen to produce a new, simplified object model for domain data representation. caCORE was used for the initial generation of these POJOs.

The domain classes are principally designed to support the entity model described by the MAGE-TAB 1.0 specification. The underlying object model described by MAGE-TAB is considerably more understandable than MAGE-OM while still providing a complete enough model to support MIAME compliance. The most central entities in the domain model are pictured in *Figure 4.5*.

Annotation for array design elements are represented by a hierarchy of annotation classes based on the array design type. Each array design element that reports on a biological sequence is related to an instance of AbstractProbeAnnotation. The hierarchy of annotation information is shown in *Figure 4.6*

Figure 4.7

As has been noted earlier, array data needs to be represented in way that allows for efficient storage and transport when required by remote clients. The classes used to represent array data are shown in *Figure 4.8*. caArray is designed to represent array data at two levels:

- The `AbstractArrayData` hierarchy represents individual data files that have been imported into caArray, describing their type and relationships to hybridizations. These are high level representations that do not contain the actual data values.
- The `DataSet` class and the classes it is related to by composition (`HybridizationData` and the `AbstractDataColumn` hierarchy). These classes ultimately contain the array data values, specifically as arrays of primitive or string values within the `AbstractDataColumn subclasses.`

The DataSet classes are used both to persist the data contained in array data files and as a container for custom data sets requested by clients. As an example, a given Affymetrix CEL file imported into the system will have a single persistent DataSet containing a single persistent HybridizationData instance that contains several AbstractDataColumn instances (IntegerColumns for CELX and CELY, FloatColumn for CELIntensity, etc.). If a remote API client requests the data for all CEL files within an experiment, a transient, compound DataSet is created that contains multiple HybridizationDatas where each HybridizationData is retrieved from persistent storage.

A columnar approach to data representation allows for efficient retrieval and storage when compared with a row-based representation. This represents a significant change from caArray 1.x where an entire BioDataCube must be retrieved in its entirety, allowing for a significant reduction of network transfer overhead. This columnar approach is preferable for two reasons:

1. Array data files typically contain relatively few columns but a large number of rows, typically in the tens of thousands or larger. When returning data to remote clients, it is far more efficient to serialize a large array of primitives when compared to returning a large object graph.
2. Clients typically require only a small subset of the columns represented by an array data file, so organizing data by column allows for much more efficient custom DataSet assembly. Clients may indicate which columns to select by specifying QuantitationTypes to retrieve. The semantics of the various QuantitationTypes will be registered in caDSR to make them meaningful and comparable to clients that don't have advance knowledge of the context of specific QuantitationTypes.

In addition to efficient storage and transfer, this approach is also intended to meet the needs of caB2B and other tools that need everything navigable in the model (for example, require the domain model semantics -- aren't aware of the data retrieval API). Making the columns themselves persistent with their data allows these clients to navigate to the raw data values while we still retain an efficient mechanism for storage and retrieval (the columns' compressed, serialized value arrays are transparently expanded on request).

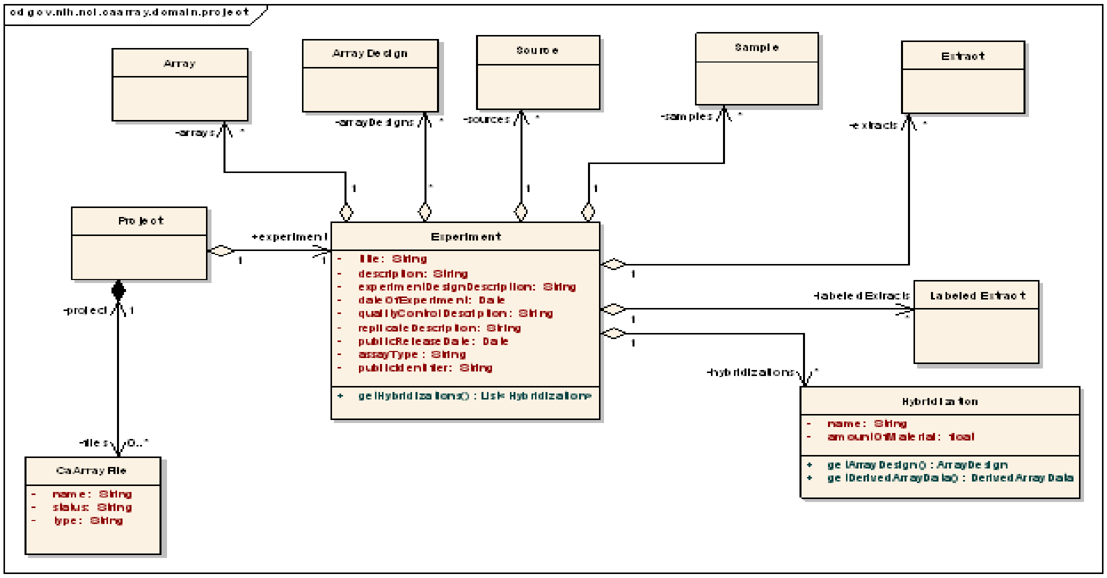


Figure 4.5 Experiment Overview

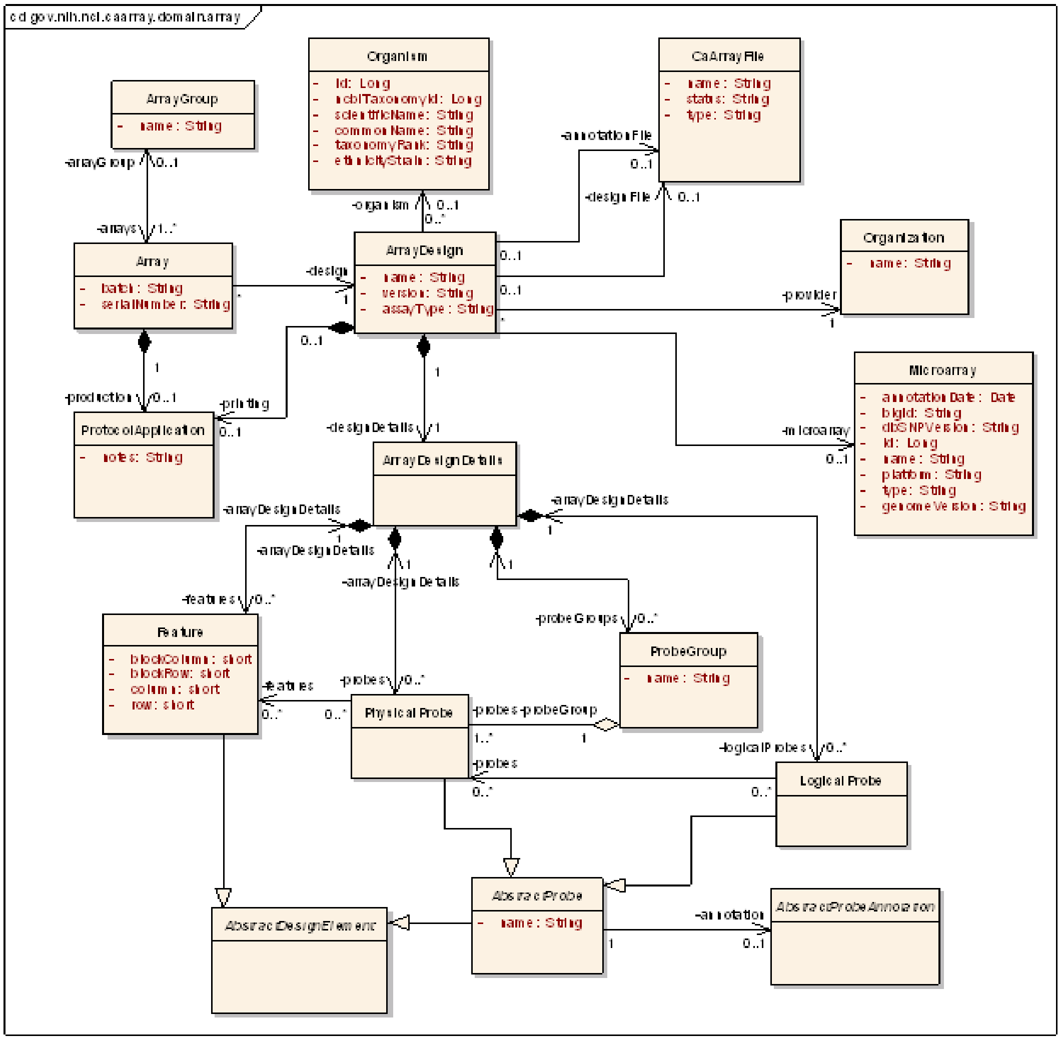


Figure 4.6 caArray Array Design Classes

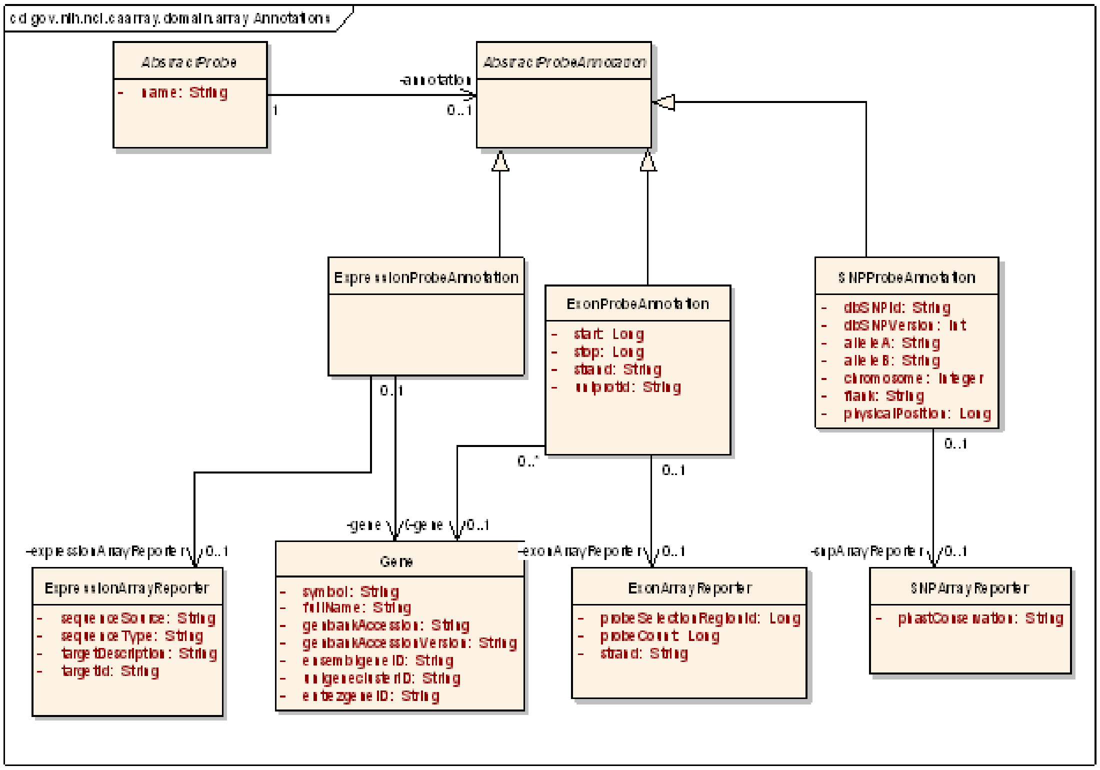


Figure 4.7 caBIO Array Reporter Annotation Object Model

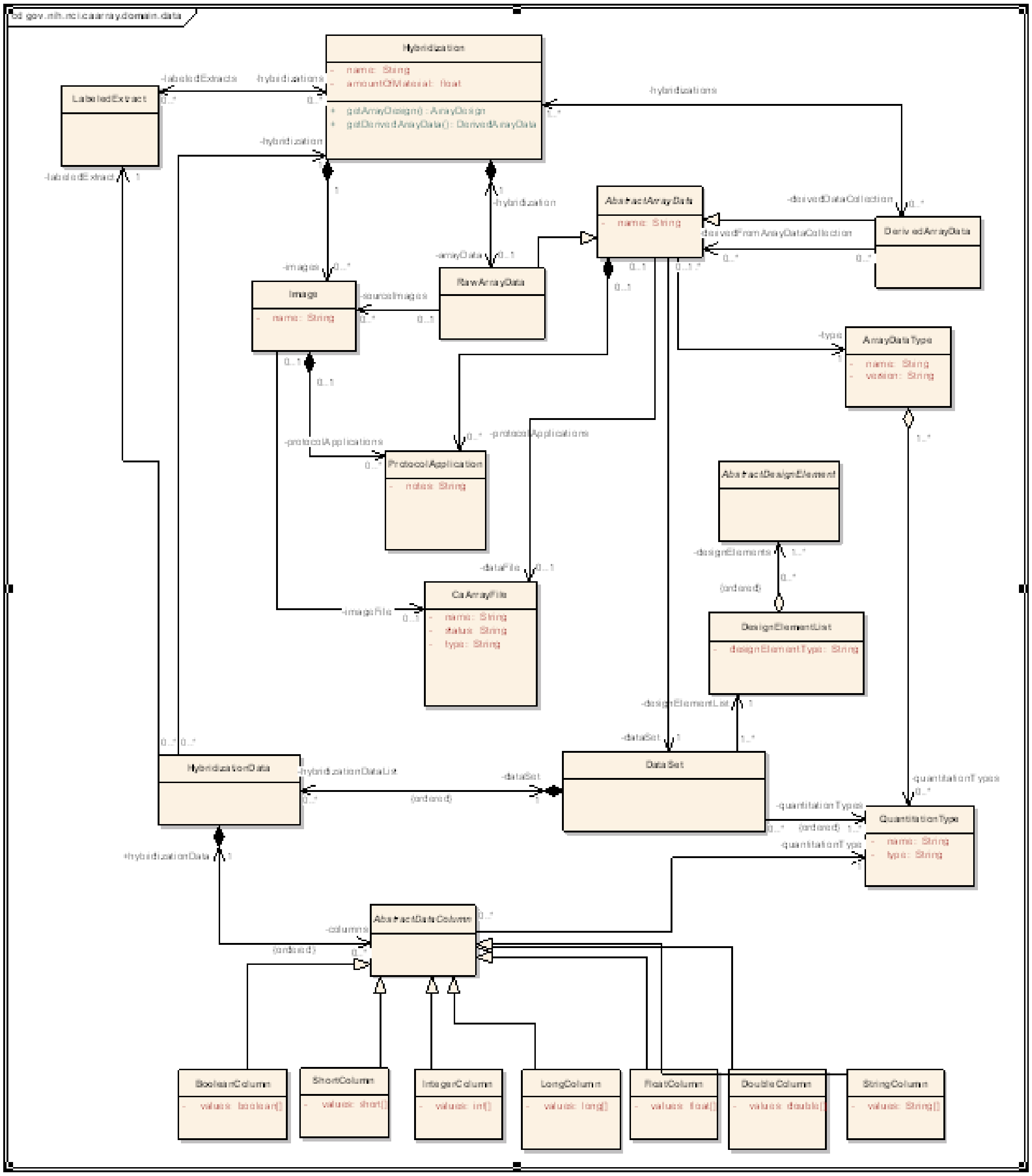


Figure 4.8 Array Data Classes

Access Control using CSM 4.0

caArray allows experiment owners to define fine grained access constraints on both whole experiments and individual samples (and the BioSources and Hybridization data derived from those samples). By default, non-experiment owners (including anonymous, non-logged in users) have access to a small set of overview information about an experiment. Read access to experiments and/or samples can be granted to the public as well as to defined groups of users (known as collaboration groups). The collaboration groups can also be granted write permissions to experiments and/or individual samples. Finally, an experiment can be removed from visibility entirely, making it completely inaccessible to users who have not been granted special permissions as described above.

This permissions system is implemented via integration with CSM 4.0. CSM provides a rich, fine grained domain model for expressing security constraints, including instance and attribute level security. The concepts described above map nicely onto the classes available in CSM in a very natural way. The architecturally interesting points about the integration, described below, involve synchronization between the caArray and CSM data models and the enforcement of the security constraints defined in the model.

Synchronizing the caArray and CSM data models requires creation and modification of CSM data structures expressing appropriate security constraints in response to corresponding operations on the caArray data model. This is accomplished via `SecurityInterceptor`, which takes advantage of a Hibernate API that allows application code to respond to Hibernate lifecycle events. `SecurityInterceptor` detects creation, modification and deletion of caArray domain objects and in response creates or modifies the CSM data structures which store security constraints on those objects.

Enforcement of the security constraints is done in two ways. Hibernate filters are used to enforce read permissions and visibility control for experiments. The filters are defined for any caArray domain classes which are covered by the security model, and act as essentially additional `WHERE` clauses that limit any queries against those classes to instances to which the user has access. These filters are applied transparently by Hibernate, and are automatically parameterized by CSM with the current user. This provides for a clean separation of concerns, as business logic can be written without the clutter of security considerations.

To enforce write permissions, we instead use the API provided by CSM's `AuthorizationManager` class. The logic for doing so is centralized in the `SecurityUtils` class.

It is important to note that security constraints are checked twice. First, during display of data, security constraints are checked to determine whether to display certain user interface elements. For instance, on the Work Queue page, the edit link is only displayed for an experiment if the current user has write permissions to the experiment. Second, security constraints are checked and enforced before any actual operations against protected data are performed. This ensures that the user interface shows the user only the actions they have permissions to perform, but still enforces those permissions if a malicious user circumvents the normal user interface (for instance by URL hacking).

ProjectManagementService

The ProjectManagementService subsystem is implemented as a façade to allow the user interface to create and retrieve experiments. The implementation of this subsystem delegates directly to the CaArrayDataAccess service for entity management and to the FileAccessService for file management. The subsystem contents are shown in *Figure 4.9*.

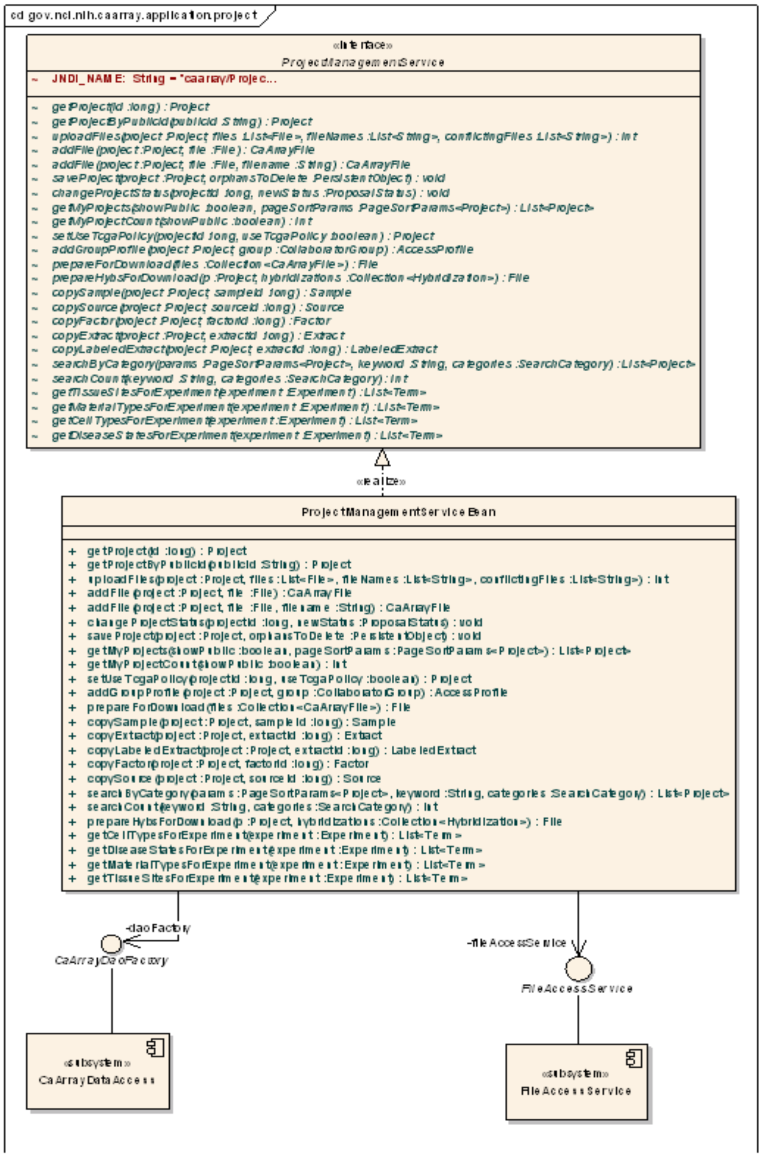

cd gov.nci.nih.caarray.application.project
«interface»
ProjectManagementService
~ JNDI_NAME: String = "caarray/Projec...
~ getProject(id :long) : Project
~ getProjectByPublicId(publicId :String) : Project
~ uploadFiles(project :Project, files :List<File>, fileNames :List<String>, conflictingFiles :List<String>) : int
~ addFile(project :Project, file :File) : CaArrayFile
~ addFile(project :Project, file :File, filename :String) : CaArrayFile
~ saveProject(project :Project, orphansToDelete :PersistentObject) : void
~ changeProjectStatus(projectId :long, newStatus :ProposalStatus) : void
~ getMyProjects(showPublic :boolean, pageSortParams :PageSortParams<Project>) : List<Project>
~ getMyProjectCount(showPublic :boolean) : int
~ setUseTcgaPolicy(projectId :long, useTcgaPolicy :boolean) : Project
~ addGroupProfile(project :Project, group :CollaboratorGroup) : AccessProfile
~ prepareForDownload(files :Collection<CaArrayFile>) : File
~ prepareHybsForDownload(p :Project, hybridizations :Collection<Hybridization>) : File
~ copySample(project :Project, sampleId :long) : Sample
~ copySource(project :Project, sourceId :long) : Source
~ copyFactor(project :Project, factorId :long) : Factor
~ copyExtract(project :Project, extractId :long) : Extract
~ copyLabeledExtract(project :Project, extractId :long) : LabeledExtract
~ searchByCategory(params :PageSortParams<Project>, keyword :String, categories :SearchCategory) : List<Project>
~ searchCount(keyword :String, categories :SearchCategory) : int
~ getTissueSitesForExperiment(experiment :Experiment) : List<Term>
~ getMaterialTypesForExperiment(experiment :Experiment) : List<Term>
~ getCellTypesForExperiment(experiment :Experiment) : List<Term>
~ getDiseaseStatesForExperiment(experiment :Experiment) : List<Term>
«realize»
ProjectManagementServiceBean
+ getProject(id :long) : Project
+ getProjectByPublicId(publicId :String) : Project
+ uploadFiles(project :Project, files :List<File>, fileNames :List<String>, conflictingFiles :List<String>) : int
+ addFile(project :Project, file :File) : CaArrayFile
+ addFile(project :Project, file :File, filename :String) : CaArrayFile
+ changeProjectStatus(projectId :long, newStatus :ProposalStatus) : void
+ saveProject(project :Project, orphansToDelete :PersistentObject) : void
+ getMyProjects(showPublic :boolean, pageSortParams :PageSortParams<Project>) : List<Project>
+ getMyProjectCount(showPublic :boolean) : int
+ setUseTcgaPolicy(projectId :long, useTcgaPolicy :boolean) : Project
+ addGroupProfile(project :Project, group :CollaboratorGroup) : AccessProfile
+ prepareForDownload(files :Collection<CaArrayFile>) : File
+ copySample(project :Project, sampleId :long) : Sample
+ copyExtract(project :Project, extractId :long) : Extract
+ copyLabeledExtract(project :Project, extractId :long) : LabeledExtract
+ copyFactor(project :Project, factorId :long) : Factor
+ copySource(project :Project, sourceId :long) : Source
+ searchByCategory(params :PageSortParams<Project>, keyword :String, categories :SearchCategory) : List<Project>
+ searchCount(keyword :String, categories :SearchCategory) : int
+ prepareHybsForDownload(p :Project, hybridizations :Collection<Hybridization>) : File
+ getCellTypesForExperiment(experiment :Experiment) : List<Term>
+ getDiseaseStatesForExperiment(experiment :Experiment) : List<Term>
+ getMaterialTypesForExperiment(experiment :Experiment) : List<Term>
+ getTissueSitesForExperiment(experiment :Experiment) : List<Term>
-daoFactory
CaArrayDaoFactory
-fileAccessService
FileAccessService
«subsystem»
CaArrayDataAccess
«subsystem»
FileAccessService

Figure 4.9 ProjectManagementService implementation

FileAccessService

The FileAccessService subsystem is implemented as a stateful session bean and is responsible for storage of all files managed within caArray (annotation, array design and data). Files that are uploaded to caArray are registered with the FileAccessService which reads the files, compresses the contents and stores the contents as BLOB(s) (in the database) associated with a CaArrayFile instance. Due to limitations in MySQL when storing very large blobs (>250MB), caArray breaks very large files into multiple blobs for storage in the database. The storage of multiple blobs is transparent to users of the CaArrayFile class.

File retrieval is performed through the `TemporaryFileCache` interface. The implementation of this interface is stored in a `ThreadLocal` and maintains a `Map` of opened files so that any given client only needs to retrieve file contents from the database once per overall client transaction. Client subsystems that require access to file contents call the `getFile(caArrayFile : CaArrayFile) : File` method which performs the inverse operation; reading the contents from the BLOB(s), decompressing the contents and writing them to temporary file reading area. Clients are expected to call `closeFile(file : File)` when done using files so that the subsystem can remove the files from temporary file system storage, but the subsystem also performs clean up when it is finalized. The static structure of the `FileAccessService` subsystem is shown in *Figure 4.10* and the act of storing file contents is shown in *Figure 4.11*..

This `TemporaryFileCache` implementation extracts each requested file to a different temporary location. This does mean potentially having duplicates of temporarily uncompressed files, but this should be the exception as files are only needed on download and when parsed. After weighing the potential approaches, the minor overhead of temporary duplicates was definitely preferable to the overhead of maintaining file reference counters across multiple sessions.

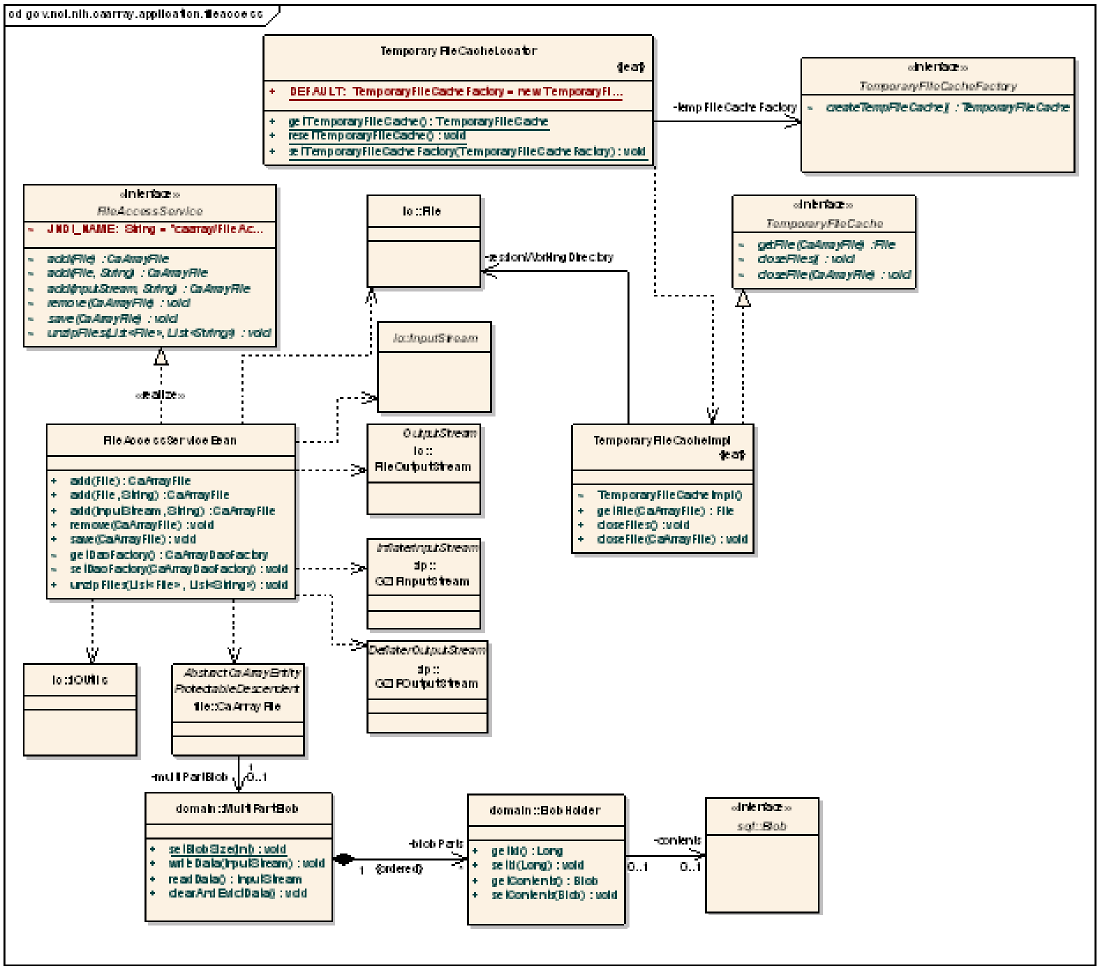
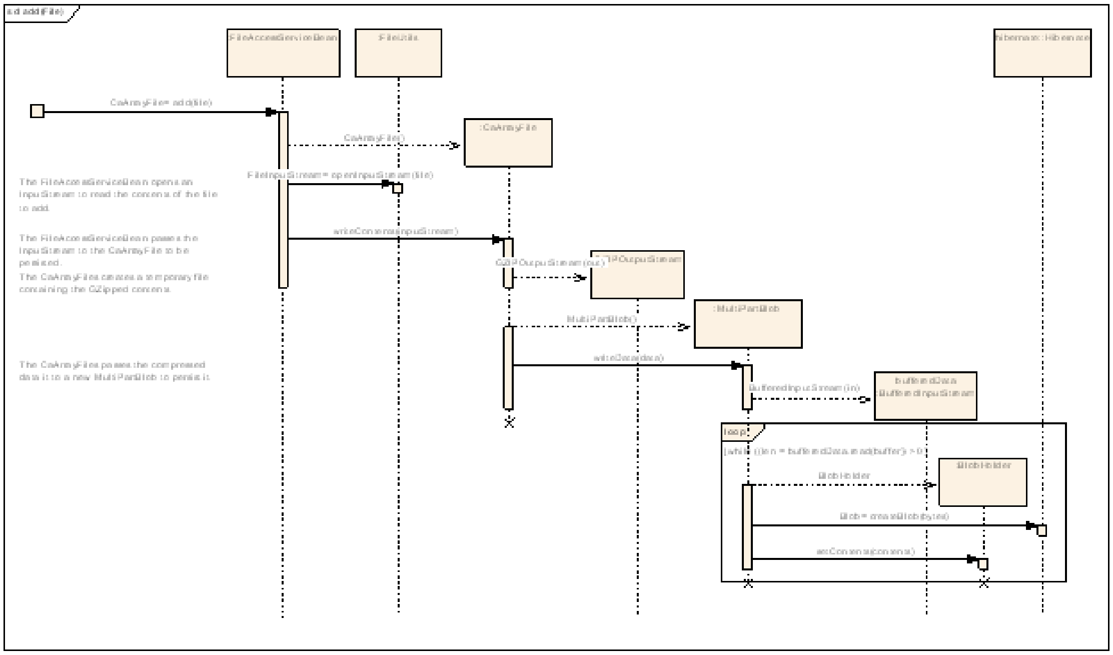

cd gov.nci.nih.caarray.application.fileaccess
TemporaryFileCacheLocator
{leaf}
+ DEFAULT: TemporaryFileCacheFactory = new TemporaryFi...
+ getTemporaryFileCache() : TemporaryFileCache
+ resetTemporaryFileCache() : void
+ setTemporaryFileCacheFactory(TemporaryFileCacheFactory) : void
-tempFileCacheFactory
«interface»
TemporaryFileCacheFactory
~ createTempFileCache() : TemporaryFileCache
«interface»
FileAccessService
~ JNDI_NAME: String = "caarray/FileAc...
~ add(File) : CaArrayFile
~ add(File, String) : CaArrayFile
~ add(InputStream, String) : CaArrayFile
~ remove(CaArrayFile) : void
~ save(CaArrayFile) : void
~ unzipFiles(List<File>, List<String>) : void
io::File
-sessionWorkingDirectory
«interface»
TemporaryFileCache
~ getFile(CaArrayFile) : File
~ closeFiles() : void
~ closeFile(CaArrayFile) : void
io::InputStream
«realize»
FileAccessServiceBean
+ add(File) : CaArrayFile
+ add(File, String) : CaArrayFile
+ add(InputStream, String) : CaArrayFile
+ remove(CaArrayFile) : void
+ save(CaArrayFile) : void
~ getDaoFactory() : CaArrayDaoFactory
~ setDaoFactory(CaArrayDaoFactory) : void
+ unzipFiles(List<File>, List<String>) : void
OutputStream
io::
FileOutputStream
TemporaryFileCacheImpl
{leaf}
~ TemporaryFileCacheImpl()
+ getFile(CaArrayFile) : File
+ closeFiles() : void
+ closeFile(CaArrayFile) : void
InflaterInputStream
zip::
GZIPInputStream
DeflaterOutputStream
zip::
GZIPOutputStream
io::IOUtils
AbstractCaArrayEntity
ProtectableDescendent
file::CaArrayFile
-multiPartBlob
1
0..1
domain::MultiPartBlob
+ setBlobSize(int) : void
+ writeData(InputStream) : void
+ readData() : InputStream
+ clearAndExistData() : void
-blobParts
1
{ordered}
domain::BlobHolder
+ getId() : Long
+ setId(Long) : void
+ getContents() : Blob
+ setContents(Blob) : void
-contents
0..1
0..1
«interface»
sql::Blob

Figure 4.10 FileAccessService implementation class diagram



The FileAccessServiceBean opens an inputStream to read the contents of the file to add.

The FileAccessServiceBean passes the inputStream to the CaArrayFile to be persisted.
The CaArrayFiles creates a temporary file containing the GZipped contents.

The CaArrayFiles passes the compressed data it to a new MultiPartBlob to persist it.

Figure 4.11 FileAccessService operation AddFile(file : File)

FileManagementService

Whereas the FileAccessService handles the lower level functionality of file storage and retrieval, the FileManagementService subsystem is responsible for performing higher level logical file operations, specifically, the validation and import of MAGE-TAB annotation, array design files and array data files. The implementation of the subsystem does this through delegation to subsystems responsible for handling these various types of data. The organization of the FileManagementService implementation is shown in *Figure 4.12* where the central bean delegates import and validation functionality to a set of importer classes that in turn delegate to the lower-level subsystems.

Validation results are instantiated by the lower-level subsystems and then the FileManagementService associates these with the CaArrayFile object that represents the validated annotation or data file.

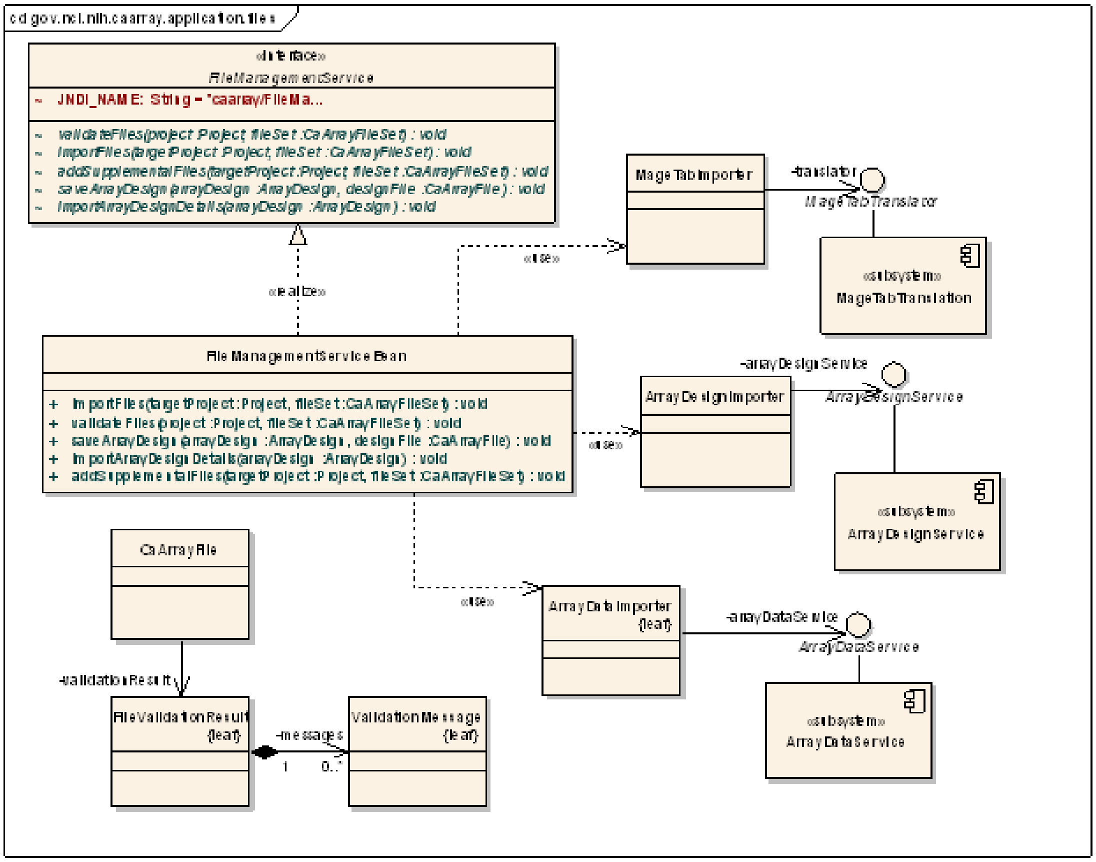

cd gov.nci.nih.caarray.application.file
«interface»
FileManagementService
~ JNDI_NAME: String = "caarray/FileMa...
~ validateFiles(project :Project, fileSet :CaArrayFileSet) : void
~ importFiles(targetProject :Project, fileSet :CaArrayFileSet) : void
~ addSupplementalFiles(targetProject :Project, fileSet :CaArrayFileSet) : void
~ saveArrayDesign(arrayDesign :ArrayDesign, designFile :CaArrayFile) : void
~ importArrayDesignDetails(arrayDesign :ArrayDesign) : void
«realize»
«use»
MageTabImporter
-translator
MageTabTranslator
«subsystem»
MageTabTranslation
FileManagementServiceBean
+ importFiles(targetProject :Project, fileSet :CaArrayFileSet) : void
+ validateFiles(project :Project, fileSet :CaArrayFileSet) : void
+ saveArrayDesign(arrayDesign :ArrayDesign, designFile :CaArrayFile) : void
+ importArrayDesignDetails(arrayDesign :ArrayDesign) : void
+ addSupplementalFiles(targetProject :Project, fileSet :CaArrayFileSet) : void
-arrayDesignService
ArrayDesignImporter
«use»
ArrayDesignService
«subsystem»
ArrayDesignService
CaArrayFile
«use»
ArrayDataImporter
{leaf}
-arrayDataService
ArrayDataService
«subsystem»
ArrayDataService
-validationResult
FileValidationResult
{leaf}
-messages
1
0..*
ValidationMessage
{leaf}

Figure 4.12 FileManagementService implementation

ArrayDataService

The ArrayDataService subsystem is responsible for validating array data files, storing array data and retrieving array data when requested by clients. The typical order of events related to a given array data file is as follows:

- An array data file is validated using the `validate(arrayDataFile : CaArrayDataFile) : ValidationResult` operation. Only the generated FileValidationResult is created and persisted.
- The data file is imported using the `import(arrayData : AbstractArrayData) : void` operation. At this point, a DataSet and associated HybridizationData and AbstractDataColumn instances are created, but individual data values are not retrieved or persisted.
- A client requests data via the `getData(arrayData : AbstractArrayData, types : List<QuantitationType>) : DataSet` method. If this is the first request for the AbstractDataColumns associated with the provided QuantitationTypes, the requested data is parsed from the files as columnar arrays of primitives and stored persistently as serialized, GZipped byte[] representations of the arrays. The data is then returned to the client. If the data had

previously been loaded as a result of earlier calls to getData the existing serialized byte[] is deserialized[1].

Though caArray 1.x and 2.0 do ultimately use the database to persist array data, the approaches are radically different. The 2.0 design does not exhibit the same performance and resource consumption when compared to 1.x. For illustration purposes, the 1.x design stores the array data from a file as a large number of rows (one per design element) with a column per data value, maintaining a complete relational representation of the entire data set. caArray 2.0 uses a single BLOB entry to store a large primitive array of data corresponding to a complete column's worth of data from a data file. For example, whereas a CEL file consumes hundreds of thousands of rows of seven columns apiece in 1.x, the new design creates 7 rows, each with a single serialized, compressed representation of hundreds of thousands of data points.

[1] The delayed parsing of the data sets was deferred from the initial release of caArray 2.0. The data sets are stored as part of the import process. After delayed parsing is implemented, additional space savings will be achieved; only requested columns of data will be parsed and persisted. This is important since only a small subset of the quantitation types of a given data type are of interest to clients (for example, 2 columns out of 20 such as "detection' and "chip signal" of a .chp file in an Affymetrix experiment).

The method import(arrayData : AbstractArrayData) : void is illustrated in *Figure 4.13* .

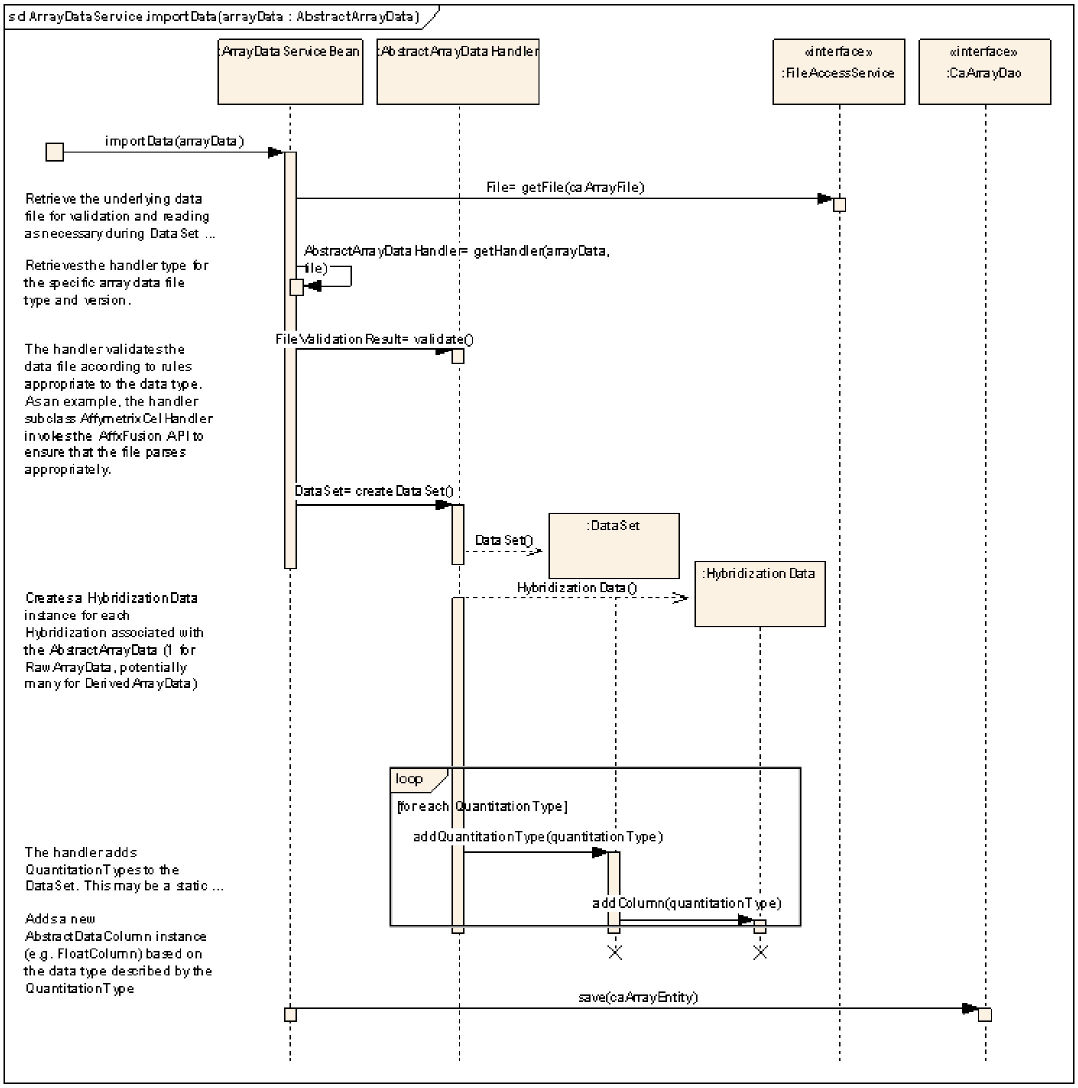

sd ArrayDataService.importData(arrayData : AbstractArrayData)
:ArrayData Service Bean
:AbstractArrayData Handler
«interface»
:FileAccessService
«interface»
:CaArrayDao
importData(arrayData)
File= getFile(caArrayFile)
Retrieve the underlying data file for validation and reading as necessary during Data Set ...
Retrieves the handler type for the specific array data file type and version.
AbstractArrayData Handler= getHandler(arrayData, file)
FileValidation Result= validate()
The handler validates the data file according to rules appropriate to the data type. As an example, the handler subclass AffymetrixCel Handler invokes the AffxFusion API to ensure that the file parses appropriately.
DataSet= create Data Set()
:DataSet
Data Set()
:Hybridization Data
Hybridization Data()
Creates a HybridizationData instance for each Hybridization associated with the AbstractArrayData (1 for RawArrayData, potentially many for Derived ArrayData)
loop
[for each QuantitationType]
addQuantitationType(quantitationType)
addColumn(quantitationType)
The handler adds QuantitationTypes to the DataSet. This may be a static ...
Adds a new AbstractDataColumn instance (e.g. FloatColumn) based on the data type described by the QuantitationType
save(caArrayEntity)

Figure 4.13 Import Operation implementation

ArrayDesignService

The ArrayDesignService is responsible for parsing, persisting, and retrieving array design annotation from the various array annotation file types. In the implementation of the subsystem, each file format is handled by a specific subclass of AbstractArray-DesignHandler. These subclasses contain the logic to parse, validate, and persist the details of a given format. The array annotation is stored in the ArrayDesignDetails and caBIO reporter annotation structures described earlier in the section on the caArray

Domain Classes package. The major classes and dependencies are represented in *Figure 4.14*

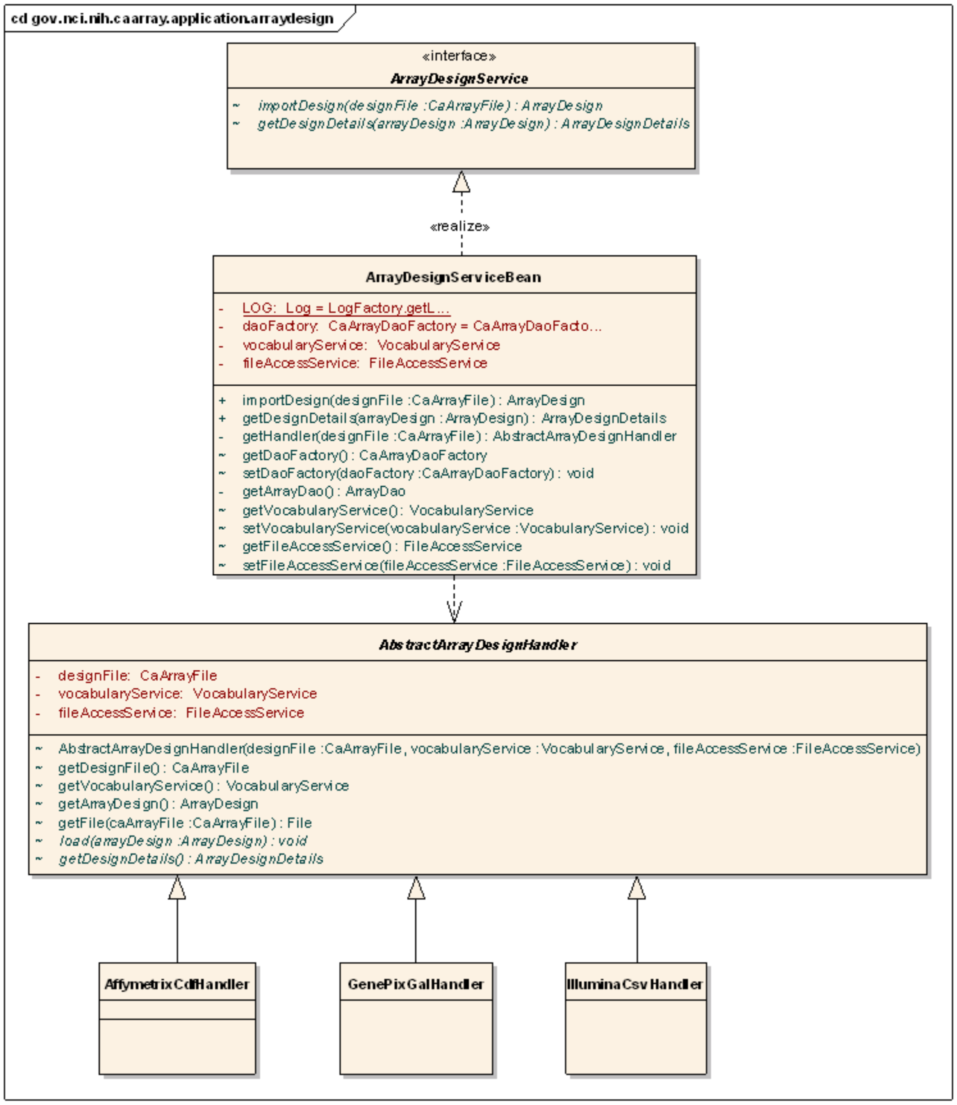

cd gov.nci.nih.caarray.application.arraydesign
«interface»
ArrayDesignService
~ importDesign(designFile :CaArrayFile) : ArrayDesign
~ getDesignDetails(arrayDesign :ArrayDesign) : ArrayDesignDetails
«realize»
ArrayDesignServiceBean
- LOG: Log = LogFactory.getL...
- daoFactory: CaArrayDaoFactory = CaArrayDaoFacto...
- vocabularyService: VocabularyService
- fileAccessService: FileAccessService
+ importDesign(designFile :CaArrayFile) : ArrayDesign
+ getDesignDetails(arrayDesign : ArrayDesign) : ArrayDesignDetails
- getHandler(designFile :CaArrayFile) : AbstractArrayDesignHandler
~ getDaoFactory() : CaArrayDaoFactory
~ setDaoFactory(daoFactory :CaArrayDaoFactory) : void
- getArrayDao() : ArrayDao
~ getVocabularyService() : VocabularyService
~ setVocabularyService(vocabularyService :VocabularyService) : void
~ getFileAccessService() : FileAccessService
~ setFileAccessService(fileAccessService :FileAccessService) : void
AbstractArrayDesignHandler
- designFile: CaArrayFile
- vocabularyService: VocabularyService
- fileAccessService: FileAccessService
~ AbstractArrayDesignHandler(designFile :CaArrayFile, vocabularyService : VocabularyService, fileAccessService :FileAccessService)
~ getDesignFile() : CaArrayFile
~ getVocabularyService() : VocabularyService
~ getArrayDesign() : ArrayDesign
~ getFile(caArrayFile :CaArrayFile) : File
~ load(arrayDesign :ArrayDesign) : void
~ getDesignDetails() : ArrayDesignDetails
AffymetrixCdfHandler
GenePixGalHandler
IlluminaCsvHandler

Figure 4.14 ArrayDesignService subservice implementation

MageTabParser

The MageTabParser subsystem is responsible for reading a set of files in MAGE-TAB format, validating the files and ultimately representing the contents of the files in object model based on MAGE-TAB concepts. The major implementation classes are shown in *Figure 4.15*..

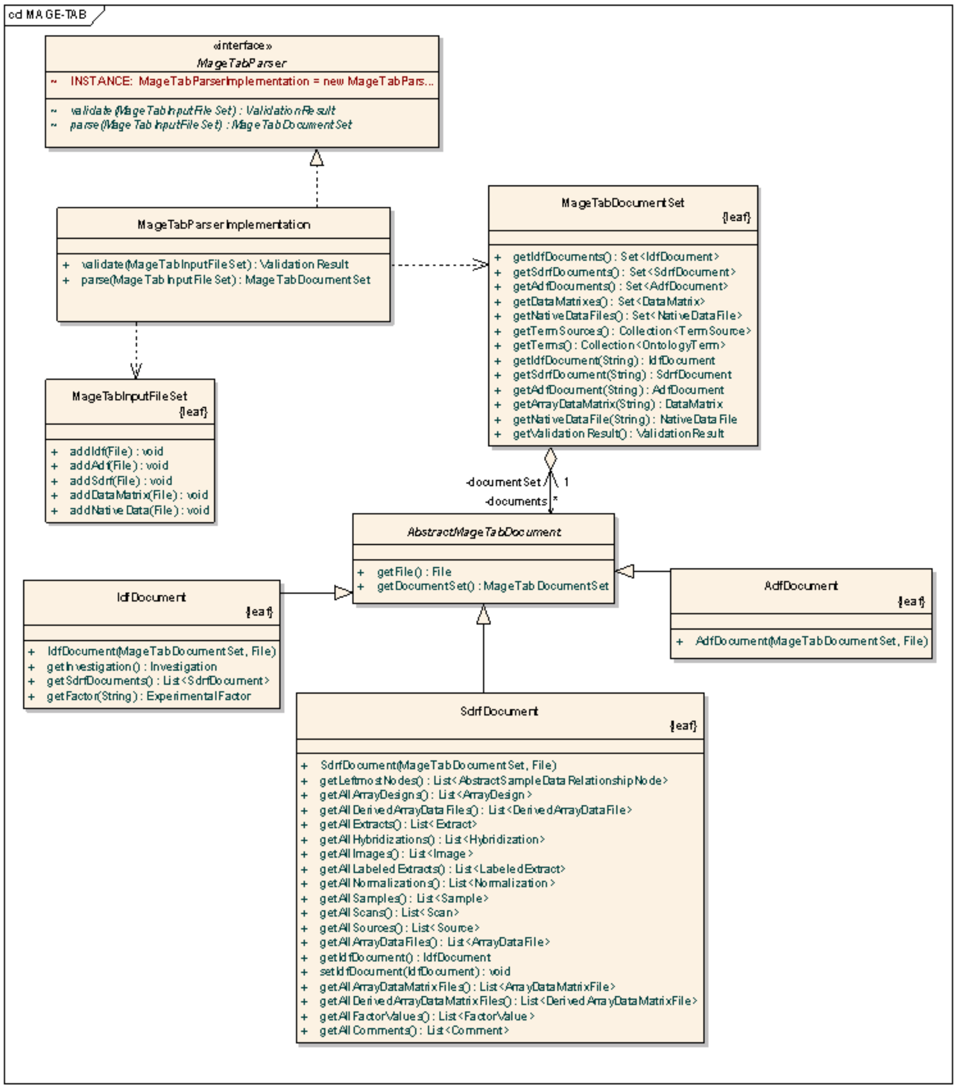

cd MAGE-TAB
«interface»
MageTabParser
~ INSTANCE: MageTabParserImplementation = new MageTabPars...
~ validate (MageTabInputFile Set) : ValidationResult
~ parse (MageTab InputFile Set) : MageTabDocumentSet
MageTabParserImplementation
+ validate(MageTabInputFileSet) : Validation Result
+ parse(MageTab InputFile Set) : MageTabDocumentSet
MageTabInputFileSet
{leaf}
+ addIdf(File) : void
+ addAdf(File) : void
+ addSdrf(File) : void
+ addDataMatrix(File) : void
+ addNative Data(File) : void
MageTabDocumentSet
{leaf}
+ getIdfDocuments() : Set<IdfDocument>
+ getSdrfDocuments() : Set<SdrfDocument>
+ getAdfDocuments() : Set<AdfDocument>
+ getDataMatrixes() : Set<DataMatrix>
+ getNativeDataFiles() : Set< NativeDataFile>
+ getTermSources() : Collection<TermSource>
+ getTerms() : Collection<OntologyTerm>
+ getIdfDocument(String) : IdfDocument
+ getSdrfDocument(String) : SdrfDocument
+ getAdfDocument(String) : AdfDocument
+ getArrayDataMatrix(String) : DataMatrix
+ getNativeDataFile(String) : NativeData File
+ getValidation Result() : ValidationResult
-documentSet
1
-documents
*
AbstractMageTabDocument
+ getFile() : File
+ getDocumentSet() : MageTab DocumentSet
IdfDocument
{leaf}
+ IdfDocument(MageTabDocumentSet, File)
+ getInvestigation() : Investigation
+ getSdrfDocuments() : List<SdrfDocument>
+ getFactor(String) : ExperimentalFactor
AdfDocument
{leaf}
+ AdfDocument(MageTabDocumentSet, File)
SdrfDocument
{leaf}
+ SdrfDocument(MageTabDocumentSet, File)
+ getLeftmostNodes() : List<AbstractSampleDataRelationshipNode>
+ getAllArrayDesigns() : List<ArrayDesign>
+ getAllDerivedArrayDataFiles() : List<DerivedArrayDataFile>
+ getAllExtracts() : List< Extract>
+ getAllHybridizations() : List<Hybridization>
+ getAllImages() : List<Image>
+ getAllLabeledExtracts() : List<LabeledExtract>
+ getAllNormalizations() : List<Normalization>
+ getAllSamples() : List<Sample>
+ getAllScans() : List< Scan>
+ getAllSources() : List< Source>
+ getAllArrayDataFiles() : List<ArrayDataFile>
+ getIdfDocument() : IdfDocument
+ setIdfDocument(IdfDocument) : void
+ getAllArrayDataMatrixFiles() : List<ArrayDataMatrixFile>
+ getAllDerivedArrayDataMatrixFiles() : List<DerivedArrayDataMatrixFile>
+ getAllFactorValues() : List<FactorValue>
+ getAllComments() : List<Comment>

Figure 4.15 MageTabParser Subsystem implementation

MageTabTranslation

The MageTabTranslation subsystem of caArray is invoked to translate from the MAGE-TAB object model generated by the MageTabParser system to a corollary caArray Domain Class representation. It implements a set of translator classes for each MAGE-

TAB document type and for shared data types (i.e. Terms and TermSources). The major classes and dependencies are represented in *Figure 4.16*.

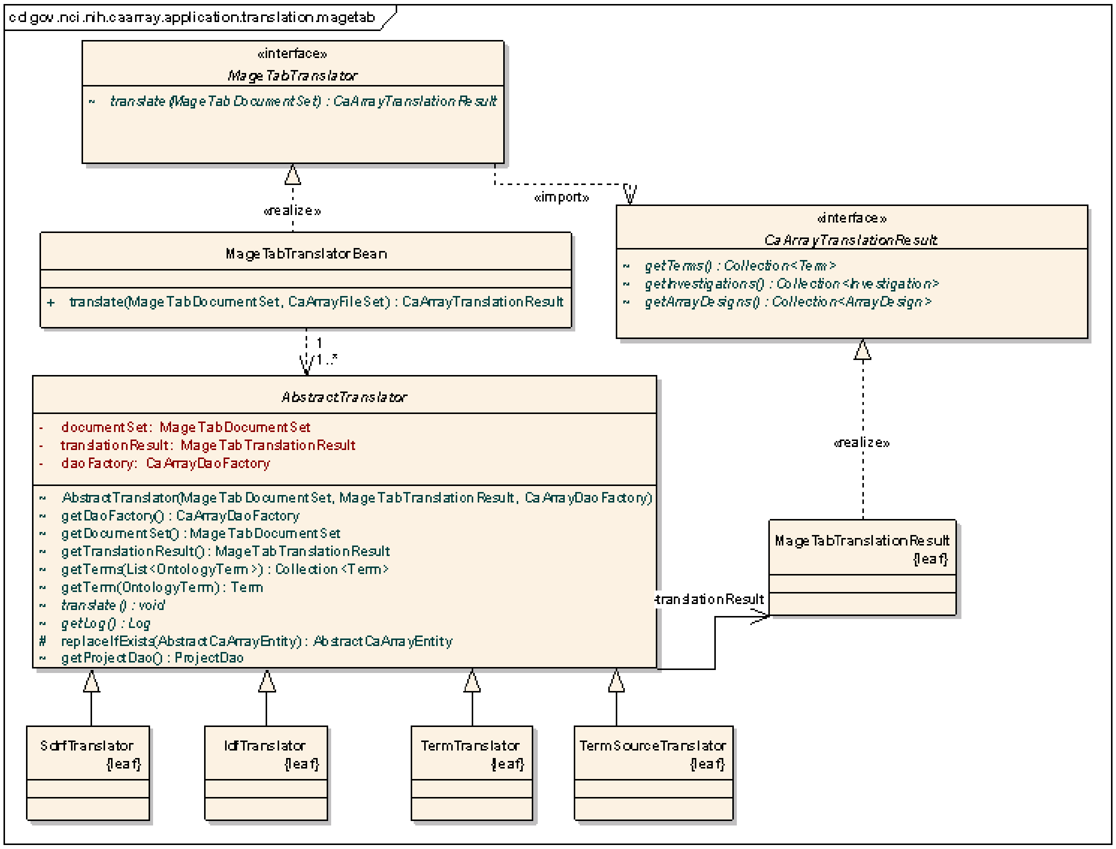

cd gov.nci.nih.caarray.application.translation.magetab
«interface»
MageTabTranslator
~ translate (MageTabDocumentSet) : CaArrayTranslationResult
«realize»
«import»
MageTabTranslatorBean
+ translate(MageTabDocumentSet, CaArrayFileSet) : CaArrayTranslationResult
«interface»
CaArrayTranslationResult
~ getTerms() : Collection<Term>
~ getInvestigations() : Collection<Investigation>
~ getArrayDesigns() : Collection<ArrayDesign>
1
1..*
AbstractTranslator
- documentSet: MageTabDocumentSet
- translationResult: MageTabTranslationResult
- daoFactory: CaArrayDaoFactory
~ AbstractTranslator(MageTabDocumentSet, MageTabTranslationResult, CaArrayDaoFactory)
~ getDaoFactory() : CaArrayDaoFactory
~ getDocumentSet() : MageTabDocumentSet
~ getTranslationResult() : MageTabTranslationResult
~ getTerms(List<OntologyTerm>) : Collection<Term>
~ getTerm(OntologyTerm) : Term
~ translate () : void
~ getLog() : Log
# replaceIfExists(AbstractCaArrayEntity) : AbstractCaArrayEntity
~ getProjectDao() : ProjectDao
«realize»
MageTabTranslationResult
{leaf}
-translationResult
SdrfTranslator
{leaf}
IdfTranslator
{leaf}
TermTranslator
{leaf}
TermSourceTranslator
{leaf}

Figure 4.16 MageTabTranslator Subsystem implementation

CaArrayDataAccess

caArray uses the standard Data Access Object pattern to provide data updates and retrievals. The DAOs are exposed as Java interfaces accessed through a Factory class. The implementations of the DAOs use Hibernate 3.2 as the underlying persistence mechanism.

VocabularyService Subsystem

The VocabularyService subsystem is implemented as a façade to allow the user interface to manage controlled vocabularies. The implementation of this subsystem delegates directly to the CaArrayDataAccess service for entity management. The data model accessed by this service represents a subset of concepts present in external vocabularies such as the MGED Ontology. These concepts provide a consistent Term and Category view of vocabularies, and allow the service to manage both locally and externally controlled vocabularies. However, at this time, external vocabularies are not communicated with directly. Instead, both local and external vocabularies are stored within the caArray database for maximum efficiency. The user interface and mage tab translation both access this service to store and retrieve terms as needed.

The subsystem contents are represented in *Figure 4.17*.

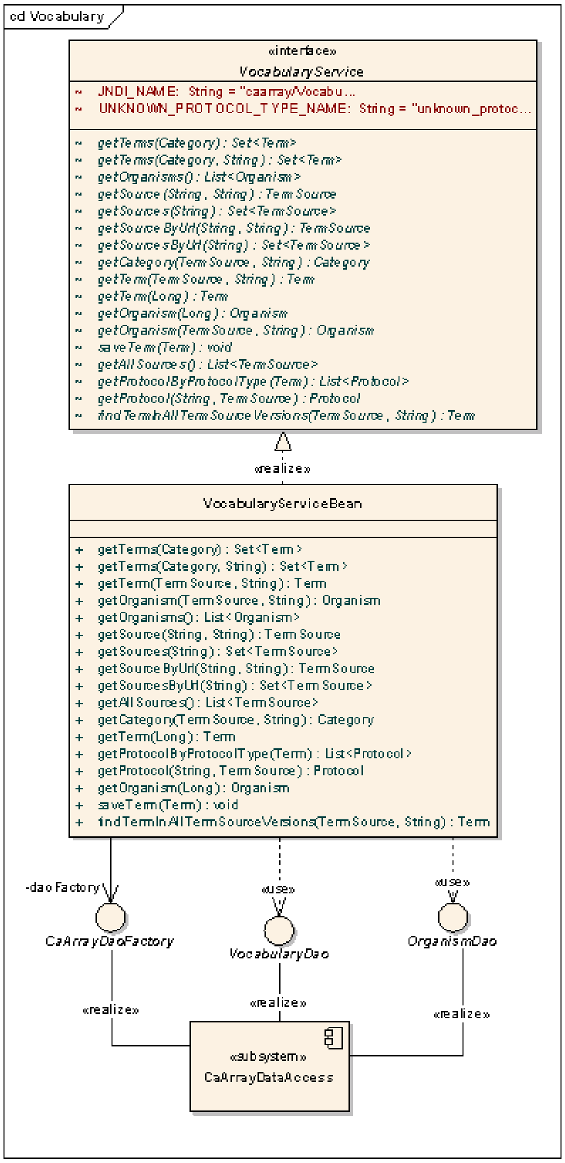

cd Vocabulary
«interface»
VocabularyService
~ JNDI_NAME: String = "caarray/Vocabu...
~ UNKNOWN_PROTOCOL_TYPE_NAME: String = "unknown_protoc...
~ getTerms(Category) : Set<Term>
~ getTerms(Category, String) : Set<Term>
~ getOrganisms() : List<Organism>
~ getSource (String , String) : TermSource
~ getSources(String) : Set<TermSource>
~ getSourceByUrl(String , String) : TermSource
~ getSourcesByUrl(String) : Set<TermSource>
~ getCategory(TermSource , String) : Category
~ getTerm(TermSource , String) : Term
~ getTerm(Long) : Term
~ getOrganism(Long) : Organism
~ getOrganism(TermSource , String) : Organism
~ saveTerm(Term) : void
~ getAllSources() : List<TermSource>
~ getProtocolByProtocolType (Term) : List<Protocol>
~ getProtocol(String , TermSource) : Protocol
~ findTermInAllTermSourceVersions(TermSource , String) : Term
«realize»
VocabularyServiceBean
+ getTerms(Category) : Set<Term>
+ getTerms(Category, String) : Set<Term>
+ getTerm(TermSource , String) : Term
+ getOrganism(TermSource , String) : Organism
+ getOrganisms() : List<Organism>
+ getSource(String , String) : TermSource
+ getSources(String) : Set<TermSource>
+ getSourceByUrl(String , String) : TermSource
+ getSourcesByUrl(String) : Set<TermSource>
+ getAllSources() : List<TermSource>
+ getCategory(TermSource , String) : Category
+ getTerm(Long) : Term
+ getProtocolByProtocolType(Term) : List<Protocol>
+ getProtocol(String , TermSource) : Protocol
+ getOrganism(Long) : Organism
+ saveTerm(Term) : void
+ findTermInAllTermSourceVersions(TermSource, String) : Term
-daoFactory
«use»
«use»
CaArrayDaoFactory
VocabularyDao
OrganismDao
«realize»
«realize»
«realize»
«subsystem»
CaArrayDataAccess

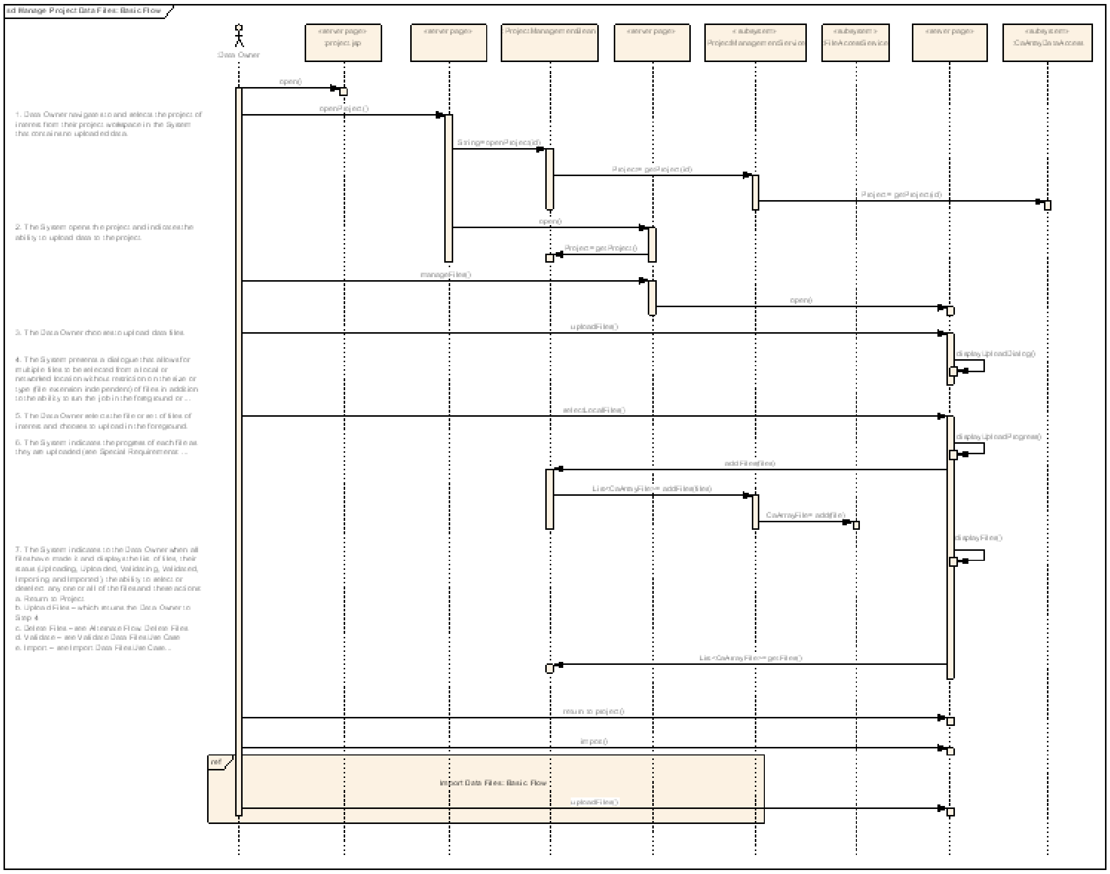
Figure 4.17 VocabularyService implementation

caarraydb

The caarraydb component represents the MySQL database schema that is used to store all of caArray's persistent data. The schema is generated directly from Hibernate annotations recorded in the caArray Domain Classes so that the schema and domain classes are kept easily synchronized. caArray supports MySQL version 5.0 and uses InnoDB tables for transactional behavior.

Use-Case Realizations

Manage Experiment Data Files



1. Data Owner navigates to and selects the project of interest from their project workspace in the System that contains no uploaded data.

2. The System opens the project and indicates the ability to upload data to the project.

3. The Data Owner chooses to upload data files.

4. The System presents a dialogue that allows for multiple files to be selected from a local or networked location without restriction on the size or type (file extension independent) of files in addition to the ability to run the job in the foreground or ...

5. The Data Owner selects the file or set of files of interest and chooses to upload in the foreground.

6. The System indicates the progress of each file as they are uploaded (see Special Requirements: ...

7. The System indicates to the Data Owner when all files have made it and displays the list of files, their status (Uploading, Uploaded, Validating, Validated, Importing and Imported) the ability to select or deselect any one or all of the files and these actions:
a. Return to Project
b. Upload Files – which returns the Data Owner to Step 4
c. Delete Files – see Alternate Flow: Delete Files
d. Validate – see Validate Data Files Use Case
e. Import – see Import Data Files Use Case...

Figure 4.18 Manage Experiment Data Files basic flow sequence diagram

Import Data Files

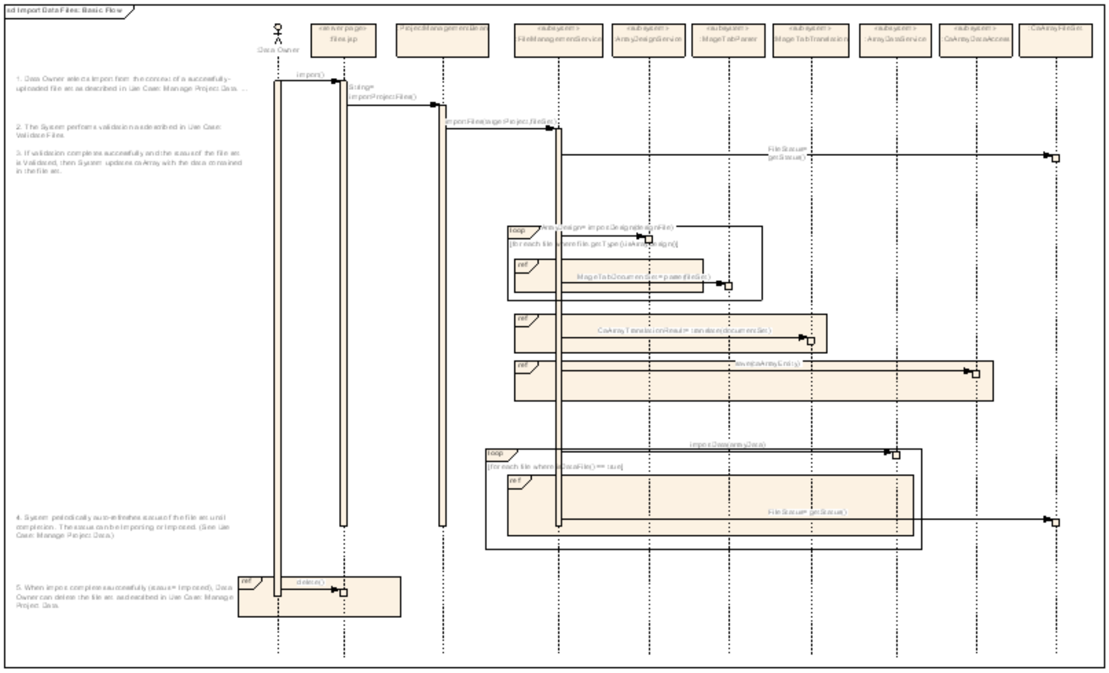

sd Import Data Files: Basic Flow
:Data Owner
«server page» :files.jsp
:ProjectManagementBean
«subsystem» :FileManagementService
«subsystem» :ArrayDesignService
«subsystem» :MageTabParser
«subsystem» :MageTabTranslator
«subsystem» :ArrayDataService
«subsystem» :CaArrayDaoAccess
:CaArrayFileSet
1. Data Owner selects import from the context of a successfully-uploaded file set as described in Use Case: Manage Project Data. ...
2. The System performs validation as described in Use Case: Validate Files.
3. If validation completes successfully and the status of the file set is Validated, then System updates caArray with the data contained in the file set.
4. System periodically auto-refreshes status of the file set until completion. The status can be Importing or Imported. (See Use Case: Manage Project Data.)
5. When import completes successfully (status = Imported), Data Owner can delete the file set as described in Use Case: Manage Project Data.
import()
String= importProjectFiles()
importFiles(targetProject,fileSet)
FileStatus= getStatus()
loop
[for each file where file.getType().isArrayDesign()]
ArrayDesign= importDesign(designFile)
ref
MageTabDocumentSet= parse(fileSet)
ref
CaArrayTranslationResult= translate(documentSet)
ref
save(caArrayEntity)
loop
[for each file where isDataFile() == true]
importData(arrayData)
ref
FileStatus= getStatus()
ref
delete()

Figure 4.19 Import Data Files basic flow sequence diagram

Acquire Experiment Data via API

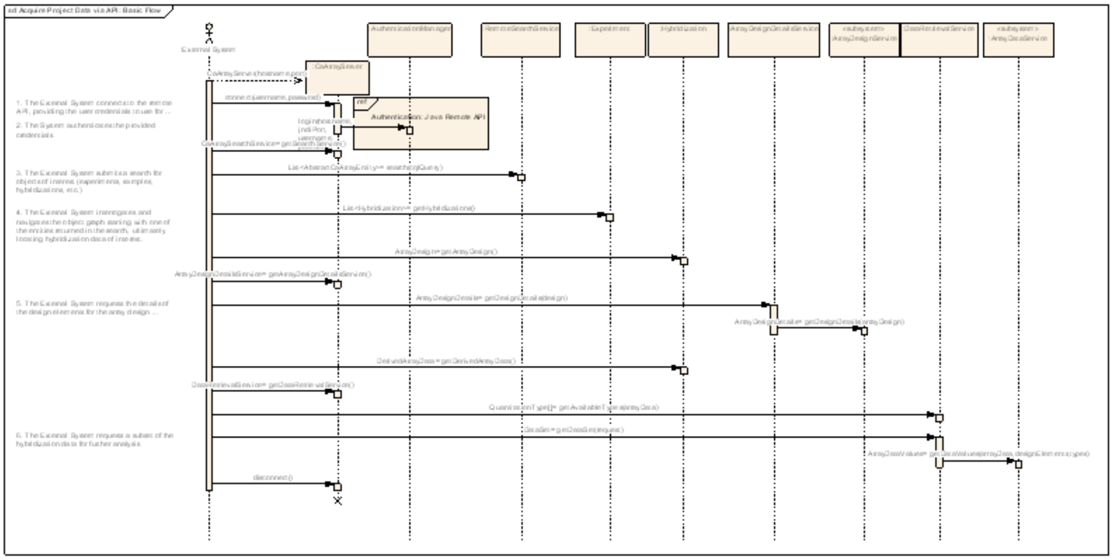

sd Acquire Project Data via API: Basic Flow
:External System
:CaArrayServer
:AuthenticationManager
:RemoteSearchService
:Experiment
:Hybridization
:ArrayDesignDetailsService
«subsystem» :ArrayDesignService
:DataRetrievalService
«subsystem» :ArrayDataService
1. The External System connects to the remote API, providing the user credentials to use for ...
2. The System authenticates the provided credentials.
3. The External System submits a search for objects of interest (experiments, samples, hybridizations, etc.)
4. The External System interrogates and navigates the object graph starting with one of the entities returned in the search, ultimately locating hybridization data of interest.
5. The External System requests the details of the design elements for the array design ...
6. The External System requests a subset of the hybridization data for further analysis.
CaArrayServer(hostname,port)
connect(username, password)
login(hostname, jndiPort, username, password)
ref
Authentication: Java Remote API
CaArraySearchService= getSearchService()
List<AbstractCaArrayEntity>= search(objQuery)
List<Hybridization>= getHybridizations()
ArrayDesign= getArrayDesign()
ArrayDesignDetailsService= getArrayDesignDetailsService()
ArrayDesignDetails= getDesignDetails(design)
ArrayDesignDetails= getDesignDetails(arrayDesign)
DerivedArrayData = getDerivedArrayData()
DataRetrievalService= getDataRetrievalService()
QuantitationType[]= getAvailableTypes(arrayData)
DataSet= getDataSet(request)
ArrayDataValues= getDataValues(arrayData, designElements,types)
disconnect()

Figure 4.20 Acquire Experiment Data via API basic flow

Grid access follows a similar flow, with the CaArraySvc delegating calls to these same remote session beans.

CHAPTER 5

IMPLEMENTATION VIEW

Overview

The major physical artifacts that comprise the caArray software deployment units are illustrated in *Figure 5.1*. The major artifacts and their relationships to the subsystems they realize are described in the following section, *Artifacts*.

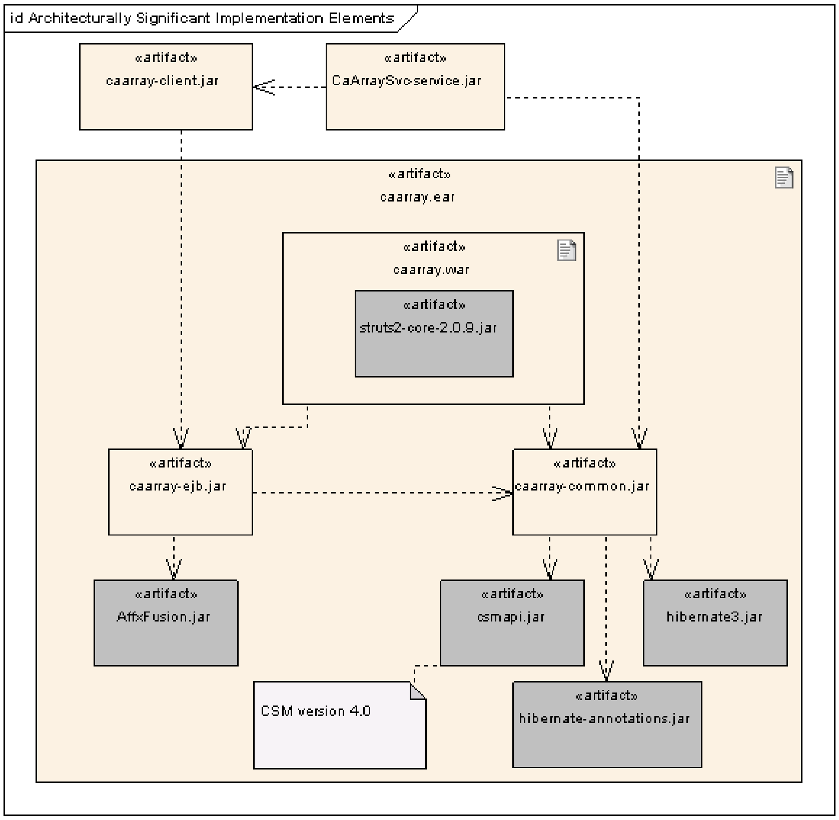

id Architecturally Significant Implementation Elements
«artifact» caarray-client.jar
«artifact» CaArraySvc-service.jar
«artifact» caarray.ear
«artifact» caarray.war
«artifact» struts2-core-2.0.9.jar
«artifact» caarray-ejb.jar
«artifact» caarray-common.jar
«artifact» AffxFusion.jar
«artifact» csmapi.jar
«artifact» hibernate3.jar
CSM version 4.0
«artifact» hibernate-annotations.jar

Figure 5.1 Architecturally significant implementation elements

Artifacts

caarray.ear

The caarray.ear artifact is the J2EE Enterprise Application Archive (EAR) that contains all of the web portal application and EJB components that make up the User Interface, Remote Java API, Application Logic, and Business Logic layers of the application. The EAR also contains the third-party JARs necessary to support the Application and Business Logic Layers of the application.

caarray.war

The caarray.war artifact packages the JSPs and caArray Struts 2 classes that comprise the User Interface layer of caArray. The WAR also contains the Struts 2 third-party JARs and necessary supporting JARs.

caarray-ejb.jar

The caarray-ejb.jar packages the implementation of all of the EJB subsystems and the implementation of the remote API interfaces defined in caarray-client.jar. This includes all of the subsystems in the Application Layer and VocabularyService subsystem from the Business Logic Layer of the logical model. The important third-party dependency to note is the dependency on AffxFusion.jar which provides Affymetrix file format parsing support.

caarray-common.jar

The caarray-common.jar contains the caArray Domain Classes packages, the CaArray Data Access subsystem and the MageTabParser subsystem. The major third-party component dependencies noted are hibernate3.jar and hibernate-annotations.jar to support annotation-based Hibernate ORM mapping and to csmapi.jar to support entity access authorization.

caarray-client.jar

The caarray-client.jar contains the remote EJB interfaces required by Java Remote API clients and the caArray Grid Service. This JAR also repackages other third-party classes required by remote clients.

CaArraySvc-service.jar

The CaArraySvc-service.jar contains the caGrid API implementation classes.

CHAPTER 6 DEPLOYMENT VIEW

The typical deployment configuration for caArray is documented in *Figure 6.1*. The NCICB deployment of caArray is similar to the scenario modeled in the figure, with the addition of a front-end Apache web server that receives the HTTPS requests and then delegates these requests to the JBoss server where caArray is deployed. The JBoss container configures the datasource (for db connections) and SMTP information (for email).

While Globus is shown as the grid service execution environment, the NCICB deployment environment uses JBoss 4.0.4 as the container and Globus runs inside of JBoss

4.0.4. External adopters might also choose to deploy application components and execution environments to a single server.

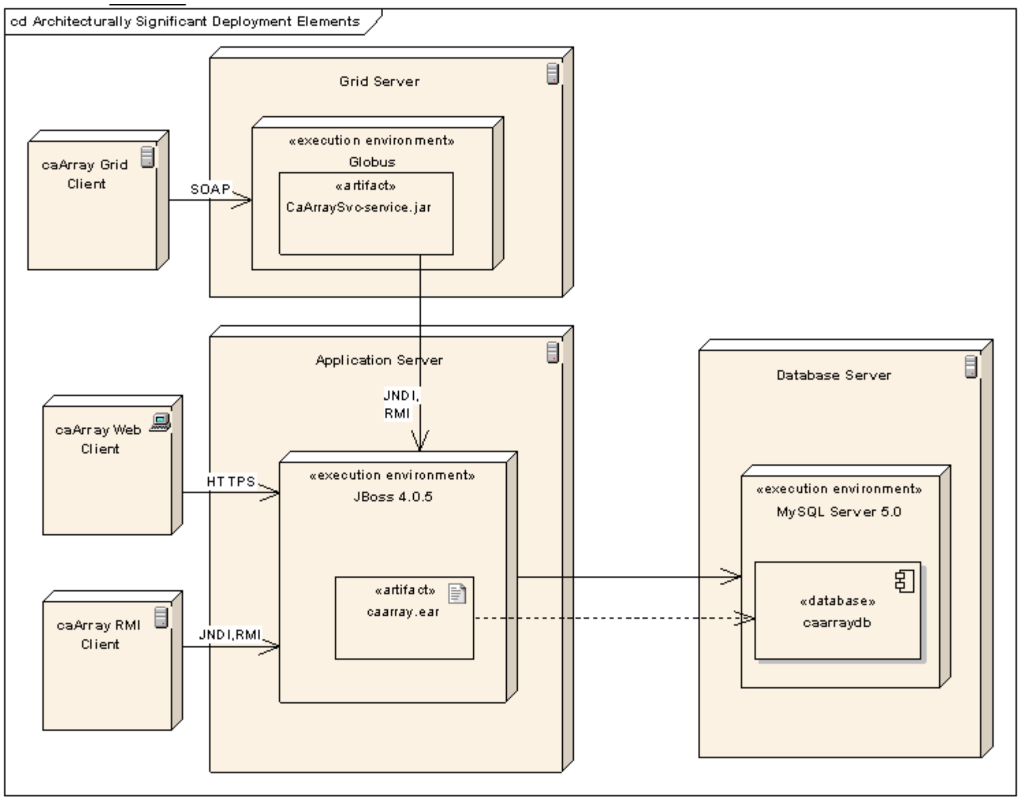


Figure 6.1 Architecturally Significant Deployment Elements

APPENDIX A

CAARRAY REFERENCES

- **caArray UML Models**
  //cm/caarray2/docs/analysis_and_design/models/caaray.eap
- **caArray Use Case Summary**
  //cm/caarray2/docs/analysis_and_design/models/caaray.eap
- **Philippe Kruchten 1995, "The 4+1 view model of architecture," IEEE Software. 12(6), November 1995.**
  //cm/caarray2/docs/analysis_and_design/references/architecture/Kruchten4+1.pdf

INDEX