

# caArray 2.0

## *Technical Guide*



Center for Bioinformatics

January 27, 2008



# CREDITS AND RESOURCES

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

<b>Contacts and Support</b>	
NCICB Application Support	<a href="http://ncicbsupport.nci.nih.gov/sw/">http://ncicbsupport.nci.nih.gov/sw/</a> Telephone: 301-451-4384 Toll free: 888-478-4423



# TABLE OF CONTENTS

<b>Credits and Resources .....</b>	<b>i</b>
<b>Using the caArray Technical Guide .....</b>	<b>1</b>
Introduction to caArray .....	1
Purpose of this Manual .....	1
Definitions and Acronyms .....	2
References .....	2
Organization of the Manual .....	2
Document Text Conventions .....	2
<b>Chapter 1</b>	
<b>Architectural Representation of caArray .....</b>	<b>5</b>
Architectural Representation .....	5
Architectural Goals and Constraints .....	6
<b>Chapter 2</b>	
<b>Use Case View .....</b>	<b>7</b>
caArray Use Cases .....	8
<b>Chapter 3</b>	
<b>Logical View .....</b>	<b>11</b>
Overview .....	11
Architecturally Significant Design Elements .....	13
caarraydb .....	33
Use-Case Realizations .....	34
<b>Chapter 4</b>	
<b>Implementation View .....</b>	<b>37</b>
Overview .....	37
Artifacts .....	38

## Chapter 5

Deployment View .....39

Index .....41

# 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
- *References* 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/svn-root/caarray2/trunk/docs/analysis\\_and\\_design/models/caarray.EAP](https://gforge.nci.nih.gov/svn-root/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

## References

---

- **caArray UML Models**  
[https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/analysis\\_and\\_design/models/caarray.EAP](https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/analysis_and_design/models/caarray.EAP)
- **caArray Use Case Summary**  
[https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/requirements/caarray\\_use\\_case\\_summary.doc](https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/requirements/caarray_use_case_summary.doc)
- **Philippe Kruchten 1995, "The 4+1 view model of architecture," IEEE Software. 12(6), November 1995.**  
[https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/analysis\\_and\\_design/references/architecture/Kruchten4+1.pdf](https://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/analysis_and_design/references/architecture/Kruchten4+1.pdf)

## Organization of the Manual

---

The *caArray Technical Guide* contains the following chapters:

- *Using the caArray Technical Guide*
- *Chapter 1, Architectural Representation of caArray*, on page 5
- *Chapter 2, Extracting Data Using an API*, on page 7
- *Chapter 2, Use Case View*, on page 7
- *Chapter 3, Logical View*, on page 11
- *Chapter 4, Implementation View*, on page 37
- *Chapter 5, Deployment View*, on page 39
- *Appendix A, caArray References*, on page 45

## 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.

<i>Convention</i>	<i>Description</i>	<i>Example</i>
<b>Bold &amp; Capitalized Command</b> <b>Capitalized command &gt; Capitalized command</b>	Indicates a Menu command Indicates Sequential Menu commands	<b>New Array Design</b>
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.
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
<b>Boldface type</b>	Options that you select in dialog boxes or drop-down menus. Buttons or icons that you click.	From the Experiment Details page, click <b>Generate MAGE-ML</b> .
<i>Italics</i>	Used to reference other documents, sections, figures, and tables.	<i>caArray User's Guide</i>
<b>Boldface monospace type</b>	Text that you type	In the New Subset text box, enter <b>Array Manufacture Software</b> .
<b>Note:</b>	Highlights a concept of particular interest	<b>Note:</b> This concept is used throughout the installation manual.
<b>Warning!</b>	Highlights information of which you should be particularly aware.	<b>Warning!</b> 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



# 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 2, Use Case View*, on page 7.
- 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 3, Logical View*, on page 11.
- The **Process View** - Illustrates the process decomposition of the system, including the mapping of classes and subsystems on to processes and threads.
- The **Deployment View** – Describes how the processes are allocated to hardware and execution environments and the communication paths between hardware nodes. *Chapter 4, Deployment View*, on page 37.
- 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 39.

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 USE CASE VIEW

The use cases represented in *Figure 2.1* 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 document: <https://>

[gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/requirements/caarray\\_use\\_case\\_summary.doc...](http://gforge.nci.nih.gov/svnroot/caarray2/trunk/docs/requirements/caarray_use_case_summary.doc...)

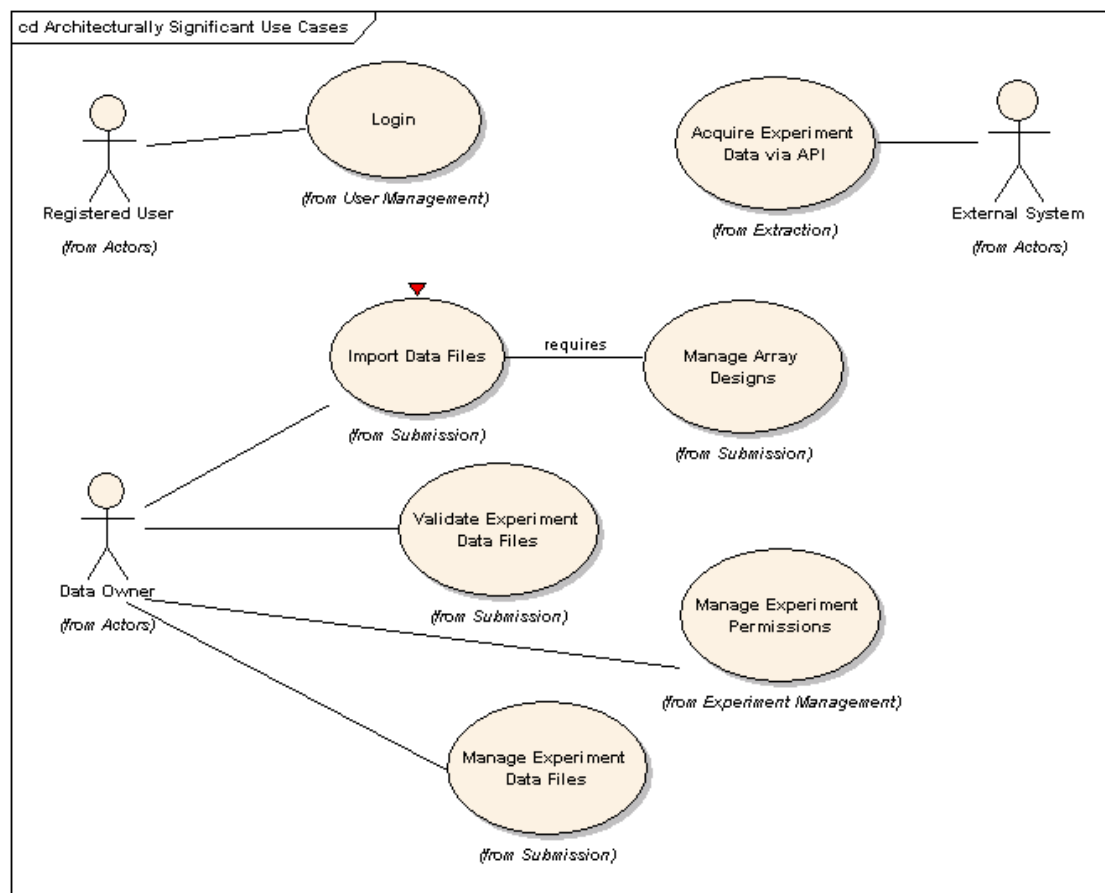


Figure 2.1 Use case summary

## 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 3 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 3.1](#) 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 3.1](#) 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 33 which documents the use-case realizations that employ these subsystems.

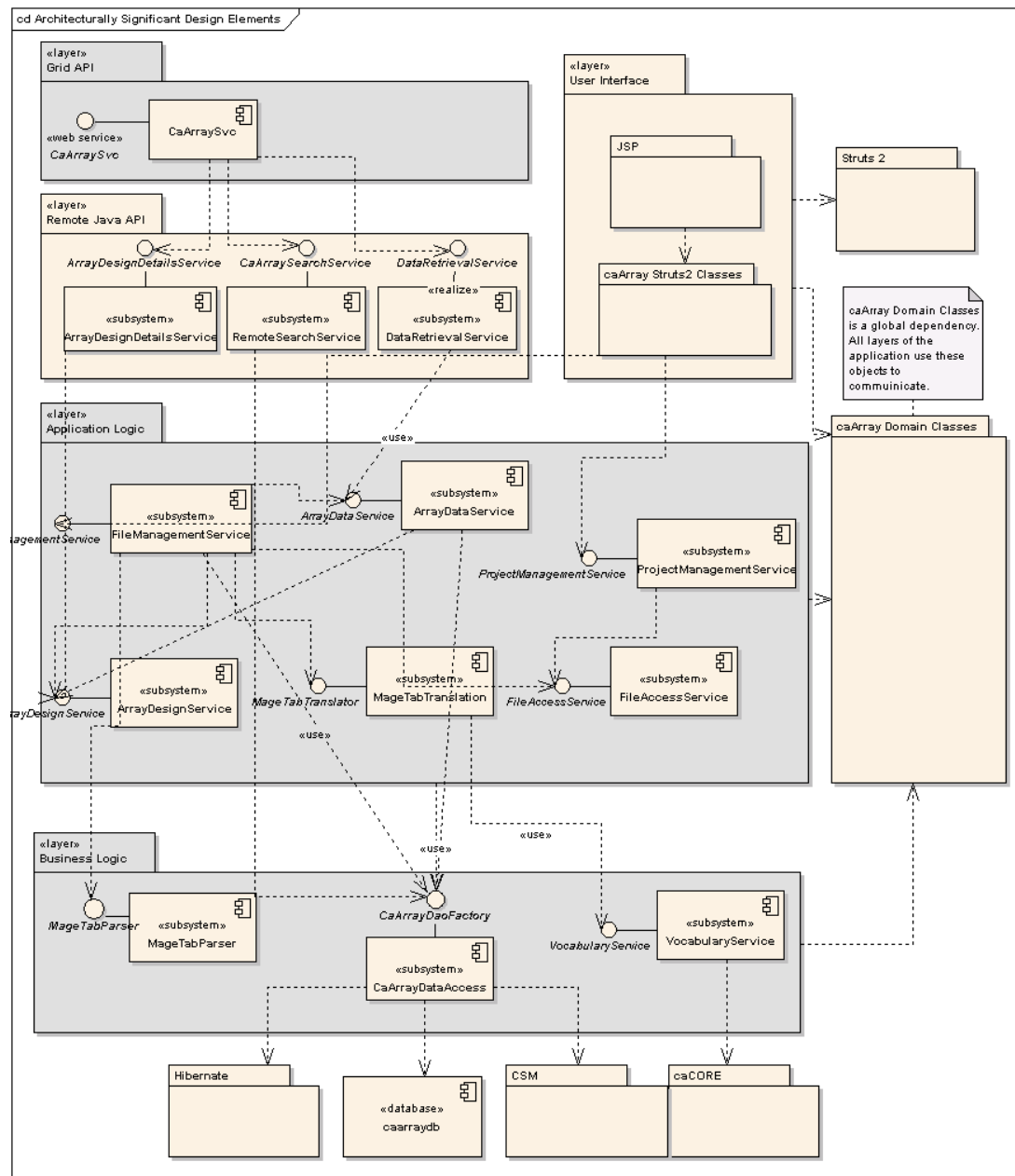


Figure 3.1 Architecturally Significant Design Elements

## 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 Struts 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 3.2](#).

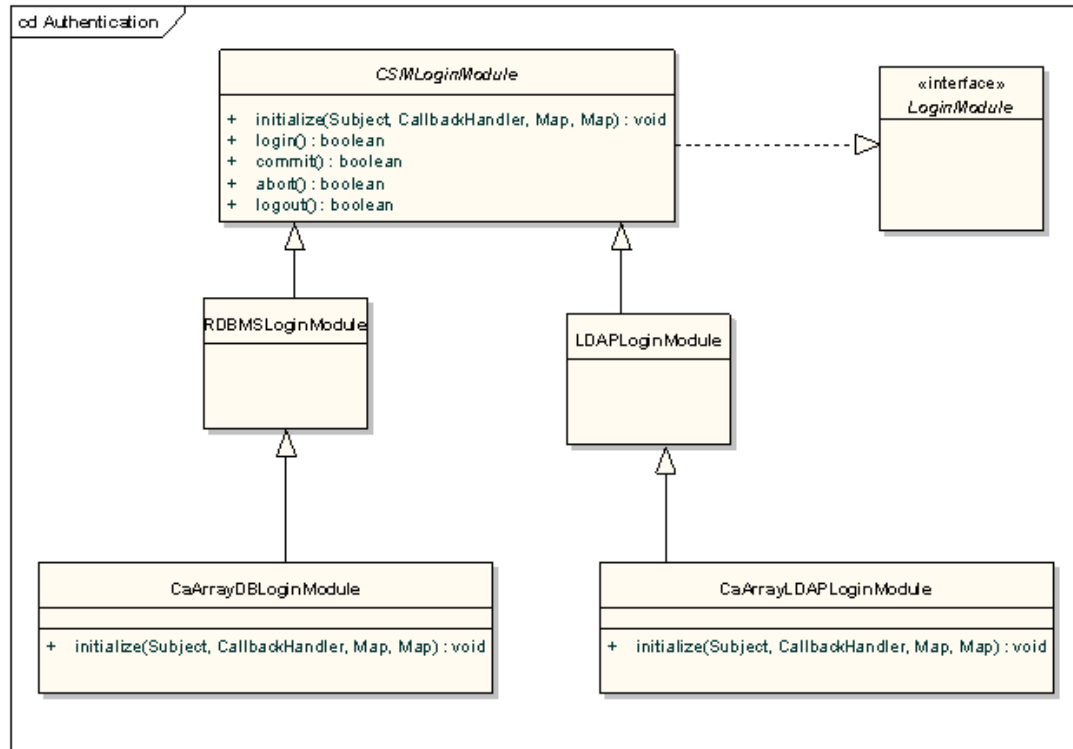


Figure 3.2 Authentication classes

## 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<AbstractCaArrayObject> 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 Introduced 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.

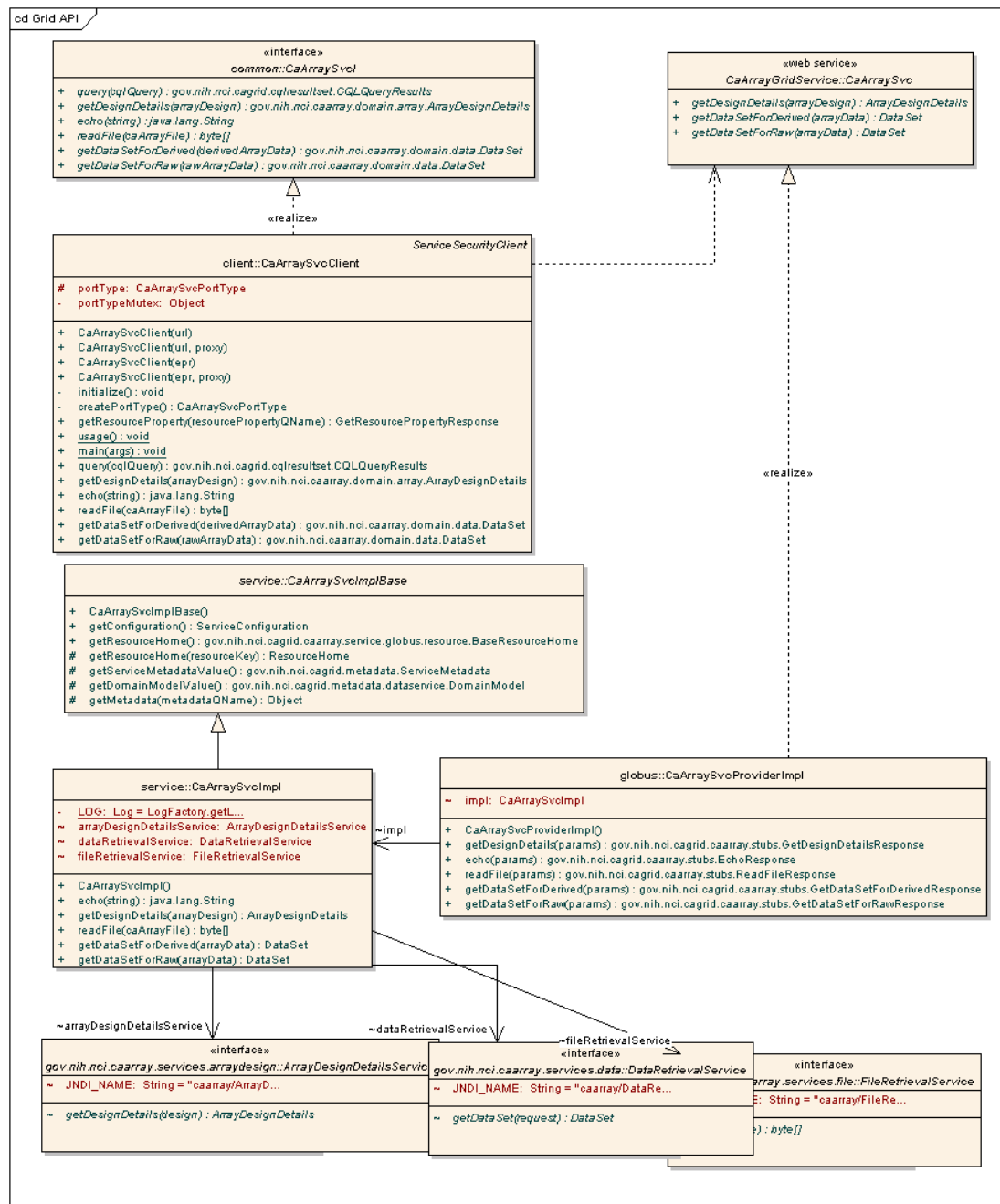


Figure 3.3 CaArraySvc Grid API implementation

## 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 3.4](#).

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.

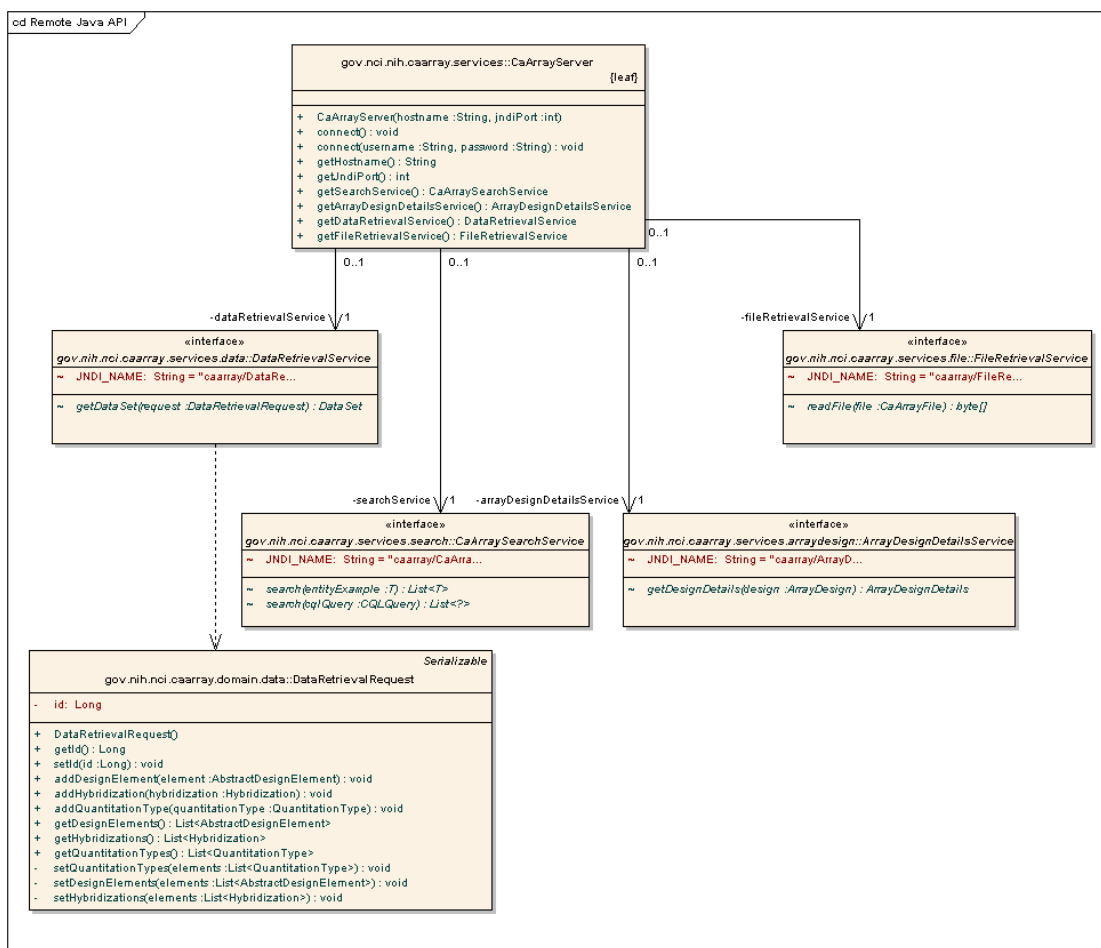


Figure 3.4 caArray Remote Java API

## 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<sup>1</sup>). 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 use 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 3.5](#).

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 3.6](#).

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 3.8](#). 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 `HybridizationData`s 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).

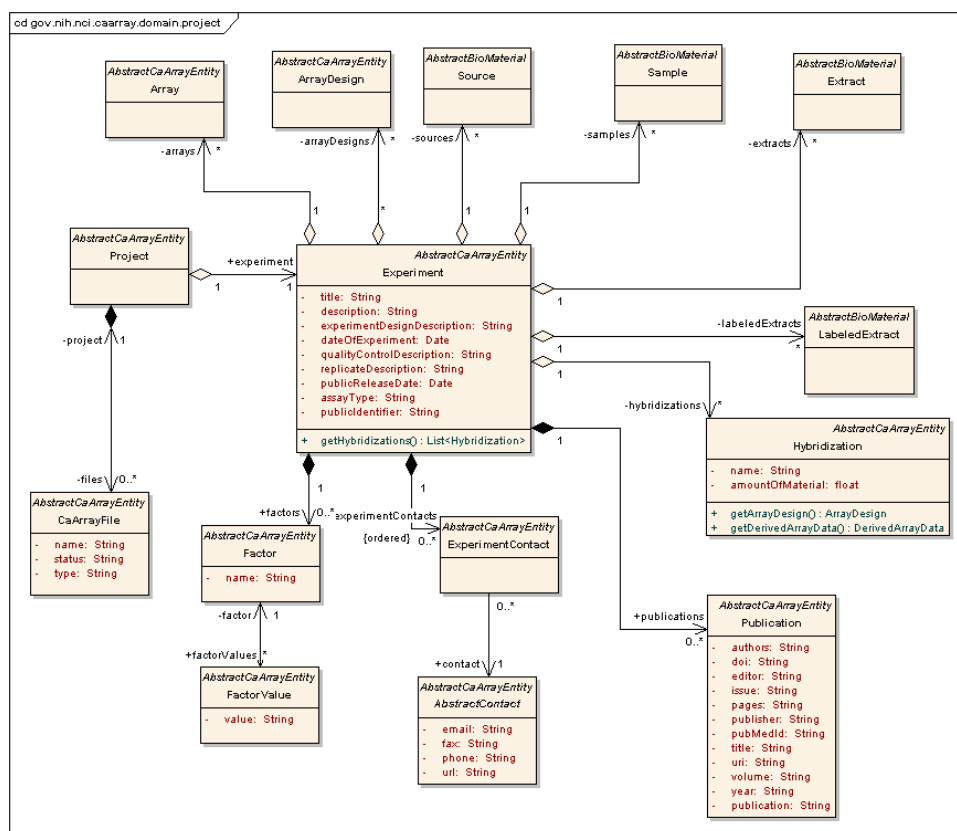


Figure 3.5 Experiment Overview



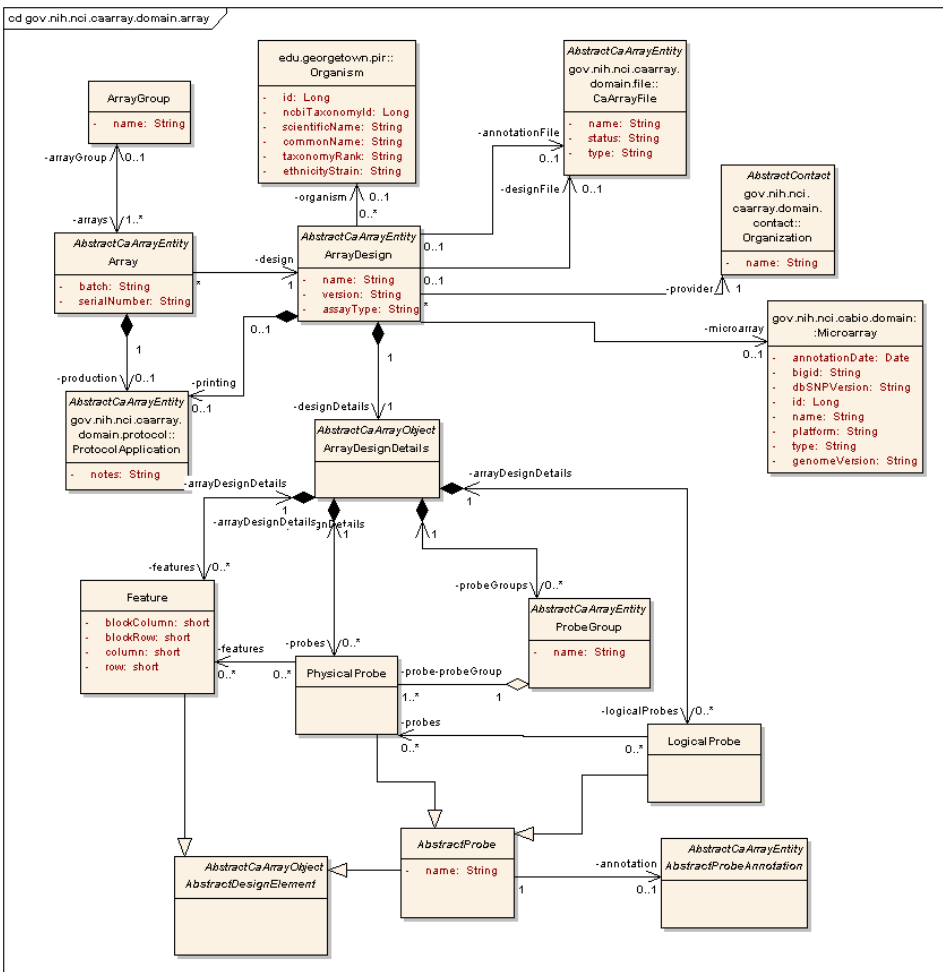


Figure 3.7 *caBIO* Array Reporter Annotation Object Model



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 3.9](#).

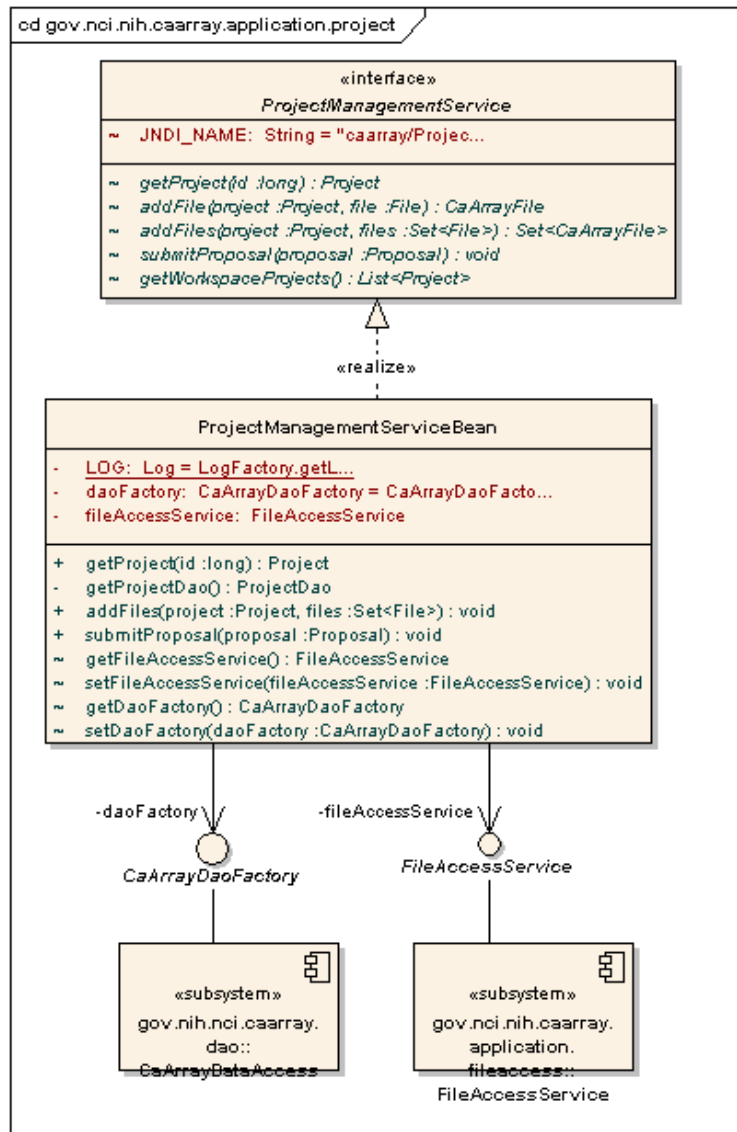


Figure 3.9 *ProjectManagementService* implementation

## 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 3.10](#) and the act of storing file contents is shown in [Figure 3.11](#).

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.

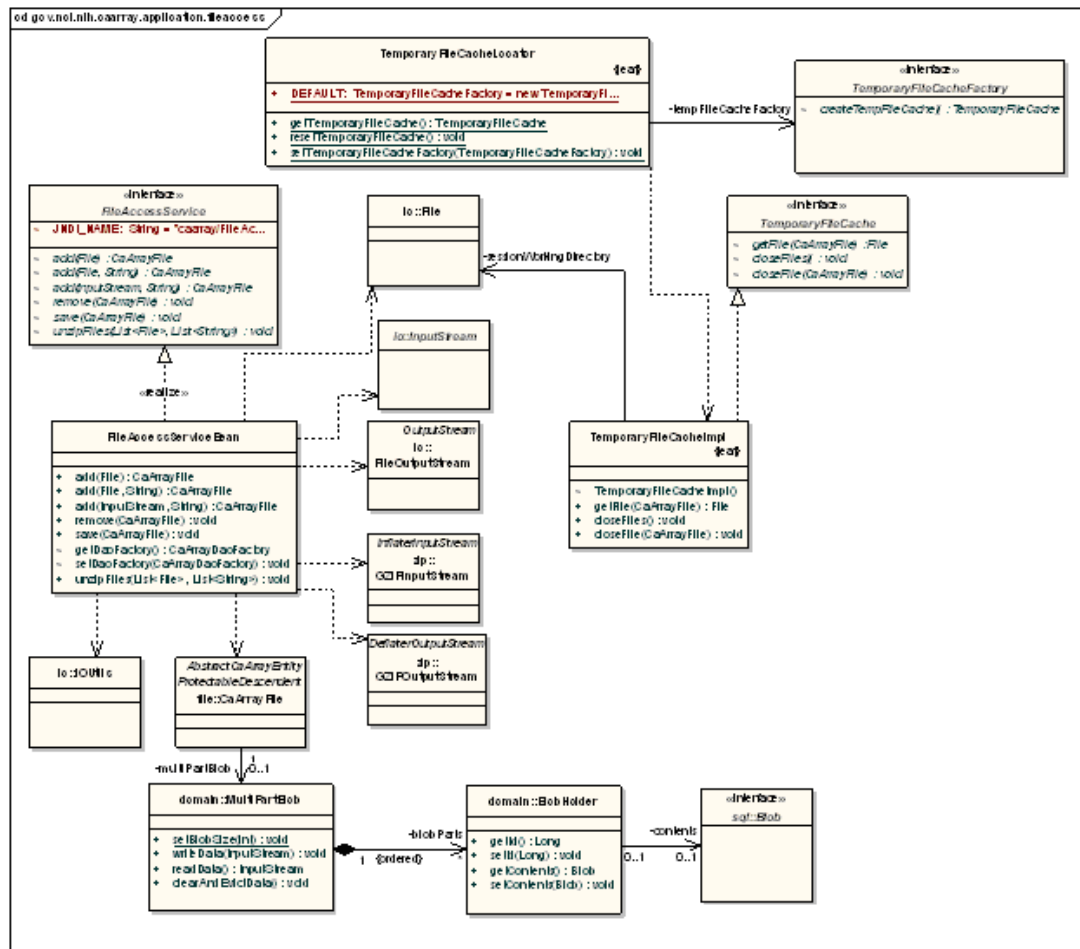


Figure 3.10 FileAccessService implementation class diagram

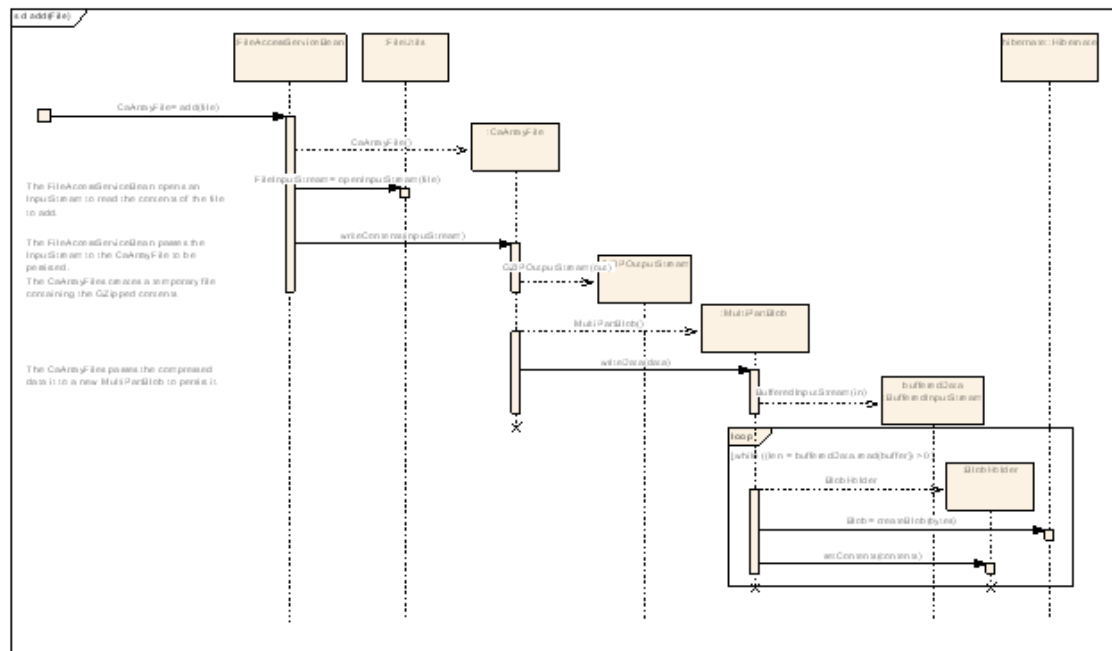


Figure 3.11 FileAccessService operation AddFile(file : File)

## 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 3.12](#) 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.

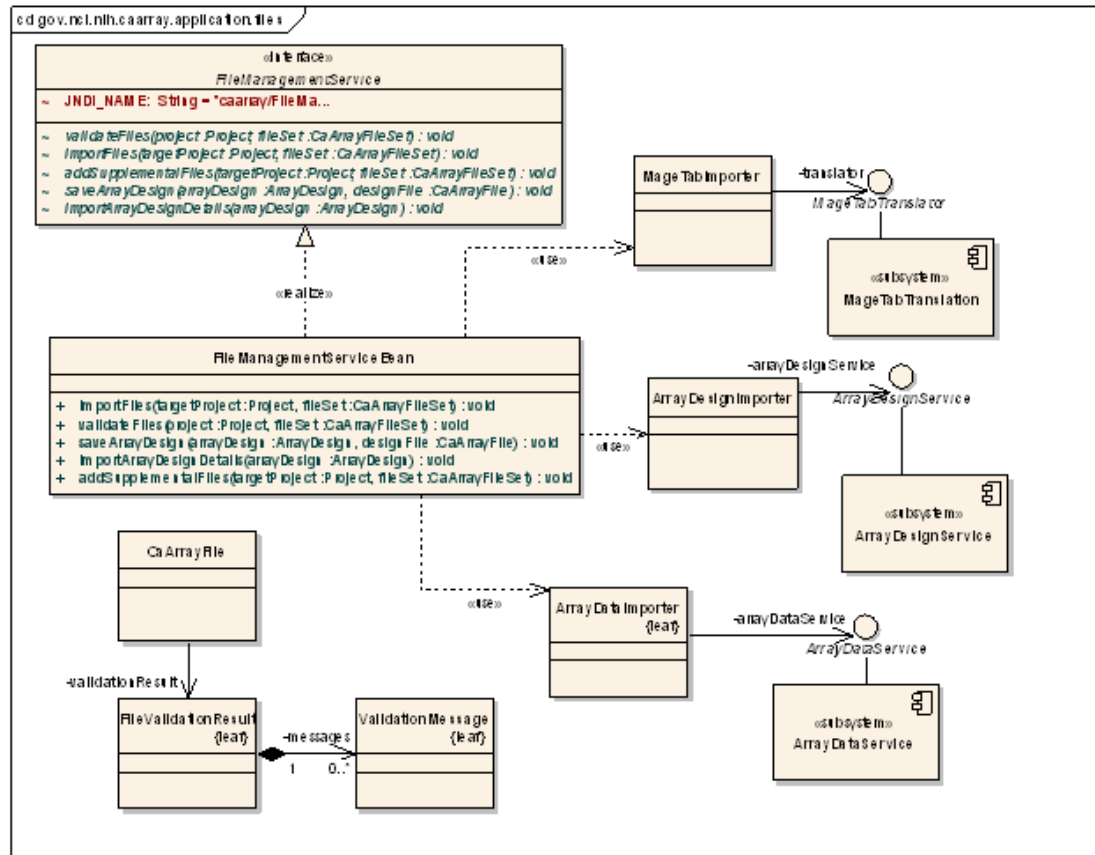


Figure 3.12 FileManagementService implementation

## 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<sup>1</sup>.

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 3.13](#).

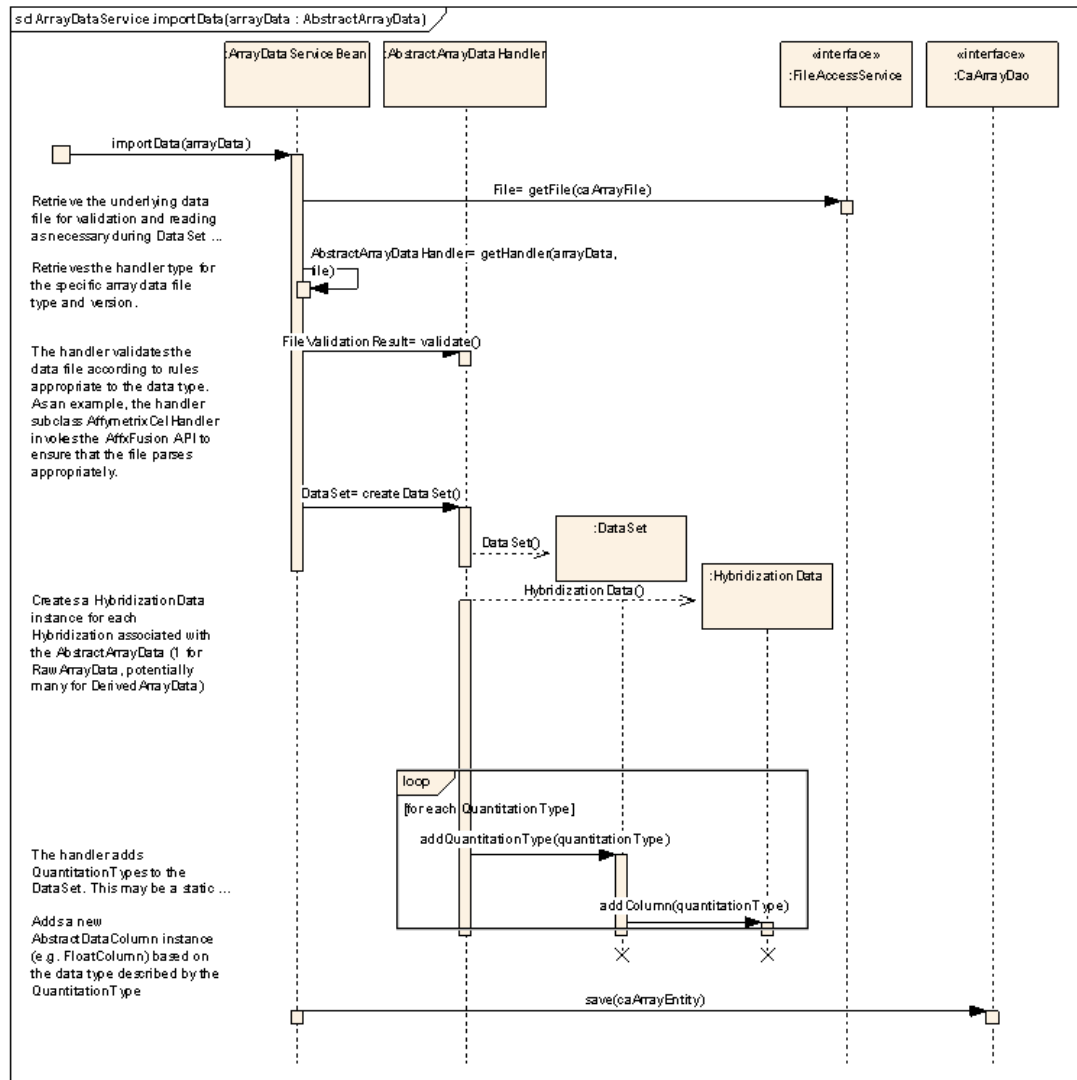


Figure 3.13 Import Operation implementation

## 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 AbstractArrayDesignHandler. 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 3.14](#)

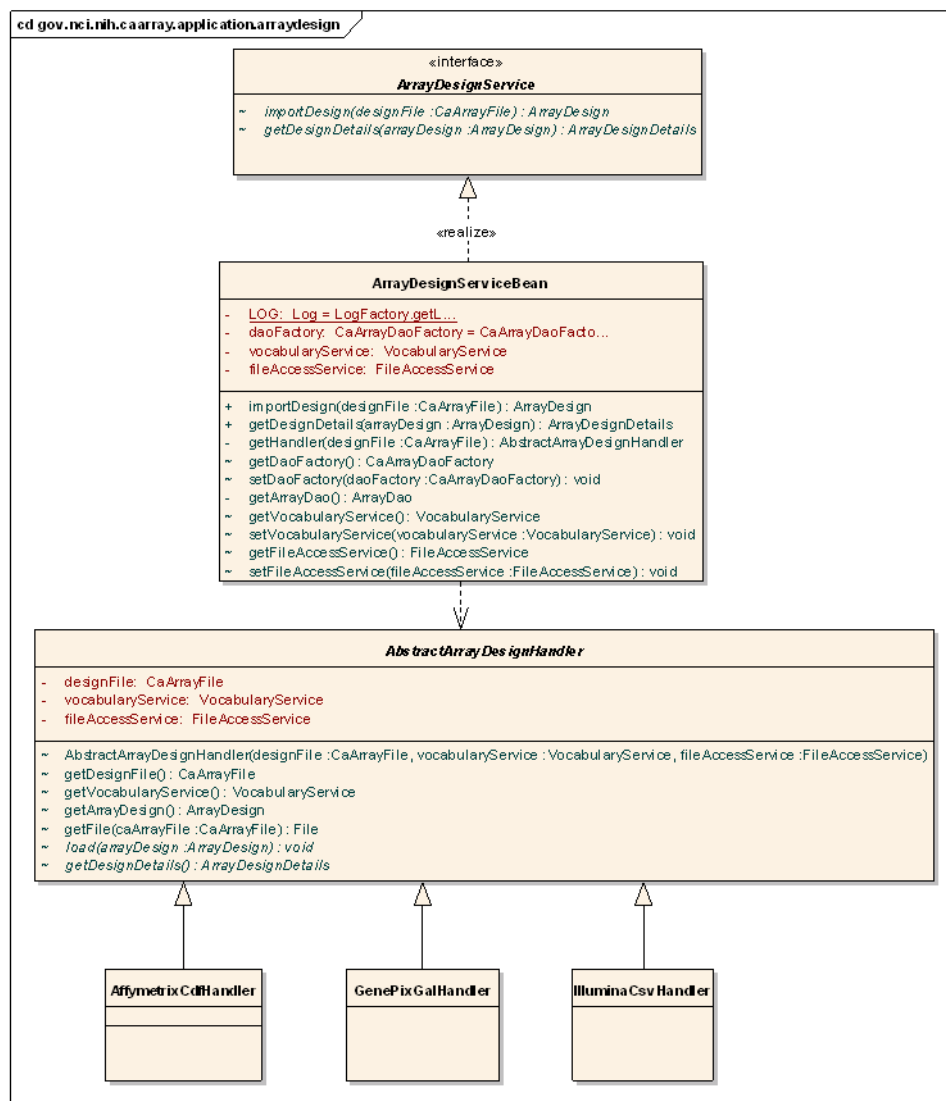


Figure 3.14 ArrayDesignService subservice implementation

## 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 3.15.](#)

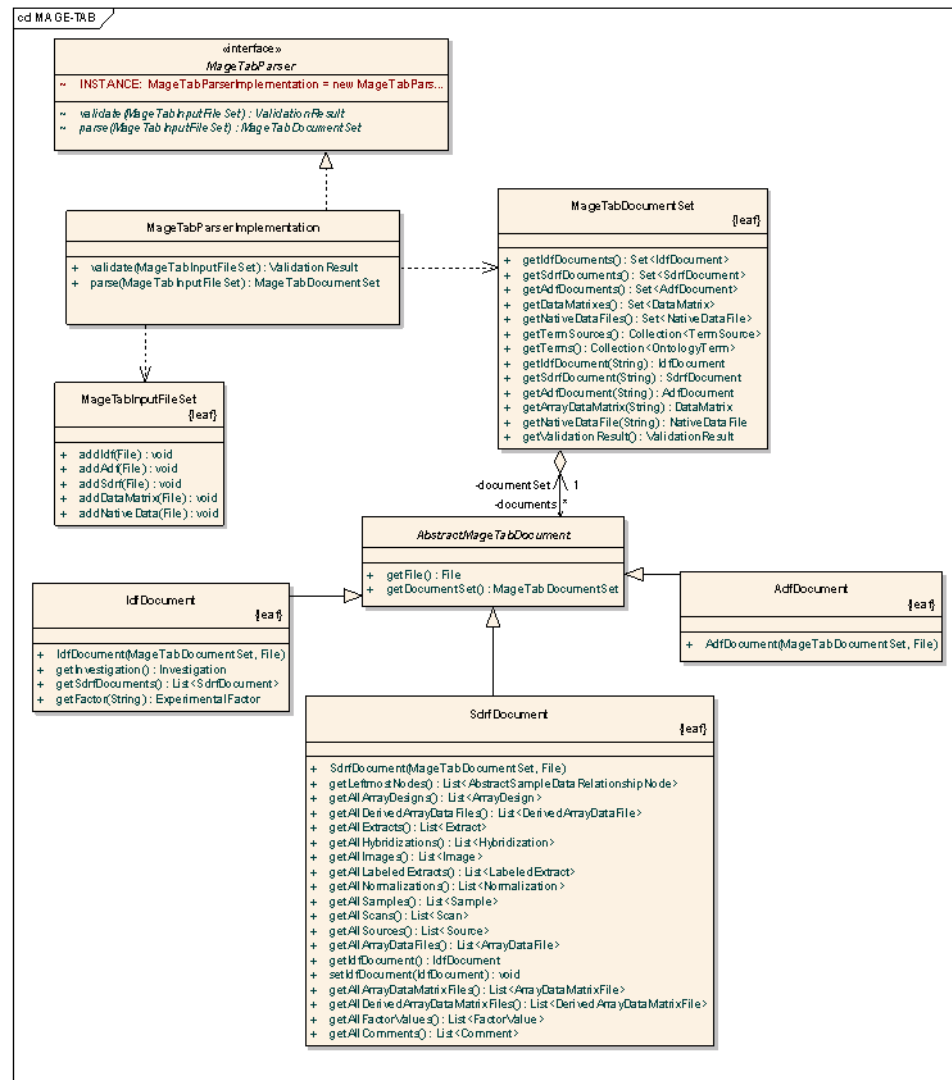


Figure 3.15 MageTabParser Subsystem implementation

## 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 3.16](#).

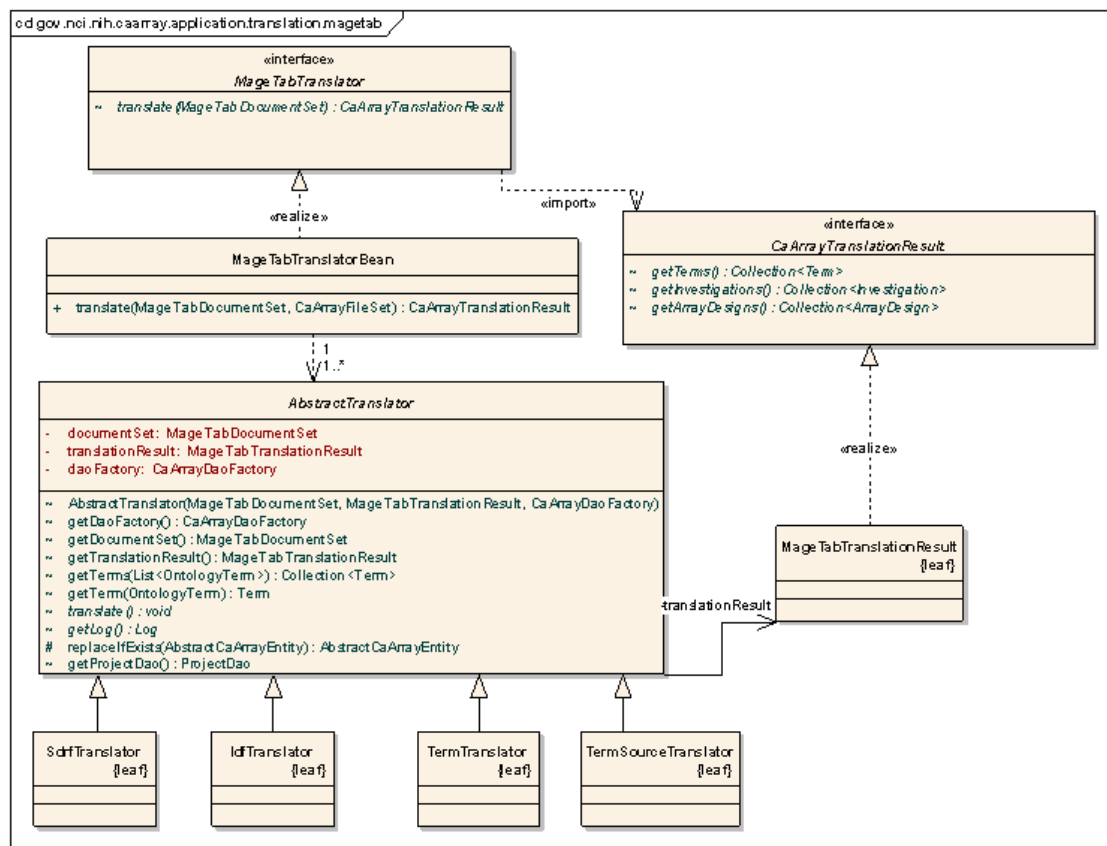


Figure 3.16 MageTabTranslator Subsystem implementation

## 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 3.17](#).

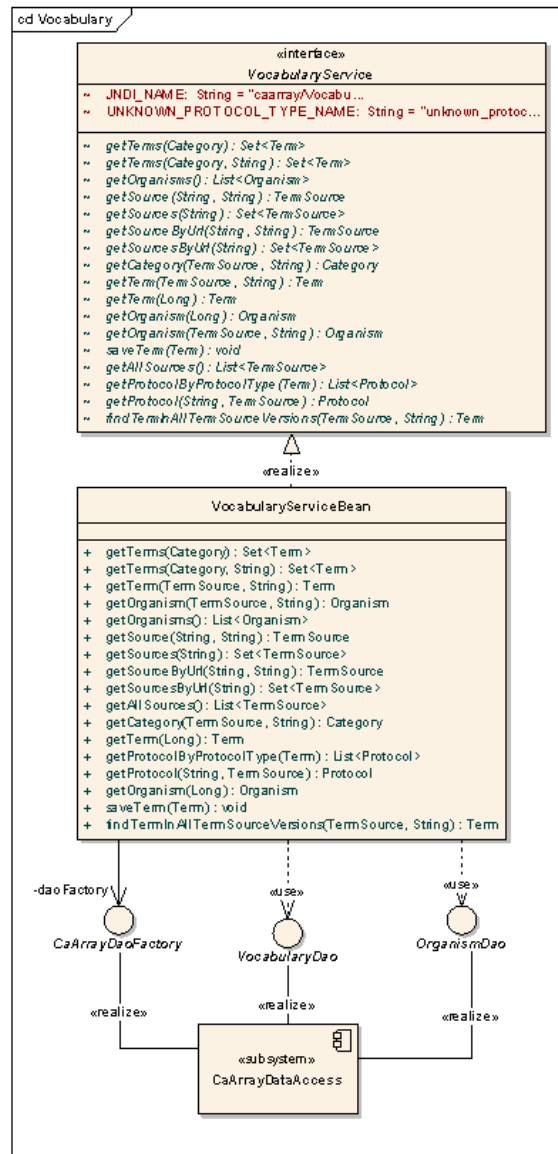


Figure 3.17 *VocabularyService* implementation

## 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

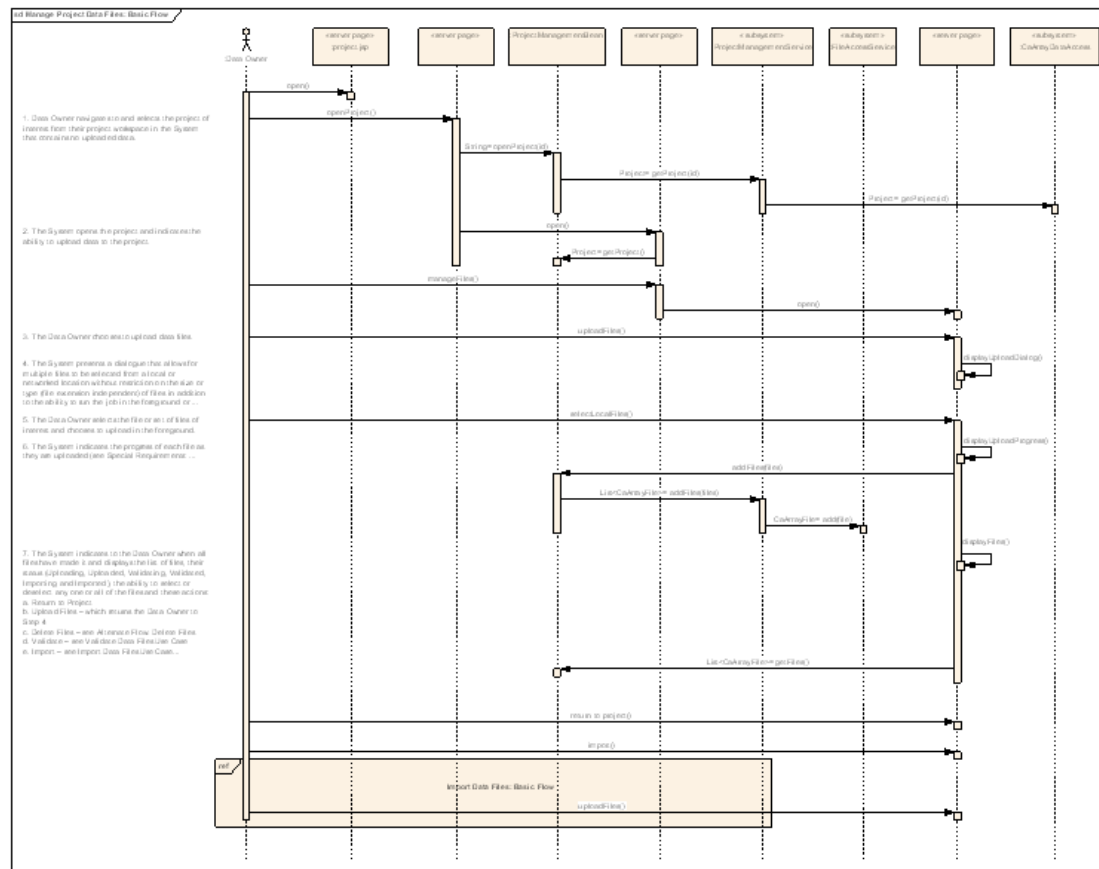


Figure 3.18 Manage Experiment Data Files basic flow sequence diagram







# CHAPTER 4

## IMPLEMENTATION VIEW

### Overview

The major physical artifacts that comprise the caArray software deployment units are illustrated in [Figure 4.1](#). The major artifacts and their relationships to the subsystems they realize are described in the following section, [Artifacts](#).

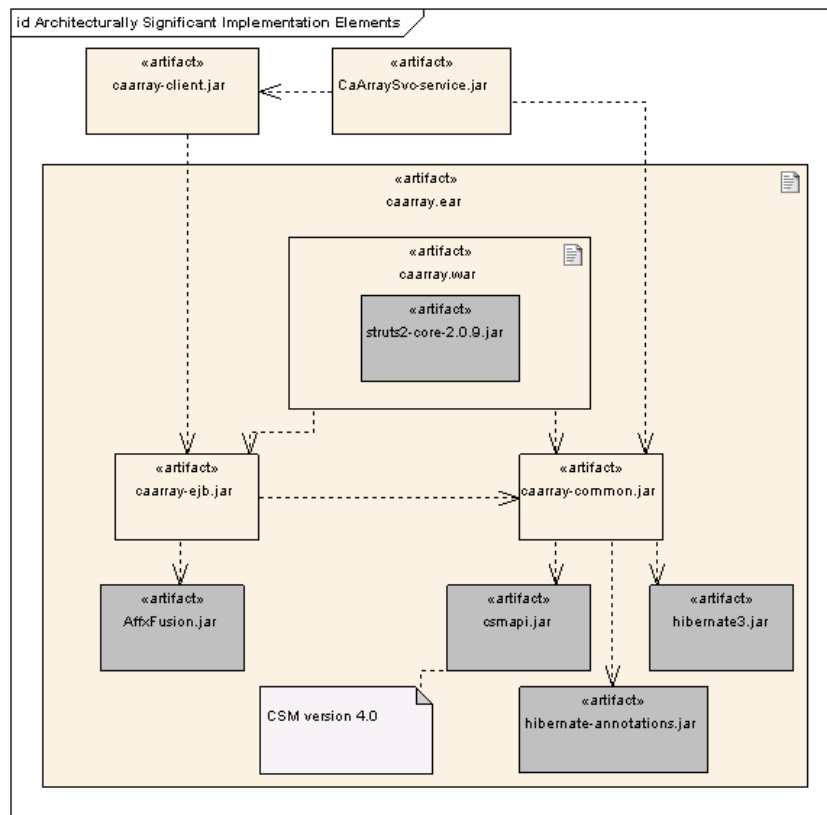


Figure 4.1 Architecturally significant implementation elements

## 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

# 5

## DEPLOYMENT VIEW

The typical deployment configuration for caArray is documented in [Figure 5.1](#). 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.

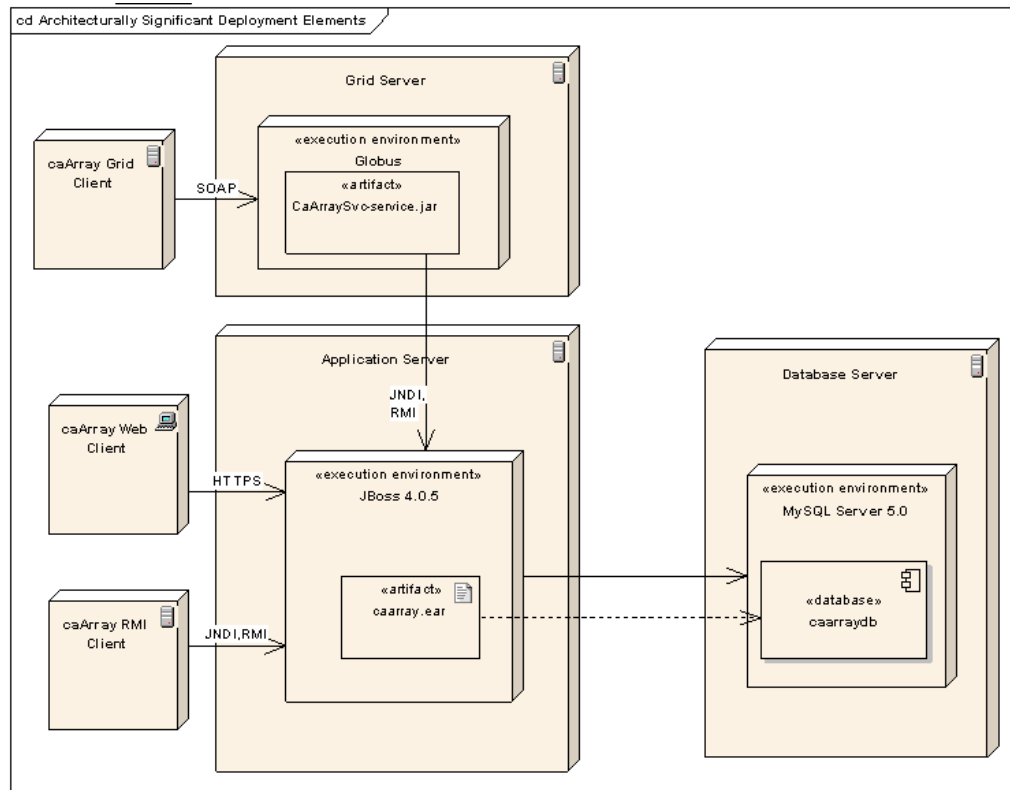


Figure 5.1 Architecturally Significant Deployment Elements

# INDEX

## A

- access control, CSM 4.0 [22](#)
- Acquire Experiment Data via API [35](#)
- acquire experiment data via API, use case [9](#)
- API
  - grid [14](#)
  - Remote Java [16](#)
- API, extracting data using [9](#)
- architectural
  - constraints [6](#)
  - goals [6](#)
- architecture
  - description of views [5](#)
  - logical view overview [11](#)
  - use case view [7](#)
- ArrayDataService [27](#)
- ArrayDesignService [29](#)

## C

- caArray
  - description [1](#)
  - domain classes [17](#)
  - Technical Guide description [1](#)
  - User's Guide text conventions [2](#)
- caarray.ear [38](#)
- caarray.war [38](#)
- caarray-client.jar [38](#)
- caarray-common.jar [38](#)
- CaArrayDataAccess [32](#)
- caarraydb [33](#)
- caarray-ejb.jar [38](#)
- CaArraySvc-service.jar [38](#)
- caBIG silver compliance [6](#)
- CSM 4.0 access control [22](#)

## D

- data retrieval [6](#)
- data storage [6](#)
- deployment view [39](#)
- document conventions [3](#)

- domain classes [17](#)

## E

- extracting data via API [9](#)

## F

- FileAccessService [24](#)
- FileManagementService [26](#)

## G

- grid API [14](#)

## H

- High Performance Data Parsing [6](#)

## I

- Import Data Files [35](#)
- import experiment data, use case [9](#)

## J

- J2EE 1.4 application [11](#)

## L

- logical view, overview [11](#)
- login use case [8](#)

## M

- manage array designs [9](#)
- Manage Experiment Data Files [34](#)
- manage experiment data files, use case [8](#)

## P

- ProjectManagementService [23](#)

## R

- remote API usability [6](#)
- Remote Java API [16](#)

## S

silver compliance, caBIG [6](#)

structural hierarchy [11](#)

subsystems

dependencies [11](#)

description [11](#)

figure [12](#)

## T

text conventions in user guide [3](#)

## U

use case

acquire experiment data via API [9](#)

import experiment data [9](#)

login, description [8](#)

manage array designs [9](#)

manage experiment data files [8](#)

validate experiment data files [9](#)

use case view [7](#)

user interface layer [13](#)

## V

validate experiment data files, use case [9](#)

view, architecture descriptions [5](#)