caCORE Software Developer Kit (SDK)

Version 4.1 - Programmer's Guide

NATIONAL® CANCER INSTITUTE

Center for Biomedical Informatics
and Information Technology

November 14, 2008

Table of Contents

About This Guide

This preface introduces you to the caCORE SDK 4.1 Programmer's Guide. Topics in this preface include:

Intended Audience

The *caCORE Software Developer Kit (SDK) V 4.1 Programmer's Guide* is the companion documentation to the Cancer Common Ontologic Representation Environment Software Development Kit (caCORE SDK). The caCORE SDK is a set of development resources that allows you to create, compile, and run caCORE-like software. The SDK is designed to aid programmers with life science backgrounds who are interested in using or extending the capabilities of caCORE.

For more information about caCORE, see the NCICB website: http://ncicb-dev.nci.nih.gov/infrastructure/cacore_overview.

Recommended Reading

Following is a list of recommended reading materials and resources that can be useful for familiarizing oneself with concepts contained within this guide.

- Java Programming
- Enterprise Architect Online Manual
- ArgoUML Online Manual
- Hibernate

Uniform Resource Locators (URLs) are also included throughout the document to provide more detail on a subject or product.

Organization of this Guide

The caCORE SDK 4.1 Developer's Guide contains the following chapters:

- *Chapter 1 Overview of caCORE SDK* - This chapter provides an overview of caCORE SDK 4.1, describes new features of the 4.1 release, and provides instructions for obtaining the release.

- *Chapter 2 Code Generation Technical Overview*- This chapter describes the code generation process in the context of the caCORE SDK. It also describes how the caCORE SDK's code generation module works.
- *Chapter 3 Runtime System Technical Overview* - This chapter describes the architecture of the caCORE system. It includes information about the major components, such as security, logging, database object-relational mappings (ORM), client-server communication, and system connection to non-ORM systems.
- *Chapter 4 Security* – This chapter provides information regarding the available security implementations for an SDK generated system.
- *Chapter 5 Writable API* – This chapter provides information on using the newly available Writable API to access an SDK generated system.
- *Chapter 6 Data Validation* – This chapter provides information regarding the data validation settings available with the Writable API in the SDK generated system.
- *Chapter 7 Logging/Audit Trail Management* - This chapter describes the CLM-based auditing/logging functionality that becomes available when the Writable API is enabled in the SDK.
- *Chapter 8 Using SDK Client Interfaces* – This chapter provides examples to access the generated system's client interfaces by a client application or a user. It includes information on how to access the system when security is enabled.
- *Chapter 9 Utilities* – This chapter describes a class that can be used to serialize and deserialize generated Java Beans to XML and back again.
- *Chapter 10 Creating the UML Model for caCORE SDK* – This chapter provides information on how to create UML models that can be used by the caCORE SDK to generate the system.
- *Chapter 11 Configuring and Running the SDK* – This chapter describes how to configure the SDK Code Generator and generate the system.
- *Chapter 12 Configuring Security* – This chapter provides instructions for configuring different types and different levels of security for an SDK generated system.
- *Appendix A Troubleshooting* - This appendix includes questions and scenarios that have been reported by SDK users and may be helpful in troubleshooting a problem when setting up the SDK.
- *Appendix B Performance Tuning the Java API* – The SDK development team and many of the SDK users have encountered problems when applying the SDK to their own use cases and workflows and have discovered solutions to improve performance. This chapter includes some of the solutions discovered by these users.
- *Appendix C Planned Features for Future Releases* - This appendix contains a short summary of some of the major features under consideration for a future release.

- *Appendix D Example Model and Mapping* - The caCORE SDK release package contains the example model included in this appendix, which can be used by the user as a reference to model a particular scenario for a system.
-

Text Conventions Used

This section explains conventions used in this guide. The various typefaces represent interface components, keyboard shortcuts, toolbar buttons, dialog box options, and text that you type.

Convention	***Description***	***Example***
Bold	Highlights names of option buttons, check boxes, drop-down menus, menu commands, command buttons, or icons.	Click **Search**.
URL	Indicates a Web address.	http://domain.com
text in SMALL CAPS	Indicates a keyboard shortcut.	Press ENTER.
text in SMALL CAPS + text in SMALL CAPS	Indicates keys that are pressed simultaneously.	Press SHIFT + CTRL.
Italics	Highlights references to other documents, sections, figures, and tables.	See *Figure 4.5*.
Italic boldface monospace type	Represents text that you type.	In the **New Subset** text box, enter ***Proprietary Proteins***.
Note:	Highlights information of particular importance.	**Note:** This concept is used throughout this document.
{ }	Surrounds replaceable items.	Replace {last name, first name} with the Principal Investigator's name.

Credits and Resources

caCORE SDK Development and Management Teams			
SDK Development Team	***Other Development Teams***	***Documentation***	***Program Management***
Satish Patel [1]	Kunal Modi [1]	Satish Patel [1]	Denise Warzel [4]
Dan Dumitru [1]	Vijay Parmar [1]	Dan Dumitru [1]	Avinash Shanbhag [4]
Aynur Abdurazik [2]	Shaziya Muhsin [2]	Charles Griffin [1]	George Komatsoulis [4]
Santhosh Garmilla[1]	Konrad Rokicki [2]	Bronwyn Gagne[5]	Charles Griffin [1]
Xiaoling Chen [2]	Ye Wu [2]		Dave Hau [4]
	Christophe Ludet [3]		Bilal Elahi [6]
	Eugene Wang [2]		
[1]Ekagra Software Technologies	[2] Science Applications International Corporation (SAIC)	[3] Oracle Corporation	[4] Nat'l Cancer Inst. (NCI) Center for Biomedical Informatics and Information Technology (CBIIT)
[5] Lockheed Martin	[6] Sapient		

SDK Resources	
Name	***URL***
Mailing List	CACORESDK_USERS-L@mail.nih.gov
Mailing List Archive	https://list.nih.gov/archives/ cacore_sdk_users-l.html
Project Home (GForge)	https://gforge.nci.nih.gov/projects/cacoresdk/
SDK Support Tracker (GForge)	https://gforge.nci.nih.gov/tracker/?group_id=148&atid=731

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

Submitting a Support Issue

A GForge Support tracker group, which is actively monitored by caCORE SDK developers, has been created to track any support requests. If you believe there is a bug/issue in the caCORE SDK software itself, or have a technical issue that cannot be resolved by contacting the NCICB Application Support group, please submit a new support tracker using the following link:

https://gforge.nci.nih.gov/tracker/?group_id=148&atid=731.

Prior to submitting a new tracker, review any existing support request trackers in order to help avoid duplicate submissions.

Release Schedule

This guide has been updated for the caCORE SDK 4.1 release. It may be updated between releases if errors or omissions are found. The current document refers to the 4.1 version of caCORE SDK, released in November 2008 by CBIIT.

Chapter 1 Overview of caCORE SDK

This chapter provides an overview of caCORE SDK 4.1, describes new features of the 4.1 release, and provides instructions for obtaining the release.

Topics in this chapter include:

Introduction

The National Cancer Institute (NCI) Center for Biomedical Informatics and Information Technology (CBIIT) provides biomedical informatics support and integration capabilities to the cancer research community. CBIIT has created the caCORE Software Development Kit or caCORE SDK, a data management framework designed for researchers who need to be able to navigate through a large number of data sources. caCORE SDK is CBIIT's platform for data management and semantic integration, built using formal techniques from the software engineering and computer science communities.

By providing a common data management framework, caCORE SDK helps streamline the informatics development throughout academic, government and private research labs and clinics. A caCORE SDK generated system is built on the principles of Model Driven Architecture (MDA) and n-tier architecture and consistent API. Model Driven Architecture (MDA) is a software development practice that uses a structured modeling language to describe the requirements, objects, and interactions of a data system prior to its construction. The use of MDA and *n*-tier architecture, both standard software engineering practices, allows for easy access to data, particularly by other applications.

caCORE SDK Modules

The caCORE SDK is comprised of two modules, as shown in Figure 1-1 below. The first module is the Code Generation Module, which accepts a UML model as input and produces various artifacts corresponding to the model as output. The second module is the Runtime System, which is a pre-built system and utilizes the artifacts generated by the code generation module in order to serve the data to the client application.

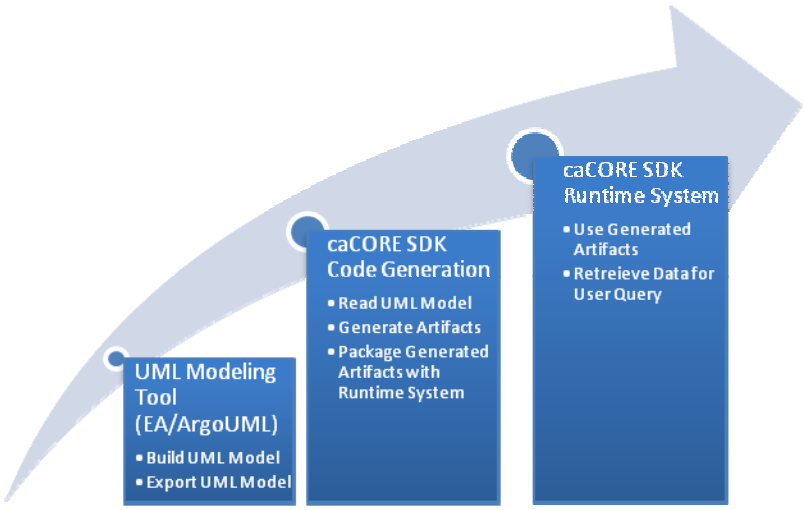


Figure 1-1 SDK System Generation Process

Chapter 2, Code Generation Technical Overview beginning on page 15 describes the architecture of the code generation module and an overview of the artifacts that the caCORE SDK generates. *Chapter 3, Runtime System Technical Overview* beginning on page 23 provides an overview of the architecture of the runtime system and describes the variety of ways it can deliver the data to the client.

caCORE SDK Users

There are basically two types of caCORE SDK users, grouped by which module they will use: 1) users of the code generation module and 2) users of the runtime system. Users of the code generation module focus primarily on preparing the UML model and running it through the caCORE SDK, using appropriate settings to generate the runtime system. Users of the runtime system focus primarily on writing queries against the runtime system to retrieve the data from the data source. *Chapter 11, Configuring and Running the SDK* beginning on page 115 provides information regarding use of the code generation. *Chapter 8, Using SDK Client Interfaces* beginning on page 47 provides information about how to access and use the runtime system through the available client interfaces.

SDK within the caCORE Environment

The caCORE SDK can be utilized to quickly generate a system from a caBIG® silver-level compatible UML model. For more information on caBIG compatibility levels, refer to the caBIG website at https://cabig.nci.nih.gov/guidelines_documentation.

The use of the SDK, however, is not limited to generating a system from silver-level compatible models. The SDK can be used outside of the caCORE environment to create a system that is generated from a UML model and which runs on standardized query languages. Within the caCORE application development process, the caCORE SDK serves the purpose of generating the system from the UML model after semantic integration is completed. More details on how to create a UML model for use with the SDK can be found in *Chapter 10, Creating the UML Model for caCORE SDK* beginning on page 83.

Benefits of Using the caCORE SDK

Users of the caCORE SDK benefit in numerous ways. The primary benefits of using the caCORE SDK includes:

- **Consistent UML representation of the data** – Users of the caCORE SDK are required to represent their data in UML format. As a user of the SDK, the user is likely to maintain their UML model throughout the life cycle of the application. The same UML model can be used to quickly learn about the organization of the data at various levels in the application.
- **Rapid data service generation** – The SDK can generate caBIG silver-level compatible APIs quickly from the UML model. Once the UML model and the database are ready, the data service can be generated in a matter of hours. Manually building the application from the ground up can take several months to achieve the same functionality.
- **Uniform way to access data** – SDK-generated systems provide uniform access to the data stores. Other applications developed using the caCORE SDK have similar mechanisms to retrieve the data. Thus common data representation allows multiple applications to share data.
- **Query using information model** – SDK-generated systems allow queries to be written in various ways including using Query-By-Example. Since the query is independent of the system's implementation, changes in the runtime systems do not affect the client application.
- **Integration with caGrid** – SDK-generated systems can be easily integrated with the caGrid using caGrid's Introduce Toolkit. Developing caGrid-compatible data services without using caCORE SDK can result in error-prone and lengthy processes.

New Features for caCORE SDK 4.1

The SDK 4.1 release contains many new features and enhancements. This section provides an overview of the major changes in the 4.1 release.

If you are currently using the SDK 3.2 version and are not yet familiar with the changes made for the 4.0 SDK, please refer to *Features Introduced in caCORE SDK 4.0* beginning on page 9 for information on the features introduced for the 4.0 release.

Code Generation

The features that are part of the code generation module work in harmony with the SDK generated middleware system to meet the system use cases. This subsection includes brief overview of the code generation features that are either newly added or have been enhanced.

- **Abstract Classes**

  Previous versions of the caCORE SDK ignored the property of the UML class that indicated that the class was abstract (i.e. one cannot create an instance of the generated class). The SDK 4.1 is now capable of recognizing the abstract property of the UML class and generating appropriate artifacts.

- **Market Interfaces**

  Interfaces without any constant member variables and methods are known as marker interfaces. For example, *java.io.Serializable* is a marker interface which tells the JVM that the class data can be serialized into a byte stream. Using a similar principle, the SDK allows users to specify the marker interfaces in their UML model that the domain objects can implement.

- **Implicit Inheritance**

  Abstract classes are classes which cannot be instantiated. When they are mapped to the database, they do not possess independent mapping in the database. In these cases, the attributes and the associations of the parent class are mapped in the table for the child class. The SDK 4.1 release now supports this mapping strategy.

- **Enhanced Hibernate Mapping file generation**

  Hibernate mapping files generated by the caCORE SDK had previously been capable of efficiently serving the data to the users of the SDK generated system. However these generated files were not capable of supporting writes or updates in the database. In the 4.1 release, the SDK code generator has been enhanced to support the missing features needed for the write/update functionality. Users can now specify the needed settings at the UML model level, which include:

  - Primary Key generator setting
  - Cascade style setting
  - Inverse-of setting
  - Eager loading setting

- **XSD and XML Mapping file generation based on GME Namespace**

  The users of the caCORE tools register their UML models in the caDSR registry for semantic integration purposes. While creating the grid service, the model owners also register the information in the Grid Metadata Exchange (GME) registry. A new feature developed across the caCORE toolset allows users to harmonize the model information across the metadata registries using cross-linking information (e.g., GME Namespace). The caCORE SDK has been enhanced to generate the XSDs and XML mapping files based on the same registered namespace information.

- **Validation**

  The caCORE SDK 4.1 has a new feature which allows users to perform validation against the caDSR's permissible values. The SDK downloads the permissible values and injects them in the generated beans as annotations so that they can be used at runtime.

Generated System

The generated system or middleware system of the caCORE SDK has been significantly enhanced in the 4.1 release, including the new features listed below.

- **Grid CQL Integration in SDK**

  The caCORE SDK and caGrid both allow users to form queries and get results from the underlying system. The syntax of queries for both systems, however, differed, creating a difficult path for migration for users who wanted to move from the SDK to caGrid. The caCORE SDK now provides mechanism to pass caGrid's CQL query to the SDK. When migrating to caGrid, users can fire the same CQL query against the caGrid data service.

- **Writable API**

  The API generated by previous releases of the caCORE SDK was capable of serving data in read-only fashion. In 4.1, the caCORE SDK provides a fully functional API, capable of performing operations like create, update, and delete.

- **Auditing/Logging**

  As the data is being manipulated through the writable API, it is important to maintain a trail of what is being manipulated. The caCORE SDK provides an optional logging capability, which inserts the log statements into the database with help of Common Logging Module (CLM). CLM also provides a companion application called Log Locator Tool, which allows users to review the inserted log statements in a secured fashion.

- **Grid Security Infrastructure support**

  As more and more users start using the caGrid infrastructure, the user accounts managed at the local level using Common Security Module (CSM) need to be migrated to the grid level. At the grid level, each user is assigned a unique Grid Identity for authorization purposes. caCORE SDK 4.1 now contains a feature that allows users to use their caGrid user accounts in the SDK generated system's environment, across all tiers.

- **Validation**

  Validation was mentioned earlier in the new Code Generator feature section. The SDK-generated middleware system at runtime utilizes the generated validation annotations to validate the data that is being manipulated through the API.

Features Introduced in caCORE SDK 4.0

The 4.0 release of the caCORE SDK was a major release containing many new features. Some of the features strengthened the infrastructure while others supported new requirements. The purpose of this section is to highlight the major functionality, performance enhancements, and improvements introduced into caCORE SDK 4.0.

Code Generation

The architecture and the core of the code generation module of the caCORE SDK was completely rewritten for version 4.0. The entire code generation framework now runs from a single configuration file based on the Spring Framework, as opposed to the

individual configuration files used by the previous releases. Some of the visible improvements in the code generation module are highlighted below.

- **Support for Enterprise Architect and ArgoUML**

  Previous releases of caCORE SDK supported only Enterprise Architect for UML modeling. With SDK 4.0, users can choose between ArgoUML and Enterprise Architect. The added support for ArgoUML provides users an open source alternative to commercial software like Enterprise Architect.

- **Performance Improvement in Code Generation**

  The caCORE SDK 4.0 contains significant improvements in the performance of the code generation module. Average users should notice completion of the system generation process in approximately 15% of the time of previous SDK versions.

- **Support for Validators**

  The caCORE SDK code generator now supports validators. Validators serve the purpose of validating the object model and object relational mapping information before the code generator starts. These validators provide descriptive messages to users, allowing users to more quickly identify the root cause of any code generation failures.

- **Reduced and Improved Generated Artifacts**

  The artifacts generated by the caCORE SDK 4.0 were completely redesigned to suit the needs of the redesigned runtime system. Artifacts generated by previous SDK releases were not reusable outside of it due to certain dependencies. Artifacts generated by SDK 4.0 can be reused anywhere. For example, Java beans generated by the previous SDK had getter methods to connect to the server; they were not simple POJOs. In SDK 4.0 these are simple POJO beans.

  The following table lists the artifacts that changed between 3.2.x and 4.0:

Artifact	*SDK 4.0 Includes*	*SDK 3.2.x Included*
POJO beans for domain objects (*.java)	yes	yes
SDK specific Java beans for domain objects (*.java)	yes	no
“Impl” classes for Java beans (*.Impl.*.java)	yes	no
Web Service beans (*.ws.*.java)	yes	no
“Impl” classes for web service beans (*.ws.impl.*.java)	yes	no
JUnit test cases for domain objects	yes	no
Hibernate O/R mapping files for domain objects (*.hbm.xml)	yes	yes
Hibernate O/R mapping files for “Impl” classes (*Impl.hbm.xml)	yes	no
Hibernate configuration file (*.cfg.xml)	yes	yes
Hibernate cache configuration file (ehcache.xml)	yes	yes
SDK DAO configuration file (DAOConfig.xml)	yes	no
Domain object list (coreBeans.properties)	yes	no
Association mapping file (roleLookup.properties)	yes	no
XML Schema for domain model (*.XSD)	yes	yes
Castor mapping files (xml-mapping.xml, xml-unmapping.xml)	yes	yes
Web service deployment descriptor (server-config.wsdd)	yes	yes

Table 1-1 SDK 4.0 Artifacts

Additional UML Features Supported

caCORE SDK 4.0 supports many new UML features in the object model and in the object relational mapping aspect.

- **Object Model**
  - **ID attribute** – Users of the caCORE SDK do not have to use the name "ID" for the attribute that maps to the primary key column of the corresponding table for the class. Users can now specify the attribute mapping to the primary key column using a tag value on the class in the domain model.
  - **Primitives support** – SDK 4.0 allows users to specify Java's primitive type for any attribute's data type. SDK 4.0 interprets these primitives in the wrapper data type during code generation.
  - **Collection of primitives** –Users of the SDK can now use a collection of primitives or wrapper data types as the type for the attribute.
- **Object Relational Mapping**
  - **Inheritance** – caCORE SDK 4.0 supports an alternate way of mapping inheritance hierarchy in the database. SDK users can choose the previous *Table per class* mechanism to map inheritance in the database, or they can choose *Table per inheritance* hierarchy for the mapping.
  - **Join tables** – Previous releases of the SDK supported join tables only for the many-to-many type associations. With SDK 4.0 users can choose to use join tables for any type of association.

Generated System

In addition to the new code generator module, caCORE SDK 4.0 introduced significant changes in the runtime system. Since many of the changes are in the infrastructure, typically only users utilizing the advanced options will notice or be affected by the restructuring of the SDK's runtime system.

- **Client Server Infrastructure**
  - The client-server infrastructure of the SDK used to rely on the Java beans developed specifically for SDK. These specialized java beans had the capability to connect to the server when required to fetch the associated objects. With SDK 4.0, regular POJOs are used in conjunction with concepts from Aspect Oriented Programming (AOP) to facilitate a similar mechanism. With this design approach, domain object beans generated by the SDK are true POJOs and can easily be used outside of the SDK.
  - In addition to the restructuring of the Java beans with AOP, SDK 4.0 can also connect to various SDK-generated systems from within the same client JVM. In previous versions, users of the SDK could connect to only one remote service at a time; with this feature, developers can retrieve data from multiple data services.
- **Simplified Application Service**

Many of the existing methods of the ApplicationService interface have been deprecated. Newly added methods have syntax similar to the existing methods

but they now require less information. The simplified Application Service will be easier to work with.

- **Web Services**
  - SDK 4.0 generated web services work on the simple POJO beans. The web service from previous SDK versions required specialized POJO beans in the .ws package, whereas SDK 4.0 generated web services utilize the same Java beans that are used by other tiers of the application.
  - SDK 4.0 web services also have additional methods to allow users to fetch the associations of a domain object. Users can now specify which specific association they would like to fetch from the server.
  - Starting with version 4.0, users of the SDK will not have to deploy the web service independently. The SDK 4.0 generated web services are embedded in the .war file and are deployed automatically when the application server starts
- **Graphical User Interface**
  - The caCORE SDK 4.0 generated system has a newly developed graphical user interface, providing users a richer experience.
  - Security of the new user interface has been enhanced. Users now have access to built-in security capabilities such that when the security is enabled in the system, users experience a completely secured system and not just a secured interface.
  - The caCORE SDK 4.0 generated GUI now has embedded JavaDocs for the domain objects for which the system was originally generated. Users of the web interface can browse the JavaDocs by visiting a link on the generated system’s home page.
  - Previous releases of the SDK did not allow fetching of an associated object that had more than one association with another object. The newly generated web interface allows users to retrieve associations regardless of the number of associations between objects.
- **Security**
  - The caCORE SDK 4.0 has a security implementation that is based on Acegi security framework. The previous implementation of security in the caCORE SDK was weaved into the application logic. As of caCORE SDK 4.0, security implementation is kept outside of the application and is managed through Aspect Oriented Programming principles. SDK users can now change the implementation of security without going into the details of the SDK’s code base.
  - Instance level security – The caCORE SDK 4.0 supports instance level security utilizing CSM, which provides flexibility to provide more granular access to the data. For example, users can be given access to only a subset of records from a particular table versus all the records of a particular table.
  - Attribute level security – In addition to the instance level security, the caCORE SDK 4.0 also provides very granular attribute level security to

the users. For example, only certain users may be allowed to see the Social Security Numbers of a Person object.

- **Concurrent user access in secured API**

Users of the SDK generated java client in the previous releases were constrained to use the same user account throughout the lifecycle of the ApplicationService. In SDK 4.0, users can create many different instances of the ApplicationService and login with different user accounts at the same time from different threads of the client application.

Obtaining the caCORE SDK

The caCORE SDK is released periodically in .zip file format and .tar file format. Updates are released frequently on the NCICB's GForge website. The latest releases and archives can be obtained from https://gforge.nci.nih.gov/frs/?group_id=148.

caCORE SDK Minimum System Requirements

In addition to the caCORE SDK files that must be downloaded from the link above, additional hardware and software is also required.

Minimum Hardware Requirements

The caCORE SDK 4.1 has been built and tested on the platforms shown in Table 1-2.

Users of the caCORE SDK need a computer system for two purposes: first, to generate a system using the caCORE SDK, and second, to host the generated system in a production environment.

Users can use the tested configurations listed below as a reference to determine their appropriate hardware configuration. Hardware selections should be based on the amount of data the system is expected to handle.

	Linux Server	***Solaris***	***Windows***
Model	HP Proliant ML 330	Sunfire 480R	Dell GX 270
CPU	1 x Intel® Xeon™ Processor 2.80GHz	2 x 1050MHz	1 x Intel® Pentium™ Processor 2.80GHz
Memory	4 GB	4 GB	1 GB
Local Disk	System 2 x 36GB	(RAID 1) Data = 2 x 146 (RAID 1) System 2 x 72GB	System 1 x 36GB
Operating System	Red Hat Linux ES 3 (RPM 2.4.21-20.0.1)	Solaris 8	Windows XP/2000 Professional

Table 1-2 Minimum Hardware Requirements

Software Requirements

The software listed in Table 1-3 is required to use the SDK but is not included with the caCORE SDK download. Users must download and install the appropriate listed software.

The software name, version, description, and URL hyperlinks for download are indicated in the table.

Software	Description	Version	URL
JDK	The J2SE Software Development Kit (SDK) supports creating J2SE applications	1.5.0_11 or higher	http://java.sun.com/j2se/1.5.0/download.html
Enterprise Architect (EA)	UML Modeling Tool†	6.0 or higher	http://www.sparxsystems.com.au/
ArgoUML		0.24 or higher	http://argouml.tigris.org/
Oracle	Database Server†	9i	http://www.oracle.com/technology/products/oracle9i/index.html
MySQL		5.0.27	http://dev.mysql.com/downloads/mysql/5.0.html
JBoss	Application Server†	4.0.5	http://labs.jboss.com/jbossas/downloads
Tomcat		5.5.20	http://tomcat.apache.org/download-55.cgi
Ant	Build Tool	1.6.5 or higher	http://ant.apache.org/bindownload.cgi

Table 1-3 Minimum Software Requirements

† Only one is required.

Contributing to caCORE SDK Development

The caCORE SDK project is managed by a CBIIT project manager. If you would like to contribute by providing a patch for a particular defect, email the caCORE SDK Users' mailing list (CACORE_SDK_USERS-L@list.nih.gov). Users interested in participating in the development process can contact CBIIT management for more details.

Chapter 2 Code Generation Technical Overview

This chapter describes the code generation process in the context of the caCORE SDK. It also describes how the caCORE SDK's code generation module works.

Topics in this chapter include:

Introduction

Code generation is a systematic process of converting a model into a series of instructions or programs that can be executed by a machine. The principle of code generation is primarily popular in programming language compilers (for example, a C compiler or a Java compiler) in which the code generation stage is responsible for generating machine specific instructions or assembly language instructions.

The input to the code generation stage typically consists of parsed source code or an abstract syntax tree that is prepared by the source code parser. In the context of the caCORE SDK, the code generator generates the artifacts from a UML model using principles of Model Driven Architecture that are consumed by the SDK's runtime system.

The Role of Code Generation in the caCORE SDK

While other tools and programming language compilers use the code that the SDK generates, the SDK itself can be viewed as a level above the other compilers and tools. The code generation module is responsible for generating various artifacts from the UML model. Like output from the code generation stage of compilers, the output from the code generation stage of SDK is specific; the output of the caCORE SDK consists of artifacts like Java source code, O/R mapping files etc. In other words, the caCORE SDK transforms the UML model into system specific artifacts and the code generation engine is simply a complex transformer for the UML model.

Figure 2-1 Code Generation

The primary purpose of the caCORE SDK is to allow users to quickly build data services. One of the ways the SDK implements this requirement is to generate the application for

the user based on specified settings. The SDK takes a UML model, which consists of an object model and a data model, as input, and generates a complete application using the generation settings.

Features and Limitations of Code Generation

UML provides a generic mechanism to represent the various parts of a software system and its lifecycle. However, UML by itself is unable to describe how the complete system works after implementation. To efficiently generate code from a UML model, the SDK specifies additional information (in the form of tag values) that needs to be embedded inside the model. This additional information allows the SDK to determine how the code generation should proceed.

The SDK code generation sub-system can interpret only a set of well-known features from the model, which currently includes following:

1. UML packages
2. UML class
3. UML class attributes
4. UML attribute’s data type
5. UML association
6. UML dependency
7. UML tag values
8. UML generalization
9. UML Interface

The SDK cannot read and interpret unsupported features included in the UML model. To interpret unsupported UML features, the SDK code generator must be modified. In addition to modifying the code generator, the runtime system also requires modification in order to consume the modified artifacts from code generator.

Code Generation Process

The SDK code generation process can be viewed as a layer of different processes. In order to generate code from a UML model, the constructed model must first be exported from a UML modeling tool. Then, the exported model can be used by the SDK to generate the code.

The code generation process, illustrated in Figure 2-2, involves the following high-level steps to generate the artifacts required.

1. Read UML model.
2. Generate artifacts.
3. Manage Output.

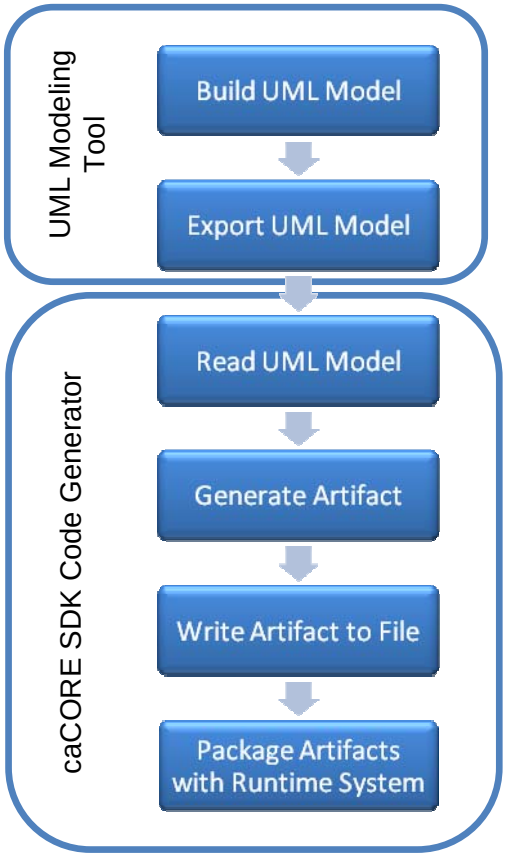


Figure 2-2 Code generation process details

Reading the UML Model

When the UML model is constructed in the UML modeling tool, the information about the model is stored in a proprietary formatted file. In order for the SDK to read the model, the model information needs to be translated into a standard format that can be interpreted by other modeling tools. Once the model information is exported in the standard format, the SDK can read the model information and convert it into internal data structures. These internal data structures can then be used by other stages of the SDK to generate the desired code. Having the internal data structures also gives additional flexibility to the SDK. In the event that a modeling tool adopts a new standard or starts exporting information in some format other than the one recognized by SDK, a new model reader can be developed without affecting other stages of code generation.

Currently the SDK uses a UML model reader that was developed as a separate project at CBIIT. The UML Model Reader, also known as the XMI Handler, can interpret model

information created by either Enterprise Architect or ArgoUML. Because the UML Model Reader is separate, if the SDK code generator ever needs to support a different UML modeling tool, the UML Model Reader can be extended without the need to change the code generator.

Artifact Generation (Model Transformation)

The SDK generates various artifacts based on the information that it obtains from the UML model. Artifacts can be Java beans, O/R mapping files, or web service deployment descriptors. Most artifacts are generated from information obtained from the UML model. (Other artifacts are generated from the property files and configuration files supplied at the time the SDK is run.)

The UML model contains various complex elements. The artifact generation stage reads all of the elements in the UML model and then constructs a collection of relevant elements from which a particular artifact can be generated. The artifact generation process must be repeated for each type of artifact.

Output Management

When an artifact is generated, the output must be written to a file. The file content can be Java source code or XML. If the artifact is a Java program then it must be written in a particular folder hierarchy to preserve namespace. In addition, all Java program files require ".java" as a file extension. Similarly, generated XML documents must be placed in appropriate folders and assigned appropriate file names and extensions.

Code Generation Framework

As explained earlier in the section, the code generation process involves various steps in order to generate an artifact. If there are many different types of artifacts to be generated, the model transformation process must be executed for each type of artifact. In the case of multiple artifacts, it becomes necessary to automate these steps so that the artifact generation process can be handled efficiently.

In the current design, the artifact generation process is controlled by means of a control or configuration file. The control file specifies what combination of components will be used to generate a particular type of artifact. The execution engine (Generator), which understands the information specified in the control file, can then read the control file and orchestrate the workflow as desired.

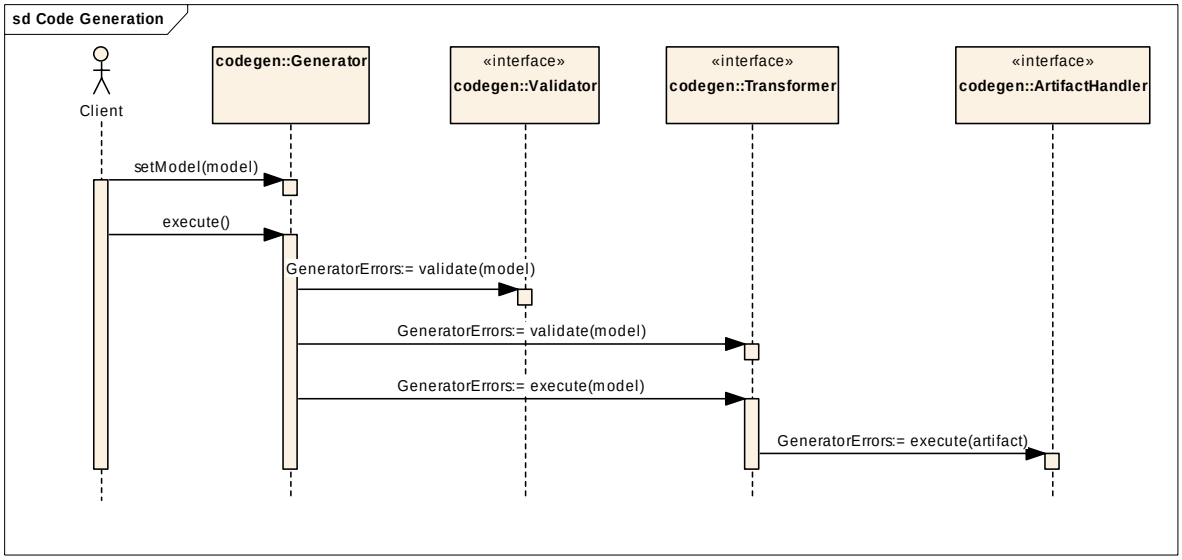


Figure 2-3: Code Generation Workflow Automation Sequence Diagram

As described in Figure 2-3, when the code generation execution engine initializes, it reads the control file and configures itself with the information obtained from the control file. The execution engine configuration involves initializing the components defined in the control file as sub-elements and configuring them one at a time. Once the configuration of the execution engine and components is finished, the code generation execution engine executes the workflow as described by the pseudo-code below.

1. Open UML Model file.
2. Read UML Model file containing various UML models.
3. Set the UML Model in the Generator.
4. Set the Validators in the Generator.
5. Set the Transformers in the Generator.
6. Execute the Generator.
    a. Execute all the registered validators.
    b. If the errors are present during the previous validation, then stop executing and log the errors.
    c. Execute the validate method of all the registered transformers.
    d. If the errors are present during the previous validation, then stop executing and log the errors.
    e. Execute all the registered transformers.
        i. Generate artifact from the UML Model.
        ii. If errors are discovered during code generation then return the errors.
        iii. Pass the generated Artifact to the registered ArtifactHandler.
            1. Write the artifact to the respective file.

Reusable Components of the Code Generation Workflow

In order to complete the code generation process, the components used in the code generation workflow must implement specific behaviors. As the code generation engine executes the workflow, it instantiates and configures components required for the workflow (as specified in the control file).

Each type of component needs some information to configure itself. This information is supplied from the control file to the component. The class that implements the interface (Validator or Transformer) is recognized by the code generation execution engine and is responsible for following certain behaviors expected by the engine. This mechanism allows the new implementation of the components to be plugged into the workflow by simply modifying the control file.

Since the SDK code generator uses a Spring Framework's bean configuration file, configuring each component becomes easy. It is up to the developer of a component to specify what information the component needs to execute itself.

Overview of SDK Generated Artifacts

As part of the code generation process, the caCORE SDK generates the following artifacts with the help of different transformers.

- **Beans** – For each object defined in the object model, a Java bean (POJO) is generated. The generated bean follows the same package structure as the folder structure in the object model. The generated Java beans are compiled and packaged in a JAR file. The JAR file is named *project_name-beans.jar.*
- **Hibernate files** – The following hibernate files are packaged in a separate JAR file after generation. The JAR file is named *project_name-orm.jar*
    - **Hibernate mapping files** - For each object defined in the object model, the caCORE SDK generates a Hibernate mapping file (Object Relational mapping file) by reading tag values that maps object and attributes to tables and columns in the data model. In the case of inheritance in the object model, the mapping file is created for the root level class in the inheritance hierarchy. The generated files follow the same package structure as the folder structure in the object model.
    - **Hibernate configuration file** – The SDK generates a configuration file named *hibernate.cfg.xml* for Hibernate that contains a list of generated Hibernate mapping files as well as the database connection settings.
    - **EHCache configuration file** – A cache configuration file for Hibernate.
- **XSD and XML Mapping files** – For each package defined in the object model, the caCORE SDK generates an XSD file. The XSD file is named after the fully qualified name of the package name for which the file was generated. If the UML model is annotated with semantic tags (CDE information from caDSR) then the generated XSD files will include this information as XSD documentation.

The SDK also generates XML mapping files (castor mapping files) for the entire object model. There are two XML mapping files that are generated: *xml-mapping.xml* and *xml-unmapping.xml*. These files are primarily used by the caGrid project to create a grid service from the SDK generated system.

- **Web Service deployment descriptor file** – A deployment configuration file for the AXIS-based web service is generated for the entire object model.

Chapter 3 Runtime System Technical Overview

This chapter describes the architecture of the caCORE system. It includes information about the major components such as security, logging, database object-relational mappings (ORM), client-server communication, and system connection to non-ORM systems.

Topics in this chapter include:

High-Level Architecture

The caCORE SDK generated runtime system's infrastructure exhibits an n-tiered architecture with client interfaces, server components, backend objects, data sources, and additional backend systems.

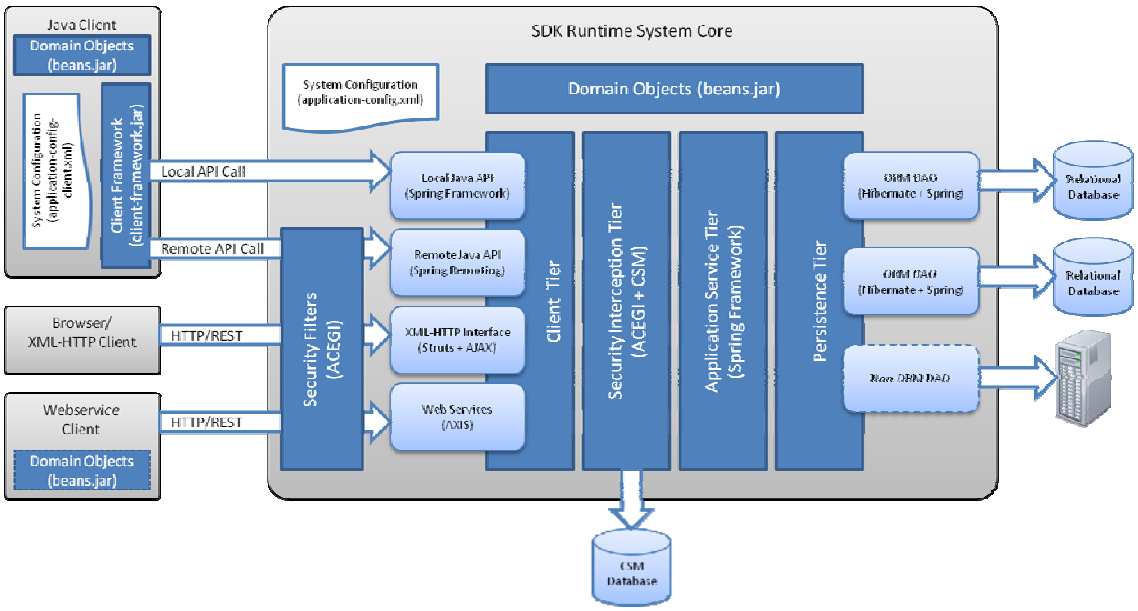


Figure 3-1 SDK Generated Runtime System Architecture

This n-tiered system divides tasks or requests among different servers and data stores. This isolates the client from the details of how or from where data is retrieved.

The system also performs common tasks such as logging and provides different levels of security. Clients (browsers, applications) receive information from backend objects. Java applications also communicate with backend objects via domain objects packaged within the client.jar. Non-Java applications can communicate via SOAP (Simple Object Access Protocol). Back-end objects communicate directly with data sources, either relational databases (using Hibernate) or non-relational systems (using, for example, the Java RMI API).

N-Tier System

The SDK generated system can be viewed as a typical n-tier architecture system where each tier in the system is responsible for a set of defined activities. In an SDK generated system, the layers, starting from the lowest layer, are as follows:

- *Persistence Tier*
- *Application Service Tier*
- *Security Interception Tier*
- *Client Interface Tier*
- *Security Filters*

Each of these layers is discussed in detail in the sections that follow.

Persistence Tier

The persistence tier is responsible for understanding the query that has been sent and for fetching the results. The SDK currently supports persistence tiers created in two ways; object-relational mapping (ORM) based persistence tiers and non-object-relational mapping (non-ORM) based persistence tiers.

To access the data stored in the persistence tier with the ORM-based mechanism, the SDK provides a pre-constructed DAO (*ORMDAOImpl*). The *ORMDAOImpl* is written specifically for Hibernate-based object relational mapping. This DAO converts the query into a Hibernate-specific query and executes it using Hibernate APIs. Each DAO also provides a list of domain objects, when requested by the Application Service tier, by using the *getAllClassNames()* method of the DAO.

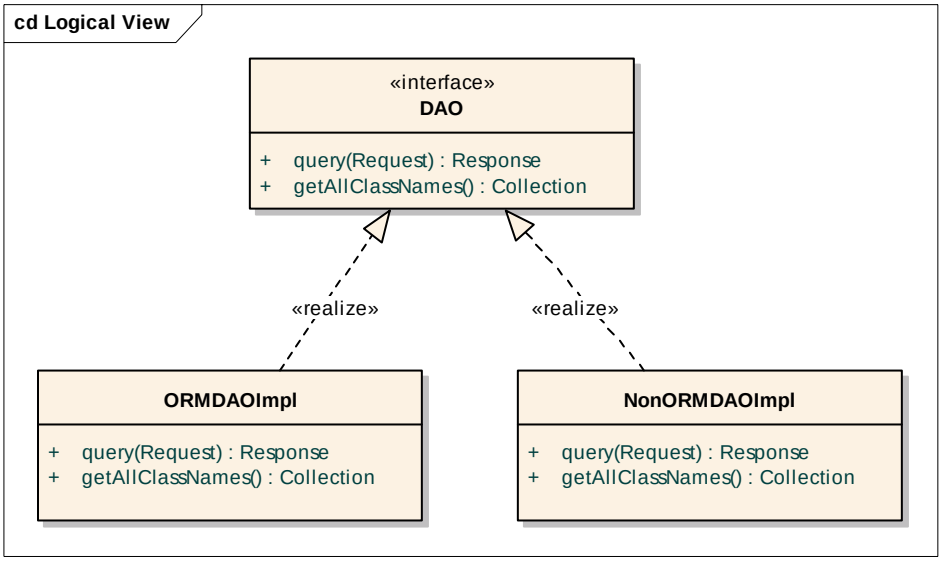


Figure 3-2 Persistence Tier Classes

If the Application Service tier determines that there is an overlap between the lists of domain objects provided by the DAOs then the application will not be loaded. Details on how each DAO works are provided in the following sections.

Object Relational Mapping

The SDK code generator performs the object relational mapping for Hibernate (http://www.hibernate.org) as its underlying technology. Hibernate allows the objects to be mapped to the relational database by means of ORM files. These ORM files are generated by the caCORE SDK during the code generation process. If a user does not intend to use one or all of these mapping files and instead provide independently developed mapping files, they can do so by altering the code generation process. Alteration of the code generation process can be done through the configuration files and is explained in *Chapter 11, Configuring and Running the SDK* beginning on page 115.

Non-Object Relational Mapping

SDK users can choose not to use ORM to map the relational database to the objects or the data for the objects residing on a remote server. In this scenario, the SDK user bears the responsibility of populating the objects based on the query. The user must develop a custom non-ORM DAO that can perform the task of retrieving the data. The custom non-ORM DAO is required to implement the interface expected by the caCORE SDK. Other than supporting the method to retrieve results using the query, the non-ORM DAO needs to implement another required method, *getAllClassNames()*, to return a list of the domain objects supported by that DAO.

Application Service Tier

The Application Service tier consolidates incoming requests from the various interfaces and forwards them to the appropriate persistence tier implementation. The Application Service tier is the main tier that facilitates the operations within the SDK generated system, and its methods in the ApplicationService interface are exposed to the Java Clients.

The class *ApplicationServiceImpl* has the concrete implementation of the ApplicationService interface. When any of the client interfaces in the SDK code requests a handle to the ApplicationService, the default implementation of ApplicationServiceImpl is returned. When the remote Java client requests a handle to the ApplicationService, a remote handle to the ApplicationService is wrapped inside the ApplicationServiceProxy and returned to the client.

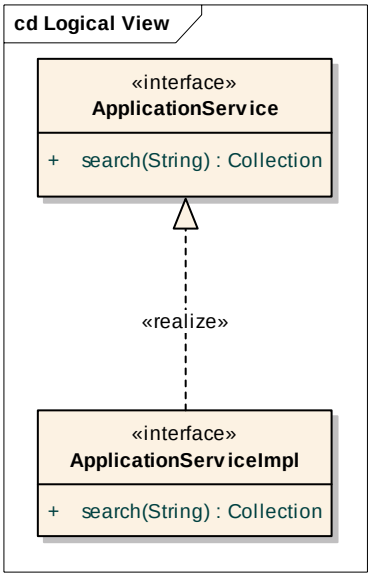


Figure 3-3 Application Service

The default Application Service tier has methods to support ORM based systems. These methods are sufficient to support requirements for most applications. However, users can extend the Application Service Tier by adding additional methods as described below.

Extending the Application Service Tier

If additional methods are needed in the Application Service tier, one possible approach is to modify the source code to add additional methods. Another option is to extend the Application Service and modify the configuration files to work with the extended Application Service. As shown in Figure 3-4 below, a *CustomApplicationService* can extend the ApplicationService interface, and the class *CustomApplicationServiceImpl* provides a concrete implementation of the method inside the *CustomApplicationService* interface.

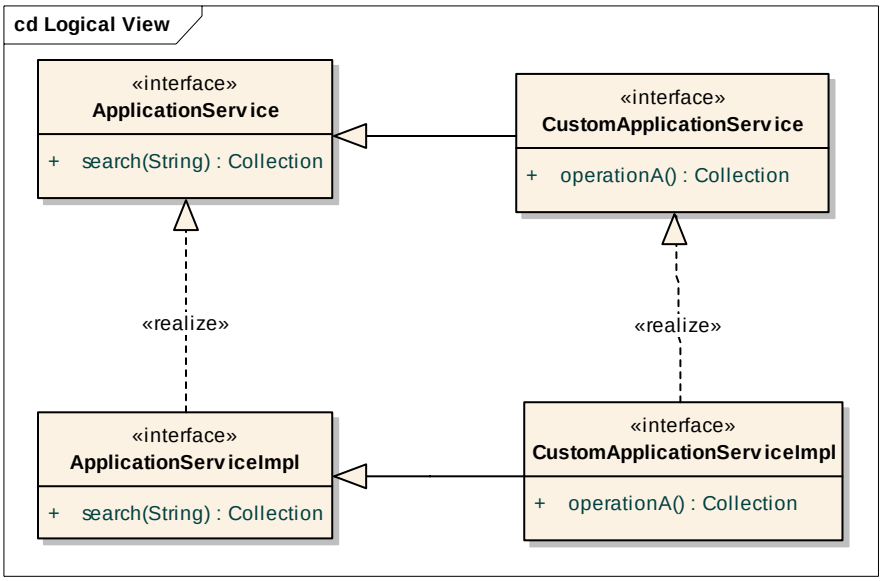


Figure 3-4 Extending the Application Service

As the new methods are added to the Application Service, it is also necessary to modify client tiers to expose the additional methods to their respective clients. The configuration files on the client and the server side also must be modified to reflect the extension of the Application Service.

Security Interception Tier

The Security Interception tier ensures that only authorized users are allowed to access the system. The security configuration in the SDK is done using the Acegi and the Common Security Module (CSM) developed by CBIIT. In the case of an unsecured system, this layer is disabled through the configuration files. Additional details on implementing security in the SDK can be found in *Chapter 4, Security* beginning on page 35.

Client Interface Tier

The SDK provides four distinct client interface methods to reach the Application Service Tier:

- XML-HTTP Interface (browsers, thin clients)
- Web Services Client
- Local Java API Client
- Remote Java API Client

Each of these methods of retrieving data involves preparing the query in the format that the interface understands, sending the request to the corresponding interface, and retrieving the results from the interface to which the query was submitted. The following sections provide details about system usage for each method.

XML-HTTP Client

The XML-HTTP client accesses data using two types of clients 1) a web browser to view data in the form of a web page and 2) a thin client to get data in XML format. The web based GUI interface or the Domain Model Browser was also known as Happy.jsp in the previous release. Using this interface, clients can form a query using Query-By-Example (QBE) syntax and are provided data for the result object. If a client intends to fetch data for the associated object then the client is required to make a second query.

In a web browser, a user can click on a link to fetch the associated object. In the thin client, the client application is required to form the query to fetch the associated object and send it to the server. If the query executed by the client returns a large number of records, the server returns only the results allowed per page size. The client is required to make a second call to fetch the next page from the server. If the client intends to fetch the data in the form of XML instead of a web page format, they can use a different URL (GetXML) with the same query parameters.

NOTE: The page size can be configured in the configuration file `application-config.xml`.

Web Services Client

SDK generated web services run on Apache Axis using a literal-based RPC web service protocol. The Web Services client uses this protocol to fetch data. When a query returns a large amount of data, the Web Service client only receives the maximum number of allowed records per call. The client application is required to make an additional call to the server to fetch the next chunk of data. The server also does not return the association to the client application. If the client needs to fetch the association then the client application has to make an additional call with specific details on which association the client application would like to fetch.

Java API Local and Remote Client

The Java API client can access the SDK generated application using two different mechanisms; 1) a local API call and 2) a remote client server API call. Regardless of the type of call the Java client application chooses, the interaction of the client application remains the same. Typical Java API client communication with the SDK generated application is illustrated in Figure 3-5 below.

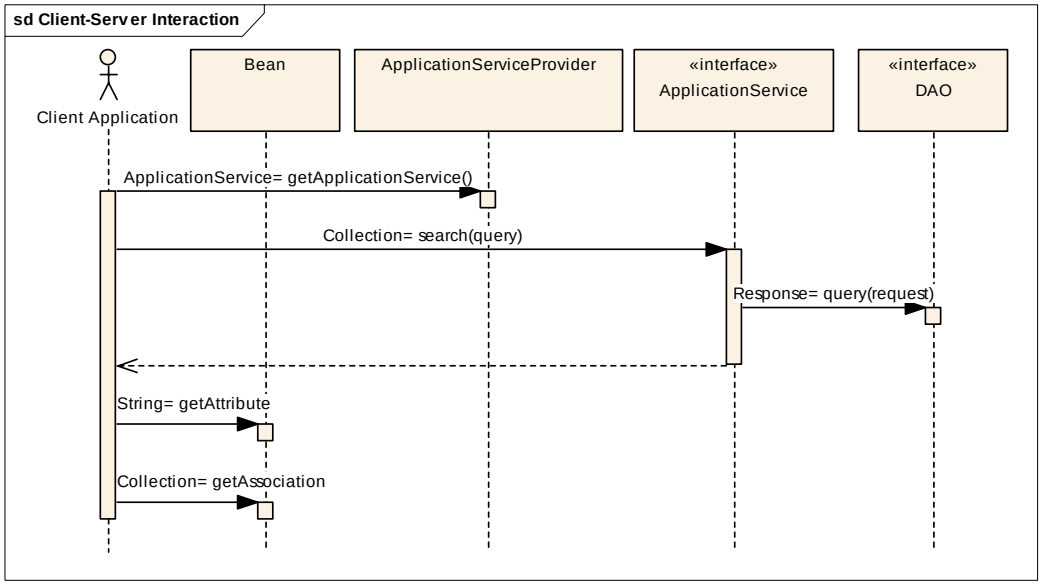


Figure 3-5 Java API communication with an SDK generated application

The client of the generated application intends to fetch data from the database and use the data in the desired manner in their respective application. The client application intends to achieve this behavior using the following steps:

1. Obtain a reference to the service that can deliver the data.
2. Form the query and search the database using the prepared query on the service obtained in step one.
3. Iterate through the results and obtain the attributes/associations of the result object.

If the client application uses the generated system in remote client server mode, the generated client must connect to the remote service using the remote client. On the other hand, if the client application uses the generated system locally, the service must be present in the local environment and remote calls should be avoided. Since the client application is developed in a different environment, it is best to isolate the client from knowledge about what type of client (remote or local) is used to fetch the data.

Technical Challenges of the Client Tier

There are many ways to implement the expected behavior of the client. Technologies include 1) Java RMI 2) Web Services 3) EJB 4) CORBA 5) Remoting etc. From the client perspective, which technology is adopted to solve the problem of client-server communication is less relevant than the underlying system implementation, because users are never exposed to it.

One problem being faced is how to fetch a large result set and its associated objects. Regardless of which technology is used to implement the application framework, the problem of loading a large result set and its associated objects remains. In order to resolve these problems, the data is required to be loaded on demand (lazy-loading). In order to lazy-load the objects, the developed application framework must recognize when there are remaining objects are to be loaded from the database. Events that

require lazy loading are 1) iterating through the large result set and, 2) accessing attributes/associations of the retrieved objects.

The retrieved objects are required to trigger the event where the client application makes an attempt to access the attribute/association of an object. One possible way to achieve this functionality is to hardwire the event triggers in the result objects. This approach makes the result objects tied to the SDK generated application. Another way to achieve the same functionality is to dynamically inject the event triggers in the result objects. The next section describes how lazy-loading behavior is achieved in SDK.

Dynamic Proxy-Based SDK Generated Client API

In order for the client application to obtain the handle to the service tier (Application Service) of the generated application, the SDK provides a helper class called *ApplicationServiceProvider* (ASP). ASP instantiates the service based on the settings in the configuration file (`application-config.xml`).

The sequence diagram below demonstrates how ASP retrieves the service. When the client application requests a handle to the service, ASP retrieves the handle using the configuration file and adds an interceptor to the service resulting in *ApplicationServiceProxy*, which is a dynamic proxy generated using the AOP feature of the Spring Framework (http://www.SpringFramework.org). *ApplicationServiceProxy* intercepts all the calls to the actual ApplicationService and takes action to facilitate the lazy-loading mechanism described earlier. When this occurs, the client application is expecting a handle to the ApplicationService to be received from ASP but receives ApplicationServiceProxy instead.

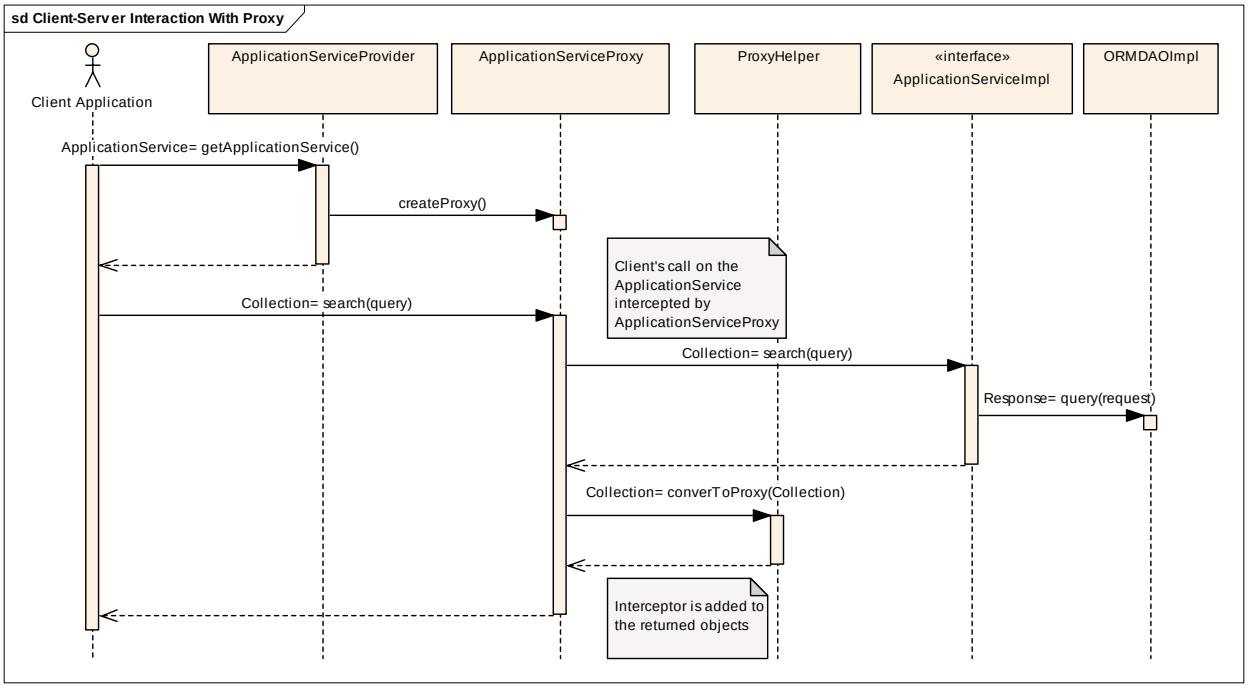


Figure 3-6 Actual Behavior of the SDK Generated Application - 1

After the invocation made by the client on the ApplicationService, the *ApplicationServiceProxy* obtains the result from the actual ApplicationService. The obtained result set can be primitive objects: Java, domain objects, or Java collections.

Since domain objects or collections of domain objects can be required to lazily load their associated objects, *ApplicationServiceProxy* is required to add an interceptor on the domain objects.

After obtaining the results from the ApplicationService, *ApplicationServiceProxy* uses the class *ProxyHelper* to add the appropriate interceptor (BeanProxy) to the domain objects, so that the domain objects can trigger the event to lazily load attribute/associated objects.

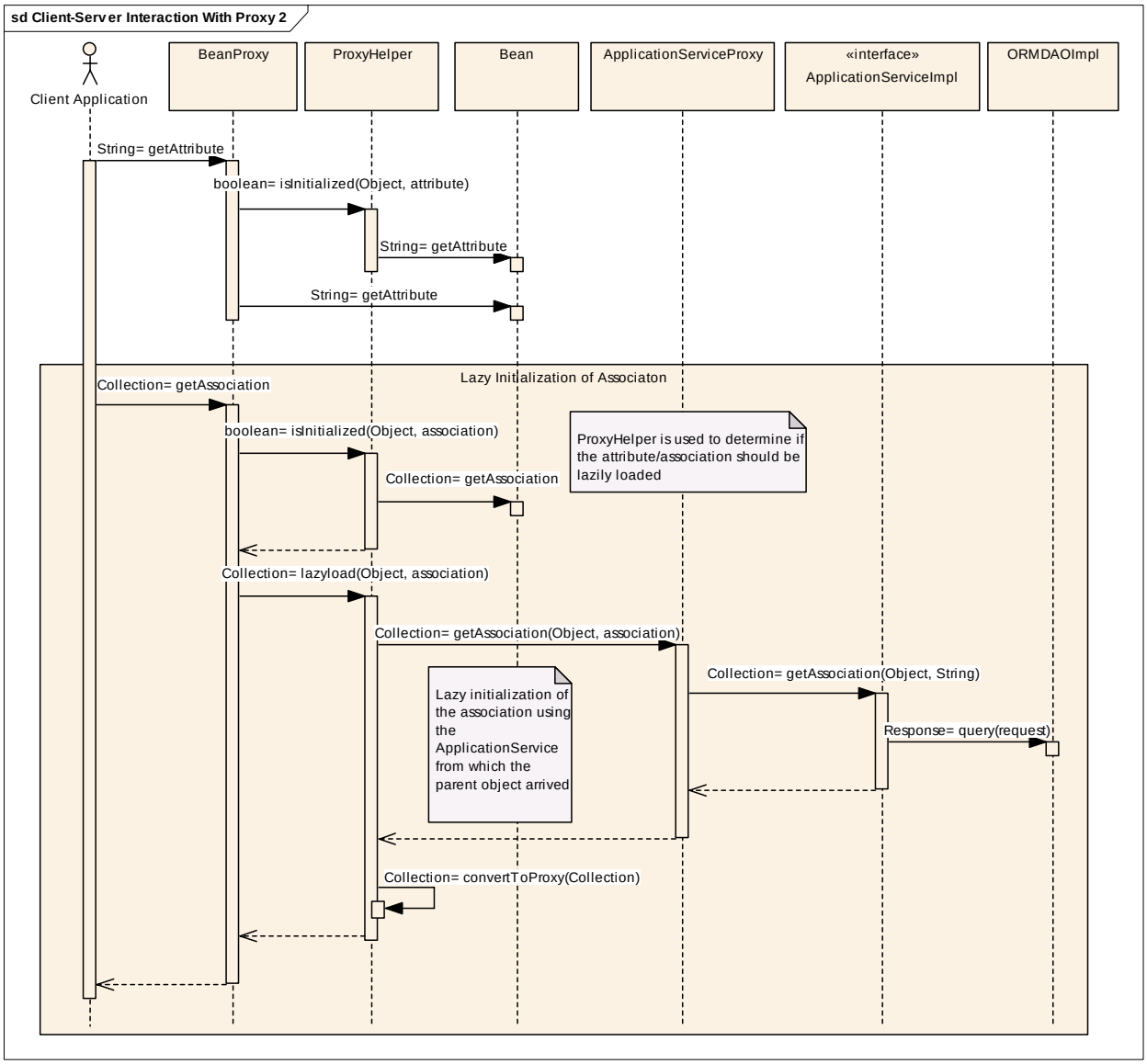


Figure 3-7 Actual Behavior of the SDK Generated Application - 2

The results returned from the ApplicationServiceProxy to the client application have an added interceptor (BeanProxy). The interceptor holds a reference to the ApplicationService where the result objects were loaded.

When the client application invokes any of the methods on the result objects to retrieve attributes/associations, the interceptor (BeanProxy) of the domain object triggers an

event. Subsequent to this event, the ProxyHelper class is used as a decision maker to determine if the attribute/association should be lazily loaded. If the ProxyHelper class indicates that the method should be lazily loaded (i.e. the method should not be executed locally and ApplicationService should be used to obtain the return value), BeanProxy again uses ProxyHelper to execute the method and load the result from the correct ApplicationService.

For ORM-based applications, the ProxyHelper class always checks for the presence of HibernateProxy for the associations. If HibernateProxy is present instead of the actual associated objects, the ApplicationService is called (via ApplicationServiceProxy) to fetch the associated objects. ProxyHelper is responsible for preparing the query and calling the ApplicationService with the appropriate parameters.

For a non-ORM system, the ApplicationService may have been extended to support additional query methods, and these methods can return domain objects that are not the same as regular POJOs. In that case, the implementer of the non-ORM system must intercept all the method calls to the result objects and resolve the lazy initialization routine. The non-ORM system can configure the custom ProxyHelper through the configuration file `application-config-client.xml`.

Connecting to Multiple Remote Application Services

The client application framework can be used to connect to multiple application services at the same time; that is, the client can connect to various SDK generated services at the same time using the same framework.

Note: This feature can be used only with the remote client, and not with the local client API.

In order to facilitate multiple service connections, the ApplicationServiceProvider (ASP) class is used in conjunction with the proxy framework mentioned earlier. ASP reads information from the file `application-config.xml` to create a new instance of the Application Service. If the client application does not mention the service it needs to connect to, the ASP initializes the service described under the “ServiceInfo” bean in the configuration file. However, if the client application mentions the name of the service, the ASP locates the configuration entry for that service, instantiates the service handler, and returns it to the client after adding the interceptor.

When using the client framework in multiple services mode, the developer of the client application must ensure the following:

1. Domain objects corresponding to all the services to which they are trying to connect must be present in the local environment.
2. The services to which the client application intends to connect should be based on the ApplicationService interface of the SDK core.
3. The remote services can be an extension of the ApplicationService interface provided by the SDK. If one or more services have the same extension interface name (e.g. com.xyz.CustomService) then they should have the same method signatures as well.
4. All the extensions of the ApplicationService interface corresponding to different remote services should be present in the local environment.
5. If any of the remote services has been modified in the ApplicationService interface, the client framework will fail to operate.

6. Appropriate entries should be made in the `application-config.xml` for each of the remote services.

Security Filters

Security filters are HTTP servlet filters configured through Acegi (http://www.acegisecurity.org/) in the file `application-config-web-security.xml`. The filters are used in a chained fashion to ensure reusability of the filters. For different client interfaces, the purpose of the filter is to 1) retrieve a user's security credentials from the HTTP message 2) log a user into the application by putting information in the ThreadLocal variable (http://java.sun.com/j2se/1.4.2/docs/api/java/lang/ThreadLocal.html) and 3) clear a user's security information from ThreadLocal at the end of the request. For a web interface, a user's security information is stored in the HTTPSession so that it can be retrieved on a subsequent call. For all other interfaces, a user is required to resubmit login information for each request to be processed.

Chapter 4 Security

The SDK generated middleware system has built-in security that is capable of performing authentication and authorization. In order to manage the authentication, the user has a variety of options, which are supported by the Common Security Module (CSM). The CSM API allows the users to use LDAP, RDBMS and other JAAS based login modules for authentication purposes.

In addition to the CSM-based authentication, the SDK also allows the users to use caGrid-based user accounts/credentials for authentication. The remainder of this chapter provides details on the features and implementation of security in the SDK generated system.

Security Overview

Security is disabled in the default SDK configuration. When a user intends to enable security, they can do so by enabling the security flag in the configuration file and then generating the system. Users can also enable the security in an already generated system by modifying multiple configuration files. However, since this process is error prone, it is not recommended to all users of the SDK.

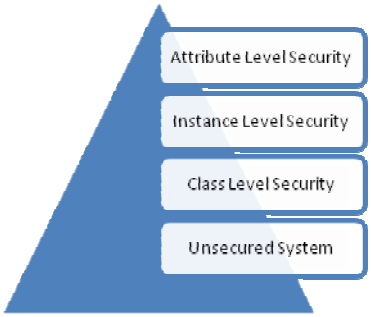


Figure 4-1: Security Levels in caCORE SDK

The caCORE SDK provides security at various levels: Class level security, Instance level security, Attribute level security or no security. Below is brief description of each level.

- Unsecured system/No Security – All the users of the SDK have equal access to the data that the runtime system serves.
- Class level security – Only users who have access to certain objects in the system can query for the data. For example, doctors can view patient data whereas an administrator cannot.
- Instance level security – Users are allowed to access data for only the records to which they are given access. For example, doctors can view data for their patients only and for no other patients in the system.
- Attribute level security – Users are allowed to only see data for which they are authorized. For example, doctors are allowed to see a patient's medical record number but they cannot see the patient's social security number.

Users of the SDK can choose the appropriate level of security for their system based on their requirements. Configuration of the level of security to be enabled is managed through the configuration file (`deploy.properties`) at the time of system generation. When security is enabled, the system applies class level security by default, however if instance and/or attribute level security is desired, users can choose to do so. Details on how to configure the security in CSM for SDK generated system can be found in *Chapter 12, Configuring Security* beginning on page 133.

Authentication

Authentication is a process of determining whether the person using the system is a valid user or not. The SDK supports two different methods of authentication. One option is to use the CSM based authentication which allows for RDBMS, LDAP, or any other JAAS based username/password authentication. The other option is to use caGrid-based user accounts where the user's account information is accessed via the caGrid infrastructure. The following sections provide more details on each type of authentication supported by the SDK.

CSM Authentication

SDK generated applications provide a mechanism to log users into the application. It takes the user credentials from the client and supplies it to the Acegi framework, so that Acegi framework can validate the user credentials and decide if the user can proceed with the operation or not.

Since security policy is managed at the CSM level, a bridge is prepared between CSM and Acegi. The Acegi-CSM bridge retrieves the user information from the security database and logs the user into the application. If the user successfully logs into the system, his/her security policy is cached at the application level. In the case of unsuccessful login, an exception is returned to the user indicating the cause of the error.

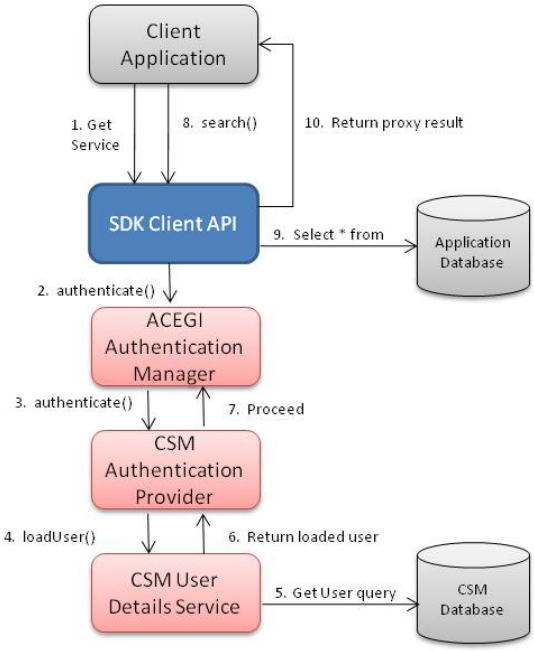


Figure 4-2 Acegi-CSM based authentication in SDK generated system

Figure 4-2 above provides a high level overview of how Acegi and CSM are integrated to provide security in SDK. When the user requests a handle to the secured service by entering a username and password, the SDK generated system internally calls Acegi's Authentication Manager, which in-turn invokes CSM's Authentication Provider to determine validity of the user. If the user is valid and the password in the database matches the user-supplied password, the user is considered authenticated and allowed to proceed.

Grid Authentication

Users with caGrid-based accounts authenticate themselves using the caGrid infrastructure (e.g., caGrid's AuthenticationService client) and obtain their grid credentials using Dorian client. In order to provide a comprehensive security solution for these users, the caCORE SDK has implemented functionality in all tiers of the generated application. The user of the SDK can provide their Grid account's username and password to the generated system, and the generated system can communicate with the configured caGrid services to authenticate the user.

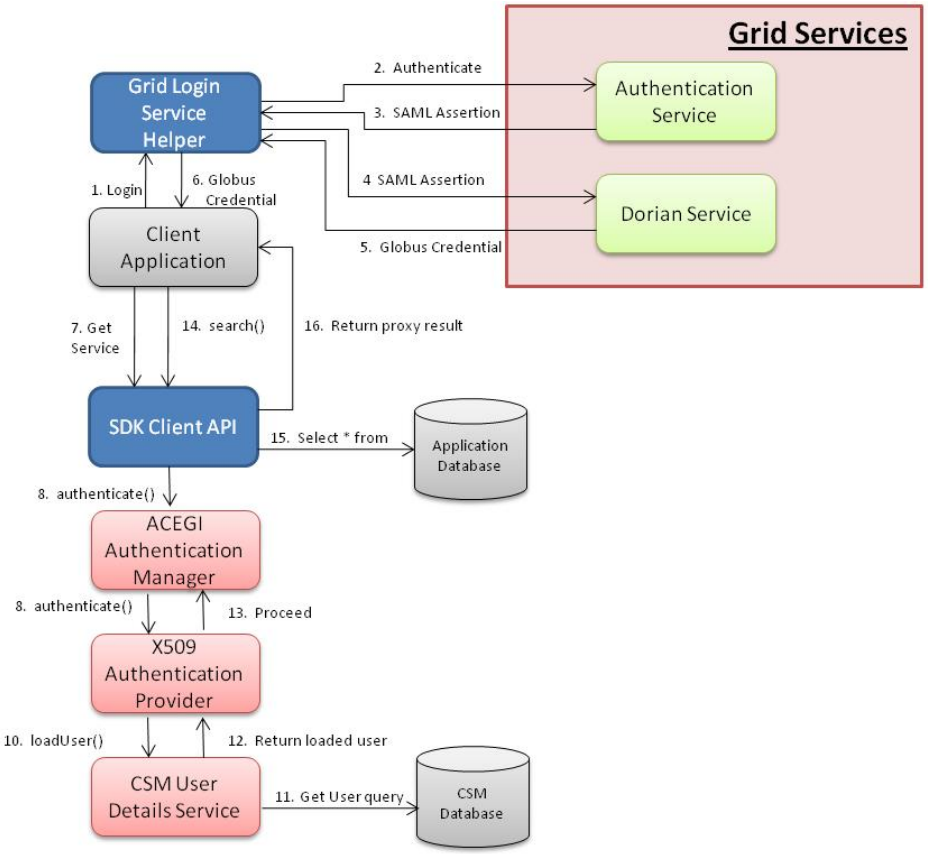


Figure 4-3 caGrid-based user authentication in SDK generated system

Figure 4-3 above shows the authentication workflow when using caGrid-based user credentials. The user first invokes into the SDK generated Grid Login Service Helper using caGrid user account information. This in-turn invokes into the Authentication and Dorian services to complete the grid authentication workflow and retrieve a Globus Credential. The client application can subsequently invoke into SDK's client API with the

retrieved Globus Credential. The SDK's client API internally retrieves the user's account information using the X509 Authentication Provider. The X509 Authentication Provider computes the username to be looked up in the CSM schema using a specific algorithm. If the user account exists in CSM, then the authentication process is considered complete and user is allowed to proceed.

Authorization

Authorization is the process of determining whether the authenticated user has access rights to the resource they are asking for. When security is enabled, Class level security is invoked by default. However the SDK allows for Instance level and Attribute level security as well. All three levels are described briefly below.

Class Level Security

Whenever a user is trying to execute any operation on the service layer, the Acegi framework intercepts the call to the operation, and with the help of a SecurityHelper class, determines what classes the user is trying to access. Subsequent to that, the Acegi framework decides if the user has access to that class or not by checking against the security policy of the user. If the user does not have access, an access denied exception is thrown back to the user.

Instance Level Security

If the system is a secured system with instance level security enabled, the SDK utilizes CSM's services to provide the instance level security. CSM provides instance level security by altering the query using Hibernate filters. The modified query has additional criteria in the "where clause" which goes against the CSM security configuration and restricts the user to retrieving only the records to which they have access. In order to use this feature, the CSM security configuration has to be available on the application's primary database schema, which is used by the SDK generated system.

Attribute Level Security

If the system is a secured system with attribute level security enabled, the SDK utilizes CSM's services to provide the attribute level security. CSM provides attribute level security by altering result objects using Hibernate interceptors. When any object is loaded in memory from the database, the CSM interceptor checks to see whether the user has access to a certain attribute or not. If user does not have access, the CSM interceptor nullifies that attribute.

Chapter 5 Writable API

The Writable API is an important addition to an SDK generated system. The user of the SDK can generate a writable API with minimal effort from a simple UML model. In order to effectively use the writable API, users have to be familiar with few basic principles of how the CRUD operations work in an O/R mapping based system.

This chapter provides a high level overview of how the writable API module works in conjunction with the read API in an SDK generated system. It also provides an overview of other features that become available with use of the writable API module.

Writable API Architecture

The Writable API is made available to SDK users as an extension to the Read API. All the functionality of the read API is always available to the users of the writable API but inverse is not true for read API users.

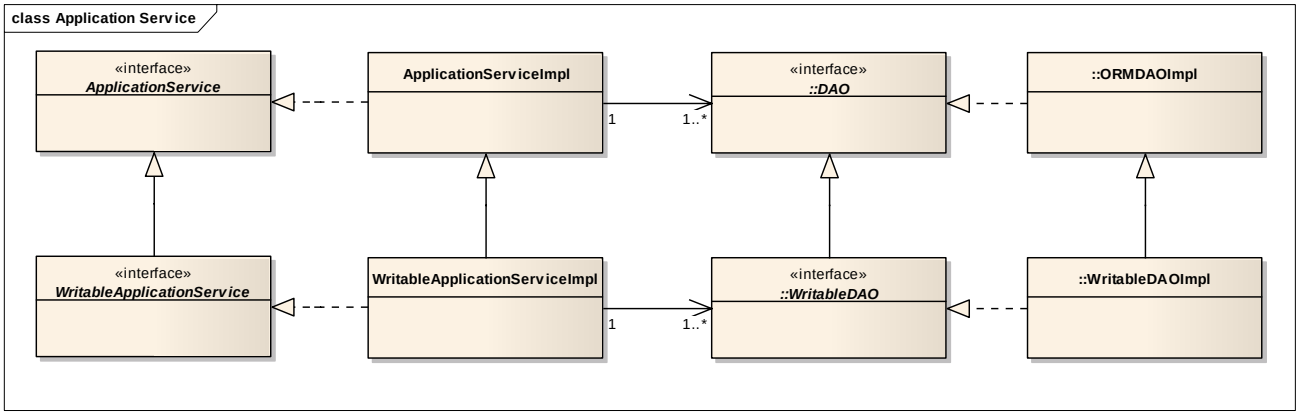


Figure 5-1 Writable API implementation model

As shown in the diagram above, the *WritableApplicationService* interface of the SDK generated system extends the read API interface (e.g., *ApplicationService*). At the same time, the implementation of the writable API (e.g., *WritableApplicationServiceImpl*) extends the read API implementation (e.g., *ApplicationServiceImpl*).

The persistence tier of the SDK is represented by the DAO interface. The default implementation of the persistence tier is the Hibernate-based *ORMDAOImpl*. To support the writable operations, the DAO is extended by *WritableDAO* and its implementation, *WritableDAOImpl*, extends the default DAO implementation, *ORMDAOImpl*.

This extended architecture ensures that the core functionality of the read only API is not affected in any way by the writable API. When a user decides to generate the writable API, the SDK's code generation process creates a configuration file that uses the *WritableApplicationService* and *WritableDAO* instead of *ApplicationService* and *DAO* as core modules of the system. The user of the generated API can access the read and write functionality by getting a handle for the service from the ApplicationServiceProvider.

An additional advantage of this approach is that it allows the SDK to leverage existing infrastructure for the remoting and security of the API without any major changes.

Object Relational Consideration for Writable API

When enabling the writable API, the SDK user has to take few items into consideration required for proper functioning of the API. This section describes the O/R mapping settings needed for the API functioning, including:

- Primary Key generator settings
- Cascade settings
- Inverse settings

The O/R mapping settings mentioned above need to be specified at time of code generation. The SDK allows user to specify all of these settings at the UML model level in form of tag values. The following sections provide details for each of these settings as well as how to properly specify them.

Primary Key Generator Settings

Every object to be persisted in the database is required to have a unique identity. In a database, this identity is normally maintained using a primary key, which is unique by definition. There are many different ways the primary key can be generated, including using a database-specific primary key generator (e.g. Oracle sequence and MySQL identity) or custom generators (e.g. HiLo and GUID generator). Since the SDK uses Hibernate as underlying technology, it supports all the different generators that Hibernate supports.

The SDK allows specifying the primary key at two levels. The global primary key generator can be specified in the `deploy.properties`. If any individual class does not have a primary key generator specified, the SDK uses the global generator. The user can override the global primary key generator by specifying a class level key generator using UML tag values in the model. The SDK code generator reads the UML tag values at the time of code generation and generates appropriate Hibernate mapping files.

Cascade Settings

While persisting an object graph, the system has to know whether or not to update the associations. Hibernate allows specifying the update settings on the associations through the mapping files. The SDK generates the required mapping files as part of the code generation process. During the code generation process, the SDK reads the UML tag values specified on the associations, indicating the cascade styles to be used for a particular association. If the user has not specified any cascade style, the SDK generates the mapping files with no cascade style. The setting of no cascade style indicates that if an object is being persisted then its associations are not to be updated. When the user specifies the cascade style, the SDK simply copies the cascade style into the generated mapping file.

Inverse Settings

The Hibernate technology used by the SDK to manage the persistence tier uses a notion of inverse settings to remove cyclical dependency of the object graph being persisted. Hibernate treats bi-directional associations as two separate associations going in opposite directions, and disables the association that is marked as “inverse-of=true.” While building the mapping files, the user should review the use case of

create/update/delete and determine which side of the association will be used when performing write operation. Depending on the use case, the user can insert an inverse setting in the mapping file. Since the SDK generates the mapping files from the UML model file (on behalf of the user), the UML model should include information on appropriate inverse settings.

Transactions

Transaction management becomes very important when the API updates the underlying data in the database. The SDK generated writable API uses an external transaction manager to handle all the transactions within the system.

By default, the configured transaction manager is Hibernate transaction manager implementation:

```
<bean id="HibernateTransactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="SessionFactory" />
</bean>
```

When integrating the generated API with other systems or custom code, one may need to keep both the generated API and the other system in the same transaction boundary.

Since the entire configuration for transactions is managed through an external configuration file, it can easily be changed. For example, you use JBoss Transaction Manager in SDK by changing the SDK's Spring configuration file to lookup the transaction manager in JNDI context instead of creating a new instance of Hibernate Transaction Manager as shown in the snippet above. This approach allows the users to use the SDK generated writable API within the same transaction as their existing API.

Chapter 6 Data Validation

When creating new records and performing updates on the existing records, it is very important to perform validation of the data being inserted or updated in the tables. caCORE SDK provides a convenient way to perform validation on the data being manipulated by allowing the SDK user to enable validation at the time of code generation. When enabled, the SDK generates the validation settings using the permissible values stored in caDSR.

In addition to validation against a list of values, the SDK validation framework is capable of performing other validations like Date, Range, Min, Max etc. However, as there does not exist a system in caBIG which stores all of this information, SDK provides a very flexible approach to meet each user's needs. The user can prepare a configuration file in a format that SDK expects and provide it during code generation. SDK reads the user-provided configuration file and uses that, in addition to the caDSR enumerated values, to prepare validation settings.

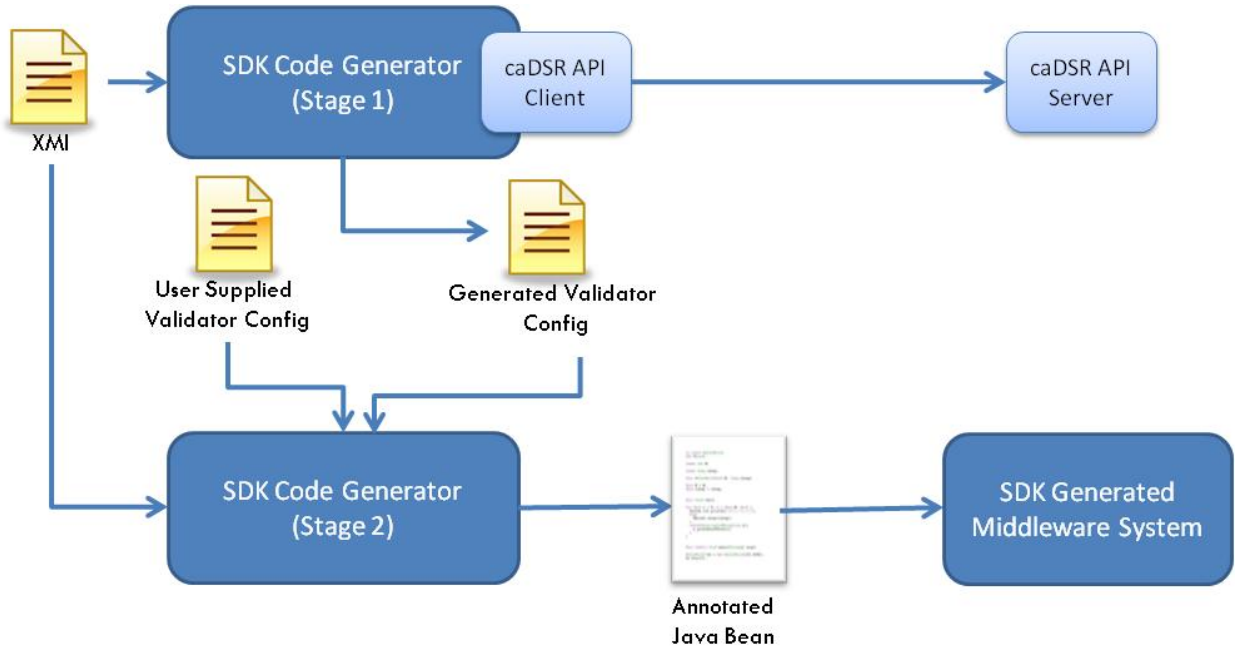


Figure 6-1 Validation generated process in SDK code generator

Validation generation in the code generator module of the SDK is a two stage process. In the first stage, when the data validation feature is enabled, the code generation module downloads the caDSR Value Domain information from the caDSR production system and prepares a validation configuration setting file.

The second stage of code generation reads the generated validator configuration file as well as any user-supplied configuration file. Once both the files are read, the code generator injects the information into the generated Java bean as JDK 5 annotations to be used during runtime. At runtime, when the user of the API is attempting to manipulate the data, the underlying validation framework (Hibernate Validator configured in the generated middleware system), intercepts all the calls and performs validations of the objects before executing the queries.

More details on the types of validations being supported and how to prepare custom validations can be found on the Hibernate Validator wiki page, located at: https://wiki.nci.nih.gov/x/_BSZ.

Future consideration: As the data validation feature in SDK matures, the SDK team is planning to add more sources for data validation information, such as the ability to read UML enumerations from the XMI file.

Chapter 7 Logging/Audit Trail Management

As the data is being manipulated through the Writable API, users may need to keep track of the changes being made. The SDK generated middleware system provides a convenient way to manage the audit trail. During the code generation phase of the SDK, the user can select the option to enable the audit trail. When this option is enabled, the SDK code generator injects a Common Logging Module (CLM) based interceptor in the persistence tier of the system. The CLM interceptor, with help of a Log4j appender, creates a log statement when it detects a change in the state of an object.

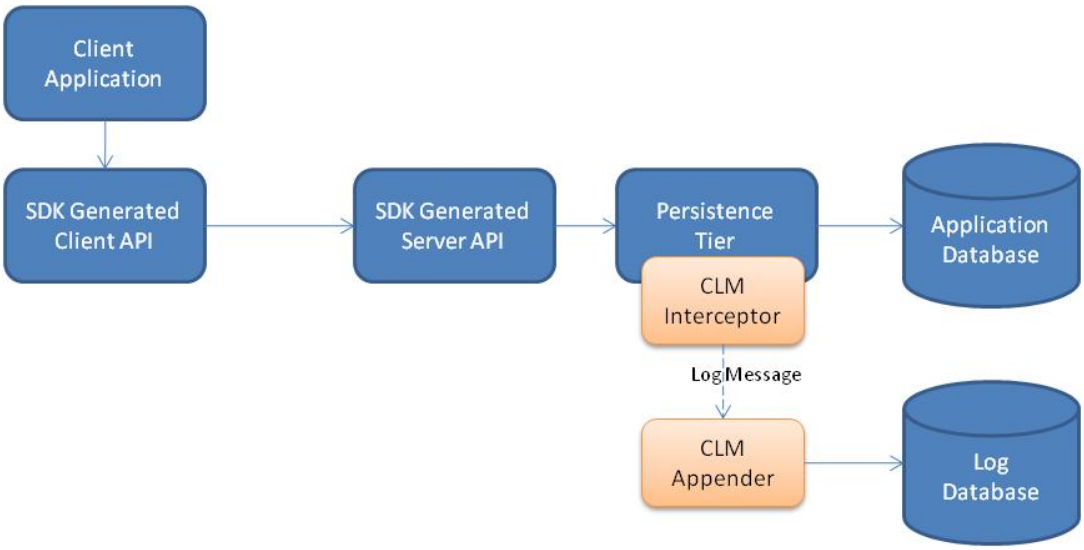


Figure 7-1 Overview of Logging/Audit Trail management implementation

In order to detect a change in the state of an object, the CLM's object state logger intercepts all the events related to the domain object change whenever an attempt is made to persist the changed domain object in the database. As CLM detects the change in the object's attributes, it creates a new log message, which is handled by the Log4J appender.

Currently CLM provides a database appender which is capable of persisting log messages in a relational database. Since the CLM's database appender inserts the log statements in asynchronous manner, the performance impact at runtime is negligible.

CLM also provides a companion application called Log Locater Tool (LLT) for viewing the log statements previously inserted by the database appender. The LLT is a web-based application which can be installed and configured using CSM to allow only certain users to review the log statements. LLT also allows users to search records based on criteria such as date range. More information on CLM and LLT can be found at: http://ncicb.nci.nih.gov/NCICB/infrastructure/cacore_overview/clm.

Chapter 8 Using SDK Client Interfaces

This chapter describes the available client interfaces for an SDK generated runtime system, and how to access the data using the same. For those client interfaces, this chapter also provides instructions for how to gain access when the generated system has security enabled.

Topics in this chapter include:

Introduction

By default, security for an SDK generated system is disabled, meaning that users do not have provide credentials in order to access or query the system.

When the generated system is secured, the user of the system is required to perform the login operation before making any query to the system. The login operation varies depending on the client interface used.

Once the login operation is complete, querying the system is done in the same fashion as for an unsecured system.

XML-HTTP Interface

The XML-HTTP interface can be accessed in two ways: 1) from a web browser or 2) from a thin client application that can fetch data in XML format from the server using the REST-like syntax.

Accessing Data from a Web Browser

The URL used by this interface uses the following pattern:

SDK GUI URL Pattern	http://<server_name><server_port>/<project_name>
Sample SDK GUI URL	http://localhost8080/example

Figure 8-1 below shows a sample SDK *Home* page. The SDK web page contains links to several other pages that facilitate access to domain data.

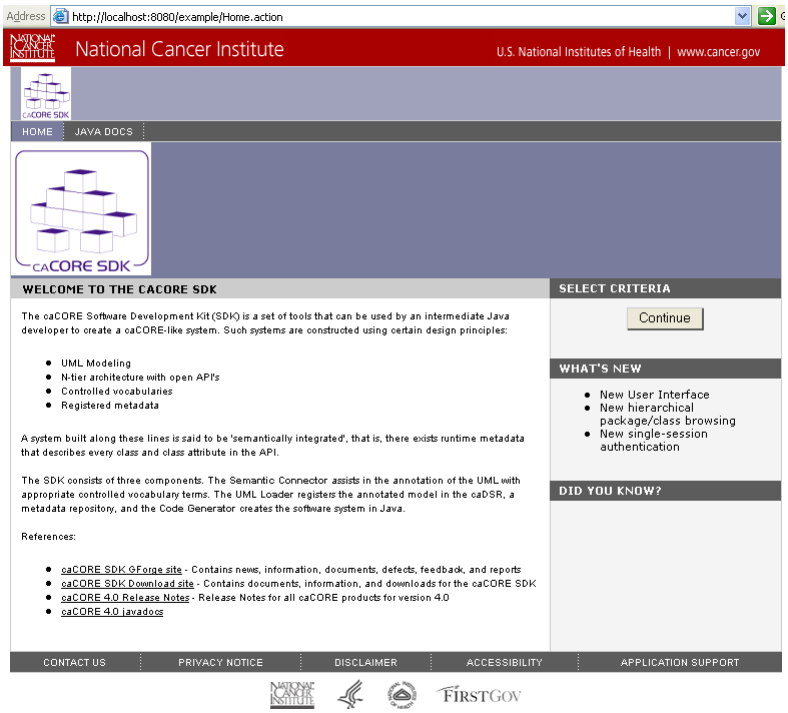


Figure 8-1 caCORE SDK Home page

The *Home* page contains various links to SDK-related sites and documentation, such as:

- the SDK GForge site
- the SDK Download site
- the SDK Release Notes
- JavaDocs for the domain objects of the generated system.

When security is disabled (which is the default), a **Continue** button appears on the Home page.

If security is enabled, a Login form requesting a User ID and Password appears instead. Browser-based clients must provide security configured through form-based authentication, meaning that a user must enter a username and password in order to access the application. Figure 8-2 below shows the login form on the SDK Home page.

More about enabling security is provided in *Chapter 12, Configuring Security* beginning on page 133.

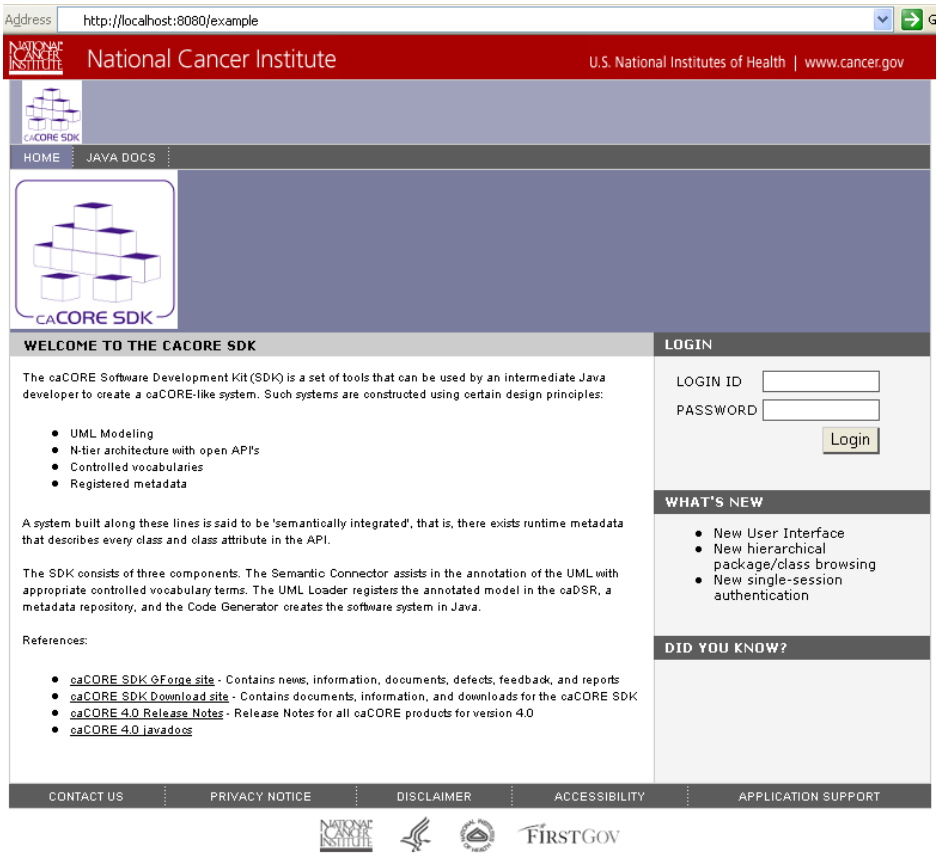


Figure 8-2 Security login form in the web browser

If login is unsuccessful, an error message appears in the login section of the screen. After three consecutive unsuccessful login attempts, CSM locks out the user's account for 30 minutes.

Successfully logging in or clicking the **Continue** button displays a hierarchical domain package/class browser tree known as the *Content* page. The Content page contains both a domain class browser and a *Criteria* form to search for records. The Content page for the sample SDK model is shown in Figure 8-3 below.

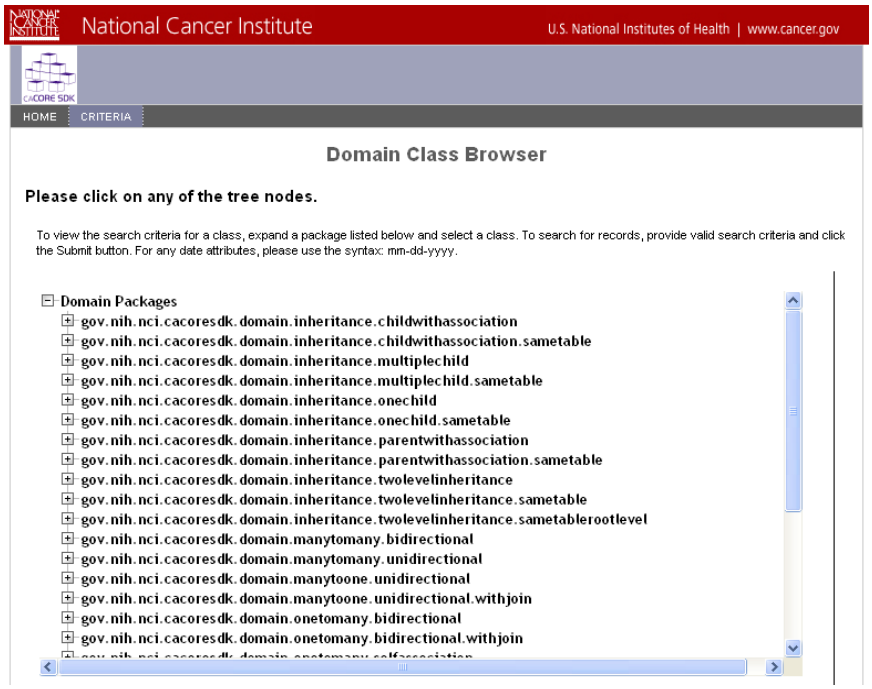


Figure 8-3 SDK Content page

Expand or collapse the items in the Domain Class Browser tree by clicking on the **+** or **-** symbols to the left of a domain package name. To view the *Search Criteria* for a particular class, expand a domain package so that its classes are listed, then select the desired class name node. A *Search Criteria* form listing the searchable class fields appears to the right of the browser tree.

Figure 8-4 below illustrates the *Search Criteria* form for the Professor class of the sample SDK model.

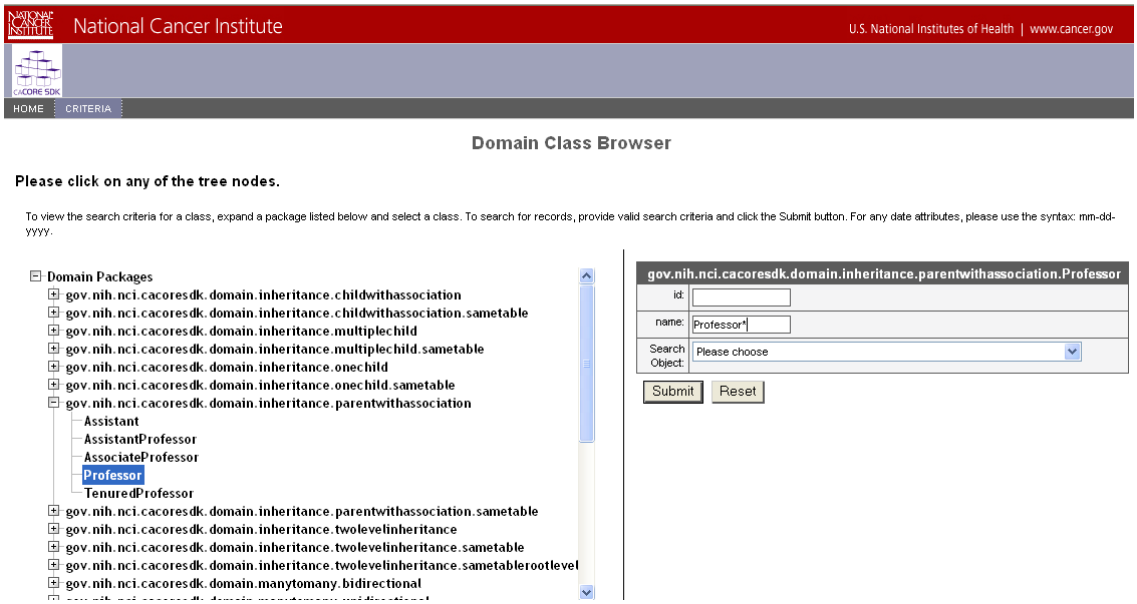
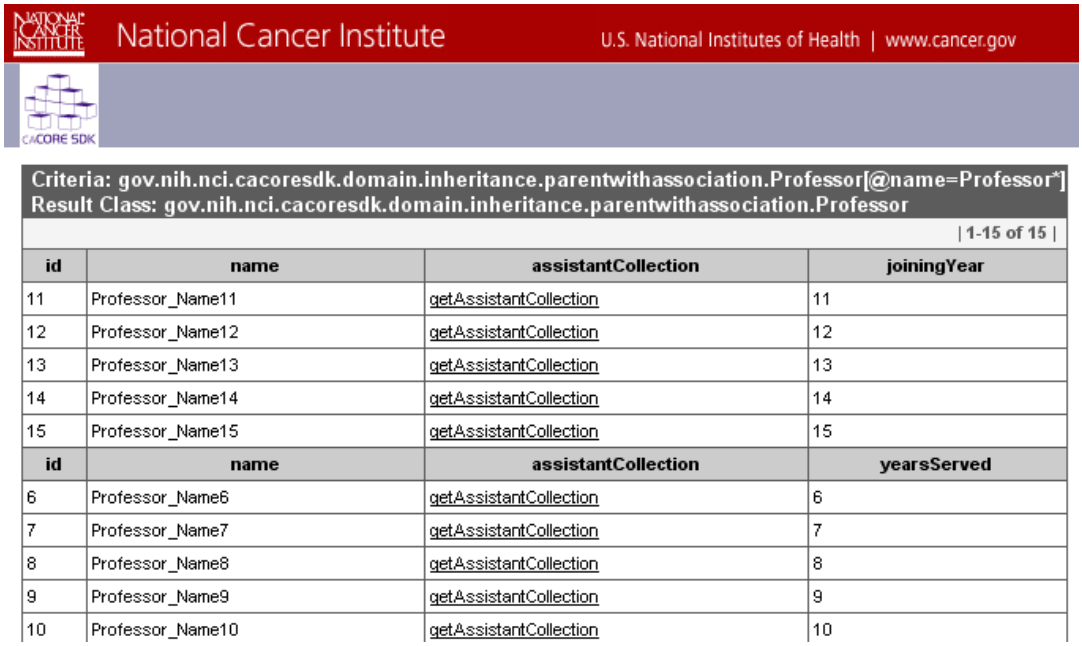


Figure 8-4 Search Criteria Form

Notes Regarding Search Criteria:

- To search for date attributes, use the syntax: mm-dd-yyyy.
- The Search Criteria form accepts the asterisk (*) as a wildcard character.
- The Search Criteria form also contains a drop-down list containing Search Objects (domain classes) that are associated with the current domain class. Selecting a Search Object from the drop-down list causes the query to return records of the type represented by the Search Object, and *not* records of the type represented by the selected class, which is the default if no Search Object is selected.

When you click **Submit** on the *Search Criteria* form, a new window appears containing the Result Data Table page listing all matching records (Figure 8-5).



Criteria: gov.nih.nci.cacoresdk.domain.inheritance.parentwithassociation.Professor[@name=Professor*]
Result Class: gov.nih.nci.cacoresdk.domain.inheritance.parentwithassociation.Professor

| 1-15 of 15 |

id	name	assistantCollection	joiningYear
11	Professor_Name11	getAssistantCollection	11
12	Professor_Name12	getAssistantCollection	12
13	Professor_Name13	getAssistantCollection	13
14	Professor_Name14	getAssistantCollection	14
15	Professor_Name15	getAssistantCollection	15
id	**name**	**assistantCollection**	**yearsServed**
6	Professor_Name6	getAssistantCollection	6
7	Professor_Name7	getAssistantCollection	7
8	Professor_Name8	getAssistantCollection	8
9	Professor_Name9	getAssistantCollection	9
10	Professor_Name10	getAssistantCollection	10

Figure 8-5 Result Data Table page

On the Results page, the table column headers correspond to fields from the resulting domain class type. A collection of wrappers of primitive object types and field values appear as strings within the corresponding table cells. Fields that represent an association to another domain class appear as links, which can be clicked to retrieve any associated domain object records.

Accessing Data from a Thin Client

The Representational State Transfer (REST) interface provided by the SDK is a simple URL interface that transmits domain-specific data over HTTP without an additional messaging layer, such as SOAP, or session tracking via HTTP cookies.

For more information on REST, see http://en.wikipedia.org/wiki/REST.

If the SDK generated system is secured, the thin client application is required to provide the username and the password using BASIC authentication:

(http://www.ietf.org/rfc/rfc2617.txt, http://en.wikipedia.org/wiki/Basic_access_authentication).

Under BASIC authentication, the username and password are encrypted using Base64 encryption and are supplied as part of the HTTP header to the server. The server side component decrypts the headers using the corresponding decryption logic and attempts to log the user into the application. The code snippet shown below demonstrates how to set up BASIC authentication using the Java API.

```
URLConnection conn = url.openConnection();
String base64 = "userId" + ":" + "password";
conn.setRequestProperty("Authorization", "Basic " + new String(

        org.apache.commons.codec.binary.Base64.encodeBase64(base64.getBytes())));
```

Figure 8-6 Security Login in Java based REST (XML) client

NOTE: For the REST interface, the thin client application is required to provide the username and password in every call made to the server.

The URL used by the REST interface adheres to the following pattern:

REST Interface URL Pattern	http//<server_name><server_port>/<project_name>/ GetXML?query=<target>&<criteria>[&rolename=<rolename>]

The following table describes each of the variable properties of the REST URL:

Parameter	*Description*
server_name	A string identifying the server, or host, name. Examples include localhost and 127.0.0.1.
server_port	A string identifying the port number to which the SDK server is listening. Examples include 80 or 8080.
Project_name	A string identifying the project name used when building and deploying the SDK application. Examples include example and myproject. **NOTE:** This value coincides with the PROJECT_NAME property found within the `deploy.properties file`.
Target	A string identifying the qualified or non-qualified query target/result class name. Examples include: gov.nih.nci.cacoresdk.domain.inheritance.childwithassociation.Bank
Criteria	A string identifying the qualified or non-qualified criteria class name to be used as a filter/constraint on the result set. An example is the SDK sample model Credit class that has an association to the Bank class via its issuingBank attribute. If desired, the value of the id attribute of the criteria class instance can also be supplied in order to further constrain the result set. The pattern for such a criteria string is <criteria_class_name>[@id=<id_value>]. An example might be Credit[@id=3], which indicates that only target/result class instances are returned that are associated to the Credit record with an id value of 3.

Parameter	Description
Rolename	The name of the attribute within the criteria class that identifies the association to be traversed when retrieving the target/result class(es). An example is the issuingBank attribute of the Credit class found within the sample SDK model. The rolename property must be specified whenever the Criteria class has two or more attributes representing associations to the same target/result class type. One example would be the Child class within the sample SDK model that contains two attributes, mother and father, that both represent instances of the Parent class. In this scenario, specifying a value of rolename=mother or rolename=father within the REST URL would ensure that the correct Parent instance would be returned.

Table 8-1 Variable properties of the REST URL

A sample URL from the sample SDK model is provided below:

Sample REST URL	http://localhost:8080/example/GetXML?query=Bank&Credit[@id=3] &roleName=issuingBank

While such a URL can be invoked directly from a browser, it is most frequently done so programmatically via a remote client program. An example of such a program, *TestGetXMLClient.java*, is provided in the *\output\example\package\remote-client\src* folder created by the SDK Code Generator. Figure 8-7 below provides a sample of the XML output produced from invoking the *Sample REST URL* shown above.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xlink:httpQuery xmlns:xlink="http://www.w3.org/1999/xlink">
  - <queryRequest>
    - <query>
        <queryString>query=Bank&Credit[@id=3]&roleName=issuingBank</queryString>
        <class>gov.nih.nci.cacoresdk.domain.inheritance.childwithassociation.Bank</class>
      </query>
      <criteria>Credit[@id=3]</criteria>
    </queryRequest>
  - <queryResponse>
      <recordCounter>1</recordCounter>
    - <class name="gov.nih.nci.cacoresdk.domain.inheritance.childwithassociation.Bank"
        recordNumber="1">
        <field name="id">3</field>
        <field name="name">Bank3</field>
      </class>
    - <pages count="1">
        <page number="1" xlink:type="simple"
          xlink:href="http://localhost:8080/example/GetXML?query=Bank&Credit[@id=3]
          &pageNumber=1&resultCounter=200&startIndex=0">1</page>
      </pages>
      <recordCounter>1</recordCounter>
      <start>1</start>
      <end>1</end>
    </queryResponse>
</xlink:httpQuery>
```

Figure 8-7 Sample XML output from REST call

Java API Interface

The use of the Java API client has two main steps. The first step involves obtaining a reference to the instance of the *ApplicationService* interface from the *ApplicationServiceProvider* class. The second step involves invoking one of the interface methods in order to fetch the results from the SDK generated server component (local in the case of a local client).

The following test programs illustrates how the SDK Java API can be used are provided as samples:

- *TestClient.java:* A sample *local* client located in the folder *output**example**package**local-client**src*.
- *TestClient.java:* A sample *remote* client located in the folder *output**example**package**remote-client**src*.

More information about these test programs is provided in *Testing the Java API* on page 126.

Using security with the Java API client is a simple one-step process. The user is required to use the methods that accept the username and password to obtain the reference to the *ApplicationService* from the *ApplicationServiceProvider* class.

When CSM-based authentication is used, the username and password combination is used to authenticate the user. When the username and password are passed as parameters, the *ApplicationServiceProvider* class validates the username and password against the authentication service, and if successful, logs the user in the application.

```
ApplicationService appService =
ApplicationServiceProvider.getApplicationService(“userId”, “password”)
```

Figure 8-8 Security Login in Java API client using CSM based authentication

When the system is generated to use caGrid-based authentication, the user can retrieve the Globus Credential object by passing their caGrid username and password combination to the SDK’s Grid Authentication Helper. After retrieving the Globus Credential, the user passes the Globus Credential to the SDK generated Java API to be used for further authentication and authorization purpose.

```
GridAuthenticationHelper loginHelper = new GridAuthenticationHelper("grid");
GlobusCredential proxy = loginHelper.login(username, password);

ApplicationService appService =
ApplicationServiceProvider.getApplicationService(proxy);
```

Figure 8-9 Security Login in Java API client using caGrid based authentication

NOTE: In addition to the example shown above, there are other convenience methods in the *ApplicationServiceProvider* class that allow a user to log in on a different service or different URL.

Obtaining ApplicationService

Access to the *ApplicationService* interface is provided via the *ApplicationServiceProvider* class, which provides several variations of a single method as shown below.

Primary Application Service Provider Method	getApplicationService(service, url, username, password)

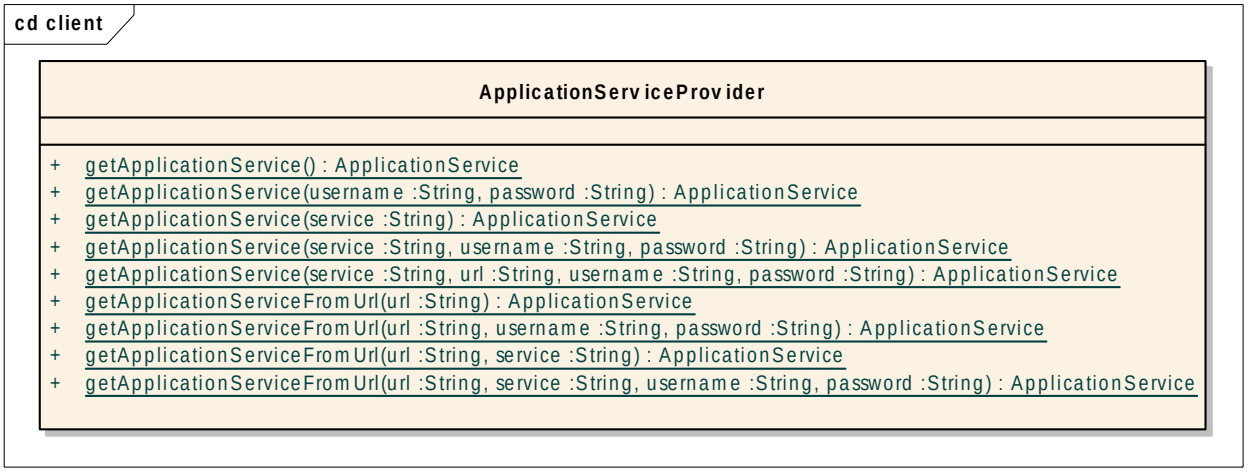


Figure 8-10 ApplicationServiceProvider Methods

The four parameters required by the methods of the *ApplicationServiceProvider* class are described in the following table:

ApplicationService Parameter	***Description***
Service	A string identifying the name of the Spring bean to use when configuring the ApplicationService instance. The bean represents a hash map and is defined within the configuration file, `application-config-client.xml`, located within the folder `/output/<project_name>/package/[local|remote]-client/conf/`. The default bean name (for those methods that do not require the service parameter) is ServiceInfo. This default hash map defines the following configuration properties: • APPLICATION_SERVICE_BEAN: A reference to another Spring bean within the same configuration file that identifies the ApplicationService class to instantiate. • AUTHENTICATION_SERVICE_BEAN: A reference to another Spring bean within the same configuration file that identifies the authentication provider class to use when security is enabled. • APPLICATION_SERVICE_URL: The URL to the Spring DispatcherServlet configured within the SDK to handle remote Java API calls. The URL must conform to the following pattern : http://<server_name>:<server_port>/<project_name> • APPLICATION_SERVICE_CONFIG: A reference to another Spring bean within the same configuration file that identifies a configuration string used when instantiating the ApplicationService instance. **NOTE:** This is an advanced property setting, and should rarely need to be changed, if ever.

ApplicationService Parameter	Description
url	A string identifying the URL to the remote service configured within the SDK to handle remote Java API calls. The URL must conform to the following pattern: http://<server_name>:<server_port>/<project_name>.
username	A string identifying the username to use for both authentication and authorization purposes. Only required and valid when security is enabled.
password	A string identifying the password to use for authentication purposes. Only required and valid when security is enabled

Table 8-2 Primary ApplicationServiceProvider method parameters

The *ApplicationServiceProvider* method can be classified into two method groupings. The first group of methods returns an *ApplicationService* instance without requiring an Application Service URL. The second group, in contrast, requires that an Application Service URL be provided.

NOTE: The ApplicationServiceProvider methods requiring a URL are useful when overriding the default URL. These methods are also useful when multiple ApplicationService instances to different SDK applications are desired.

ApplicationService API Methods

The SDK Java API consists of several query/search methods and a few other convenience methods that facilitate *read-only* access to domain data. Figure 8-11 below shows a class diagram that highlights these methods.

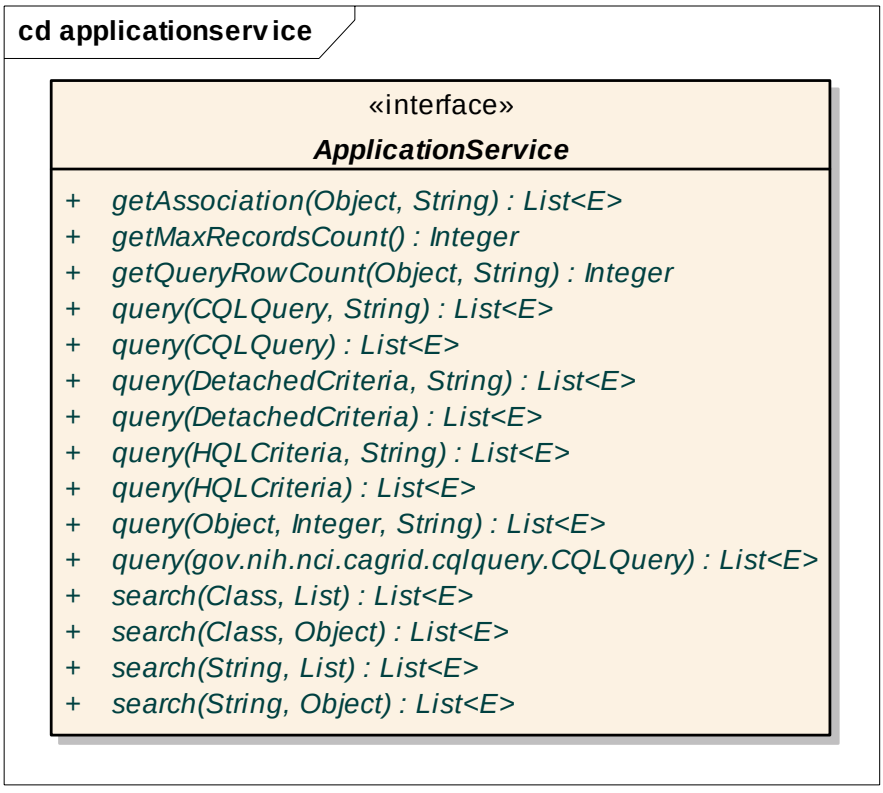


Figure 8-11 ApplicationService Interface Methods

The *ApplicationService* methods are grouped into different categories and are discussed in the sections that follow.

Convenience Query

The *ApplicationService* interface provides various convenience query methods, which can be SDK users, but which are typically used by the SDK infrastructure. Table 8-3 below highlights these methods.

ApplicationService Method	***Description***
getMaxRecordsCount()	Returns the maximum number of records the ApplicationService interface has been configured to return at one time.
getQueryRowCount(Object criteria, String targetClassName)	Returns the number of records that meet the search criteria. This method is used by the client framework to determine the number of list chunks in the result set. SDK users can also invoke this method in conjunction with the getMaxRecordsCount() method; however, this is not typical.
getAssociation(Object source, String associationName)	Retrieves an associated object for the example object specified by the source parameter.

Table 8-3 ApplicationService interface query methods

HQL Query

Hibernate is equipped with a powerful query language, called Hibernate Query Language (HQL) that is similar to SQL. However, though the syntax is SQL-like, HQL is still fully object-oriented and understands concepts like inheritance, polymorphism, and association.

See http://www.hibernate.org/hib_docs/v3/reference/en/html/queryhql.html for more information on the Hibernate Query Language.

The SDK contains a wrapper class called *HQLCriteria*, which is used for submitting HQL queries. A diagram of this class is shown in Figure 8-12.

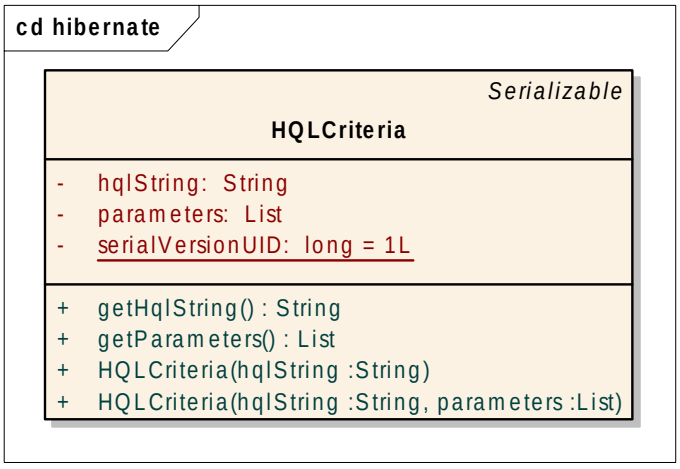


Figure 8-12 HQLCriteria Class Diagram

The following table highlights the HQL related *ApplicationService* methods.

ApplicationService Method	Description
query(HQLCriteria hqlCriteria)	This method retrieves the results obtained by querying the data source using the Hibernate Query Language (HQL). As such, the data source must use Hibernate at the persistence tier. Internally, Hibernate executes the HQL query against the relational database and fetches the results. **Note:** The retrieved results are converted into a list that may not be completely loaded. If the number of retrieved records is more than the maximum number of supported records as indicated by the getMaxRecordsCount() method, then the result set will only contain a subset of the total records. The client framework will execute a subsequent query (transparent to the client application) against the ApplicationService to load the remaining results in the list chunk.
query(HQLCriteria hqlCriteria, String targetClassName)	Deprecated. Internally calls the query(HQLCriteria hqlCriteria) method without the targetClassName parameter.

Table 8-4 HQL ApplicationService methods

Figure 8-13 below shows how an SDK *HQLCriteria* object representing an HQL query might be instantiated and submitted. Figure 8-14 below that shows how the results would be returned.

```
ApplicationService appService = ApplicationServiceProvider.getApplicationService();

HQLCriteria hqlCrit = new HQLCriteria("from gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit");

Collection results = appService.query(hqlCrit);
for(Object obj : results)
{
    printObject(obj, Suit.class);
    break;
}
```

Figure 8-13 Sample HQL Query

```
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit$$EnhancerB
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1,
[java] Name:Spade
[java] Id:1
```

Figure 8-14 Sample HQL Query Results

Detached Criteria Query

While HQL is extremely powerful, some developers prefer to build queries dynamically using an object-oriented API, rather than building query strings. To this end, Hibernate provides an intuitive *Criteria query API*.

See http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#querycriteria for more information on Hibernate Criteria queries. See section *15.8. Detached Queries and Subqueries* of the same chapter for details on the Hibernate *DetachedCriteria* itself.

The Hibernate *Detached Criteria* extends the Criteria concept, allowing Criteria queries to be created outside of the scope of a session, to be executed later using some arbitrary Hibernate Session.

Table 8-5 highlights the *Detached Criteria* related *ApplicationService* methods.

ApplicationService Method	***Description***
query(DetachedCriteria detachedCriteria)	Retrieves the result from the data source using the DetachedCriteria query object. The DetachedCriteria query structure can be used only by the Object Relational Mapping based persistence tier. Hibernate executes it against the relational database and fetches the results. **Note:** The retrieved results are converted into a list that may not be completely loaded. If the number of retrieved records is more than the maximum number of supported records as indicated by the getMaxRecordsCount() method, then the result set will only contain a subset of the total records. The client framework will execute a subsequent query (transparent to the client application) against the ApplicationService to load the remaining results in the list chunk.
query(DetachedCriteria detachedCriteria, String targetClassName)	Deprecated. Internally calls the query(DetachedCriteria detachedCriteria)method without the targetClassName parameter.

Table 8-5 Detached Criteria related ApplicationService methods

Figure 8-15 shows how a Hibernate *DetachedCriteria* object might be instantiated and the query submitted. Figure 8-16 shows how the results would be returned.

```
ApplicationService appService = ApplicationServiceProvider.getApplicationService();

DetachedCriteria detachedCrit = DetachedCriteria.forClass(Suit.class)
        .add( Property.forName("id").eq(1) );

Collection results = appService.query(detachedCrit);
for(Object obj : results)
{
    printObject(obj, Suit.class);
    break;
}
```

Figure 8-15 Sample DetachedCriteria Query

```
[echo] ************************************************************
[echo] ***  caCORESDK: Running the Detached Criteria Query test
[echo] ************************************************************
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit$$EnhancerB
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1,
[java] Name:Spade
[java] Id:1
```

Figure 8-16 Sample DetachedCriteria Query Results

CQL Query

In addition to providing access to Hibernate-specific queries, SDK also provides language neutral SDK-specific queries. CQL is one of such two query mechanisms. SDK CQL queries are modeled similarly to the object representation of the caBIG Query Language (CQL), which uses syntax similar to the Query-by-Example (QBE) query language to specify the way results are to be retrieved.

NOTE: QBE is a database query language for relational databases. It was devised by Moshé M. Zloof at IBM Research during the mid 1970s, in parallel to the development of SQL. It is the first graphical query language, using visual tables where the user would enter commands, example elements and conditions. For more information on QBE, see: http://en.wikipedia.org/wiki/Query_by_Example.

The system formulates the query based on the navigation path specified in the query search criteria. The query mechanism allows the user to search for the objects using platform-independent query syntax.

The CQL query is represented by a complex object structure as shown in Figure 8-17.

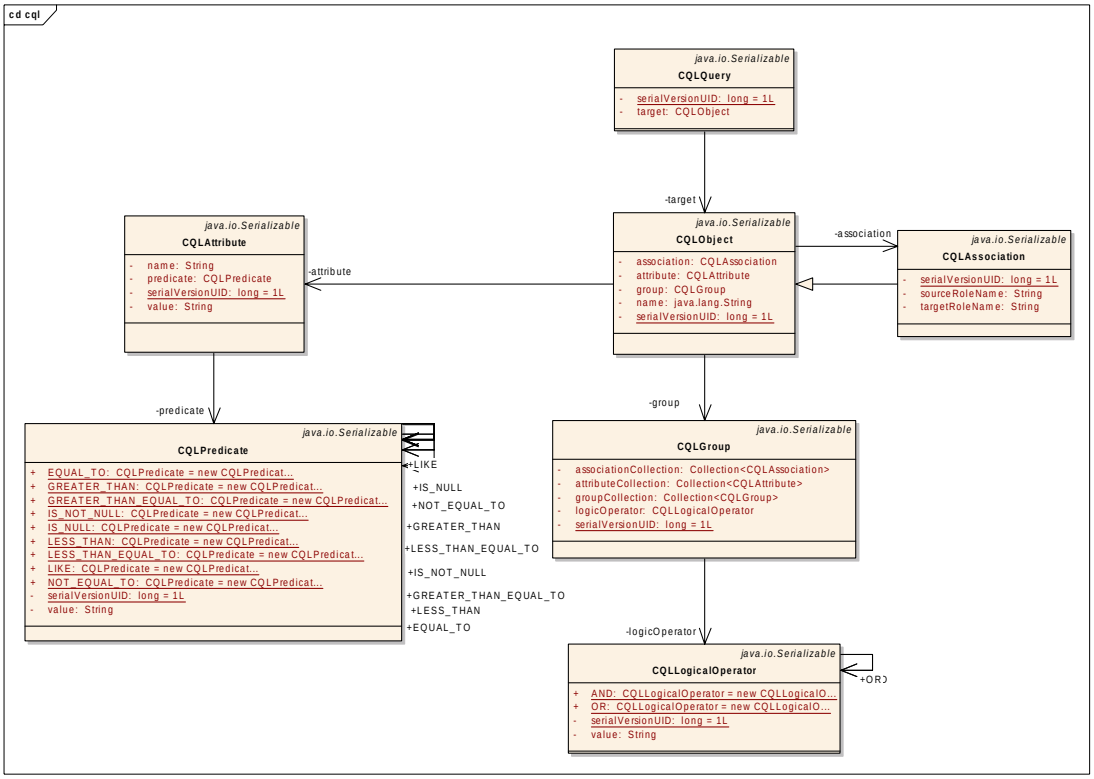


Figure 8-17 CQL Query Association Diagram

The starting object for a CQL query is always a *CQLQuery* object, in which the user has to specify which object (target object) is to be fetched from the database. The target object (CQLObject) is an example of the object that a user intends to search. The example query object has space for:

1. an attribute (CQLAttribute)

2. an association (CQLAssociation) and
3. a group (CQLGroup) of association collection and attributes collection.

For example, to search for an object with one of its attributes called *zipcode* with a value equal to *20852*, a CQLObject must be created with a CQLAttribute object populated inside it. The CQLAttribute object will have its *name* attribute's value as *zipcode* and *value* attribute's value as *20852*. The CQLAttribute object will also require a CQLPredicate for comparison between CQLAttribute and the database value. In this example, the CQLPredicate of EQUAL_TO will be selected and is equivalent to "where zipcode=20852". CQLGroup allows the logical grouping of other groups, attributes, or associations. CQLGroup can be utilized to create a query like "where zipcode=20852 and name like '%Dav%' ".

The following table highlights the *CQLQuery* related *ApplicationService* query methods.

ApplicationService Method	***Description***
query(CQLQuery cqlQuery)	Retrieves the query result from the data source using the CQL query syntax. Internally, CQL query structure is converted into Hibernate Query Language (HQL). Hibernate in turn converts the HQL into SQL and executes it against the relational database. Also see NOTE below.
query(gov.nih.nci.cagrid.cqlquery.CQLQuery cqlQuery)	Retrieves the query result from the data source using the CQL query syntax. Internally, CQL query structure is converted into Hibernate Query Language (HQL). Hibernate in turn converts the HQL into SQL and executes it against the relational database. Also see NOTE below.
query(CQLQuery cqlQuery, String targetClassName)	Deprecated. Internally calls the query(CQLQuery cqlQuery) method without the targetClassName parameter.

Table 8-6 CQLQuery related ApplicationService query methods

NOTE: For the above listed methods, the retrieved results are converted into a list that may not be completely loaded. If the number of retrieved records is greater than the maximum number of supported records as indicated by the getMaxRecordsCount() method, the result set will only contain a subset of the total records. The client framework will execute a subsequent query against the ApplicationService (transparent to the client application) to load the remaining results in the list chunk.

The following examples provide instances of how to create and execute a CQL query using the *ApplicationService* interface. Figure 8-18 shows classes from the sample SDK model package *gov.nih.nci.cacoresdk.domain.other.levelassociation*, and is provided as a point of reference.

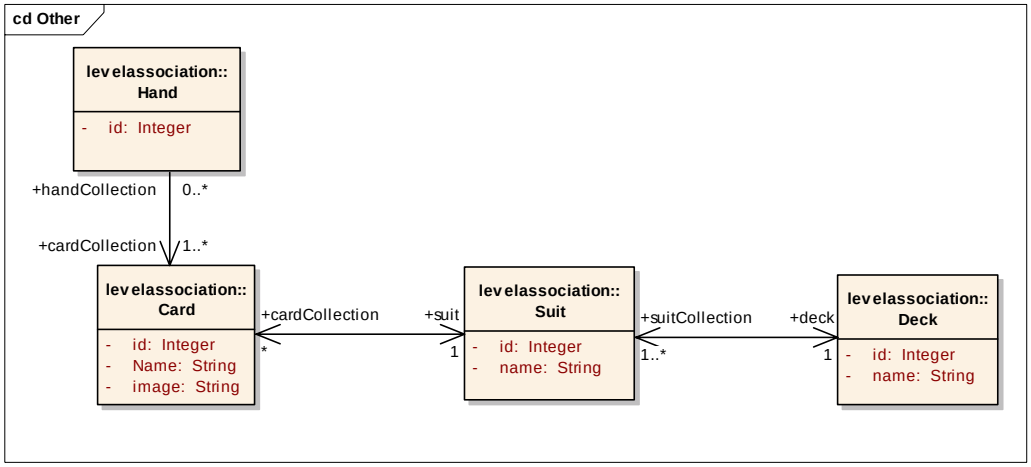


Figure 8-18 Sample Domain Class Diagram

Figure 8-19 below shows how an SDK CQL query object might be instantiated and the query submitted as “select * from Suit where id=1”. Figure 8-20 below that shows how the results would be returned.

```
ApplicationService appService = ApplicationServiceProvider.getApplicationService();

CQLQuery cqlQuery = new CQLQuery();

CQLObject target = new CQLObject();
target.setName("gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit");

CQLAttribute attribute = new CQLAttribute();
attribute.setName("id");
attribute.setValue("1");
attribute.setPredicate(CQLPredicate.EQUAL_TO);

target.setAttribute(attribute);

cqlQuery.setTarget(target);

Collection results = appService.query(cqlQuery);
for(Object obj : results)
{
    printObject(obj, Suit.class);
    break;
}
```

Figure 8-19 Sample CQL Query without Association

```
[echo] ****************************************************************
[echo] ***  caCORESDK: Running the CQL Query
[echo] ****************************************************************
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit$$Enhancer
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1,
[java] Name:Spade
[java] Id:1
```

Figure 8-20 Sample CQL Query without Association Results

Figure 8-21 below shows how an SDK CQL query object might be instantiated and the query submitted as “select * from Suit (select suit from card where id=2 or id=32)”. Figure 8-22 below that shows how the results would be returned.

```
public void testSearch() throws Exception
{
    ApplicationService appService = ApplicationServiceProvider.getApplicationService();

    CQLQuery cqlQuery = new CQLQuery();

    CQLObject target = new CQLObject();
    target.setName("gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit");

    //Create an Association to a Card instance in the Spade Suit
    CQLAssociation association1 = new CQLAssociation();
    association1.setName("gov.nih.nci.cacoresdk.domain.other.levelassociation.Card");
    CQLAttribute attribute1 = new CQLAttribute();
    attribute1.setName("id");
    attribute1.setValue("2"); //Part of the Spade suit
    attribute1.setPredicate(CQLPredicate.EQUAL_TO);

    association1.setTargetRoleName("cardCollection");
    association1.setSourceRoleName("suit");
    association1.setAttribute(attribute1);

    //Create a second Association to a Card instance in the Diamond Suit
    CQLAssociation association2 = new CQLAssociation();
    association2.setName("gov.nih.nci.cacoresdk.domain.other.levelassociation.Card");
    CQLAttribute attribute2 = new CQLAttribute();
    attribute2.setName("id");
    attribute2.setValue("32");  //Part of the Diamond suit
    attribute2.setPredicate(CQLPredicate.EQUAL_TO);

    association2.setTargetRoleName("cardCollection");
    association2.setSourceRoleName("suit");
    association2.setAttribute(attribute2);

    //Add both associations to a Group
    CQLGroup group = new CQLGroup();
    group.addAssociation(association1);
    group.addAssociation(association2);
    group.setLogicOperator(CQLLogicalOperator.OR);

    target.setGroup(group);

    cqlQuery.setTarget(target);

    Collection results = appService.query(cqlQuery);
    System.out.println("Number of qualifying records: " + results.size());
    for(Object obj : results)
    {
        printObject(obj, Suit.class);
    }
}
```

Figure 8-21 Sample CQL Query with Association

```
[echo] ****************************************************************
[echo] ***  caCORESDK: Running the CQL With Association Query Test
[echo] ****************************************************************
[java] Number of qualifying records: 2
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1, gov.
[java] Name:Spade
[java] Id:1
[java] --------
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1b, gov
[java] Name:Diamond
[java] Id:3
[java] --------
```

Figure 8-22 Sample CQL Query with Association Results

For more information related to CQL, please refer to the Data Services:CQL wiki located at: http://www.cagrid.org/mwiki/index.php?title=Data_Services:CQL.

NOTE: The Data Services:CQL wiki describes an XML representation of the CQL. The SDK Code Generator, however, consumes a corresponding object representation of the query instead of the XML version.

Nested Search Criteria Query

SDK Nested Search Criteria queries are developed specifically for SDK and have two parts: 1) a comma separated path to the target search object and 2) an example of the source object.

The comma separated path starts with the target object to be retrieved from the database (using the fully qualified name of the class). The next item in the comma-separated path is a link in the chain to an element that connects the element on its left to the element on its right (also using the fully qualified name of the class). The element on the right could be the example object or another element in the chain. The linked element provides a mechanism to traverse from the example object to the desired object using a comma separated path.

Table 8-7 highlights the *Nested Search Criteria* related *ApplicationService* methods.

NOTE: For the methods listed in the table below, the retrieved results are converted into a list that may not be completely loaded. If the number of retrieved records is greater than the maximum number of supported records as indicated by the getMaxRecordsCount() method, the result set will only contain a subset of the total records. The client framework will execute a subsequent query (transparent to the client application) against the ApplicationService to load the remaining results in the list chunk.

ApplicationService Method	Description
search(String path, List<?> objList)	Retrieves the result from the data source using a Nested Search Criteria. The path specifies the list of objects (separated by commas), which should be used to reach the target object from the example objects passed in the objList, or the associated object for the example object. Internally, the Nested Search Criteria is converted into the data source specific query language. For the Object Relational Mapping based persistence tier, the query structure is first converted into the Hibernate Query Language (HQL). Hibernate then converts the HQL into SQL and executes it against the relational database. Also see NOTE above.
search(Class targetClass, List<?> objList)	Retrieves the result from the data source using the Query by Example query language. The targetClass specifies the object that to fetch after executing the query. The targetClass should be the same as the object specified in the objList or associated object for the example object. All the objects in the objList have to be the same type. The example query is converted into the data source specific query language. For the Object Relational Mapping based persistence tier, the example query structure is first converted to a Nested Search Criteria, and then to Hibernate Query Language (HQL). Hibernate then converts the HQL into SQL and executes it against the relational database. Also see NOTE above.
search(Class targetClass, Object obj)	Retrieves the result from the data source using the Query by Example query language. The targetClass specifies the object that the user intends to fetch after executing the query. The targetClass should be same as the example object or associated object for the example object. The example query is first converted into the data source specific query language. For the Object Relational Mapping based persistence tier, the example query structure is first converted to a Nested Search Criteria, and then to Hibernate Query Language (HQL). Hibernate finally converts the HQL into SQL and executes it against the relational database. Also see NOTE above.
search(String path, Object obj)	Retrieves the result from the data source using the Nested Search Criteria. The path specifies the list of objects (separated by commas) which should be used to reach the target object from the example object passed as obj, or the associated object for the example object. Internally, the Nested Search Criteria is converted into the data source specific query language. For the Object Relational Mapping based persistence tier, the query structure is first converted into the Hibernate Query Language (HQL). Hibernate then converts the HQL into SQL and executes it against the relational database. Also see NOTE above.

Table 8-7 Nested Search Criteria related ApplicationService methods

Figure 8-23 below demonstrates how to use the nested search criteria. Figure 8-24 below that shows how the results would be returned.

In this example, the Suit class is retrieved from the database from the Card object. There are two different instances of the Card object inside the cardCollection that will be ORed,

and their corresponding Suit will be retrieved. The resulting query will be as “select * from Suit where suit in (select suit from card where id=2 or id=6)”:

```
public void testSearch() throws Exception
{
    ApplicationService appService = ApplicationServiceProvider.getApplicationService();

    Card card1 = new Card();
    card1.setId(6);

    Card card2 = new Card();
    card2.setId(2);

    List<Card> cardCollection = new ArrayList<Card>();
    cardCollection.add(card1);
    cardCollection.add(card2);

    String path = "gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit," +
                  "gov.nih.nci.cacoresdk.domain.other.levelassociation.Card";

    Collection results = appService.search(path, cardCollection);
    System.out.println("Number of qualifying records: " + results.size());
    for(Object obj : results)
    {
        printObject(obj, Suit.class);
    }
}
```

Figure 8-23 Sample Nested Search Criteria Query

```
[echo] ********************************************************************
[echo] ***  caCORESDK: Running the Nested Search Query Test
[echo] ********************************************************************
[java] Number of qualifying records: 4
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1, gov.
[java] Name:Spade
[java] Id:1
[java] --------
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@e, gov.
[java] Name:Flower
[java] Id:2
[java] --------
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@1b, gov
[java] Name:Diamond
[java] Id:3
[java] --------
[java] Printing gov.nih.nci.cacoresdk.domain.other.levelassociation.Suit
[java] Deck:gov.nih.nci.cacoresdk.domain.other.levelassociation.Deck@1
[java] CardCollection:[gov.nih.nci.cacoresdk.domain.other.levelassociation.Card@28, gov
[java] Name:Heart
[java] Id:4
[java] --------
```

Figure 8-24 Sample Nested Search Criteria Query Results

Writable API Usage

This section provides an overview of the writable API operations. In the SDK generated system, the user can perform data manipulation through one of the available operations for the writable API. The writable API operations are available in the following three categories:

- Query By Example (QBE) Operations
- Bulk operations (DML)
- Batch Operations.

NOTE: Since the SDK uses Hibernate as underlying technology, it is constrained by the rules that are enforced by same technology. For example, when performing an insert operation, Hibernate will persist the associated objects only if the cascade style on the association specifies such an action.

Query By Example (QBE) Operations

SDK allows users to perform manipulation of the queries by constructing an example of the query and performing the appropriate operation. For example, a user can create a new instance of the Person object and ask the SDK to create a new record in the database by supplying the created instance. The SDK will take the Person object from the user and using Hibernate, will insert the record into the appropriate table. Similarly, the user can perform an update or delete operation using an example-like query.

Currently there are four different query object types supported by the SDK's writable API:

- InsertExampleQuery – Inserts a record into the database using the example object.
- UpdateExampleQuery – Updates a record in the database using the example object.
- SearchExampleQuery – Searches for the object by converting the example into HQL.
- DeleteExampleQuery – Deletes a record in the database using the example object.

Insert Object Example:

```
1. Person person = new Person();
2. person.setName("Jane Doe");
3. WritableApplicationService appService = (WritableApplicationService)
        ApplicationServiceProvider.getApplicationService();
4. SDKQuery query = new InsertExampleQuery(person);
5. SDKQueryResult result = appService.executeQuery(query);
6. person = (Person) result.getObjectResult();
```

In the example above we are inserting a new Person object/record into the database with person's name set as "Jane Doe". Lines 1-2 create an instance of the Person

object. Line 3 retrieves a handle to the writable API (i.e. WritableApplicationService) from the factory (i.e. ApplicationServiceProvider). Line 4 creates an instance of the query operation. Line 5 executes the query against the SDK generated API and retrieves the result. Line 6 retrieves the result of the query from the wrapper.

Bulk operations (DML)

Bulk operations can be performed when a user wants to update multiple records with the same values, which meet certain criteria. For example, take a backup and update the last achieved date of all the Person records to the current system date. Retrieving each record and saving each after updating it is not a particularly efficient way to perform this task.

The SDK generated writable API is capable of performing bulk operations by executing a user-specified HQL query statement. This approach allows bulk updates to be performed without significant network overhead.

Similar to the example query updates, the bulk mode also allows updates by using different types of query objects as listed below:

- InsertHQLQuery – Executes an insert query command as provided by the user.
- UpdateHQLQuery – Executes an update query command as provided by the user.
- DeleteHQLQuery – Executes a delete query command as provided by the user.
- SearchHQLQuery – Executes a search query command as provided by the user and returns the results.

Update Records Example:

```
1. WritableApplicationService appService = (WritableApplicationService)
        ApplicationServiceProvider.getApplicationService();
2. SDKQuery query = new UpdateHQLQuery("update Person p set
        p.archieveDate=sysdate() ");
3. appService.executeQuery(query);
```

In the example above we are updating all the Person records and setting the archiveDate attribute's value to the system date. Line 1 retrieves a handle to the writable API (i.e. WritableApplicationService) from the factory (i.e. ApplicationServiceProvider). Line 2 creates an instance of the query operation. Line 3 executes the query against the SDK generated API.

Although this approach for updating records based on the user-specified HQL query is simple from an implementation perspective, it exposes user to the Hibernate Query Language which may not be desired by all users. In the future, the SDK will provide a custom Query-by-Example-like system for updating records.

Batch Operations

The SDK generated writable API allows user to perform writable operations in batch mode. In the batch mode, the user passes a list of queries to be executed in a single

transaction to the SDK's API. The SDK generated API begins the transaction before executing the first query from the batch and commits the transaction at the end of the last transaction. In the event of any failures, the configured transaction manager rolls back all of the transactions performed since the start of the batch.

Insert Object Example:

1. WritableApplicationService appService = (WritableApplicationService)
   ApplicationServiceProvider.getApplicationService();
2. Person person1 = new Person();
3. person1.setName("Jane Doe");
4. SDKQuery query1 = new InsertExampleQuery(person1);
5. Person person2 = new Person();
6. person2.setName("Mark Smith");
7. SDKQuery query2 = new InsertExampleQuery(person2);
8. List queryList = new ArrayList();
9. queryList.add(query1);
10. queryList.add(query2);
11. List<SDKQueryResult> results = appService.executeBatchQuery(queryList);

In the example above we are creating two Person records in the same transaction. Line 1 retrieves a handle to the writable API (i.e. WritableApplicationService) from the factory (i.e. ApplicationServiceProvider). Lines 2-3 create an instance of a new Person record. Line 4 creates an instance of the first query operation. Lines 5-6 create a second instance of a Person record. Line 7 creates an instance of the second query operation. Lines 8-10 create the batch of query operations as a List. Line 11 executes the batch query against the SDK generated API.

Web Service Interface

The SDK 4.x Web Service interface is based on the Axis 1.4 framework, which adheres to the J2EE 1.4 server programming model described by JAX-RPC and JSR 109 (that is, the SDK 4.x Web Service uses the Remote Procedure Call (RPC) Web Service style).

There are four "styles" of services in Axis:

- *RPC* services use the SOAP RPC conventions as well as the SOAP "section 5" encoding.
- *Document* services do not use any encoding but DO still do XML<->Java databinding. So in particular, you will not see multiref object serialization or SOAP-style arrays on the wire.
- *Wrapped* services are just like document services, except that rather than binding the entire SOAP body into one big structure, they "unwrap" it into individual parameters.
- *Message* services receive and return arbitrary XML in the SOAP Envelope without any type mapping/data binding.

For more information on these service styles, see http://ws.apache.org/axis/java/user-guide.html#ServiceStylesRPCDocumentWrappedAndMessage.

Note: While the SDK Web Service continues to be based on the Axis 1.4 framework, the extraneous .ws layer found in previous (3.x) SDK versions has been eliminated.

In addition, the SDK Web Service Deployment Descriptor (WSDD) is now packaged along with the rest of the SDK generated system, thus allowing for automatic deployment of the Web Service; manual deployment of the Web Service is no longer required.

A sample test program illustrating how the SDK generated Web Service can be consumed, *TestClient.java*, is provided in the *\output\example\package\ws-client\src* folder. More information about this test program is provided in *Testing the Web Service Interface* on page 129.

If the SDK generated system is secured, the web services user is required to supply user credentials in the form of a web service message header. As part of the web service message, a new header called *SecurityHeader* is required to be added to the web service call. This header has an element called *security* with two child elements named *username* and *password* respectively. The values of these child elements reflect the user's login name and login password for the underlying application.

The code snippet shown below demonstrates the usage of *SecurityHeader* in Java.

```
SOAPHeaderElement headerElement = new

        SOAPHeaderElement(call.getOperationName().getNamespaceURI(),"SecurityHeader");
headerElement.setPrefix("security");
headerElement.setMustUnderstand(false);
SOAPElement usernameElement = headerElement.addChildElement("username");
usernameElement.addTextNode("userId");
SOAPElement passwordElement = headerElement.addChildElement("password");
passwordElement.addTextNode("password");
call.addHeader(headerElement);
```

Figure 8-25 Security Login in Java based web services client

NOTE: The web service communication is stateless. Hence, the user of the web service is required to provide the login information in the header of the message each time it makes a call to the server.

The remainder of this section provides specifications for the SDK generated Web Service via the Web Services Description Language (WSDL), and includes an overview of the schema imports, service, port types, and messages found within the WSDL. For more information related to the WSDL format and structure, see http://www.w3.org/TR/wsdl.html or http://en.wikipedia.org/wiki/WSDL.

SDK WSDL Directives - Schema Imports

Figure 8-26 below provides a list of the schema imports found within the WSDL for the sample SDK model.

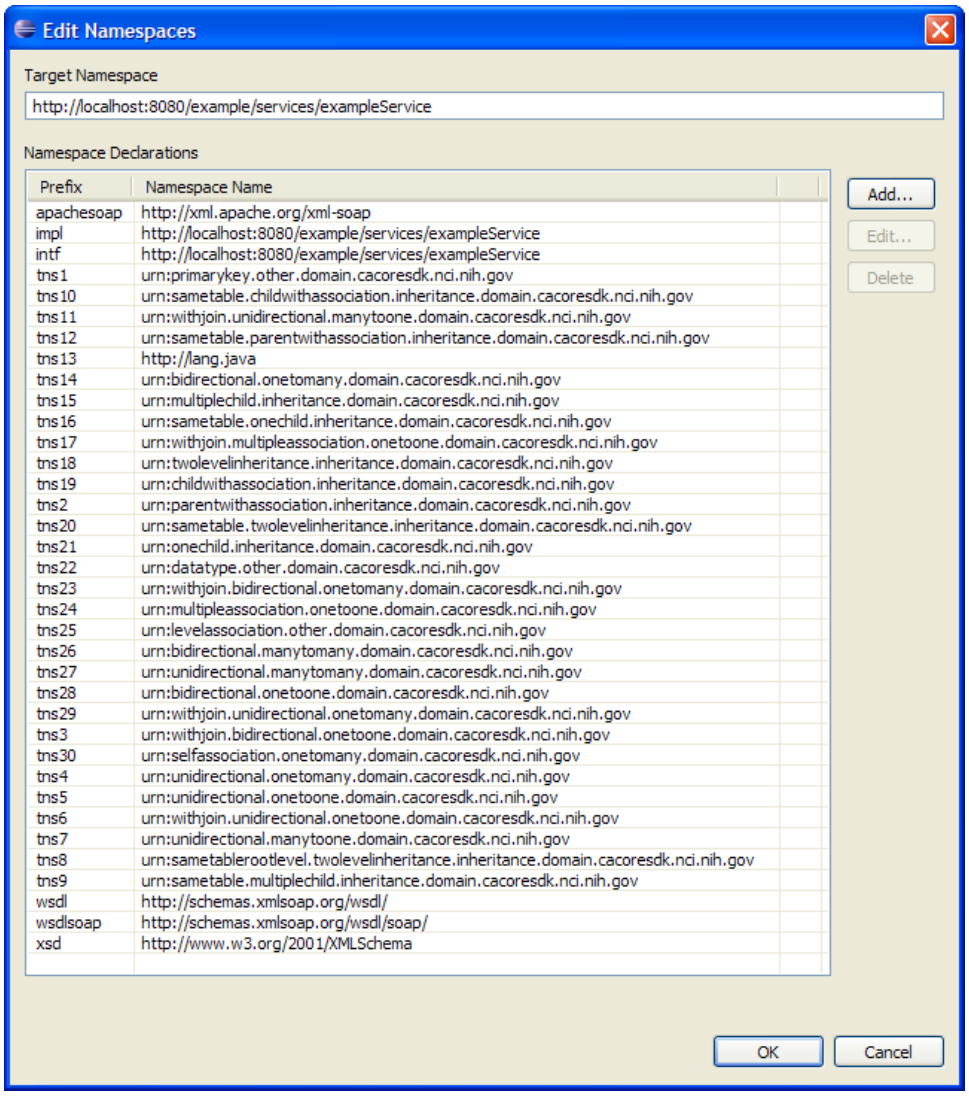


Figure 8-26 Sample WSDL Directives - Schema Imports

This graphic is provided here to emphasize the point that a schema import statement is added to the WSDL for each of the distinct domain package(s) found within the model provided to the SDK Code Generator.

WSDL Service Definition

The WSDL defines a Web Service as a collection of network endpoints, or *ports*. Figure 8-27 and Figure 8-28 below provide details for the SDK generated Web Service defined within the WSDL, which include **Name**, **Prefix**, **Target Namespace**, and **Port Information**.

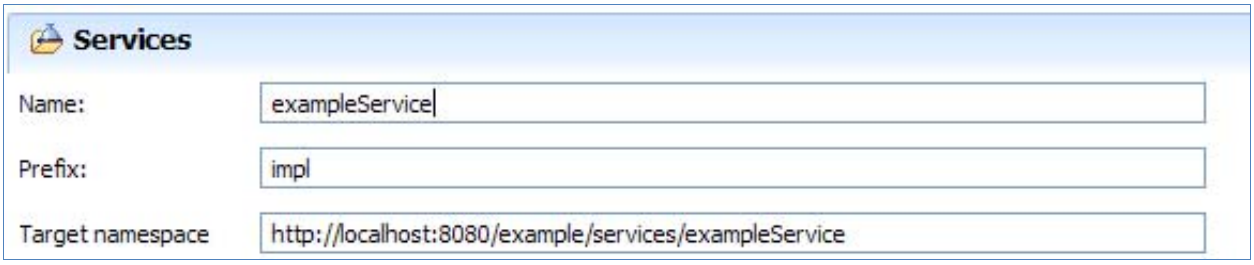


Figure 8-27 Sample WSDL Service Definition

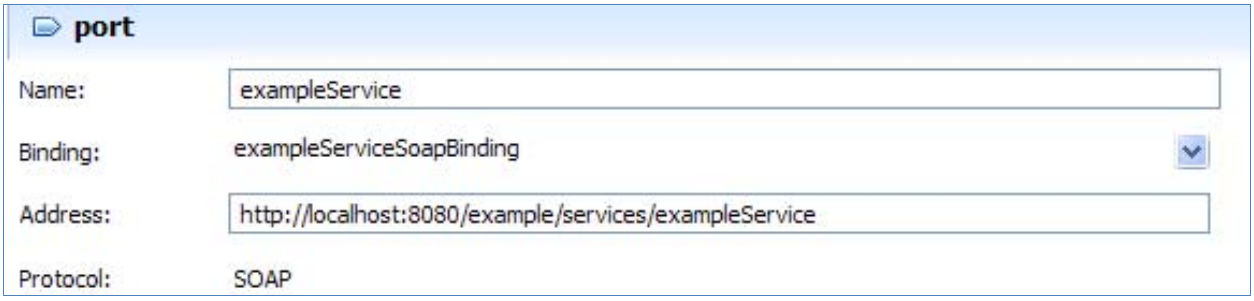


Figure 8-28 Sample WSDL Service Definition – Port

Note: The SDK Code Generator uses the value of the PROJECT_NAME property provided within the `deploy.properties` file while generating the WSDL (in this case, "example"). Therefore, while the information displayed above is specific to the sample SDK model, the same pattern is followed in the generation of the WSDL Service and Port definitions for other models.

WSDL Port Types (Network Endpoints)

The WSDL defines a port as an association of a network address with a reusable binding. Port types, in turn, are abstract collections of supported operations. Figure 8-29 below displays a summary of the collection of network endpoints (and their messages) that compose any SDK generated Web Service.

WSQueryImpl		
getAssociation		
input	source	anyType
	associationName	string
	startIndex	int
output	getAssociationReturn	ArrayOf_xsd_anyType
getTotalNumberOfRecords		
input	targetClassName	string
	criteria	anyType
output	getTotalNumberOfRecordsReturn	int
queryObject		
input	targetClassName	string
	criteria	anyType
output	queryObjectReturn	ArrayOf_xsd_anyType
query		
input	targetClassName	string
	criteria	anyType
	startIndex	int
output	queryReturn	ArrayOf_xsd_anyType

Figure 8-29 WSDL Port Types (Network Endpoints)

Messages, Elements, and Types

The WSDL defines messages as abstract descriptions of the data being exchanged. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding, where the messages and operations are then bound to a concrete network protocol and message format.

Table 8-8 below provides a summary of the messages and elements (including parameters and data types) that make up the Web Service defined in the WSDL for the sample SDK model.

Message	Description
getAssociationRequest	The getAssociationRequest message is used by a Web Service client to request object(s) associated to a given Java domain object instance. Required parameters include: • source: An instance of the Java domain object containing the association (rolename) method to be invoked; • associationName: The name of the method (rolename) that represents the associated object(s) to be returned; • startIndex: The starting index into the resulting dataset. Useful during subsequent calls when "scrolling" through a large result dataset. Initial requests should set the value of this parameter to zero (0).
getAssociationResponse	The getAssociationResponse message is used by the SDK server to provide any qualifying objects associated to the source Java domain object. The response is an array of qualifying objects.
getTotalNumberOf RecordsRequest	The getTotalNumberOfRecordsRequest message is used by a Web Service client to request a count of the total number of records that would be returned for a given search criteria. Required parameters include: • targetClassName: The fully qualified class name of the search object type to be returned. This may represent the name of the criteria object class itself or the name of a class associated to the criteria object. • criteria: a sample instance of the criteria search object, containing values for any desired field(s) (attributes) that should act as a filter (constraint) on the resulting dataset;
getTotalNumberOf RecordsResponse	The getTotalNumberOfRecordsResponse message is used by the SDK server to provide a count of the total number of records that would be returned for a given search criteria. The response type is a positive integer (int), or zero, if no qualifying records are found.
queryRequest	The queryRequest message is used by a Web Service client to request object(s) that meet the supplied search criteria. Internally, a nested search criteria query is performed. Required parameters include: • targetClassName: The fully qualified class name of the search object type to be returned. This may represent the name of the criteria object class itself or the name of a class associated to the criteria object. • criteria: a sample instance of the criteria search object, containing values for any desired field(s) (attributes) that should act as a filter (constraint) on the resulting dataset • startIndex: The starting index into the resulting dataset. Useful during subsequent calls when "scrolling" through a large result dataset. Initial requests should set the value of this parameter to zero (0).
queryResponse	The queryResponse message is used by the SDK server to return any objects that meet the search criteria passed via the queryRequest message. The response is an array of qualifying objects.

Message	Description
queryObjectRequest	The queryObjectRequest message is used by a Web Service client to request object(s) that meet the supplied search criteria. Required parameters include: • targetClassName: The fully qualified class name of the search object type to be returned. This may represent the name of the criteria object class itself or the name of a class associated to the criteria object. • criteria: a sample instance of the criteria search object, containing values for any desired field(s) (attributes) that should act as a filter (constraint) on the resulting dataset. **NOTE:** The queryObjectRequest operation has the same effect as invoking the queryRequest message with a startIndex of zero (0). A different operation/message name had to be used, as the Axis 1.4 framework does not allow the "overloading" of method signatures.
queryObjectResponse	The queryObjectResponse message is used by the SDK server to return any objects that meet the search criteria passed via the queryObjectRequest message. The response is an array of qualifying objects.

Table 8-8 Summary of messages and elements for Web Service as defined in WSDL

Web Service Error Handling

The errors that may be generated during a message exchange between a Web Service client and a generated SDK system Web Service fall into one of the two following categories:

- Those that would be generated by the generated SDK application, and
- Those that would be generated by any of the framework APIs used during the message exchange between systems.

In both instances, a SOAP Fault element handles the transport of error messages. More information related to the SOAP Fault can be found in the Simple Object Access Protocol (SOAP) 1.1 Specification: http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

The application-related errors occur when the SDK generated application cannot fulfill a request from a Web Service client. For example, a Web Service client sends a *getAssociationRequest* message but supplies an invalid *associationName* value.

In the case of the Web Services framework API, an error could occur when a message cannot reach its destination. This could be caused by any number of issues, such as an interruption in the network, an issue with the message structure or message load, etc. In these instances, the Web Services framework generates an error relevant to the incident and a SOAP Fault element transports the message to the client.

SOAP Fault Structure

As stated above, the SOAP Fault element is used to carry error and/or status information within a SOAP message. If a Fault element is present, it must appear as a child element of the Body element. A Fault element can only appear once in a SOAP message.

The SOAP Fault element has the following sub elements, shown in Table 8-9 below.

Sub Element	Description
<faultcode>	A code for identifying the fault.
<faultstring>	A human readable explanation of the fault.
<faultactor>	Information about who caused the fault to happen.
<detail>	Holds application specific error information related to the Body element.

Table 8-9 SOAP Fault Structure Element Descriptions

Chapter 9 Utilities

This chapter describes a class that can be used to serialize and deserialize generated Java Beans to XML and back again.

Topics in this chapter include:

XML Utility (Marshalling and Unmarshalling)

While used primarily by caGrid, the caCORE SDK does provide a class, *XMLUtility.java*, which can be used to marshal (serialize) the generated domain Java Beans to XML, and unmarshal (deserialize) XML data back to the generated domain Java Beans. This class is shown in Figure 9-1 below.

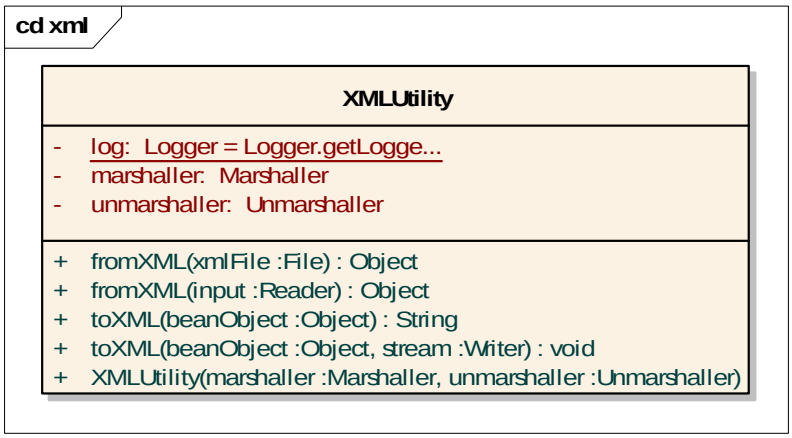


Figure 9-1 XML Utility Class Diagram

As implied by the *XMLUtility Constructor* method, the *XMLUtility* class wraps both an SDK Marshaller and Unmarshaller class, which it depends on to perform its work. These collaborating classes and their interfaces are discussed in the sections that follow.

The caCOREMarshaller Class

The SDK *caCOREMarshaller* class implements the SDK *Marshaller* interface and is used by the *XMLUtility* class to perform the actual work of marshalling (serializing) domain Java Bean objects to XML. This class is shown in Figure 9-2 below.

NOTE: The caCOREMarshaller class is used internally by the XML Utility infrastructure and is not typically invoked by the end user.

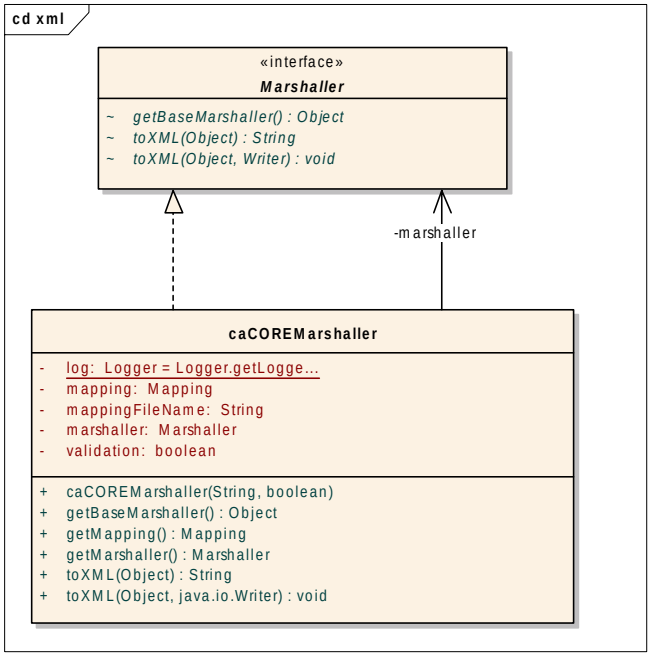


Figure 9-2 Marshaller Class Diagram

The *caCOREMarshaller* uses *Castor* technology, and utilizes the SDK generated *xml-mapping.xml* file, which provides Java-to-XML binding settings used by the Castor engine.

Castor is an Open Source data binding framework for Java, and facilitates conversion between Java Beans, XML documents, and relational tables. Castor provides Java-to-XML binding, Java-to-SQL persistence, and more. See http://www.castor.org/ for more information. Mappings are included for value attributes, collections, and associations to other domain Java Beans.

NOTE: When processing associations and collections, the caCOREMarshaller also uses custom Castor collection and domain object Field Handlers. This is done in order to prevent infinite recursion whenever domain classes have circular references/associations to each other. Consequently, associations and collections are only serialized to their first level.

Marshalling Java Objects to XML

The *XMLUtility* class provides two wrapper methods for marshaling (serializing) domain Java objects to XML, as described below in Table 9-1.

XMLUtility Method	*Description*
toXML(Object beanObject)	Accepts a domain Java Bean instance and passes it to the Marshaller instance (caCOREMarshaller, by default), which in turn marshals (serializes) the instance to XML and returns it as an XML string.
toXML(Object beanObject, Writer stream)	Accepts a domain Java Bean instance. This object is similarly passed to the Marshaller instance (caCOREMarshaller, by default), which marshals (serializes) it to XML. However, the XMLUtility then writes the serialized XML string to a character stream writer instead.

Table 9-1 Wrapper methods for marshaling domain Java objects to XML

The code snippet shown in Figure 9-3 below demonstrates how one of the XML Utility marshaling methods might be invoked.

```
Marshaller marshaller = new caCOREMarshaller("xml-mapping.xml", false);
Unmarshaller unmarshaller = new caCOREUnmarshaller("unmarshaller-xml-mapping.xml", false);
XMLUtility myUtil = new XMLUtility(marshaller, unmarshaller);

File myFile = new File(someDomainObject.getClass().getName() + "_test.xml");

FileWriter myWriter = new FileWriter(myFile);
myUtil.toXML(someDomainObject, myWriter);
myWriter.close();
```

Figure 9-3 Sample Marshaling code

A sample test program, *TestXMLClient.java*, is provided in the folder `\output\example\package\remote-client\src` folder. More information about this test program is provided in *Testing the XML Utility* on page 127.

The caCOREUnmarshaller Class

The SDK *caCOREUnmarshaller* class implements the SDK *Unmarshaller* interface and is used by the *XMLUtility* class to perform the actual work of unmarshalling (deserializing) XML to domain Java Bean objects. This class is shown in Figure 9-4 below.

NOTE: The caCOREUnmarshaller class is used internally by the XML Utility infrastructure, and is not typically invoked by the end user.

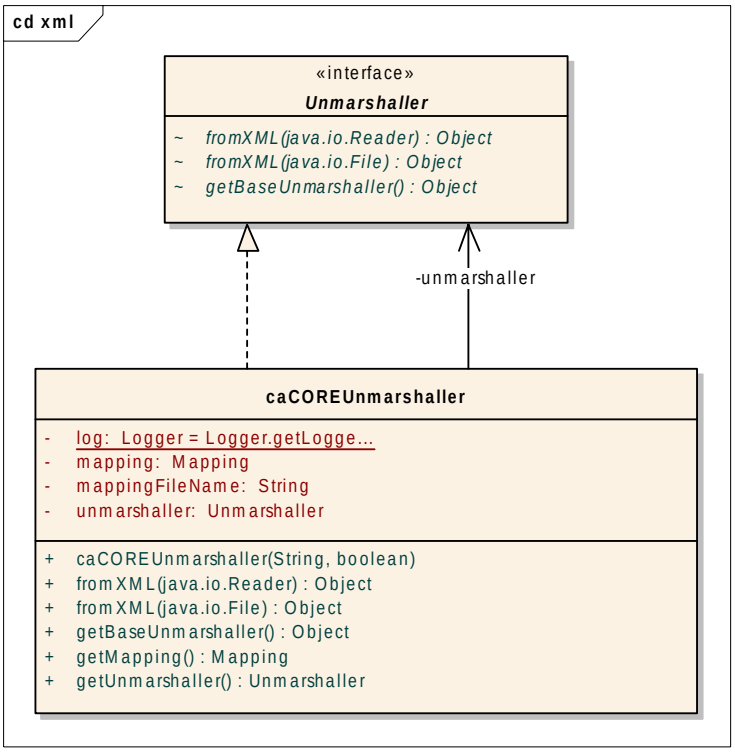


Figure 9-4 Unmarshaller Class Diagram

The *caCOREUnmarshaller* uses *Castor* technology and utilizes the SDK generated *unmarshaller-xml-mapping.xml* file, which provides XML-to-Java binding settings used by the Castor engine. Mappings are included for value attributes, collections, and associations to other domain Java Beans.

Unmarshalling XML to Java Objects

The *XMLUtility* class provides two wrapper methods for unmarshalling (deserializing) XML to domain Java objects, as described below in Table 9-2.

XMLUtility Method	*Description*
fromXML(File xmlFile)	Instantiates a domain Java Bean object from an XML file that contains the serialized output of that object.
fromXML(Reader input)	In addition, instantiates a Java Bean domain object from XML, but reads it instead from a java.io.Reader character stream.

Table 9-2 Wrapper methods for unmarshalling XML to domain Java objects

The highlighted portion of the code snippet shown in Figure 9-5 below demonstrates how one of the XML Utility unmarshalling methods, fromXML(File), can be invoked.

```
Marshaller marshaller = new caCOREMarshaller("xml-mapping.xml", false);
Unmarshaller unmarshaller = new caCOREUnmarshaller("unmarshaller-xml-mapping.xml", false);
XMLUtility myUtil = new XMLUtility(marshaller, unmarshaller);

File myFile = new File(someDomainObject.getClass().getName() + "_test.xml");

SomeDomainObject myObj = (SomeDomainObject) myUtil.fromXML(myFile);
...
```

Figure 9-5 Sample Unmarshalling Code

A sample test program, *TestXMLClient.java*, is provided in the folder `\output\example\package\remote-client\src`. More information about this test program is provided in *Testing the XML Utility* on page 127.

Chapter 10 Creating the UML Model for caCORE SDK

This chapter provides information on how to create UML models that can be used by the caCORE SDK to generate the system.

Topics in this chapter include:

Introduction

The SDK Code Generator is based upon a Model-Driven Architecture (MDA) that supports the implementation of the following scenarios specified via a UML model:

- Modeling of class attributes including :
  - A simple (primitive) attribute, such as an *integer* or *string*;
  - A collection of simple (primitive) attributes; and,
  - An identifier attribute that is named something other than the default (*ID)* in the Logical (Object) Model.
- Modeling of class associations, including:
  - Uni- and bi-directional associations;
  - Many-to-Many, Many-to-One, One-to-Many, and One-to-One associations;
  - Associations that use a Join Table;
  - Associations that *do not* use a Join Table;
- Modeling of inheritance that is implemented using:
  - One table per class in inheritance hierarchy
  - One table per inheritance hierarchy
  - One table per inheritance hierarchy, with a separate table for leaf-level child class(es)
  - One table per concrete child class
- Modeling of Interface as marker interface

The caCORE SDK distribution provides a sample model that demonstrates how these scenarios, and many others, can be modeled in a manner that is understood by the SDK Code Generator. The sample model is intended to be used as a reference when creating your own model. The sample model is located within the *\models* directory of the SDK distribution, and has been implemented in both Enterprise Architect and ArgoUML. The name of the sample model project files are:

- Enterprise Architect: *SDKTestModel.EAP*
- ArgoUML: *sdk.uml*

The following sections describe how to perform various modeling activities using both the Enterprise Architect (EA) and ArgoUML modeling tools.

Creating a New Project in Enterprise Architect (EA)

This section provides instructions for creating a new object model project file. The instructions in this and subsequent sections are separated into separate subsections for Enterprise Architect (EA) and ArgoUML.

Creating a New Project in EA

To create a new object model project file:

1. Open the *SDKEATemplate.EAP* baseline file provided in the *\models* directory of the SDK distribution. This file already contains the base *Logical View*, *Data Model*, and *Logical (Object) Model* packages, as well as classes representing the wrapper Java primitive type classes.

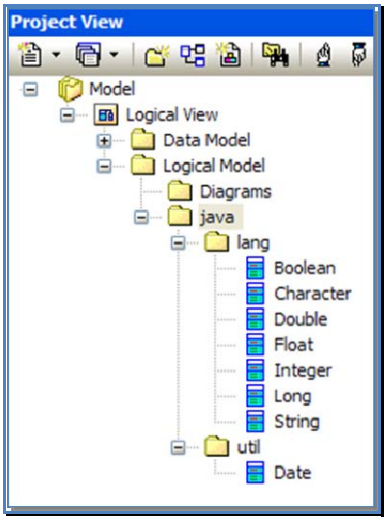


Figure 10-1 EA Project View Browser

2. Under the **File** menu, select **Save Project As**. The Save Enterprise Architect Project dialog appears.

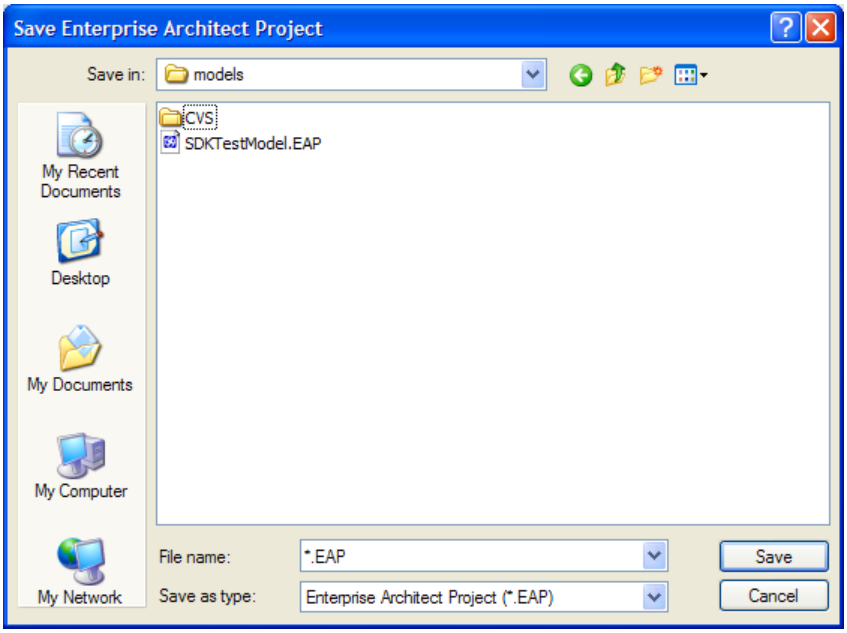


Figure 10-2 EA Save Enterprise Architect Project dialog

3. Enter a new project name in the **File name** field.
4. Click **Save**.

Alternatively, the baseline template file (*SDKEATemplate.EAP)* can be copied and renamed.

The new project file is now ready to use for creating the object and data model.

Creating a New Project in ArgoUML

To create a new object model project file:

1. Open the *SDKArgoTemplate.UML* baseline file provided in the *\models* directory of the SDK distribution. This file already contains the base *Logical View*, *Data Model*, and *Logical (Object) Model* packages, classes representing the wrapper Java primitive type classes, *Tag Definitions (TD)* for all the possible Tag Value types, and *DataTypes*.

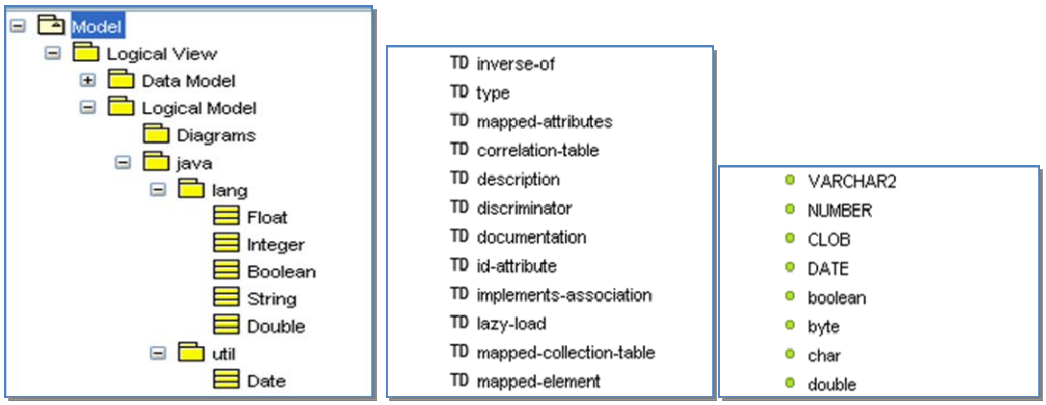


Figure 10-3 ArgoUML Explorer - showing packages/classes ,Tag definitions, data types

2. Under the **File** menu, select **Save Project As**. The Save Project dialog appears.

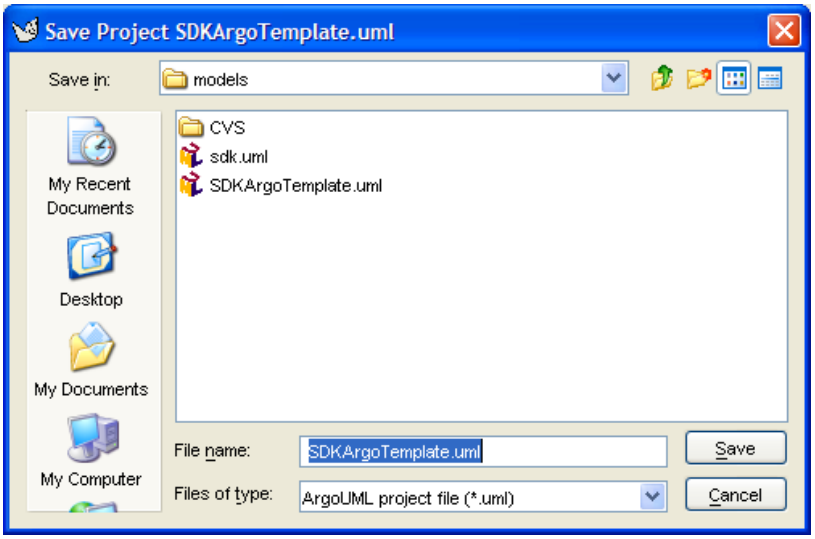


Figure 10-4 ArgoUML Save Project dialog

3. Enter a new project name in the **File name** field.
4. Click **Save**.

Alternatively, the baseline template file (*SDKArgoTemplate.uml)* can be copied and renamed.

The new project file is now ready to use for creating the object and data model.

Creating Classes and Tables

UML *Class* elements are used to represent both *Logical (object) Model* classes and *Data Model* classes (tables). Object classes are typically created using a package hierarchy within the Logical Model package, while Data Model classes (tables) are created directly within the Data Model package without the use of a package hierarchy.

Creating a Logical Model Package Structure in EA

To add a package structure to the Logical Model:

1. Select the *Logical Model* package.

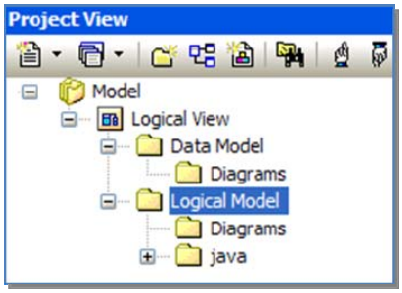


Figure 10-5 EA Project Browser

2. Right-click and select **Add > Add Package**, or from the main menu, select **Project > Add Package**. The New Package dialog box appears.

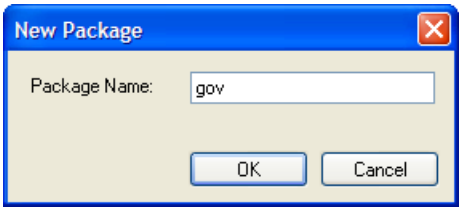


Figure 10-6 EA New Package Dialog

3. Enter a Package (folder) Name, and click **OK**.

   Notes Regarding Package Names:

   - Package names should follow Java package naming conventions; i.e., Java packages are defined using a hierarchical, *lowercase*, naming pattern, with levels in the hierarchy separated by periods (.) . Furthermore, package names are typically the organization's domain name backwards. An example, taken from the SDK sample model, is *gov.nih.nci.cacoresdk.domain.*
   - When implemented within EA, each period designates the end of one package level, and the start of a new package level (termed a *subpackage*). Each package/subpackage needs to be created individually, meaning no period(s) should be used when specifying a package name in the *New Package* dialog. Thus the fully qualified package *gov.nih.nci.cacoresdk.domain* requires a total of five (5) packages to be created within the model, one for each of the package levels. Each package is nested within the higher-level package.

4. Repeat these steps until the fully qualified package hierarchy has been created. To create a package within another package (as a sub-package/folder), select the existing package first, and then follow steps 2-3 above.

Figure 10-7 below shows most of the package hierarchy created in this manner for the SDK sample model.

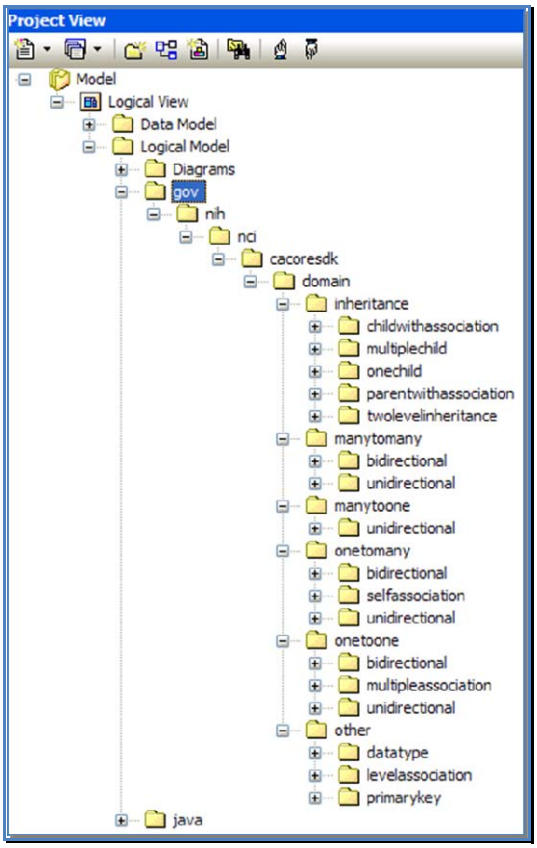


Figure 10-7 EA SDK Sample Model Packages

Creating a Logical Model Package Structure in ArgoUML

If necessary, see http://argouml-stats.tigris.org/documentation/manual-0.24/ch11.html for more information on the ArgoUML Explorer pane.

To add a package structure to the Logical Model:

1. Select the *Logical Model* package.

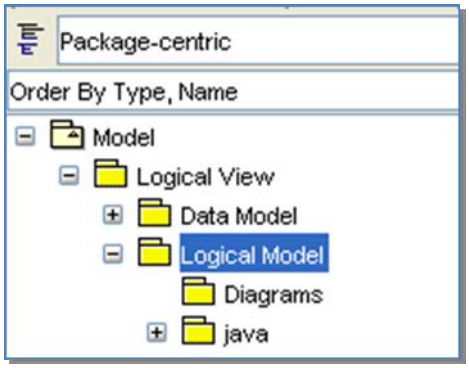


Figure 10-8 ArgoUML Explorer Pane

2. Right-click and select **Add Package**. The **Properties** tab in the Detail pane becomes active for the new package.

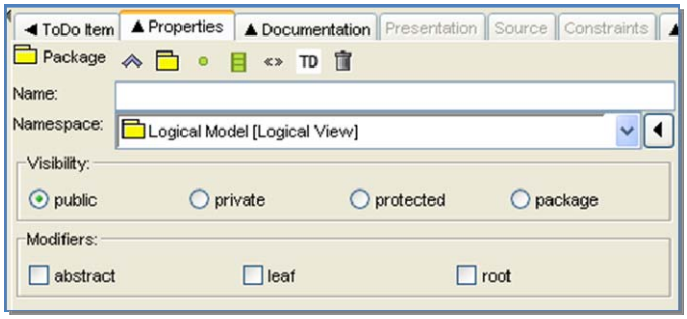


Figure 10-9 ArgoUML Package Detail Pane, Properties Tab

NOTE: See http://argouml-stats.tigris.org/documentation/manual-0.24/ch13s03.html for more information on the ArgoUML Detail Pane, Properties tab.

3. Enter a package (folder) name, and then click the **Save Project** icon ( ) or press CTRL-S.

   NOTE: Package names should follow Java package naming conventions; i.e., Java packages are defined using a hierarchical, *lowercase*, naming pattern, with levels in the hierarchy separated by periods (.). Furthermore, package names are typically the organization's domain name backwards. An example, taken from the SDK sample model, is *gov.nih.nci.cacoresdk.domain*.

4. Repeat these steps until the fully qualified package hierarchy has been created. To create a package within another package (as a sub-package/folder), select the existing package first, and then follow steps 2-3 above.

For reference, Figure 10-7 shows most of the package hierarchy created (in EA) for the SDK sample model.

Creating a Logical (Object) Model Class in EA

To add a Logical Model class to a package:

1. In the EA *Project Browser*, find the desired *Logical Model package* to which the class should be added.
2. Right-click the package and select **Add > Add Element** or from the main menu, select **Project > Add Element**. The Insert New Element dialog appears.

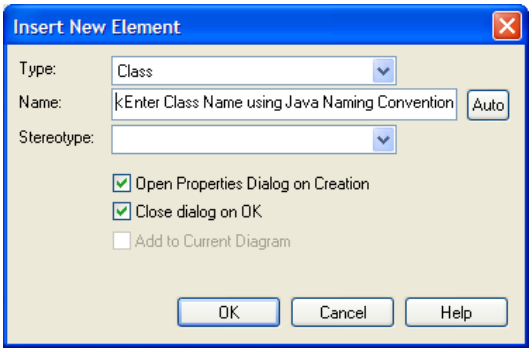


Figure 10-10 EA Insert New Element Dialog

3. In EA, set the Insert New Element options as follows:

Insert New Element Option	***Description***
Type	Select Class as the element type from the drop down list.
Name	Enter a class name according to the Java class naming conventions; i.e., class names should start with a capital letter, with embedded words capitalized.
Stereotype	Leave blank for Logical (object) Model classes.
Open Property Dialog	Check this option if you want the Property dialog to open immediately after the class is created.
Close dialog on OK	Uncheck this option if you want to add multiple classes in one session.

NOTE: Logical (Object) Model class names should follow Java class naming conventions; i.e., class names should start with a capital letter, with embedded words capitalized. An example from the SDK sample model is *GraduateStudent*.

4. Click **OK**. If the **Open Property dialog on OK** option was checked, the Property dialog opens immediately after the class is created. The Property dialog for the SDK sample *Credit* class is shown in Figure 10-11 below.

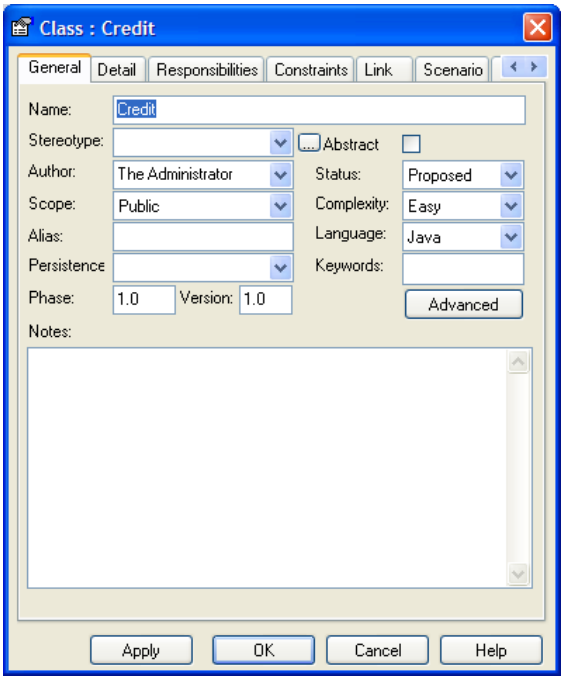


Figure 10-11 EA Class Property Dialog

NOTE: The *Stereotype* field is blank since this class represents a domain object, and not a data table.

5. Repeat these 1-5 to add other classes.

   If the **Close dialog on OK** option was unchecked in the Insert New Element dialog, additional classes can be created in the selected package by only repeating steps 4-5.

For instructions on adding attributes to classes, see *Creating Attributes and Data Types* on page 97.

Figure 10-12 below shows a series of classes that have been created in the many-to-many *bidirectional* and *unidirectional* packages of the SDK Sample model.

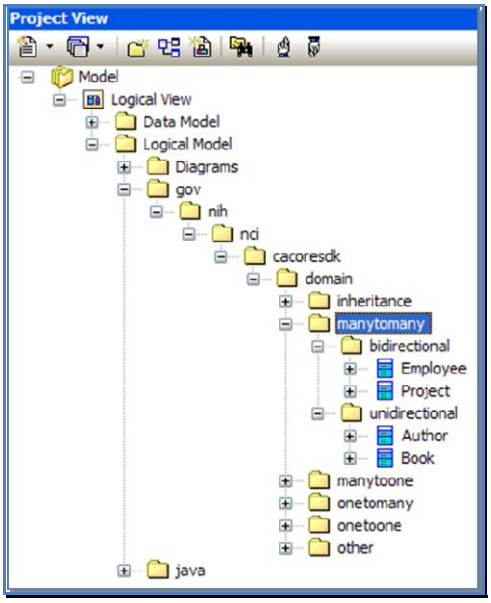


Figure 10-12 EA Project View Browser Showing SDK Sample Classes

Creating a Logical (Object) Model Class in ArgoUML

In ArgoUML, new classes are added in the context of a class diagram within the selected package.

To add a Logical Model class to a package:

1. In the ArgoUML *Explorer* pane, click on a class diagram within the package to open/activate it, or create a new class diagram within the package if none exists.
2. Select the **New Class** icon ( ) found at the top of the diagram Editing pane. The Properties tab in the Detail pane becomes active for the new class.

   Note: See http://argouml-stats.tigris.org/documentation/manual-0.24/ch12.html for more information about the Editing pane.

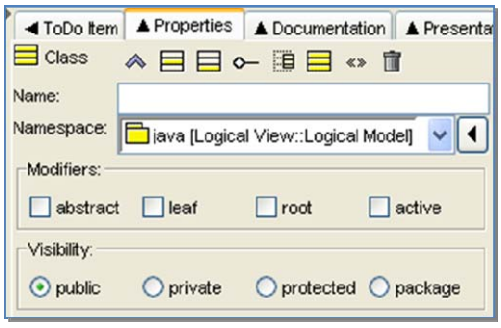


Figure 10-13 ArgoUML Class Detail Pane, Properties Tab

3. Enter a class name in the **Name** field.

   NOTE: Logical (Object) Model class names should follow Java class naming conventions; i.e., class names should start with a capital letter, with

embedded words capitalized. An example from the SDK sample model is *GraduateStudent*.

4. Click **Save Project** ( ) or press CTRL-S.
5. Repeat these steps to add other classes.

For instructions on adding attributes to classes, see *Creating Attributes and Data Types* on page 97.

For reference, Figure 10-12 above shows a series of classes that were created (using EA) in the many-to-many *bidirectional* and *unidirectional* packages of the SDK Sample model.

Creating a Data Model Table in EA

Unlike object model classes that are created using a package hierarchy, all table classes should be created within the *Data Model* package.

To add a Data Model (Table) class:

1. In the EA Project Browser, select the **Data Model** package.

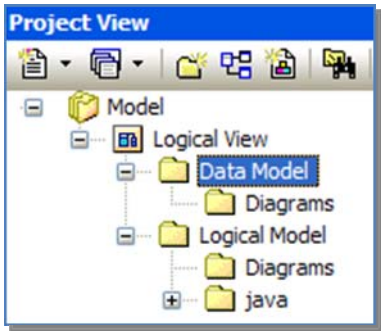


Figure 10-14 EA Data Model Package

2. Right-click and select **Add > Add Element**, or from the main menu, select **Project > Add Element**. The *Insert New Element* dialog appears.

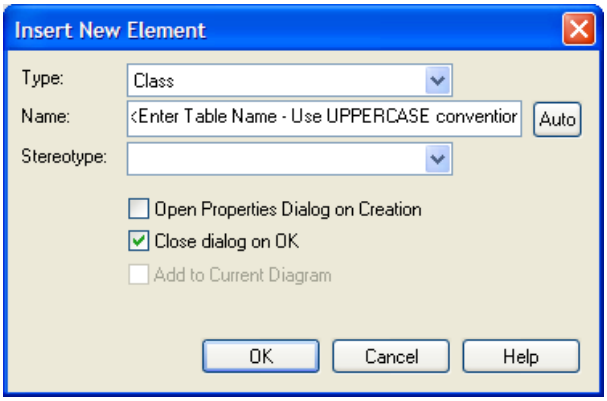


Figure 10-15 EA Insert New Element (Table) Dialog

3. Set the Insert New Element options as follows:

Insert New Element Option	Description
Type	Select Class as the element type from the drop down list.
Name	Enter a table name according to the Table naming conventions; i.e., table names should be all uppercase, with embedded words separated by an underscore (_).
Stereotype	Select table from the drop down list.
Open Property Dialog	Check the Open Property Dialog option if you want the Property dialog to open immediately after the table is created.
Close dialog on OK	Uncheck the Close dialog on OK option if you want to add multiple tables in one session.

NOTE: Table names should follow Table naming conventions; i.e., table names should be all uppercase, with embedded words separated by an underscore (_). An example from the SDK sample model: *UNDERGRADUATE_STUDENT*.

4. When finished, click **OK**.

   If the **Open Property Dialog on Creation** was checked, the Property dialog opens immediately after the class is created. Figure 10-16 shows the Property dialog for the SDK sample *Credit* table. Notice that the Stereotype field is set to *table.*

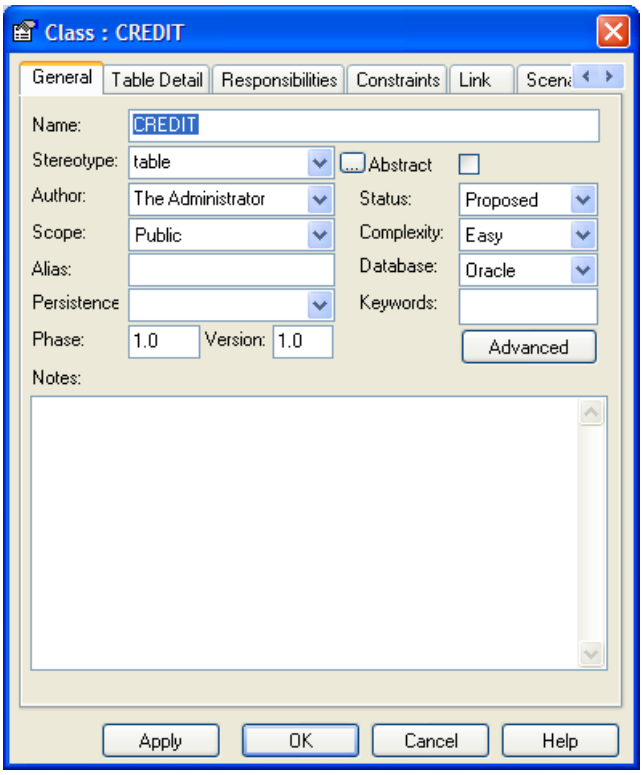


Figure 10-16 Property dialog for the SDK sample Credit

5. Repeat these steps to add other tables.

If the **Close dialog on OK** option was unchecked in the Insert New Element dialog, additional tables can be created in the Data Model package by only repeating steps 4-5.

For instructions on adding attributes (columns) to tables, see *Creating Attributes and Data Types* on page 97.

Figure 10-17 below shows various tables that have been created in the *Data Model* package of the SDK sample model.

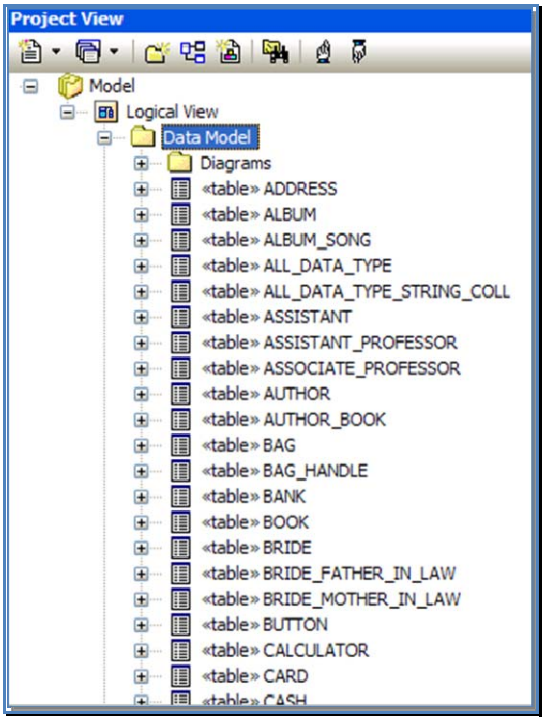


Figure 10-17 EA Various Tables from the SDK Sample Model

Creating a Data Model Table in ArgoUML

In ArgoUML, new classes (tables) are added in the context of a class diagram within the selected package. Also note that unlike object model classes that are created using a package hierarchy, all table classes should be created within the *Data Model* package.

To add a Data Model (Table) class:

1. In ArgoUML Explorer pane, select the **Data Model** package.

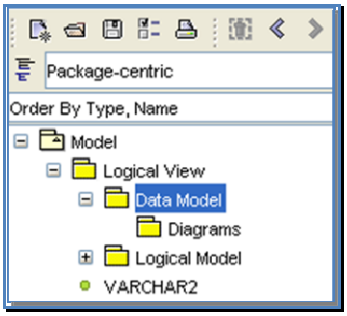


Figure 10-18 ArgoUML Data Model Package

2. Click on a class diagram within the Data Model package to open/activate it, or create a new class diagram if none exists.

3. Select the **New Class** icon ( ) found at the top of the diagram Editing pane. The Properties tab in the Detail pane becomes active for the new table (class).

   Note: See http://argouml-stats.tigris.org/documentation/manual-0.24/ch12.html for more information about the Editing pane.

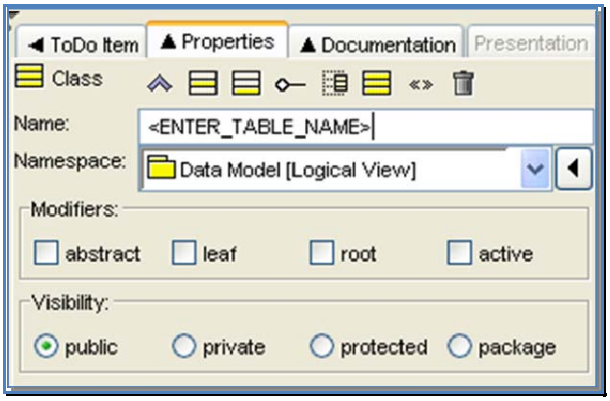


Figure 10-19 ArgoUML Table (Class) Properties Tab

4. Enter a class name in the **Name** field of the Properties tab.

   Note: Table names should follow Table naming conventions; i.e., table names should be all uppercase, with embedded words separated by an underscore (_). An example from the SDK sample model: *UNDERGRADUATE_STUDENT*.

5. Click on the **Stereotype** tab to activate it.

6. Select the **table** stereotype, and apply it to the new class by clicking the **>>** (double-arrow) button to move it to the Applied Stereotypes list.

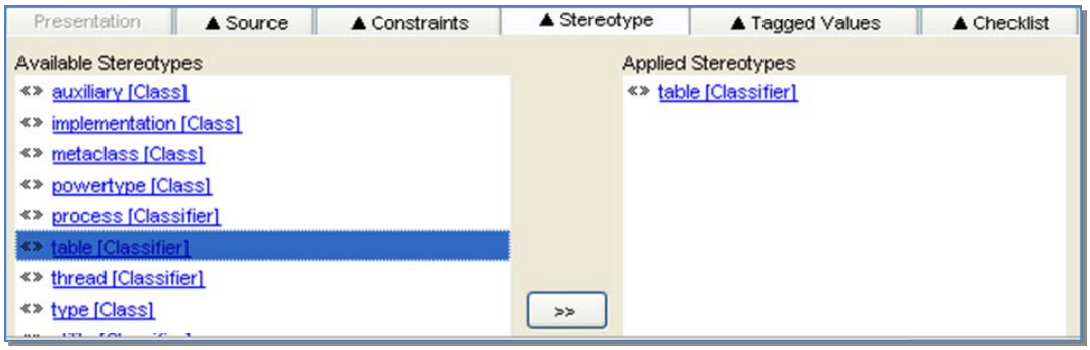


Figure 10-20 ArgoUML Applying a Stereotype to a Table Class

Alternatively, you can select the class within a diagram, right-click to open the shortcut menu, and then select **Apply Stereotypes > table**.

7. Click **Save Project** ( ) or press CTRL-S.
8. Repeat these steps to add other tables.

For instructions on adding attributes (columns) to tables, see *Creating Attributes and Data Types* on page 97.

For reference, Figure 10-17 above shows an EA-generated example of various tables that were created in the *Data Model* package of the SDK sample model.

Creating Attributes and Data Types

UML *Attribute* elements are used to represent both Logical (Object) Model class attributes and Data Model table columns (class attributes). Both Logical Model and Data Model class attributes can be added/modified using the same process outlined below.

Creating/Modifying Attributes and Data Types in EA

To add/modify a class or table attribute:

1. Select the desired **Logical Model** class or **Data Model** table (class) element. Figure 10-21 below shows the Logical Model **AllDataType** class from the SDK sample model selected.

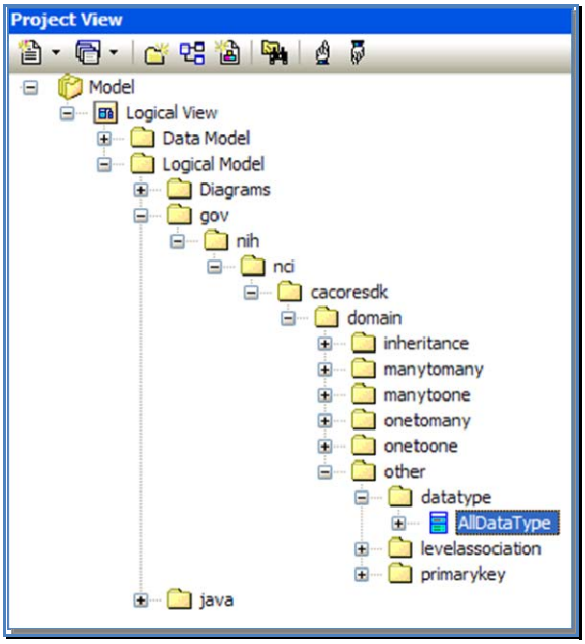


Figure 10-21 Logical Model AllDataType class

2. Right-click and select **Attributes**, or from the main menu, select **Element > Attributes**. The **Attributes** dialog appears.

   Figure 10-22 below shows the Attributes dialog for the Logical Model *AllDataType* class from the SDK sample model. This class illustrates all of the available primitive data types (including primitive collections) that can be assigned to a class attribute.

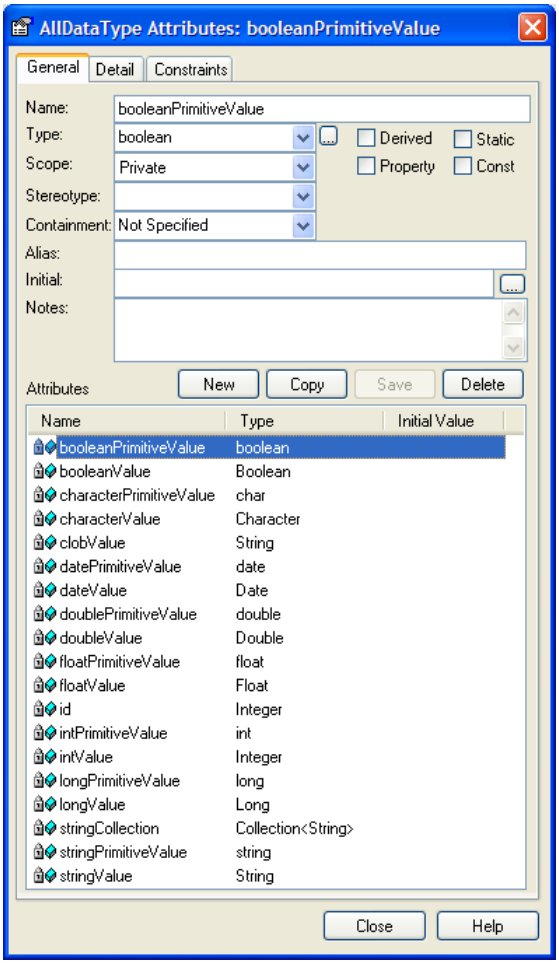


Figure 10-22 Sample AllDataType Class showing available attribute primitive data types

3. To a*dd* an attribute, click **New**.
4. Type an attribute name in the **Name** field and select a type from the **Type** drop down and then click **Save**.

   Important notes about Attributes:

   - The SDK Code Generator is only concerned with the *Name* and *Type* fields. All other fields on the EA *General* tab of the *Attributes* dialog can be ignored (left default).
   - The SDK Code generator understands both primitive wrapper class types (e.g. Boolean) and primitive types (e.g., Boolean). If a particular data type is not shown in the drop down, you can type it into the *Type* field.
   - For a list of the primitive attribute types understood by the SDK Code Generator, reference the *AllDataType* class in the SDK sample model (also shown in the diagram above). Note that primitive collection types (e.g., the *stringCollection* attribute of type *Collection<String>*) are also understood as an attribute type.

5. To *modify* an attribute, select it in the Attributes list, change the value of the *Name* and/or *Type* field, and then click **Save**.

Creating/Modifying Attributes and Data Types in ArgoUML

To add a class or table attribute:

1. Select the desired **Logical Model** class or **Data Model** table (class) element. The **Properties** tab in the Detail pane becomes active for the selected class.
2. Click the **New Attribute** ( ) icon.

   Alternatively, you can select the class within a diagram, right-click to open the shortcut menu, and then select **New Attribute** from the **Add** sub-menu.

   The **Attribute** properties tab becomes active in the Detail pane.

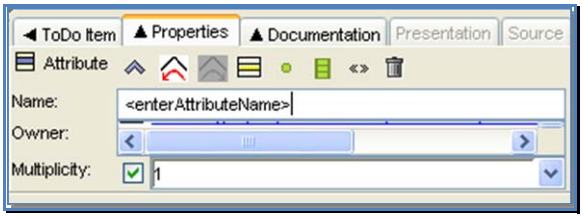


Figure 10-23 ArgoUML Attribute Properties Tab

3. Enter an attribute name in the **Name** field, and then select a type from the **Type** drop down list.
4. Click the **Save Project** icon ( ) or press *CTRL-S* to save the changes.

   Important notes about Attributes:

   - The SDK Code Generator is only concerned with the *Name* and *Type* fields. All other fields on the EA *General* tab of the *Attributes* dialog can be ignored (left default).
   - The SDK Code generator understands both primitive wrapper class types (e.g. Boolean) and primitive types (e.g., Boolean). If a particular data type is not shown in the drop down, you can type it into the *Type* field.
   - For a list of the primitive attribute types understood by the SDK Code Generator, reference the *AllDataType* class in the SDK sample model (also shown in the diagram above). Note that primitive collection types (e.g., the *stringCollection* attribute of type *Collection<String>*) are also understood as an attribute type.

To modify a class or table attribute:

1. Select the attribute in the *Explorer* pane by expanding the class to show its attributes, or if working with a diagram, click on the attribute name within the class in the *Editing* pane

   The **Attribute** properties tab in the Detail pane (Figure 10-23 above).
2. Change the value of the *Name* and/or *Type* field. Click the **Save Project** icon ( ) or press *CTRL-S* to save the changes.

Performing Object Relational Mapping

The SDK Code Generator relies on information contained within custom Tag Values to generate particular system artifacts whenever the information needed cannot be derived from the UML model elements directly (*Class*, *Attributes*, and *Associations*).

Tag Values, for instance, are used to hold class/attribute documentation (comments and/or descriptions of the element) while generating Java Docs for the object model. More importantly, however, Tag Values are used extensively when generating Hibernate Object Relational Mapping *(.hbm.xml)* files. Basically, it can be said that custom Tag Values are at the *heart* of the Logical (Object) Model-to-Data (Table) Model mapping process.

The SDK distribution provides a sample model (located within the *\models* directory) that demonstrates how various scenarios can be modeled through the use of custom Tag Values. In addition, a reference table describing each of the various custom Tag Values and their usage is provided in section *SDK Custom Tag Value Descriptions*.

NOTE: The caCORE Wiki also contains a Tag Summary page, providing information regarding tag values used by each caCORE product. See https://wiki.nci.nih.gov/x/SYl8.

For those who find working directly with Tag Values too cumbersome or error prone, you may want to consider using the **caAdapter** tool, which, among other features, provides the ability to map object models to data models via a Graphical User Interface (GUI). For more information regarding the caAdapter tool/project, see http://trials.nci.nih.gov/projects/infrastructureProject/caAdapter.

Adding/Modifying Tag Values

Both EA and ArgoUML provide the ability to easily view, add, and edit Tag Values for model elements.

In EA, Tag Values attached to a particular UML element (such as a *Class*, *Attribute*, or *Association*) can be added/modified via the *Tagged Value* browser, which is accessible by sequentially clicking and holding down the *Ctrl-Shift-6* keys.

The following diagram from the SDK sample model illustrates an association between the *Employee* and *Project* Logical Model classes.

Figure 10-24 Employee-Project Association Diagram

A sample of the EA Tagged Value browser for the *Association* (line) between both classes is shown in Figure 10-25 below.

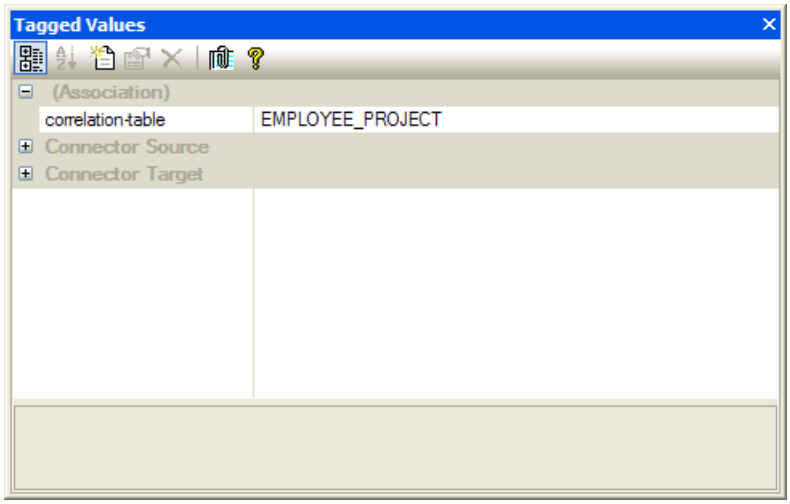


Figure 10-25 EA Tag Values Browser

Once the Tagged Values browser is open, selecting a particular UML element (such as a *Class*, *Attribute*, or *Association*) will cause the browser to display the corresponding Tag Values attached to the selected element.

In ArgoUML, Tag Values attached to a particular UML element (such as a *Class*, *Attribute*, or *Association*) can be added/modified by first selecting the element. This causes the *Detail* pane to become active for the selected element. The Detail pane contains a *Tagged Values* tab, which you can click to activate.

A sample of the Tagged Values tab for the Association between the Sample SDK Employee and Project Logical Model classes is shown below.

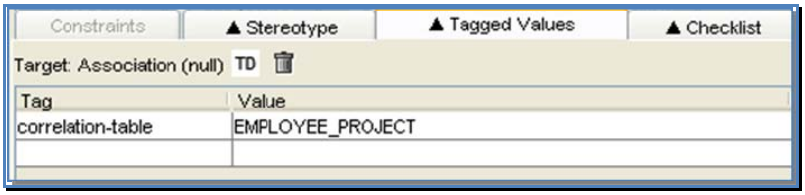


Figure 10-26 ArgoUML Detail Pane - Tagged Values Tab

SDK Custom Tag Value Descriptions

The following sections list the various Tag Values understood by the SDK Code Generator and provides a description of each value including information regarding when and where to use them:

Tag Value: correlation-table

A Tag Value added to an *Association* element (line) drawn between two Logical Model classes within the same diagram. The value specifies the correlation (join) table name.

Given the following *Many-to-Many* relationship diagram:

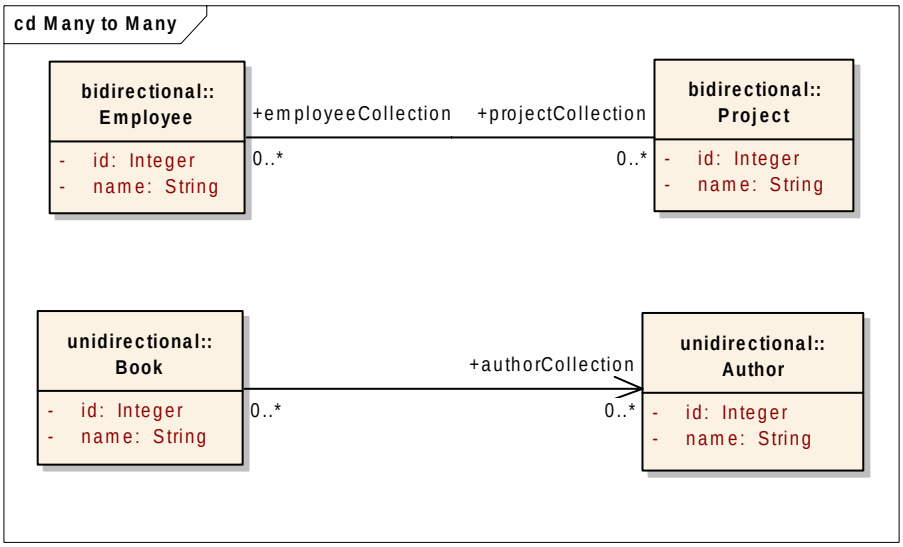


Figure 10-27 Many-to-Many association class diagram

A couple of corresponding examples from the SDK sample model are provided in the table below:

Logical Model Class (Source)	Logical Model Class (Target)	Tag Value (correlation-table) *NOTE: should be added to the Association (line) element*
Employee	Project	EMPLOYEE_PROJECT
Book	Author	AUTHOR_BOOK

Tag Value: description

An optional Tag Value added to a *Class* or *Attribute* element to store documentation/comments for the element. The value describes the element and is used when creating Java Docs for generated domain objects.

The *description* Tag Value can also be used to provide information about the element for semantic integration purposes. The text of this tag value appears as the element description in the Semantic Integration Workbench interface.

NOTE: The *description* Tag Value is only used if the *documentation* tag value for the element is empty or does not exist.

Tag Value: discriminator

A Tag Value added to a Data Model class *Attribute* element. The value of this tag represents the Logical Model class name that acts as the *discriminator* in situations when the parent and sub-class are persisted within the same database table. The value of the tag, if present, is placed within the *discriminator* element of the generated Hibernate mapping file.

A couple of examples from the SDK sample model are provided in the table below:

Data Model Class (Table)	Data Model Attribute (Column)	Tag Value (discriminator)
SHOES	DISCRIMINATOR	gov.nih.nci.cacoresdk.domain.inheritance.chi

		ldwithassociation.sametable.Shoes
GOVERNMENT	DEMOCRATIC_ DISCRIMINATOR	gov.nih.nci.cacoresdk.domain.inheritance.twolevelinheritance.sametable.DemocraticGovt

NOTE: The <discriminator> element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy, and declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row. See http://www.hibernate.org/hib_docs/v3/reference/en/html_single/#mapping-declaration-discriminator for more information.

Tag Value: documentation

An optional Tag Value added to a UML element to store documentation/comments for the element. The value will be used when creating Java Docs for the generated domain object.

The *documentation* Tag Value can also be used to provide information about the element for semantic integration purposes. The text of this tag value appears as the element description in the Semantic Integration Workbench interface.

See also the *description* Tag Value above.

Tag Value: id-attribute

A Tag Value added to a Logical Model class *Attribute*. The presence of the Tag Value indicates that the attribute is the class *identifier* attribute. This Tag Value is required when the identifier attribute is named something other than the default name, *id*. The value should specify the fully-qualified name of the Logical Model class that contains the attribute.

An example from the SDK sample model is provided in the table below:

Logical Model Class Name	Logical Model Attribute Name	Tag Value (id-attribute)
NoIdKey	myKey	gov.nih.nci.cacoresdk.domain.other.primarykey.NoIdKey

Tag Value: implements-association

A Tag Value added to a Data Model class *Attribute* (column). The value specifies the associated Logical Model class attribute that implements the association. The value must be specified using the following pattern: *<fully-qualified logical model class name>.< attribute name>*.

A couple of examples from the SDK sample model are provided in the table below:

Data Model Class (Table)	Data Model Attribute (Column)	Tag Value (implements-association)
CARD	SUIT_ID	gov.nih.nci.cacoresdk.domain.other.levelassociation.Card.suit
ASSISTANT	PROFESSOR_ID	gov.nih.nci.cacoresdk.domain.inheritance.parentwithassociation.Assistant.professor

Tag Value: inverse-of

A Tag Value added to a Data Model class *Attribute* (column). Used to identify the inverse attribute (column) of a bi-directional association. The value specifies the corresponding inverse Logical Model class attribute, and must have the same value as the *implements-association* Tag Value of the bi-directional association. The value must be specified using the following pattern: *<fully-qualified logical model class name>.< attribute name>.*

Given the following Logical Model class diagram from the sample SDK model:

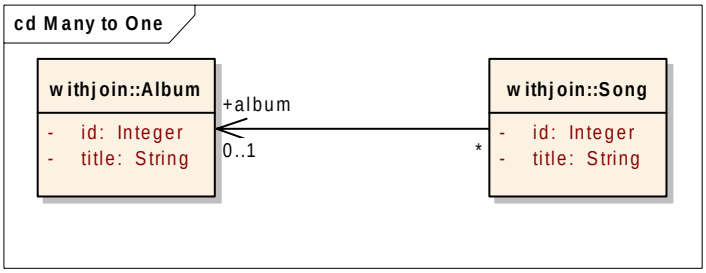


Figure 10-28 Many-to-One association class diagram

And the corresponding Data Model class diagram:

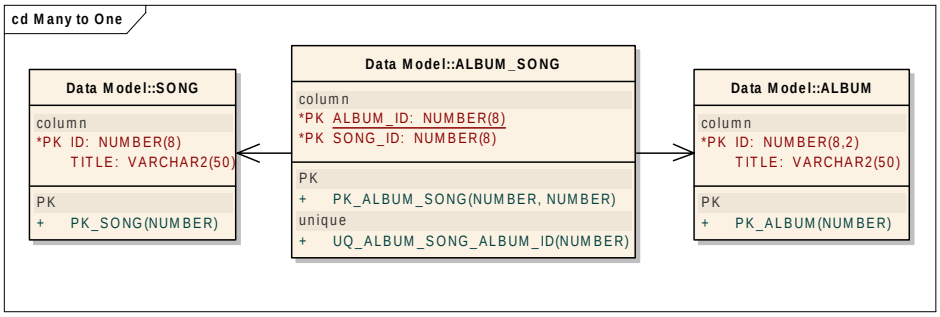


Figure 10-29 Data model diagram with correlation table

An example *inverse-of* Tag Value would be:

Data Model Class (Table)	Data Model Attribute (column)	Tag Value (inverse-of)
ALBUM_SONG	SONG_ID	gov.nih.nci.cacoresdk.domain.manytoone.unidirectional.withjoin.Song.album

This indicates that the SONG_ID attribute (column) is the inverse side of the Song/Album bi-directional association implemented by *album* attribute of the *Song* class.

NOTE: When adding an inverse-of value to a Data Model class Attribute for a Many-to-Many association, One-to-Many join table, Many-to-One join table, or a One to One - No Join Table scenario, make sure to supply the same value for both the *implements-association* and *inverse-of* tag values of the bi-directional association.

See also the related *implements-association* Tag Value above.

Tag Value: lazy-load

A Tag Value added to an *Association* element between two Logical Model classes. The value specifies whether the association should be fetched lazily or not.

Permissible values are *yes*, and *no*. That is, any value other than *yes* is treated as a *no*. This tag value sets the lazy attribute in the generated *.hbm.xml* file to either *true* or *false* accordingly.

No example is provided in the SDK sample model.

Tag Value: mapped-attributes

A Tag Value added to a Data Model class *Attribute* (Column). The value specifies the corresponding mapped Logical Model class *Attribute*. The value must be specified using the following pattern: *<fully-qualified logical model class name>.< attribute name>.*

A couple of examples from the SDK sample model are provided in the table below:

Data Model Class (Table)	Data Model Attribute (column)	Tag Value (mapped-attributes)
UNDERGRADUATE_STUDENT	STUDENT_ID	gov.nih.nci.cacoresdk.domain.inheritance.multiplechild.UndergraduateStudent.id
SHOES	ID	gov.nih.nci.cacoresdk.domain.inheritance.childwithassociation.sametable.Shoes.id

Tag Value: mapped-collection-table

A Tag Value added to a Logical Model class *Attribute*. The value specifies the name of the mapped primitive collection (non-domain class – e.g., String, Integer) table.

Given the following Data Model class diagram from the SDK sample model:

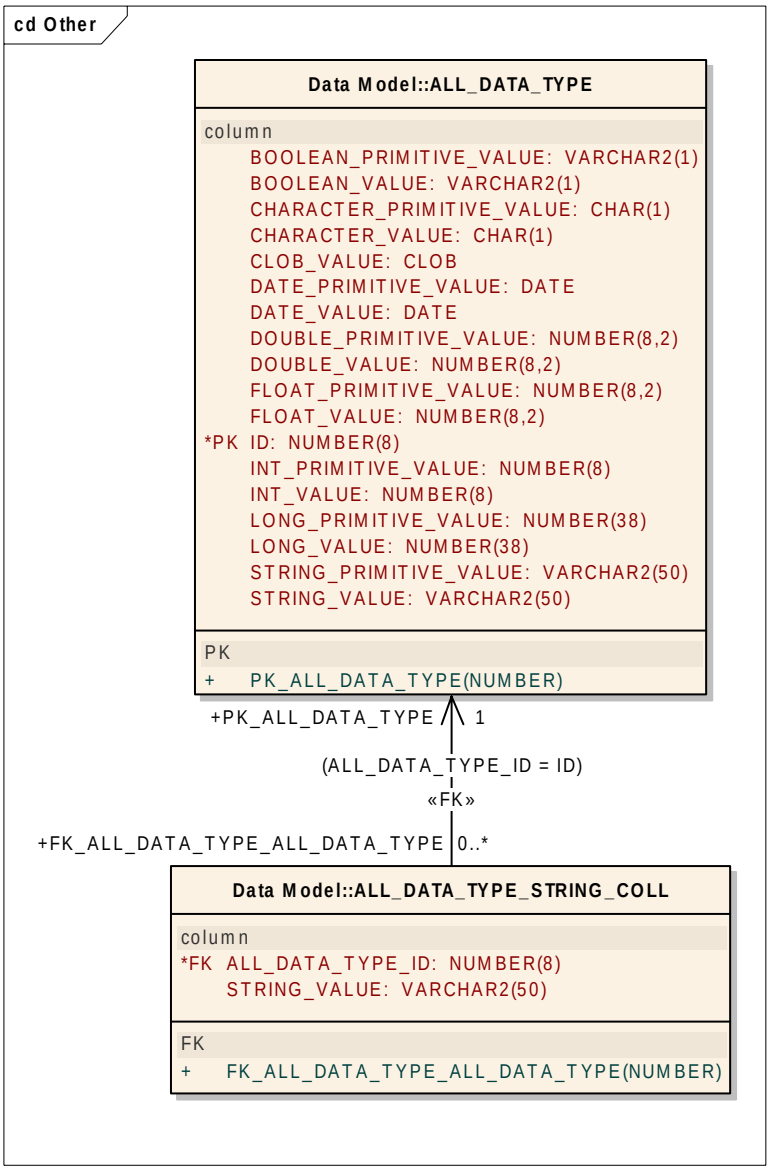


Figure 10-30 Data model for storing collections of primitives

An example *mapped-collection-table* Tag Value would be:

Logical Model Class	Logical Model Attribute	Tag Value (mapped-collection-table)
AllDataType	stringCollection	ALL_DATA_TYPE_STRING_COLL

Tag Value: mapped-element

A Tag Value added to a Data Model class *Attribute* (Column). The value specifies the name of the mapped primitive collection (non-domain class – e.g., String, Integer) Logical Model class attribute. The value must be specified using the following pattern: *<fully-qualified logical model class name>.< attribute name>.*

Given the following Logical Model class diagram:

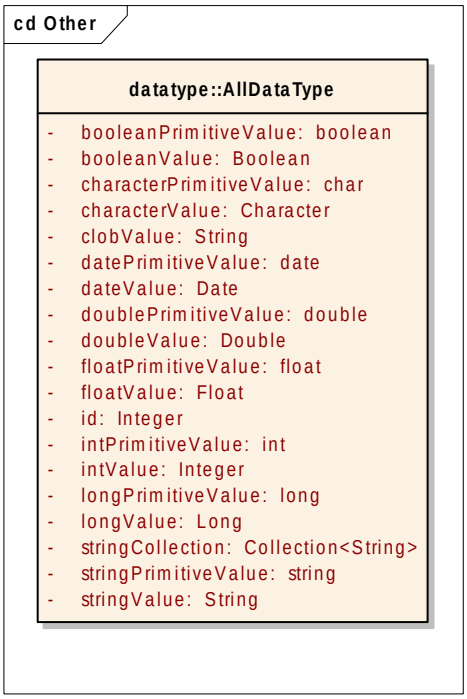


Figure 10-31 Logical model class diagram for data type

And the following Data Model class diagram from the SDK sample model:

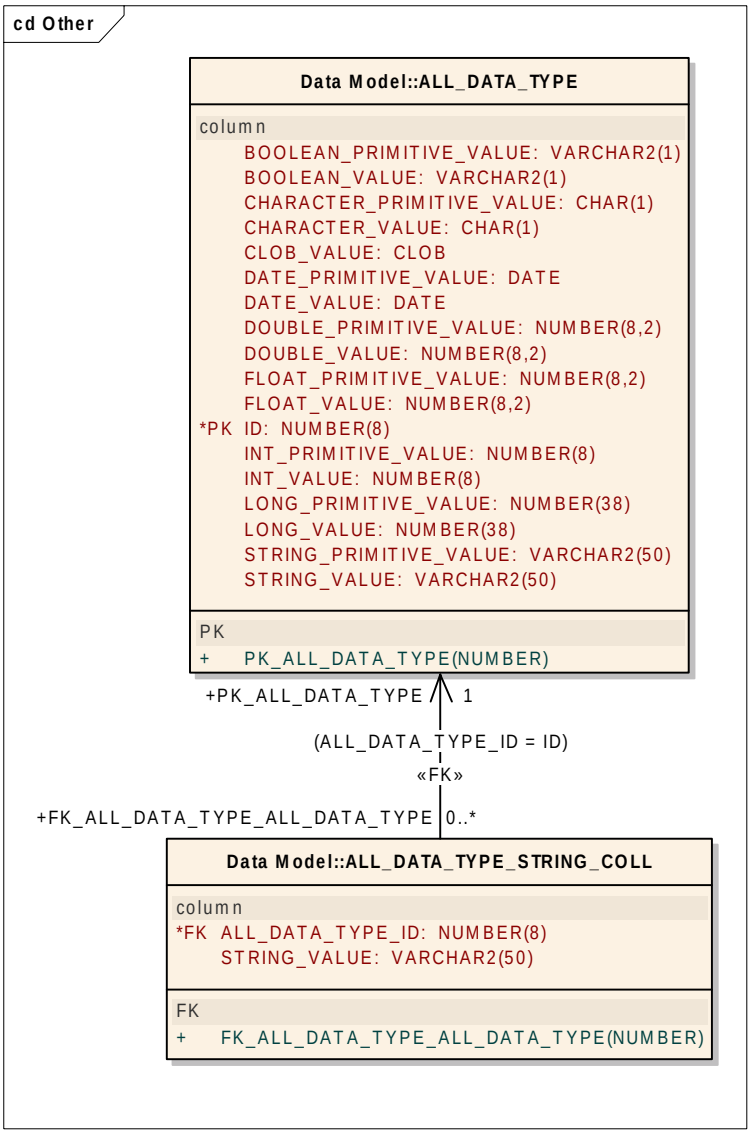


Figure 10-32 Data model class diagram for data type

An example *mapped-collection-table* Tag Value would be:

Data Model Class (Table)	Data Model Attribute (Column)	Tag Value (mapped-element)
ALL_DATA_TYPE_ STRING_COLL	STRING_VALUE	gov.nih.nci.cacoresdk.domain.other.datatype.AllDataType.stringCollection

Tag Value: NCI_GME_XML_NAMESPACE

Three Tag Values have been added to an Object Model for mapping purposes:

- One tag per project placed at the “Logical Model” package level. The default value is: "gme://{projectName}.{contextName}/{version}".
  Example: "gme://caMOD.caBIG/3.0".
- One tag per package placed at the Package level. The default value must be the full package path.
  Example: "gme://caMOD.caBIG/3.0/gov.nih.nci.camod.domain".
- One tag per class placed at the Class level. The default value must be the full package path.
  Example: "gme://caMOD.caBIG/3.0/gov.nih.nci.camod.domain".

This tag value effectively “overrides” the project namespace specified within the `deploy.properties` file when generating XSD and XML mapping artifacts.

Tag Value: NCI_GME_XML_ELEMENT

One Tag Value added to an Object Model for mapping purposes, placed at the Logical Model Class level. The value is the class name from the source model, for example, "Person". Used to effectively “rename” a class within the generated XSD and XML mapping artifacts.

Tag Value: NCI_GME_SOURCE_XML_LOC_REF

One Tag Value added to an Object Model for mapping purposes, placed on an Association link between two classes within a diagram.

Format is <<rolename>>/<<classname>>.

Example: “author/Author” or “bookCollection/BookCollection”.

Used to indicate the source class and the corresponding rolename by which it should be referenced.

NOTE: The terms “source” and “target” are relative to which association end is currently being processed. The SDK Code Generator does not care whether the NCI_GME_SOURCE_XML_LOC_REF or the corresponding NCI_GME_TARGET_XML_LOC_REF tag values are used. It looks for a match using the “other end”, or target, classname.

Tag Value: NCI_GME_TARGET_XML_LOC_REF

One Tag Value added to an Object Model for mapping purposes, placed on an Association link between two classes within a diagram.

Format is <<rolename>>/<<classname>>.

Example: “author/Author” or “bookCollection/BookCollection”.

Used to indicate the target class and the corresponding rolename by which it should be referenced.

NOTE: The terms “source” and “target” are relative to which association end is currently being processed. The SDK Code Generator does not care whether the

NCI_GME_SOURCE_XML_LOC_REF or the corresponding NCI_GME_TARGET_XML_LOC_REF tag value is used. It looks for a match using the "other end", or target, classname.

Tag Value: type

A Tag Value added to a Data Model class *Attribute* (column). The value specifies the DB column type. Valid values include (but are not limited to):

- CHAR
- CLOB
- NUMBER
- VARCHAR2

Several examples from SDK sample model include:

Data Model Class (Table)	Data Model Attribute (Column)	Tag Value (mapped-element)
CHARACTER_PRIMITIVE_KEY	ID	CHAR
CARD	IMAGE	CLOB
UNDERGRADUATE_STUDENT	STUDENT_ID	NUMBER
SHOES	COLOR	VARCHAR2

Exporting the UML Model to XMI (EA Only)

Before the SDK can process a UML model created within EA, the model needs to be exported to XMI and then copied to the *\models* directory within the SDK root folder.

This section only applies to EA because ArgoUML already stores projects in an XML format that the SDK Code Generator can understand and process.

To export an EA package to XMI:

1. In the EA Project Browser, select the **Logical View** package.

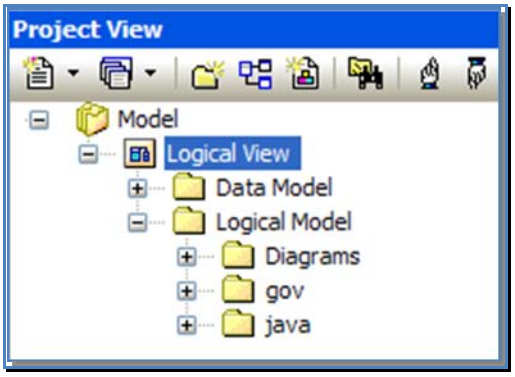


Figure 10-33 EA Logical View Package

2. Right click and select **Import/Export**, or from the main menu, select **Project > Import/Export**.
3. Select **Export Package to XMI**. The Export Package to XMI dialog appears.

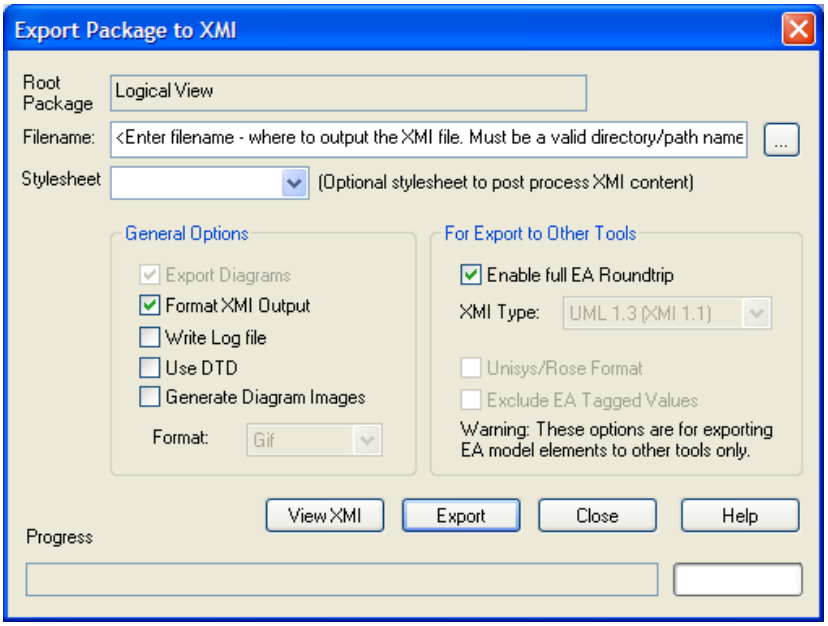


Figure 10-34 EA Exporting Package to XMI

4. Set the Export options as shown in Figure 10-34. The appropriate options are also outlined in Table 10-1 below.

Export Option	***Description***
Filename	Used to indicate where to output the XMI file. Enter a valid directory/path name. Also, make sure the file type suffix is .xmi. **NOTE:** The XMI file name and the value of the *MODEL_FILE* property within the `deploy.properties` file must match. Otherwise, a *File Not Found* error will be reported when trying to process the XMI file through the SDK Code Generator.
Stylesheet	Used to post-process XMI content before saving to file. Leave unselected.
Export Diagrams	Leave checked.
Use Unisys Rose Format	Used to indicate whether or not the Model should be exported in Rose UML 1.3, XMI 1.1 format. Leave unchecked.
Format XML output	Used to indicate whether or not to format output into readable XML (takes a few additional seconds at end of run). Leave checked.
Write log file	Used to indicate whether or not a log of export activity should be created (recommended). The log file will be saved in the same directory exported to. Optional. Leave checked if desired.
Use DTD	Used to indicate whether or not to use the UML1.3 DTD. Using this option will validate the correctness of the model and that no syntactical errors have occurred. Leave unchecked.
Exclude EA Tagged Values	Used to indicate whether or not EA specific information should be excluded from the export to other tools. The SDK now supports Full EA roundtrip. Leave unchecked.

Table 10-1 EA Export options

NOTE: The XMI export options that must be selected have changed since SDK 4.0. The options that must now be enabled are: Export Diagrams and Enable full EA Roundtrip

5. When finished, click **Export**.
6. Once the XMI file has been exported, copy it to the *\models* directory within the SDK root folder.

Importing XMI into the UML Model (EA Only)

The SDK now supports the processing of an XMI file that was exported using the *full EA roundtrip* option. Some organizations may have the need to modify the exported XMI file, perhaps to add Tag Values. As long as the XMI was exported using the *roundtrip* option, it can be synchronized with the UML model by importing it back into EA.

As with the XMI export, this section only applies to EA because ArgoUML already stores projects in an XML format that the SDK Code Generator can understand and process.

WARNING! The selection of the incorrect import options may corrupt the model file. Ensure that you back up the original model file prior to importing XMI back into the UML model.

To import an XMI package back into EA:

1. In the EA Project Browser, select the **Logical View** package.

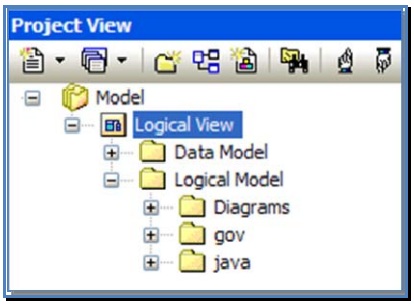


Figure 10-35 EA Logical View Package

2. Right click and select **Import/Export**, or from the main menu, select **Project > Import/Export**.
3. Select **Import Package from XMI**. The Import Package from XMI dialog appears.

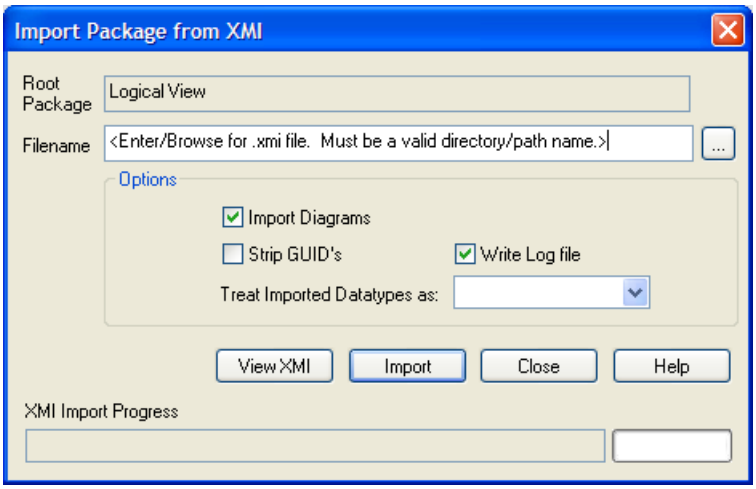


Figure 10-36 EA Import Package from XMI

4. Set the Import options as shown in Figure 10-36. The appropriate options are also outlined in Table 10-2 below.

Import Option	Description
Filename	Used to indicate where to import the XMI file. Enter a valid directory/path name.
Import Diagrams	Leave checked.
Strip GUIDS	Used to remove Universal Identifier information from the file on import. This permits the import of a package twice into the same model - the second import will require new GUIDS to avoid element collisions. Leave checked.
Treat Imported Datatypes as	Leave unselected.
Write log file	Used to indicate whether or not a log of export activity should be created (recommended). The log file will be saved in the same directory exported to. Optional. Leave checked if desired.

Table 10-2 EA Import options

5. When finished, click **Import**. A confirmation dialog appears.

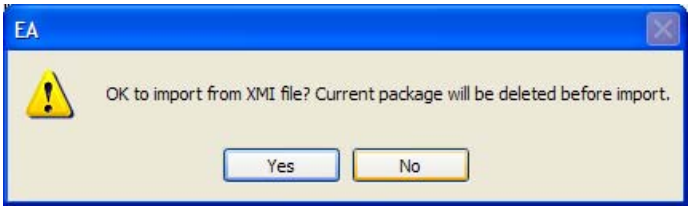


Figure 10-37 EA Confirm XMI File Import Dialog

6. Click **Yes**.

The XMI file is imported back into EA and the XMI and UML model are synchronized.

Chapter 11 Configuring and Running the SDK

Topics in this chapter include:

SDK Configuration Properties

The SDK Code Generator is configured, for the most part, by a single file, the `deploy.properties` file, which is located in the `/conf` folder in the SDK distribution.

The following table (Table 11-1) describes each of the properties (and their values) found within this file.

Property	*Default Value*	*Description*
PROJECT PROPERTIES		
PROJECT_NAME	example	Used in the creation/naming of the following items: • Output project directory folder name • Beans JAR file name • ORM JAR file name • Client JAR file name • WAR file name • Web Service Namespace • Documentation title in the generated API (JavaDocs) • Server URL context value SDK users should modify this property to reflect their own project name.
NAMESPACE_PREFIX	gme://caCORE.caCORE/3.2/	Used in the creation/naming of the following code generation artifacts: • Schemas (XSD's) • XML Marshalling and Unmarshalling Mapping files If XSDs are to be used for the caGrid, the value of the NAMESPACE_PREFIX is the same as the GME namespace value.
WEBSERVICE_NAME	${PROJECT_NAME}Service	The name of the Web Service.

Property	Default Value	Description
PROJECT SECURITY PROPERTIES		
ENABLE_SECURITY	false	Used to enable or disable security within the generated system during code generation. This applies to all of the SDK interfaces, including: Web Interface (GUI) Java API Interface (local and remote clients) Writable API Web Service Interface
ENABLE_INSTANCE_ LEVEL_SECURITY	false	Used to enable/disable CSM instance level security. Only relevant if the ENABLE_SECURITY property is set to 'true'
ENABLE_ATTRIBUTE_ LEVEL_SECURITY	false	Used to enable/disable attribute level security. Only relevant if the ENABLE_SECURITY property is set to 'true'
CSM_PROJECT_NAME	${PROJECT_NAME}	Used as a prefix when creating the CSM security configuration file name. CSM configuration should have the same application name configured. NOTE: The CSM_PROJECT_NAME value must match the project application name used when setting up security using CSM's UPT
CACHE_PROTECTION_ ELEMENTS	false	Indicates whether or not CSM Protection Elements should be cached

Property	Default Value	Description
WRITABLE API PROPERTIES		
ENABLE_WRITABLE_API_EXTENSION	false	If set to "true" or "yes", will change the application service interface and corresponding implementation to enable the Writable API
DATABASE_TYPE	oracle	A suffix appended to certain tag value keys (e.g., NCI_GENERATOR.<database-type>, NCI_GENERATOR_PROPERTY.<database-type>) added to the primary key columns within the UML model. The tag values are used by the Hibernate Mapping file transformer to generate primary key settings for a given class. The database-type suffix is necessary when supporting multiple databases through the same UML model. Only relevant if the Writable API extension is enabled.
IDENTITY_GENERATOR_TAG	<generator class="assigned"/>	If using system-wide primary key generator settings, the value for the primary key generator class. Only relevant if the Writable API extension is enabled.
CADSR_CONNECTION_URL	http://cadsrapi.nci.nih.gov/cadsrapi40	If set, will override the default connection provided in the caDSR application-config-client.xml file, located under /conf/codegen/validator. Used when generating Hibernate Validator annotations containing caDSR Permissible Value enumeration(s) for a given domain object attribute. Only relevant if the Writable API extension is enabled.
ENABLE_COMMON_LOGGING_MODULE	true	If set to "yes" or "true", will enable the Common Logging Module (CLM). Only valid if the Writable API extension is enabled.
CLM_PROJECT_NAME	${PROJECT_NAME}	Used to populated the CLM logging table (LOG_MESSAGE) application column. Only valid if CLM and the Writable API are both enabled.
APPLICATION SERVER PROPERTIES		
SERVER_TYPE	other	Used to include/exclude the log4j.jar file during the war file packaging. If set to ‘jboss’ will exclude log4j.jar from the war file, as the JBoss server already has its own instance of the log4j.jar file. Any other value will include the log4j.jar in the war file. Valid values are ‘jboss’ if deploying to a JBoss server, and ‘other’ if deploying to any other type of Servlet container such as Apache Tomcat.

Property	Default Value	Description
SERVER_URL	http://localhost:8080/ ${PROJECT_NAME}	The URL (including the application context) of the deployed application. Used as part of the URL that specifies the location of the deployed Web Service. I.e., the following pattern is used when undeploying the Web Service from the server: ${SERVER_URL}/services/${WEBSERVICE_NAME}Service
MODEL PROPERTIES		
MODEL_FILE	sdk.xmi	The name of the file which contains the object/data model be processed. SDK users should modify this property to reflect their own model file name. The file must be placed under the \models directory.
MODEL_FILE_TYPE	EA	The file type of the object/data model file to be processed. Valid values are 'EA' for Enterprise Architect files, and 'ARGO' for ArgoUML files .
LOGICAL_MODEL	Logical View.Logical Model	The logical model base (root) package/folder name containing the domain package(s) and class(es) to be processed by the Code Generator.
DATA_MODEL	Logical View.Data Model	The data model base (root) package/folder name containing the data model package(s) and class(es) to be processed by the Code Generator.
INCLUDE_PACKAGE	.*?domain.*	Specifies the regular expression (java.util.regex) pattern(s) of any packages within the object/data model that should be processed by the code generator. Separate patterns with a comma (',') as a delimiter.
EXCLUDE_PACKAGE		Specifies the regular expression (java.util.regex) pattern(s) of any fully qualified package names within the object/data model that should be ignored (not processed) by the code generator. Use a comma (',') as a delimiter to separate patterns. **NOTE:** All packages are first filtered/constrained by the INCLUDE_PACKAGE property value, and then further filtered by the EXCLUDE_PACKAGE value.
EXCLUDE_NAME		Specifies the regular expression (java.util.regex) pattern(s) of the fully qualified class name(s) within the object/data model that should be ignored (not processed) by the code generator. Use a comma (',') as a delimiter to separate patterns.

Property	Default Value	Description
EXCLUDE_NAMESPACE		Specifies the regular expression (java.util.regex) namespace pattern of the fully qualified package name(s) within the object/data model that should be ignored (not processed) by the code generator. Use a comma (',') as a delimiter to separate patterns.
NOTE: As of SDK 4.1, The INCLUDE and EXCLUDE properties above now use *java.util.regex* patterns. For details on creating regular expression patterns, see http://java.sun.com/j2se/1.4.2/docs/api/java/util/regex/Pattern.html. Also, patterns are matched against the fully qualified class name (or namespace in the case of the EXCLUDE_NAMESPACE property), so be sure to use patterns that take this into account. As an example, the INCLUDE_PACKAGE used to have a default value of 'domain'. It has now been changed to '.*?domain.*' so that it matches classes found in packages such as gov.nih.nci.cacoresdk.domain.inheritance.abstrakt.*.		
DATABASE CONNECTION PROPERTIES		
USE_JNDI_BASED _CONNECTION	false	Indicates whether or not a JNDI DB Connection should be used for the application database. If set to "true" or "yes", DB_JNDI_URL is used to obtain the connection and get data. If set to "no" then DB_DRIVER, DB_CONNECTION_URL, DB_USERNAME and DB_PASSWORD are used instead to initialize the connection and get data.
DB_JNDI_URL	java:/SDK	The DB JNDI URL value of the application database. This property is irrelevant/ignored if USE_JNDI_BASED_CONNECTION=no.
DB_CONNECTION_URL DB_USERNAME DB_PASSWORD		The application database connection properties. A sample DB_CONNECTION_URL value: jdbc:oracle:thin:@cbiodb30.nci.nih.gov :1521:CBTEST These values are purposely blank. SDK users should provide appropriate values for their database within the local.properties file located in the root folder of the SDK distribution.
DB_DIALECT	org.hibernate.dialect.OracleDialect	The Hibernate Database dialect to be used when connecting to the application database. Typical values include: org.hibernate.dialect.OracleDialect org.hibernate.dialect.MySQLDialect

Property	Default Value	Description
CSM SECURITY DATABASE CONNECTION PROPERTIES		
CSM_USE_JNDI_BASED_CONNECTION	${USE_JNDI_BASED_CONNECTION}	Indicates whether a JNDI DB connection should be used for the CSM database. If USE_JNDI_BASED_CONNECTION=true, then the DB_JNDI_URL property value is used to obtain the DB connection and retrieve data. By default, will use the same values as the application's USE_JNDI_BASED_CONNECTION.
CSM_DB_JNDI_URL	${DB_JNDI_URL}	The DB JNDI URL value for the CSM database. This property is irrelevant/ignored if CSM_USE_JNDI_BASED_CONNECTION=false. By default, will use the same value as the application's DB_JNDI_URL property.
CSM_DB_CONNECTION_URL CSM_DB_USERNAME CSM_DB_PASSWORD CSM_DB_DRIVER	${DB_CONNECTION_URL}, ${DB_USERNAME}, ${DB_PASSWORD}, ${DB_DRIVER}	The CSM database connection properties. A sample DB_CONNECTION_URL value: jdbc:oracle:thin:@cbiodb30.nci.nih.gov :1521:CBTEST These values are purposely blank. SDK users should provide appropriate values for their CSM database instance within the local.properties file located in the root folder of the SDK distribution. By default, will use the same values as the application's DB connection properties.
CSM_DB_DIALECT	${DB_DIALECT}	The Hibernate Database dialect used when connecting to the CSM database. Typical values include: org.hibernate.dialect.OracleDialect org.hibernate.dialect.MySQLDialect By default, will use the save value as the application's DB_ DIALECT property.
COMMON LOGGING MODULE (CLM) DATABASE CONNECTION PROPERTIES		
CLM_DB_CONNECTION_URL CLM_DB_USERNAME CLM_DB_PASSWORD CLM_DB_DRIVER	${DB_CONNECTION_URL}, ${DB_USERNAME}, ${DB_PASSWORD}, ${DB_DRIVER}	The Common Logging Module (CLM) database connection properties. A sample DB_CONNECTION_URL value: jdbc:oracle:thin:@cbiodb30.nci.nih.gov :1521:CBTEST These values are purposely blank. SDK users should provide appropriate values for their CLM database instance within the local.properties file located in the root folder of the SDK distribution. By default, will use the same values as the application's DB connection properties.

Property	Default Value	Description
CODE GENERATION OPTIONS The following properties are used to enable or disable code generation step(s). These properties accept values of either 'true' or 'false'. Setting the value to 'false' for a component disables the code generation of that component, while setting the value to 'true' enables it		
VALIDATE_LOGICAL _MODEL	true	Used to enable/disable the validation of the logical object model prior to code generation.
VALIDATE_MODEL _MAPPING	true	Used to enable/disable the validation of the logical object model to the data model mapping prior to code generation.
VALIDATE_GME_TAGS		
GENERATE_ HIBERNATE_MAPPING	true	Used to enable/disable the generation of the Hibernate Object-Relational Mapping files during code generation.
GENERATE_BEANS	true	Used to enable/disable the generation of the domain object beans (Java Beans) during code generation.
GENERATE_CASTOR_ MAPPING	true	Used to enable/disable the generation of the Castor XML marshalling and unmarshalling mapping files.
GENERATE_XSD	true	Used to enable/disable the generation of the XML Schemas (XSDs).
GENERATE_XSD_WITH_ GME_TAGS	false	
GENERATE_XSD_WITH_ PERMISSIBLE_VALUES	false	
GENERATE_WSDD	true	Used to enable/disable the generation of the Axis Web Service Deployment Descriptor (WSDD) file.
GENERATE_HIBERNATE_ VALIDATOR	false	
ADVANCED PROPERTIES		
CACHE_PATH	java.io.tmpdir	An advanced property used by ehcache to store its cache files on disk. A value of 'java.io.tmpdir' will create the cache files within the temporary directory. SDK users may choose to specify any absolute path instead for the cache files.
CAGRID AUTHENTICATION PROPERTIES		
CAGRID_AUTHENTICATIO N_SERVICE_URL	https://dorian.training.cagrid.org:8443/wsrf/services/cagrid/Dorian	URL for the authentication service to be used during authentication process using caGrid infrastructure
CAGRID_DORIAN_ SERVICE_URL	https://dorian.training.cagrid.org:8443/wsrf/services/cagrid/Dorian	URL for the Dorian service to be used during authentication process using caGrid infrastructure
SDK_GRID_LOGIN_ SERVICE_NAME	SDKGridLoginService	Name of the war file that performs the authentication using grid infrastructure

Property	Default Value	Description
SDK_GRID_LOGIN_ SERVICE_URL	http://localhost:8080/${SDK_ GRID_LOGIN_SERVICE_NA ME}	URL of the war file that performs the authentication using grid infrastructure
ENABLE_GRID_LOGIN_ MODULE	false	Specify if the caGrid based authentication is to be used
ENABLE_CSM_LOGIN_ MODULE	true	Specify if the CSM based authentication is to be used
CAGRID_LOGIN_ MODULE_NAME	grid	Name of the login module name in the JAAS configuration file

Table 11-1 SDK configuration properties

Generating the SDK System

Ant Build Script Targets

Apache Ant is a Java-based build tool used within the SDK to perform various build related tasks. See http://ant.apache.org/ for more information. The SDK provides an Ant script, `build.xml`, which is located in the root folder of the SDK distribution. This script contains targets for performing various system generation tasks, including building and packaging the system.

Typically speaking, most SDK users will only need to run the following two targets:

- **build-system:** Executes the SDK Code Generator using the properties configured within the `deploy.properties` file. See the above section, *SDK Configuration Properties* beginning on page 115 for more information.
- **clean-all:** Deletes all files and folders from the previous build process. It is strongly recommended that SDK users run this target prior to running the 'build-system' target.

NOTE: The SDK build process is configured by the properties found within the `deploy.properties` file as described in the section referenced above. Please review and update these properties to reflect your environment *prior* to generating the system.

For those interested in the remaining targets, the table below provides a complete list:

Ant Target	Description
build-system	Generates the SDK system using properties set within `\conf\deploy.properties`. This is the primary [default] target within the build script, and the one SDK users will most typically use when generating the system. SDK users are strongly recommended to run the 'clean-all' target prior to running the 'build-system' target.
clean	Cleans the main generated directories and files (\output) created following the execution of the build-system target.
clean-all	Cleans the generated directories and files of both the main and child projects. SDK users are strongly recommended to run the 'clean-all' target prior to running the 'build-system' target

Ant Target	Description
codegen	Runs the SDK Code Generator. The Generator is capable of selectively generating the system components. The following properties within the deploy.properties file control the behavior of the Code Generator: VALIDATE_LOGICAL_MODEL VALIDATE_MODEL_MAPPING VALIDATE_GME_TAGS GENERATE_HIBERNATE_MAPPING GENERATE_BEANS GENERATE_CASTOR_MAPPING GENERATE_XSD GENERATE_XSD_WITH_GME_TAGS GENERATE_XSD_WITH_PERMISSIBLE_VALUES GENERATE_WSDD GENERATE_HIBERNATE_VALIDATOR See *SDK Configuration Properties* on page 115 for more information. This target is run as part of the process run by the 'build-system'. SDK users should rarely, if ever need to invoke this target individually.
refresh-validator-permissible-values	Regenerates the Java beans by downloading the permissible values from caDSR

Table 11-2 Ant Script target descriptions

Selectively Generating Components

For those SDK users interested in only generating certain SDK components, the SDK Code generator is capable of selectively generating the following components:

- Hibernate O/R Mapping files
- Java Beans (domain Java objects)
- Castor XML Mapping files
- Schema (XSD) files
- Axis Web Service Deployment Descriptor (WSDD) file

To control which components are generated by the Code Generator, toggle the following respective properties within the `deploy.properties` file:

- GENERATE_HIBERNATE_MAPPING
- GENERATE_BEANS
- GENERATE_CASTOR_MAPPING
- GENERATE_XSD
- GENERATE_WSDD

Setting the value of a given property to 'true' causes the component to be generated; conversely, setting a property to 'false' causes the component to be ignored. See *SDK Configuration Properties* on page 115 for more information.

Overview of Generated Packages

During the code generation process, the SDK prepares four different packages, which are placed under a folder located at: `\output\<project_name>\package\`.

The following is a summary of the different packages created:

- **local-client** – This package contains the complete application that can be used in the local environment. It corresponds to the local-client interface of the SDK generated application. The generated binaries along with other required libraries are located in the folder `/lib folder/conf`, which contains the configuration file required by the local client to function. The folder `/src` contains a sample test program that can be used to test the generated local-client.
- **remote-client** - This package contains the remote client component of the generated application that can be used in the isolated environment. It corresponds to the remote-client interface of the SDK generated application. The generated binaries along with other required libraries are located in the folder `/lib`. The folder `/conf` contains the configuration file required by the local client to function in addition to the generated XSDs and castor mapping files. The folder `/src` contains a sample test program that can be used to test the generated remote-client. The sample programs test the following:
  - the Java API interface,
  - the XML marshalling and unmarshalling, and
  - the XML-HTTP interface's REST capabilities.
- **ws-client** - This package contains the environment to invoke the SDK generated web services with the Java based web services client. This package corresponds to the web service interface of the SDK generated application. The generated binaries along with other required libraries are located in the `/lib`. The folder `/src` contains a sample test program that can be used to test the generated client.
- **webapp** – This package contains two .war files. The file with name *<project_name>.war* is generated by the SDK and represents the server component of the SDK generated system. This file must be deployed to the application server before any of the client interfaces (except local-client) are accessed. The second file in the `webapp` folder with name *<sdk_grid_login_service_name>.war* only needs to be deployed if the grid authentication feature was enabled.
- **grid-jaas** – This package contains the JAAS based client, which is capable of connecting to caGrid to authenticate the user and retrieve user's credentials in appropriate format. This package can be used in conjunction with either the local or remote client so that the client can get access to the grid credentials.
- **server -** This package contains the files that must be copied to the server when the grid authentication feature is enabled. When copying the files to the server, the *server.xml* file in Tomcat needs to be merged with the same file inside the actual server directory.

Deploying the Generated System

The Ant build process packages the generated SDK system into a Web Archive (war) file for ease of deployment. This file is named *<project_name>.war*, and is located in the directory `\output\<project_name>\ package\webapp`. Typically, this file can be copied to the web server deployment folder and the system is automatically deployed when the web server is started.

NOTE: The generated SDK system has been tested on both JBoss v4.0.5 and Apache Tomcat v5.5.20 servers. The system should also work on other servers such as Weblogic or WebSphere; however no official testing has been done on those server types.

Deploying to JBoss

If the generated system *.war* file is to be deployed to a JBoss server instance, the SERVER_TYPE property in the `\conf\deploy.properties` file should be set to 'jboss'. This ensures that the `log4j.jar` file is excluded from the packaged war file during the build process. This is required as JBoss already has its own copy of the `log4j.jar` file, and will report an error if it finds another copy of this file within the *.war*.

To deploy to a JBoss server instance, copy the generated *.war* file to the directory `<JBoss installation directory>\server\default\deploy`, and then restart the server.

Deploying to Apache Tomcat

If the generated system war file is to be deployed to an Apache Tomcat server instance, the SERVER_TYPE property found in the file `\conf\deploy.properties` should be set to ''other". This ensures that the *log4j.jar* file is included in the packaged *.war* file.

To deploy to a Tomcat server instance, copy the generated war file to the directory `<Tomcat Installation Directory>\ webapps`, and then restart the server.

Note: When redeploying the system war file to Tomcat after an initial build, it is strongly recommended that the old war file and corresponding exploded directory be deleted before the new war file is copied to the deployment directory. This ensures that all files from the previous deployment are properly deleted.

Testing the caCORE SDK Generated System

The following sections discuss various tests for determining whether or not the SDK system has been successfully generated and deployed.

Testing the Web Interface

The SDK generated GUI consists of several web pages that facilitate access to domain data. The Home page can be accessed via the following URL pattern:

SDK Web Interface Test URL Pattern:

http://<server_name>:<server_port>/<project_name>

Thus, for the Home page of the sample SDK model, the URL might be http://localhost:8080/example. If the system has been successfully deployed, the page shown in Figure 11-1 below should appear.

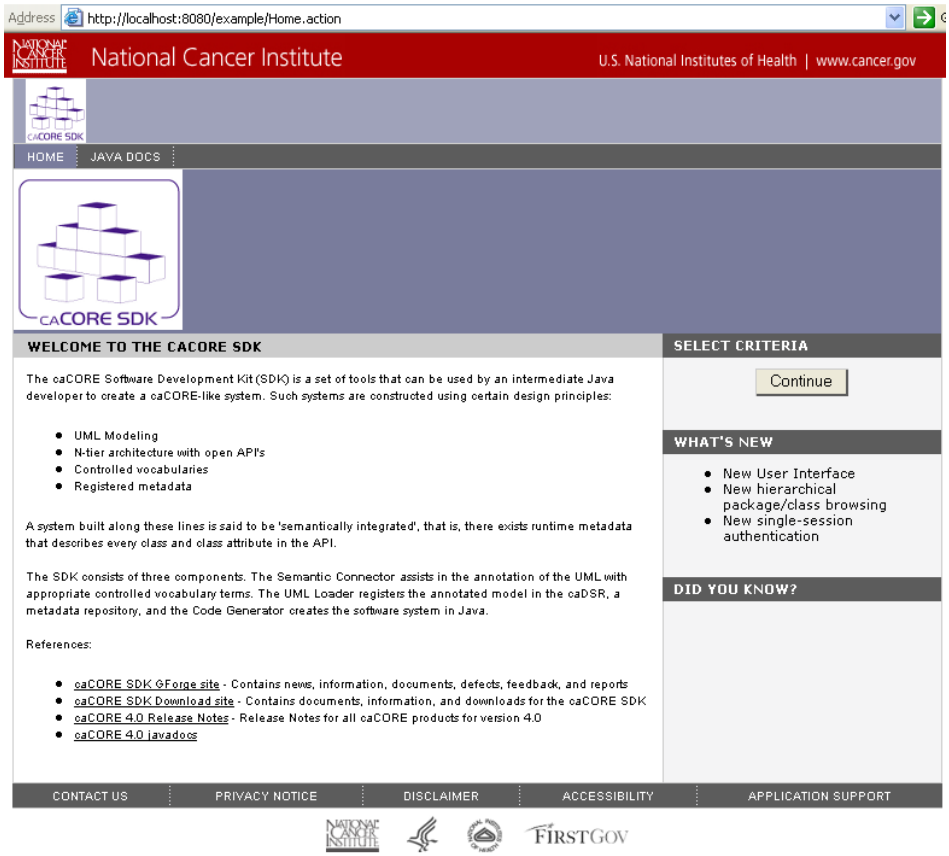


Figure 11-1 Web Interface test page

For more information, see *Accessing Data from a Web Browser* on page 47.

Testing the Java API

The program, *TestClient.java*, is provided with the SDK distribution for testing the Java API. This program is located in the folder:

`\output\<project_name>\package\remote-client\src\`.

To execute the program, run the default target of the Ant script, `build.xml`, located in the folder:

`\output\ <project_name>\package\remote-client\`.

NOTE: The generated system must be deployed to the server, and the server must be running before the test is invoked.

Figure 11-2 below shows the main test method algorithm.

```
public void testSearch() throws Exception
{
    //Application Service retrieval for secured system
    //ApplicationService appService = ApplicationServiceProvider.getApplicationService("userId","password");

    ApplicationService appService = ApplicationServiceProvider.getApplicationService();
    Collection<Class> classList = getClasses();
    for(Class klass:classList)
    {
        Object o = klass.newInstance();
        System.out.println("Searching for "+klass.getName());
        try
        {
            Collection results = appService.search(klass, o);
            for(Object obj : results)
            {
                printObject(obj, klass);
                break;
            }
        }catch(Exception e)
        {
            System.out.println(">>>"+e.getMessage());
        }
    }
}
```

Figure 11-2 Java API test algorithm

As shown, the program systematically loops through all the generated Java Bean classes and searches for each one without any filtering. It then takes the first qualifying record returned from the search and prints out its details to `stdout`, thus testing whether or not the Java API is working.

NOTE: The *TestClient.java* program is simply a client for testing the Java API. It provides only one example of how the SDK Application Service search API may be invoked. If desired, you can modify it to use a different method within the Application Service API, or to filter returned results by adding criteria data to the search object prior to the search.

See *Java API Interface* on page 54 for more information.

Testing the XML Utility

The program, *TestXMLClient.java*, is provided with the SDK distribution for testing the generated Castor XML Mapping and Schema (XSD) files. This program is located within the folder:

`\output\<project_name>\ package\remote-client\src\`.

To execute the program, run the runXML target of the Ant script, `build.xml`, located in the folder:

`\output\<project_name>\package\remote-client\`.

Note: The generated system must be deployed to the server, and the server must be running before the test is invoked.

Figure 11-3 below shows a portion of the main test method.

```
Marshaller marshaller = new caCOREMarshaller("xml-mapping.xml", false);
Unmarshaller unmarshaller = new caCOREUnmarshaller(
        "unmarshaller-xml-mapping.xml", false);
XMLUtility myUtil = new XMLUtility(marshaller, unmarshaller);
for (Class klass : classList) {
    Object o = klass.newInstance();
    System.out.println("Searching for " + klass.getName());
    try {
        Collection results = appService.search(klass, o);
        for (Object obj : results) {
            File myFile = new File("./output/" + klass.getName()
                    + "_test.xml");

            FileWriter myWriter = new FileWriter(myFile);
            myUtil.toXML(obj, myWriter);
            myWriter.close();
            printObject(obj, klass);
            DocumentBuilder parser = DocumentBuilderFactory
                    .newInstance().newDocumentBuilder();
            Document document = parser.parse(myFile);
            SchemaFactory factory = SchemaFactory
                    .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);

            try {
                System.out.println("Validating " + klass.getName()
                        + " against the schema......\n\n");
                Source schemaFile = new StreamSource(Thread
                        .currentThread().getContextClassLoader()
                        .getResourceAsStream(
                                klass.getPackage().getName() + ".xsd"));
                Schema schema = factory.newSchema(schemaFile);
                Validator validator = schema.newValidator();

                validator.validate(new DOMSource(document));
                System.out.println(klass.getName()
                        + " has been validated!!!\n\n");
            } catch (Exception e) {
                System.out
                        .println(klass.getName()
                                + " has failed validation!!!  Error reason is: \n\n"
                                + e.getMessage());
            }

            System.out.println("Un-marshalling " + klass.getName()
                    + " from " + myFile.getName() + "......\n\n");
            Object myObj = (Object) myUtil.fromXML(myFile);

            printObject(myObj, klass);
            break;
        }
    } catch (Exception e) {
        System.out.println("Exception caught: " + e.getMessage());
        e.printStackTrace();
    }
    // break;
```

Figure 11-3 XML Mapping and Schema Test Algorithm

As shown, the program systematically loops through all the generated Java Bean classes and searches for each one without any filtering. It then takes the first qualifying record returned from the search, and marshals (serializes) it to a file. Next, it reads the XML file back, parses the XML, and validates it against the generated schema. Finally, it unmarshals (deserializes) the XML back to the corresponding domain Java Bean object, thus testing that the generated XML Mapping and Schema files are working properly.

Note: The *TestXMLClient.java* program is simply a client for testing the XML Utility. It provides only one example of how the XML Utility marshalling/unmarshalling methods may be invoked. However, you can modify it to use a different method if so desired.

In addition, the same search algorithm used during the testing of the Java API is reused in this test program. See *Testing the Java API* on page 126 for more information.

Be advised that by its very nature XML processing can be memory intensive. The *TestXMLClient.java* program has been successfully run against the sample SDK model, which does not contain much data. When running the test program against a model with a lot of data, the memory specified by the *maxmemory=512m* attribute within the *runXML* target may need to be increased.

Testing the Web Service Interface

Testing the Web Service URL

A successful Web Service deployment can be tested by entering in a browser the Web Service URL that conforms to the following pattern:

SDK Web Service Test URL Pattern:
http://<server_name>:<server_port>/<project_name>/services/<project_name>Service

Thus, a successful Web Service deployment URL for the sample SDK model might be http://localhost:8080/example/services/exampleService.

Figure 11-4 below illustrates the result of a successful Web Service deployment test.

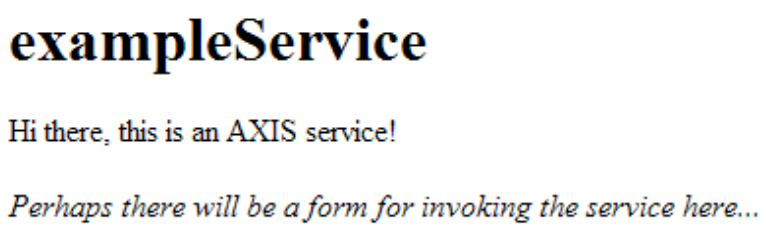


Figure 11-4 Web Service test page

NOTE: The SDK Web Service Deployment Descriptor (WSDD) is now packaged along with the rest of the SDK generated system, thus allowing for automatic deployment of the SDK Web Service whenever the system is deployed. Manual deployment of the Web Service is no longer required.

Obtaining the WSDL for Deployed Services: ?WSDL

As shown above, entering the Web Service URL in a browser causes a message to appear, indicating that the endpoint is an Axis service. However, if the suffix '*?wsdl*' is added to the end of the URL, Axis automatically generates a WSDL service description for the deployed service and returns it as XML in the browser. The URL pattern is shown below.

SDK Web Service WSDL Pattern:
http://<server_name>:<server_port>/<project_name>/services/<project_name>Service?wsdl

Figure 11-5 below illustrates a portion of the resulting XML that is generated after invoking the WSDL URL for the SDK sample Web Service.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions targetNamespace="http://localhost:8080/example/services/exampleService"
  xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://localhost:8080/example/services/exampleService"
  xmlns:intf="http://localhost:8080/example/services/exampleService"
  xmlns:tns1="urn:primarykey.other.domain.cacoresdk.nci.nih.gov"
  xmlns:tns10="urn:sametable.childwithassociation.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns11="urn:withjoin.unidirectional.manytoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns12="urn:sametable.parentwithassociation.inheritance.domain.cacoresdk.nci.nih.gov" xmlns:tns13="http://lang.java"
  xmlns:tns14="urn:bidirectional.onetomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns15="urn:multiplechild.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns16="urn:sametable.onechild.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns17="urn:withjoin.multipleassociation.onetoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns18="urn:twolevelinheritance.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns19="urn:childwithassociation.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns2="urn:parentwithassociation.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns20="urn:sametable.twolevelinheritance.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns21="urn:onechild.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns22="urn:datatype.other.domain.cacoresdk.nci.nih.gov"
  xmlns:tns23="urn:withjoin.bidirectional.onetomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns24="urn:multipleassociation.onetoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns25="urn:levelassociation.other.domain.cacoresdk.nci.nih.gov"
  xmlns:tns26="urn:bidirectional.manytomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns27="urn:unidirectional.manytomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns28="urn:bidirectional.onetoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns29="urn:withjoin.unidirectional.onetomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns3="urn:withjoin.bidirectional.onetoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns30="urn:selfassociation.onetomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns4="urn:unidirectional.onetomany.domain.cacoresdk.nci.nih.gov"
  xmlns:tns5="urn:unidirectional.onetoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns6="urn:withjoin.unidirectional.onetoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns7="urn:unidirectional.manytoone.domain.cacoresdk.nci.nih.gov"
  xmlns:tns8="urn:sametablerootlevel.twolevelinheritance.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:tns9="urn:sametable.multiplechild.inheritance.domain.cacoresdk.nci.nih.gov"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  - <!--
      WSDL created by Apache Axis version: 1.4
```

Figure 11-5 Obtaining the WSDL for Deployed Services: ?WSDL

Testing Web Services via the Client Program

The SDK distribution also provides the client program, *TestClient.java*, for testing the Web Service Interface. This program is located in the folder:

`\output\<project_name>\package\ws-client\src\`.

To execute the program, run the default run target of the Ant script, `build.xml`, located in the folder:

`\output\<project_name>\package\ws-client\`.

NOTE: The generated system must be deployed to the server and the server must be running before the Web Service test is invoked.

Figure 11-6 below shows a portion of the main test method.

```
QName searchClassQName = new QName("urn:"+getInversePackageName(klass), klass.getSimpleName());

call.setTargetEndpointAddress(new java.net.URL(url));
call.setOperationName(new QName("exampleService", "queryObject"));
call.addParameter("arg1", org.apache.axis.encoding.XMLType.XSD_STRING, ParameterMode.IN);
call.addParameter("arg2", searchClassQName, ParameterMode.IN);
call.setReturnType(org.apache.axis.encoding.XMLType.SOAP_ARRAY);

/*
//This block inserts the security headers in the service call
SOAPHeaderElement headerElement = new SOAPHeaderElement(call.getOperationName().getNamespaceURI(
headerElement.setPrefix("csm");
headerElement.setMustUnderstand(false);
SOAPElement usernameElement = headerElement.addChildElement("username");
usernameElement.addTextNode("userId");
SOAPElement passwordElement = headerElement.addChildElement("password");
passwordElement.addTextNode("password");
call.addHeader(headerElement);
*/

Object o = klass.newInstance();

System.out.println("Searching for "+klass.getName());
Object[] results = (Object[])call.invoke(new Object[] { klass.getName(), o });
```

Figure 11-6 Web Service test algorithm

The Web Service test program systematically loops through all the generated Java Bean classes and creates a Web Service *queryObject* call for each one. It then takes the first qualifying record returned from the call, and checks to see if the returned object has an association to another domain object. If it does, the program then proceeds to create and invoke a Web Service *getAssociation* call for it, thus testing multiple Web Service operations defined within the WSDL.

NOTE: The Web Service program *TestClient.java* is simply a client for testing the generated Web Service. It provides only one example of how the SDK Web Service messages may be created and invoked. However, you can modify it to use a different operation or algorithm if desired.

In addition, the same search algorithm used during the testing of the Java API's is re-used within this test program. See *Testing the Java API* on page 126 for more information. See also the *Web Service Interface* on page 69 for more information.

Chapter 12 Configuring Security

As described in the earlier section, security in the SDK is supported using various technologies, and the configurations that need to be made depend on the type and level of security you want to use. The decision process required for configuring security is based on the following options:

1. Authentication
    a. CSM based or
    b. caGrid based
2. Authorization Level
    a. Class level
    b. Instance level and/or
    c. Attribute level

In order to configure the security in SDK at the time of code generation, the following properties needs to be set in the `deploy.properties` file. Detailed descriptions for each of these properties can be found in the *SDK Configuration Properties* section of the previous chapter, beginning on page 115.

To enable or disable appropriate security levels:

- ENABLE_SECURITY
- ENABLE_INSTANCE_LEVEL_SECURITY
- ENABLE_ATTRIBUTE_LEVEL_SECURITY

Authorization policy settings

- CSM_PROJECT_NAME
- CACHE_PROTECTION_ELEMENTS
- CSM_USE_JNDI_BASED_CONNECTION
- CSM_DB_JNDI_URL
- CSM_DB_CONNECTION_URL
- CSM_DB_USERNAME
- CSM_DB_PASSWORD
- CSM_DB_DRIVER
- CSM_DB_DIALECT

CSM/caGrid based authentication settings

- ENABLE_GRID_LOGIN_MODULE
- ENABLE_CSM_LOGIN_MODULE

caGrid based authentication settings

- SERVER_TYPE
- SERVER_URL
- CAGRID_AUTHENTICATION_SERVICE_URL
- CAGRID_DORIAN_SERVICE_URL
- SDK_GRID_LOGIN_SERVICE_NAME
- SDK_GRID_LOGIN_SERVICE_URL
- CAGRID_LOGIN_MODULE_NAME

The remainder of this chapter provides more details on the configuration steps needed to enable the different types of authentication and authorization available for SDK generated systems.

Authentication Configuration

Applications dependent on JAAS-based login can configure their login procedure in several ways. (http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html)

SDK uses a JAAS-based login module for the authentication configuration. Depending on the authentication mode used (CSM or caGrid) you will need to configure a different login module in the JAAS configuration file.

Since the caCORE SDK uses Acegi and CSM as underlying security technologies, users of the SDK must perform configuration as recommended by those technologies. For an SDK generated local-client, users receive the database-based JAAS configuration prepared by the SDK. Users of the web application must configure the application server container.

Figure 12-1 below provides an example of how to configure JAAS-based CSM authentication in a JBoss server. See the CSM Technical Guide for more information on configuring JAAS-based security in different application servers and other configuration options.

```
<application-policy name="sdk">
    <authentication>
        <login-module
            code="gov.nih.nci.security.authentication.loginmodules.RDBMSLoginModule"
            flag="sufficient">
            <module-option name="driver">
                oracle.jdbc.driver.OracleDriver
            </module-option>
            <module-option name="url">
                jdbc:oracle:thin:@localhost:1521:TESTSCHEMA
            </module-option>
            <module-option name="user">TEST_USER</module-option>
            <module-option name="passwd">TEST_PASSWORD</module-option>
            <module-option name="query">
                SELECT * FROM csm_user WHERE login_name=? and password=?
            </module-option>
            <module-option name="encryption-enabled">YES</module-option>
        </login-module>
    </authentication>
</application-policy>
```

Figure 12-1 Configuring JAAS-based CSM authentication in JBoss server

SDK users must make an entry in the file <jboss-home>/server/default/conf/login-config.xml similar to the code snippet shown above. CSM reads the entry from the server's login configuration and performs authentication using the configuration.

When grid authentication is used, additional configuration on both the client and the server needs to be done. Figure 12-2 shows how grid authentication is done on a JBoss server.

```
<application-policy name = "grid">
    <authentication>
        <login-module
            code = "gov.nih.nci.system.security.authentication.cagrid.GridJAASLoginModule"
            flag = "required">
        </login-module>
    </authentication>
</application-policy>
```

Figure 12-2 Configuring JAAS-based caGrid authentication in JBoss server

The steps below provide instructions for using grid authentication.

Steps for installing certificates for the Training Grid environment

1. Follow steps listed at the following URL to download the caGrid 1.2 release: http://www.cagrid.org/wiki/CaGrid:Software:Release:1.2.
2. Execute `ant –Dtarget.grid=training-1.2 configure`
3. Check the *c:\Documents and Settings\<username>\.globus\certificates* directory. You should see some files with random names.

Steps for Configuring JBoss Server for Grid Authentication

1. Configure the grid authentication Login module as shown in Figure 12-2 above.
2. Copy all files from the `server/jboss` package to the JBoss application server.

3. Configure the machine to use the appropriate grid environment. Steps for configuring and using the training grid are provided on page 135 above.
4. Generate Host Certificates as mentioned in the following URL: http://www.cagrid.org/wiki/Dorian:1.2:Administrators_Guide:Requesting_Host_Credentials
   a. Start the GAARDS UI
   b. Log into the Grid using your grid user account (if you don't have an account, you must create one).
   c. From the **MyAccount** menu select **Request a Host Certificate**, this opens the Request a Host Certificate window.
   d. From the **Service URI** drop down select the URI of the Dorian you wish to request a host certificate from.
   e. In the **Host** text box, enter the **name** of the host for which you are requesting host credentials.
   f. Next, specify the directory on the file system where the host credentials should be written. This can be done using the **Browse** button.
   g. Click **Request Certificate**.
5. Edit `<jboss-home>\server\default\deploy\jbossweb-tomcat55.sar\server.xml`
   a. The Connector section will look something like following with exception of the certificate paths in the `server.xml` file:

```
<Connector className="org.globus.tomcat.coyote.net.HTTPSConnector" port="8443"
maxThreads="150" minSpareThreads="25" maxSpareThreads="75" autoFlush="true"
disableUploadTimeout="true" scheme="https" enableLookups="true"
acceptCount="10" debug="0"
protocolHandlerClassName="org.apache.coyote.http11.Http11Protocol"
socketFactory="org.globus.tomcat.catalina.net.BaseHTTPSServerSocketFactory"
cert="C:/Documents and Settings/<username>/.cagrid/certificates/My-cert.pem"
key="C:/Documents and Settings/<username>/.cagrid/certificates/My-key.pem"/>
```

   b. The Valve section will look something like this in the `server.xml` file

```
<Valve className="org.globus.tomcat.coyote.valves.HTTPSValve55"/>
```

Steps for Configuring Tomcat Server for Grid Authentication

1. Copy the contents of the SDK4\`<project-name>\output\package\server\tomcat` folder to the new tomcat installation.
   DO NOT copy *\example\output\package\server\tomcat\conf\server.xml*; **you need to merge it with existing** *server.xml*.
2. Generate Host Certificates as mentioned in step 4 of JBoss configuration
3. Open `<tomcat-home>\conf\server.xml`.
   a. The Connector section will look something like following with exception of the certificate paths in the `server.xml` file:

```
<Connector className="org.globus.tomcat.coyote.net.HTTPSConnector" port="8443"
maxThreads="150" minSpareThreads="25" maxSpareThreads="75" autoFlush="true"
disableUploadTimeout="true" scheme="https"
 enableLookups="true" acceptCount="10" debug="0"
cert="C:/Documents and Settings/<username>/.cagrid/certificates/My-cert.pem"
key="C:/Documents and Settings/<username>/.cagrid/certificates/My-key.pem"/>
```

b. The Valve section will look something like following in the `server.xml` file

```
<Valve className=" org.globus.tomcat.coyote.valves.HTTPSValve"/>
```

4. Edit `tomcat-home\bin\startup.bat`

   c. Locate `SET EXECUTABLE=` and enter remainder of the following sentence on the line after the located line:

   ```
   set JAVA_OPTS=%JAVA_OPTS% -
   Djava.security.auth.login.config=%CATALINA_HOME%/conf/login.co
   nfig
   ```

Steps for Local Client with Grid Authentication

1. Configure the machine to use the appropriate target grid environment. Steps for configuring and using the training grid are provided on page 135 above.
2. Merge the local client folder contents with grid-jaas folder contents as follows:
   a. Create a new folder for local-client-grid-authentication.
   b. Copy all contents of `output\<project-name>\package\grid-jaas` to <local-client-grid-authentication> folder.
   c. Copy contents of `output\<project-name>\package\local-client\lib` to `<local-client-grid-authentication>\lib` folder. Wherever there is a conflict, *keep* the files from the grid-jaas folder.
   d. Copy contents of `output\<project-name>\package\local-client\conf` to `<local-client-grid-authentication>\conf` folder *except for* the `login.config` file.
   e. Merge `Test.java` from <local-client-grid-authentication> and grid-jaas and put it in the <local-client-grid-authentication> folder.

      For example, use `grid-jaas\TestClient.java` to get *GlobusCredential* object. Pass this *GlobusCredential* object to `ApplicationServiceProvider.getApplicationService()` in `local-client\TestClient.java`.

   f. Merge `login.config` file from `<local-client-grid-authentication>\conf` and `grid-jaas\conf` and put it in the `<local-client-grid-authentication>\conf` folder.
3. Run `ant`.

Steps for Remote Client with Grid Authentication

1. Configure the machine to use the appropriate target grid environment. Steps for configuring and using the training grid are provided on page 135 above.
2. Merge the remote-client directory and grid-jaas in the same way as mentioned above for configuration of the local client. The only exception is that the `login.config` file does not need to be merged as there is only one copy.
3. Make sure the server is running.
4. Run `ant`.

Authorization Configuration

The caCORE SDK uses CSM to maintain the authorization configuration. In order to use CSM, a detailed configuration of the CSM needs to be done. The steps for configuring CSM-based authorization include:

1. Setup a CSM database schema for the application being generated.
2. Create a new application in the CSM schema using the User Provisioning Tool (UPT). The name of the application is same as the CSM_APPLICATION_NAME configured in the `deploy.properties` file.
3. Create the user accounts to be used.
4. Create the protection elements needed for different authorization levels.
5. Assign access privileges to the user accounts on the created protection elements.

NOTE: If you are planning to use instance level security, you are required to put CSM tables on the same database schema where the tables for the domain classes reside. See the CSM Technical Guide for more information on installing CSM on a particular database and using the UPT for configuring the security schema.

More information about steps 1 through 3 can be found in the CSM Technical Guide. Step 4 requires creating protection elements in CSM. These elements follow appropriate naming conventions.

Configuring CSM for Class Level Security

As shown in Figure 12-3 below, using class level security requires the existence of a protection element with the same object ID and name as the fully qualified name of the class for each of the domain objects in the generated system. Since class level security is always enforced, any user who does not have access rights on such a protection element will not be allowed to query data from that class.

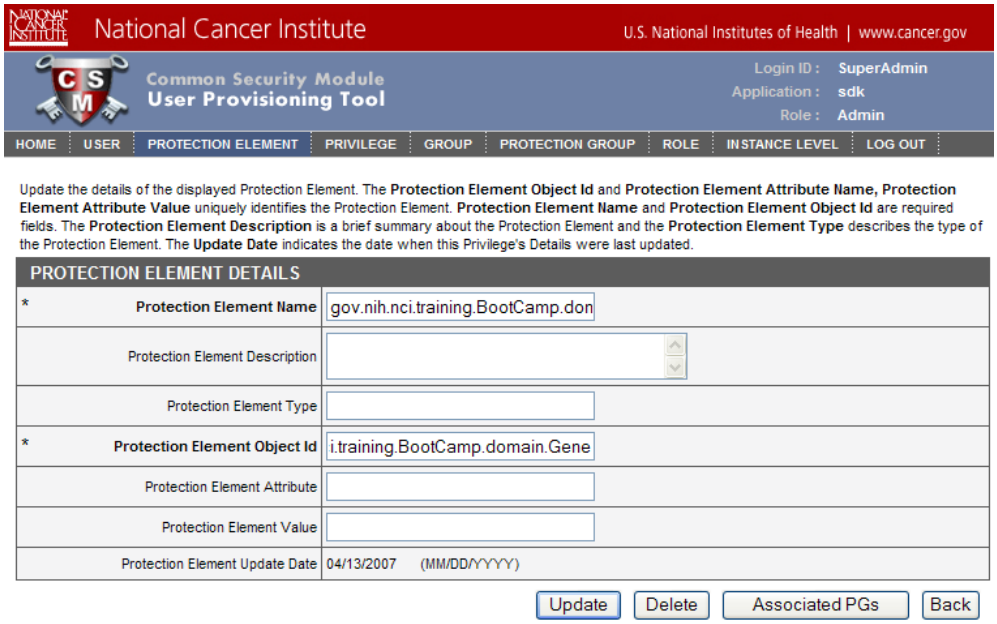


Figure 12-3 CSM UPT Screen indicating the Protection Element to be created for class level security

Configuring CSM for Attribute Level Security

As shown in Figure 12-4 below, implementing attribute level security requires the existence of a protection element with the same object ID as the fully qualified name of the class and with an attribute name the same as the name of the attribute in the class. Such protection elements must be created for each of the attributes in all the domain objects in the generated system. When attribute level security is enabled, any user who does not have access rights on such a protection element will receive nullified attributes when querying data from that class.

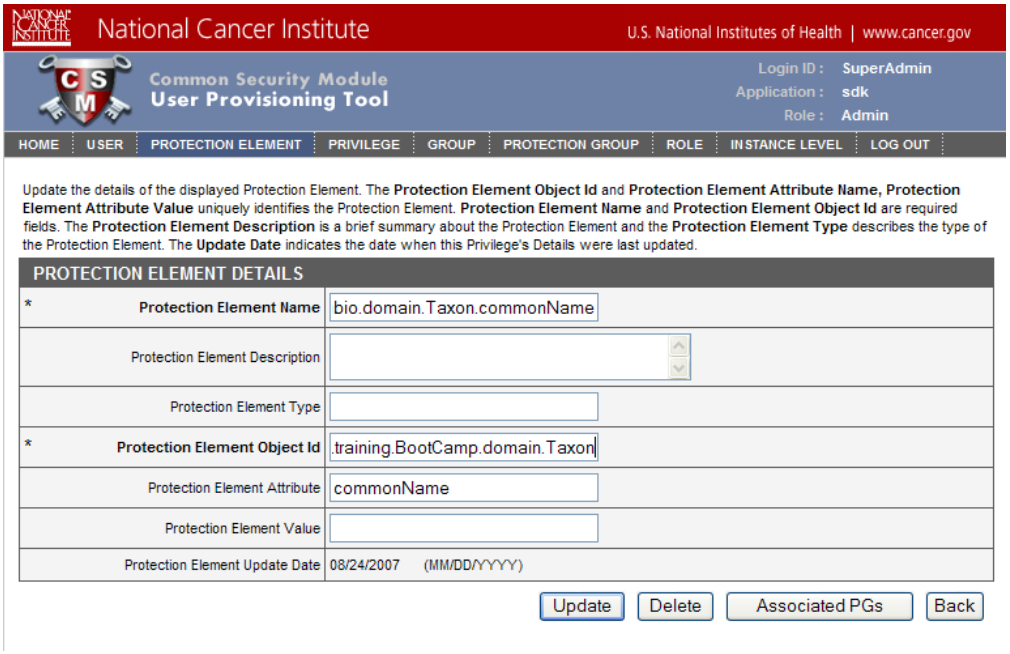


Figure 12-4 CSM UPT screen indicating the Protection Element to be created for attribute level security

Configuring CSM for Instance Level Security

When instance level security is enabled, the CSM UPT tool can be used to create the security filters. In order to create the security filters, you must upload two different JAR files through the CSM UPT.

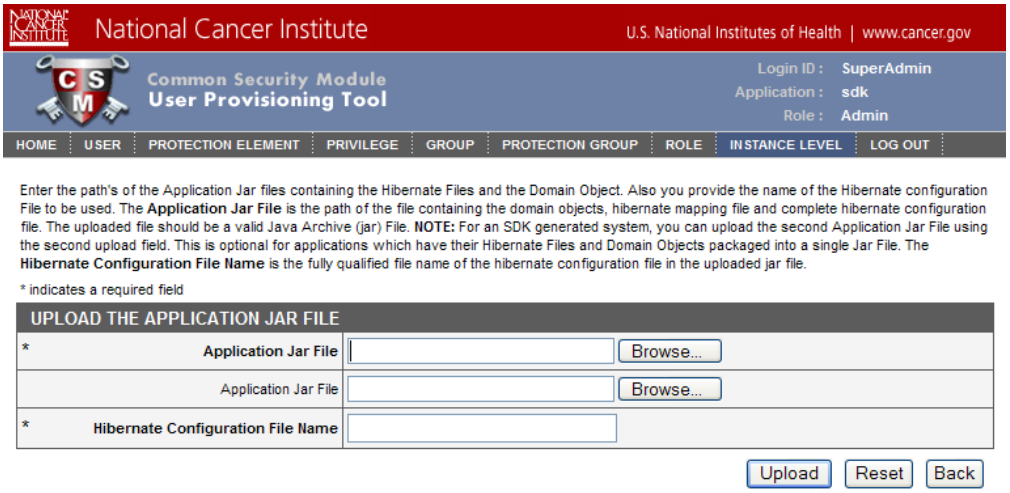


Figure 12-5 CSM UPT screen indicating the instance level security configuration tab

The JAR files are located in the `local-client/lib` folder of the caCORE SDK and their names are `<project_name>-beans.jar` and `<project_name>-orm.jar`. The CSM UPT also requires users to specify the name of the Hibernate configuration file, which in the case of an SDK generated application is `hibernate.cfg.xml`.

Once these files are uploaded using the “Instance Level” tab in the UPT, the UPT guides users to create the filters. More details about how to create the filters can be found in the CSM Technical Guide.

Once the filters are created, the query made through the SDK generated system will have additional conditions in the where clause which will limit the data coming back from the database using the established filter conditions.

Appendix A Troubleshooting

The following questions and scenarios have been reported by users and may be helpful in troubleshooting a problem when setting up the SDK.

1. ***I tried to use the SDK during code generation but I am getting just the exceptions and not error messages***

   Getting just the exceptions indicates that the SDK code generator did not initialize due to either invalid settings in the deploy.properties or an invalid UML model file. The UML model file can be considered invalid if it is not developed per the specification of the SDK or it is not exported as specified by the SDK.

2. ***I tried to generate an application with the SDK but I received validation errors. How do I make sure that model that I have created runs through the code generator?***

   The validation error messages generated from the SDK indicates specific error conditions under which the SDK cannot generate the code. Fixing the UML model and executing the code generator will solve the problem.

3. ***When running the generated application (.war file) under JBoss I am getting a Log4J exception.***

   SDK by default includes the log4j.jar and commons-logging.jar file in the generated .war file's lib directory. The JBoss server requires both of these files to be excluded from the .war file before deployment. A developer using the SDK can either remove these two jar files from the .war file before deployment or they can specify SERVER_TYPE=jboss in the deploy.properties file and regenerate the system. Specifying a server type as jboss during code generation will exclude the unnecessary jar files from being packaged in the .war file.

4. ***I successfully generated the application with the SDK. However, when running the application, I am getting database connection errors.***

   While generating the application with the SDK, the database connection parameters must be specified in the deploy.properties file. If these settings are incorrect, the SDK cannot fetch the data from the database. Make sure that the database settings are valid and the database server is running.

5. ***When I try to query the generated system, queries for some of the objects are running very slow.***

   There can be many different problems associated with slow searches. The primary problem is with the missing indexes on the primary key filed, foreign key field, or search key field. Creating these indexes should stop the database from performing full table scans and improve performance. *Appendix B* includes information on optimizing the performance of the Java API.

Appendix B Performance Tuning the Java API

The SDK generated Java API provides the ability to create a data service in a small amount of time. Because the SDK is simply a tool to generate the API, it cannot understand all the use cases for a user's application and hence cannot provide a comprehensive solution to requirements for all users. The SDK development team and many of the SDK users have encountered problems in this respect and have discovered several solutions to improve performance. This chapter includes some of the solutions discovered by these users.

Topics in this chapter include:

Database Indexes

Problem: Missing or corrupt indexes can explain performance problems for most queries. Most database modeling tools provide an option to create indexes for the primary key and foreign keys; however, the database indexes have been found to be missing or corrupted due to a variety of reasons including batch data load and recreation of the records.

Solution: Fixing the indexes should improve the performance of the queries. Proper indexes on the primary and foreign key columns will definitely improve performance for the database table joins. The user may have to create additional indexes for the columns that are more likely to be hit from the end user search.

Fine Tuning the Page Size

Problem: An SDK user can choose the page size for the SDK generated system at the time they generate the system. There are two kinds of pages for the generated system. The first is for the maximum number of records (rowCounter) that can be displayed to the user of the web interface. The second is the maximum number of records (resultCountPerQuery) that can be fetched by the Java API per call.

Solution: Both of these properties can be altered in the file `application-config.xml`, which is located in the SDK distribution folder `/conf/system/web/WEB-INF/classes`.

By default, the maximum number of records shown to the user of the web application is set to 200 and the maximum number of records that can be fetched by the Java API in one call to the server is set to 1000. Based on the nature of the underlying data, the developer of the application can choose the appropriate page size.

Hibernate Query Language (HQL)

SDK generated queries from SDK's Nested Search Criteria and SDK's CQL Search Criteria result in fetching the complete domain object from the database. At the same time, the database queries generated by the SDK specific search criteria can result in poor performance. A user of the SDK has the option to use the HQL queries to fetch the domain objects from the data service. The user can choose to retrieve selected attributes of the domain object but not the complete object by writing a more granular HQL query.

Appendix C Planned Features for Future Releases

The SDK development team constantly strives to improve the experience of using the SDK by providing new features and enhancing existing features. During the course of development for the current release, the team has come across many new features that will be considered for development immediately following the release of the current version. The following is a short summary of some of the major features under consideration.

Search Engine/Free Style search – The users of the SDK generated system can formulate a query by constructing an example. However in this approach, the user has to know which attribute he is searching for. The SDK team is planning to provide search engine in the generated system which will allow users to query using free text and retrieve all the domain objects which matches the respective criteria

GUI for installation and build process – The current SDK build process involves executing the ANT scripts to generate code with the SDK code generation module and preparing the packages for deployment and release. Although this process is geared towards novice users, many users find it difficult to use the command line script execution. A new tool is under consideration for development that will allow users to control the execution of code generation process from a graphical interface.

Robust user interface – The current user interface for the web application is a major improvement over the user interface provided by the previous release. The current version of the interface is based on the NCICB UI templates and has better integration of security then the previous version. This user interface will be expanded to provide additional features like:

- Complex Query By Example (QBE) input forms
- In line documentation for the UML class and attributes in the domain class browser
- Displaying UML diagrams in the domain class browser
- Allow editing of the records

Appendix D Example Model and Mapping

The caCORE SDK release package contains an example model which can be used by the user as a reference to model a particular scenario for their system. The example model is available in the `/models` directory of the release package. The example model is available for both Enterprise Architect (*SDKTestModel.eap*) and ArgoUML (*sdk.uml*). Users can refer to these models, which are organized in a self explanatory fashion.

The current version of the example model includes the following scenarios:

Attribute Types			
	Primary Key	**Simple Data type**	**Collection data type**
String	Yes	Yes	Yes
Integer	Yes	Yes	Yes
Double	Yes	Yes	Yes
Boolean	No	Yes	Yes
Float	Yes	Yes	Yes
Short	Yes	Yes	Yes
Long	Yes	Yes	Yes
Byte	Yes	Yes	Yes
Character	Yes	Yes	Yes
Date	Not Supported by SDK	Yes	Not Supported by SDK
String (CLOB)	Not Supported by SDK	Yes	Not Supported by SDK

Association Mapping				
	Unidirectional	**Bidirectional**	**Unidirectional with Join table**	**Bidirectional with Join table**
One to One	Yes	Yes	Yes	Yes
One to Many	Yes	Yes	Yes	Yes
Many to One	Yes	Yes	Yes	Yes
Many to Many	Yes	Yes	Yes	Yes
Self Association	Yes	Not Supported by SDK	No	Not Supported by SDK
Multiple Associations	Yes	Yes	Yes	Yes

Inheritance Mapping
Table per class
Table per hierarchy
Table per hierarchy with separate table for one of the child classes
Implicit Inheritance
Abstract Classes

Interface Mapping
Multilevel Interface Inheritance
Class Interface Realization

Other Mappings
Datatypes
Different Package
Hibernate Annotated Validation

Glossary

The following table contains a list of terms used in this document, with accompanying definitions.

Term	*Definition*
Acegi	Acegi is a security framework that provides a powerful, flexible security solution for enterprise software, with a particular emphasis on applications that use the Spring Framework. Acegi Security provides the SDK with comprehensive authentication, authorization, instance-based access control, channel security, and human user detection capabilities. See http://www.acegisecurity.org/ for more information.
Ant	Apache Ant is a Java-based build tool used within the SDK to perform various build related tasks. See the section on *Ant Build Script Targets* beginning on page 122 for more information on how Ant is used within the SDK. See http://ant.apache.org/ for more information on Ant itself.
Castor	Castor is an Open Source data-binding framework for Java, and facilitates conversion between Java Beans, XML documents and relational tables. Castor provides Java-to-XML binding, Java-to-SQL persistence, and more. See http://www.castor.org/ for more information.
Ehcache	Ehcache is a simple, fast and thread safe cache for Java that provides memory and disk stores and distributed operation for clusters. The SDK uses ehcache in conjunction with Hibernate. See http://sourceforge.net/projects/ehcache for more information.
QBE	Query by Example (QBE) is a database query language for relational databases. It was devised by Moshé M. Zloof at IBM Research during the mid 1970s, in parallel to the development of SQL. It is the first graphical query language, using visual tables where the user would enter commands, example elements and conditions. See http://en.wikipedia.org/wiki/Query_by_Example for more information.
Hibernate	Hibernate is an object-relational mapping (ORM) solution for the Java language, and provides an easy to use framework for mapping an object-oriented domain model to a traditional relational database. Its purpose is to relieve the developer from a significant amount of relational data persistence-related programming tasks. See http://www.hibernate.org/ for more information.
HQL	Hibernate Query Language (HQL) is a powerful query language that looks similar to SQL. Though the syntax is SQL-like, HQL is fully object-oriented, and understands concepts like inheritance, polymorphism and association. See http://www.hibernate.org/hib_docs/v3/reference/en/html/queryhql.html for more information.
Marshaling	The process of producing an XML document from Java Beans; i.e., the process of serializing Java Beans to XML.
ORM	An acronym for Object-Relational Mapping, a programming technique for converting data between incompatible type systems in databases and Object-oriented programming languages. This creates, in effect, a "virtual object database" which can be used from within the programming language. See http://en.wikipedia.org/wiki/Object-relational_mapping for more information. Hibernate implements this technique within the SDK.

Term	Definition
REST	"Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The term was introduced in the doctoral dissertation of Roy Fielding in 2000,[1] one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification, and has come into widespread use in the networking community. "REST strictly refers to a collection of network architecture principles that outline how resources are defined and addressed. The term is often used in a looser sense to describe any simple interface that transmits domain-specific data over HTTP without an additional messaging layer such as SOAP or session tracking via HTTP cookies. These two meanings can conflict as well as overlap. It is possible to design any large software system in accordance with Fielding's REST architectural style without using the HTTP protocol and without interacting with the world wide web. It is also possible to design simple XML+HTTP interfaces that do not conform to REST principles, and instead follow a Remote Procedure Call model. The two different uses of the term "REST" cause some confusion in technical discussions. See http://en.wikipedia.org/wiki/REST for more information.
Unmarshalling	The process of populating a generated class object from a corresponding XML document; i.e., the process of deserializing XML to Java Beans.
WSDD	An acronym for Web Service Deployment Descriptor, which can be used to specify resources that should be exposed as Web Services. See http://ws.apache.org/axis/java/user-guide.html#CustomDeploymentIntroducingWSDD for more information.
WSDL	An acronym for Web Services Definition Language, which is an XML-based language that provides a model for describing Web services. See http://www.w3.org/TR/wsdl.html or http://en.wikipedia.org/wiki/WSDL for more information.

Index

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X