caCORE SDK 4.0
Developer's Guide

Center for Bioinformatics

October 10, 2007

Credits and Resources

caCORE SDK Contributors			
SDK Development Team	**Other Development Teams**	**Guide**	**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]
	Konrad Rokicki [2]	Wendy Erickson-Hirons [5]	Charles Griffin [1]
	Ye Wu [2]		
	Christophe Ludet [3]		
[1] Ekagra Software Technologies	[2] Science Applications International Corporation (SAIC)	[3] Oracle Corporation	[4] National Cancer Institute Center for Bioinformatics
[5] Northern Taiga Ventures, Inc.			

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 4.0 release. It may be updated between releases if errors or omissions are found. The current document refers to the 4.0 version of caCORE SDK, released in October 2007 by the NCICB.

Table of Contents

Chapter 1 Using This Guide

This chapter introduces you to the caCORE SDK 4.0 Developer's Guide.

Topics in this chapter include:

Intended Audience

The *caCORE Software Development Kit 4.0 Developer's Guide* (SDK Guide) is the companion documentation to the caCORE (cancer Common Ontologic Representation Environment [http://ncicb-dev.nci.nih.gov/infrastructure/cacore_overview]) Software Development Kit (SDK). The SDK aids programmers with some life science background who are interested in using or extending the capabilities of caCORE. The caCORE SDK is a set of development resources that allows you to create, compile, and run caCORE-like software.

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 4.0 Developer's Guide contains the following chapters:

- *Chapter 1 Using This Guide* - This chapter provides an introduction to this developer's guide.
- *Chapter 2 Overview* - This chapter provides an overview of caCORE SDK 4.0, describes new features of the 4.0 release, and provides instructions for obtaining the release.
- *Chapter 3 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 4 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 5 System Usage* - This chapter provides examples to access the generated system's client interfaces by a client application or a user.
- *Chapter 6 System Usage for a Secured System* - This chapter describes how to use the SDK generated runtime system when security is enabled.
- *Chapter 7 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.
- *Chapter 8 Utilities* - This chapter describes a class that can be used to serialize and deserialize generated Java Beans to XML and back again.
- *Chapter 9 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 10 Configuring and Running the SDK* - This chapter describes how to configure the SDK Code Generator and generate the SDK 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 Planned Features for Future Releases* - This appendix contains a short summary of some of the major features under consideration for the next release.
- *Appendix C 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.

Document Text Conventions

The following table shows various typefaces to differentiate between regular text and menu commands, keyboard keys, tool bar buttons, dialog box options, and text that you type. The conventions illustrate how text is represented in this guide.

Convention	***Description***
Notes	**Notes:** Notes are enclosed for emphasis
Bold	Bold type is used for emphasis, buttons, or tabs to select on windows, and names of dialog boxes.
TEXT IN SMALL CAPS	TEXT IN SMALL CAPS is used for keyboard keys that you press (for example, ALT+F4)

This chapter introduces you to the caCORE SDK 4.0 Developer's Guide.

Topics in this chapter include:

Intended Audience

The *caCORE Software Development Kit 4.0 Developer's Guide* (SDK Guide) is the companion documentation to the caCORE (cancer Common Ontologic Representation Environment [http://ncicb-dev.nci.nih.gov/infrastructure/cacore_overview]) Software Development Kit (SDK). The SDK aids programmers with some life science background who are interested in using or extending the capabilities of caCORE. The caCORE SDK is a set of development resources that allows you to create, compile, and run caCORE-like software.

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.

- HJava ProgrammingH
- HEnterprise Architect Online Manual
- HArgoUML Online Manual
- HHibernateH

Convention	***Description***
Text in italics	Italics are used to reference other documents, sections, figures, and tables.
`Special typestyle`	Special typestyle is used for filenames, directory names, commands, file listings, and anything that would appear in a Java program, such as methods, variables, and classes.
`Bold italics typestyle`	Bold italics is used for information the user needs to enter
{ }	Curly brackets are used for replaceable items (for example, replace {home directory} with its proper value such as C:\caadapter).

Figure 1-1 Document text conventions

Chapter 2 Overview

This chapter provides an overview of caCORE SDK 4.0, describes new features of the 4.0 release, and provides instructions for obtaining the release.

Topics in this chapter include:

Introduction

NCICB provides biomedical informatics support and integration capabilities to the cancer research community. NCICB has created 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 NCICB'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. The 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 (Figure 2-1). 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. *Chapter 3* describes the architecture of the code generation module and an overview of the artifacts that the caCORE SDK generates. *Chapter 4* provides an overview of the architecture of the runtime system and describes the variety of ways it can deliver the data to the client.

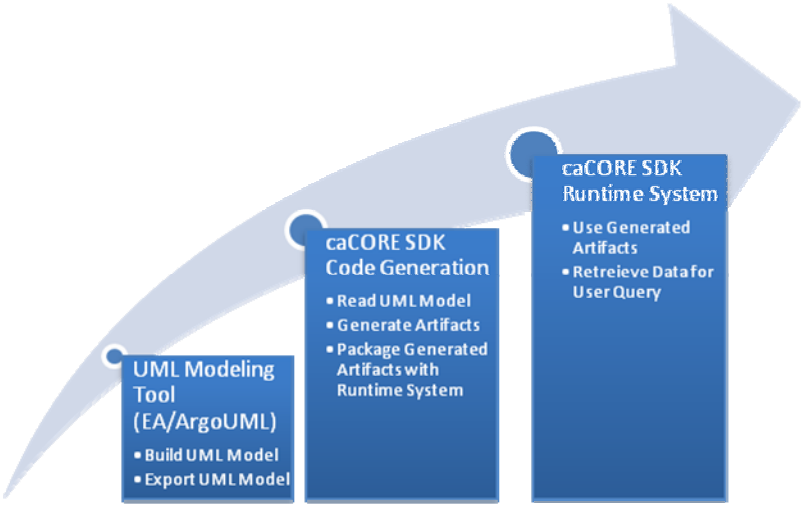


Figure 2-1 SDK System Generation Process

caCORE SDK Users

There are basically two types of caCORE SDK users that can be categorized 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 will primarily focus on preparing the UML model and running it through the caCORE SDK with appropriate settings to generate the runtime system. Users of the runtime system will primarily focus on writing queries against the runtime system to retrieve the data from the data source. *Chapter 3* provides an overview of how the code generation module is used. *Chapter 4* provides an overview of runtime system usage scenarios.

SDK within the caCORE Environment

The caCORE SDK can be utilized to quickly generate a system from the caBIG's silver-level compatible UML model. For more information on caBIG compatible system levels, see the caBIG web site at http://www.fkhealth.net/serviceweb/clients/nci/cabig_website/index.asp. However, the use of the SDK is not limited to generating a system from silver-level compatible models. The SDK can be used outside the caCORE environment to create a system that is generated from a UML model and 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 the caCORE SDK application development process can be found in *Chapter 10*.

Benefits of Using the caCORE SDK

Users of the caCORE SDK are benefited in numerous ways. The primary benefit of using 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's 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 also allows queries to be written in various ways including 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 4.0

caCORE SDK 4.0 is a major release with many new features available. Some of the newly developed features are to strengthen the infrastructure and others are to support new requirements. The purpose of this section is to highlight the major functionality and performance enhancements and improvements introduced in the caCORE SDK 4.0 release.

Code Generation

The architecture and the core of the code generation module of the caCORE SDK 4.0 has been completely rewritten. Entire code generation framework now runs from a single configuration file based on the Spring Framework as opposed to 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 the caCORE SDK used to support only Enterprise Architect as a tool 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 has significantly improved the performance of the code generation module. Average users should notice the system generation process to be

completed in approximately 15% of time of what it used to take with the previous releases of SDK.

- **Support for Validators**

  The caCORE SDK code generator now has support for validators. Validators serves the purpose of validating the object model and object relational mapping information before the code generator starts. These validators provide descriptive messages to users, which allow users to quickly identify the root cause of the code generation failure.

- **Reduced and Improved Generated Artifacts**

  The artifacts generated by caCORE SDK are completely redesigned to suit the needs of the newly redesigned runtime system. Artifacts generated by the previous SDK release were not reusable outside of it due to certain dependencies on it. However, artifacts generated by SDK 4.0 can be reused anywhere. For example, Java beans generated by the SDK had getter methods to connect to the server and were not simple POJOs, which in SDK 4.0 are simple POJO beans. The following table (Table 2-1) lists all of the artifacts that have changed since the previous release.

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

Artifact	SDK 4.0 Includes	SDK 3.2.x Included
Castor mapping files (xml-mapping.xml, xml-unmapping.xml)	yes	yes
Web service deployment descriptor (server-config.wsdd)	yes	yes

Table 2-1 SDK 4.0 artifacts

- **Additional UML Features Supported**

  caCORE SDK 4.0 now 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 name the attribute that maps to the primary key column of the corresponding table for the class as "ID". Users can now specify the attribute mapping to 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 collection of primitives or wrapper data types as the type of the attribute.

  Object Relational Mapping

  - Inheritance – caCORE SDK 4.0 now supports an alternate way of mapping inheritance hierarchy in the database. SDK users can choose between existing *Table per class* mechanism to map inheritance in the database or they can choose T*able per inheritance* hierarchy for the mapping.
  - Join tables – Previous releases of the SDK used to support join tables only for the many to many type of associations. With SDK 4.0 users can choose to use join tables for any type of associations.

Generated System

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

- **Client Server Infrastructure**

- The client-server infrastructure of 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 similar mechanism. With this design approach, domain object beans generated by SDK are true POJOs and can be used outside of SDK easily.

- In addition to the restructuring of the Java beans with AOP, SDK 4.0 now also can connect to various SDK generated system 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 will now be able to 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 version of SDK required specialized POJO beans in the .ws package whereas SDK 4.0 generated web services utilizes the same Java beans that are used by the other tiers of the application.

  - SDK 4.0 web services also have additional methods to allow users to fetch the associations of the 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 will be deployed automatically when the application server starts

- **Graphical User Interface**

  - The caCORE SDK 4.0 generated system has a newly developed graphical user interface. This new interface allows 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 will get experience of completely secured system and not just one of the 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 release 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 user to retrieve associations regardless of the number of associations between two objects.

- **Security**

  - The caCORE SDK 4.0 has a completely new security implementation that is based on ACEGI security framework. The previous implementation of security in the caCORE SDK was weaved into the application logic. For caCORE SDK 4.0, security implementation is kept outside of the application and is managed through Aspect Oriented Programming principles. SDK users can now easily change the implementation of security without going into the details of 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 are allowed to see Social Security Numbers of 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 Minimal System Requirements

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

Minimal Hardware Requirements

The caCORE SDK 4.0 has been built and tested on the platforms shown in Table 2-2.

	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 2-2 Minimal system requirements

Notes: Users of the caCORE SDK will 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. Select the appropriate hardware configuration based on the amount of data that the system is expected to handle. Users can use the hardware configurations listed above as a reference.

Software Requirements

Download and install the required software listed in Table 2-3. This software is not included with the caCORE SDK. 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

Software	Description	Version	URL
Ant	Build Tool	1.6.5 or higher	http://ant.apache.org/bindownload.cgi

Table 2-3 Minimal software requirements

† Only one is required.

Contributing to caCORE SDK Development

The caCORE SDK project is managed by an NCICB project manager. If a user 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 NCICB management for more details.

Chapter 3 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 the 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 3-1).

Figure 3-1 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

The SDK cannot read and interpret unsupported features that a user has included in the UML model. To interpret unsupported UML features, the SDK code generator must be modified. In addition to the code generator's modification, the runtime system also requires modification so that it can 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 the 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 is illustrated in Figure 3-2 and involves the following high-level steps to generate the artifacts required.

1. Read UML model
2. Artifact generation
3. Output management

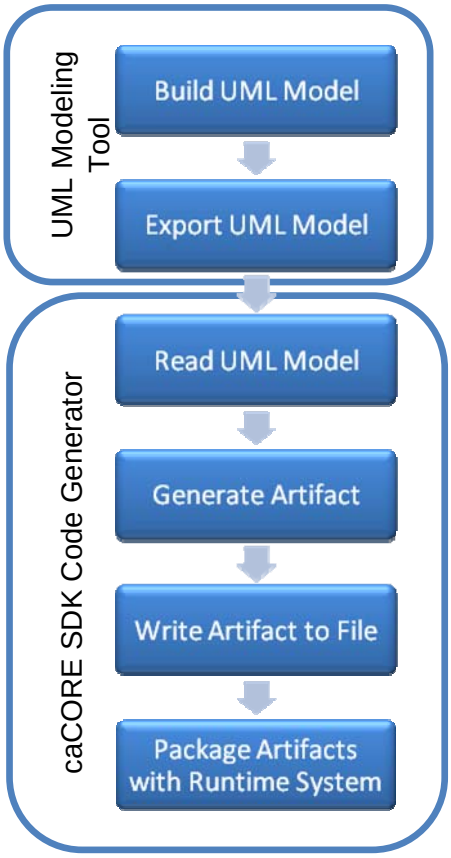


Figure 3-2 Code generation process details

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 the UML Model reader developed as a separate project at the NCICB. The UML Model Reader, also known as the XMI Handler, can interpret model information created by the tools Enterprise Architect or ArgoUML. If a UML modeling tool other than those is to be supported by SDK's code generator, then the UML Model Reader needs to be extended. However, the code generator would not need to be changed.

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 the information obtained from the UML model. Other artifacts are generated from the property files and configuration files supplied at the time of running the SDK. This section addresses artifact generation from the UML model only.

The UML model contains various complex elements. The artifact generation stage reads all the elements of the UML model and then constructs a collection of relevant elements from which a particular artifact can be generated. The artifact generation process needs to 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 to be generated. In the case of multiple artifacts, it becomes necessary to automate these steps so that the artifact generation process can be handled efficiently and effectively.

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.

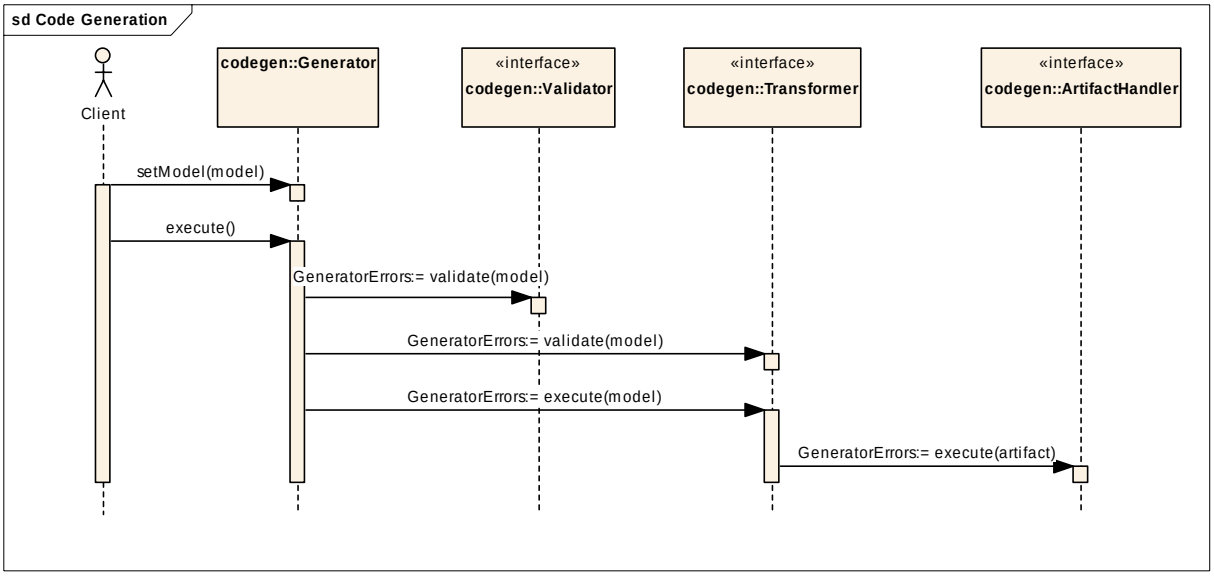


Figure 3-3: Code Generation Workflow Automation Sequence Diagram

As described in Figure 3-3, 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 controller file as sub elements and configuring them one at a time. Once the configuration of the execution engine and components finish, 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 the 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** – A configuration file named *hibernate.cfg.xml* is generated for Hibernate, which contains a list of all the generated Hibernate mapping files in addition to 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 4 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 4-1). This n-tiered system divides tasks or requests among different servers and data stores. This isolates the client from the details of where and how data is retrieved from different data stores. 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).

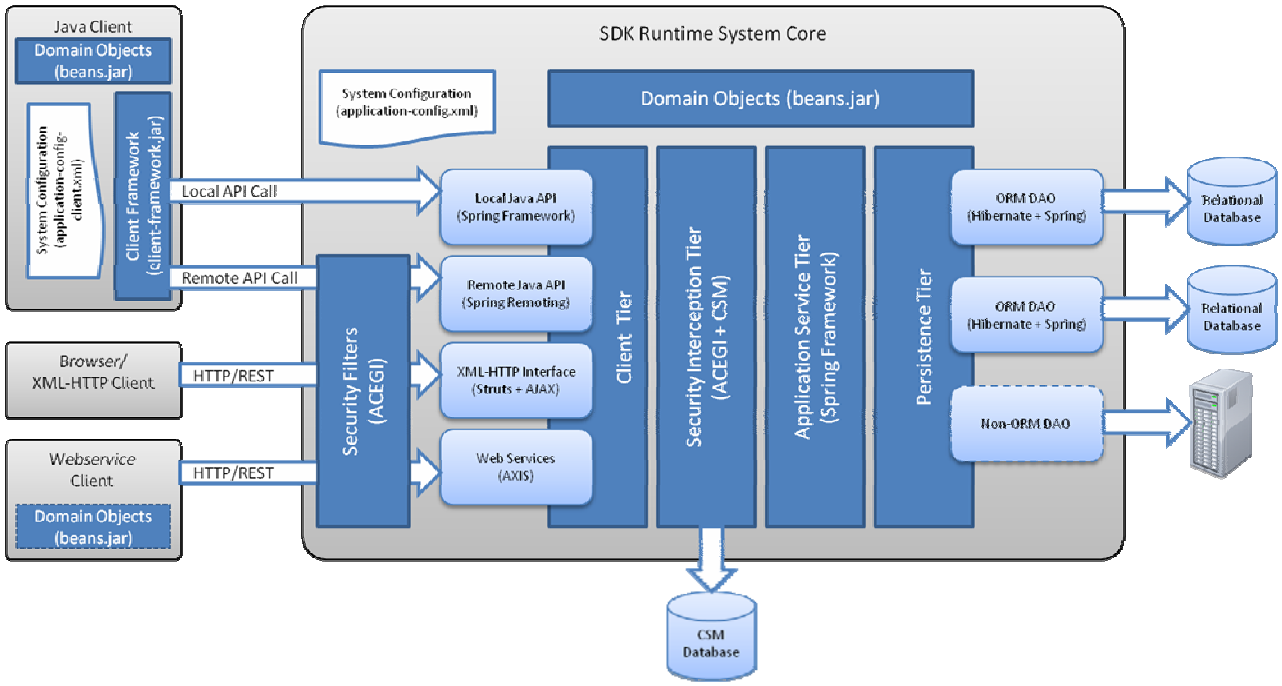


Figure 4-1 SDK Generated Runtime System Architecture

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*
- *Application Service Tier*

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 (Figure 4-2). 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 the domain objects when the Application Service tier requests it by using the *getAllClassNames()* method of the DAO. 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 described in the following sections.

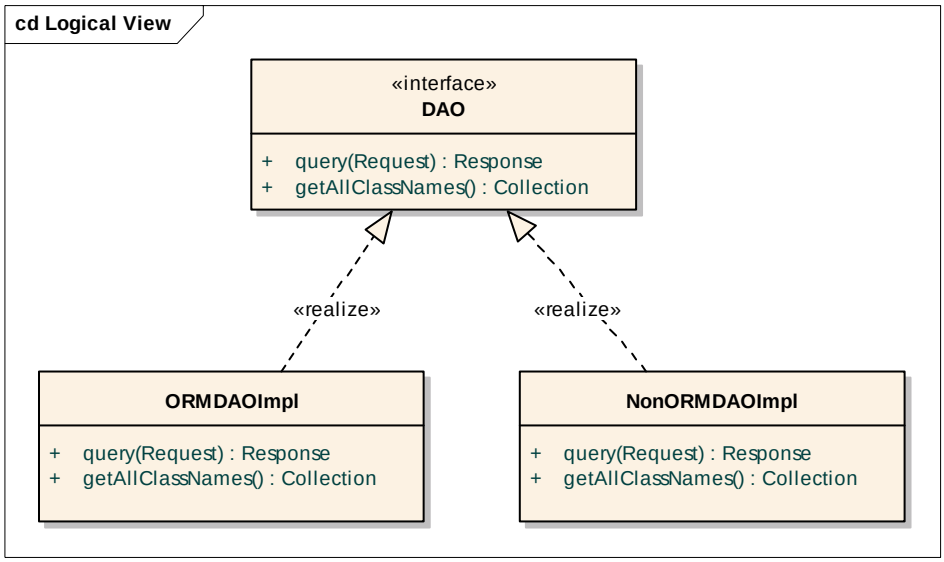


Figure 4-2 Persistence Tier Classes

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 object relational mapping (ORM) files. These ORM files are generated by the caCORE SDK during the code generation process. If a user intends to not to use one or all of the mapping files and provide the mapping files developed independently then he 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 the later section.

Non-Object Relational Mapping

SDK users can choose to not to use the ORM as a way to map the relational database to the objects or the data for the objects which reside on a remote server. In this scenario, the responsibility of populating the objects based on the query is on the user of the SDK who will have to 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 implements another required method (*getAllClassNames()*) which returns 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 (Figure 4-3). It 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. The default Application Service tier has methods to support ORM based systems. The methods are sufficient to support requirements for most applications. The following sections describe details about adding additional methods in the Application Service tier.

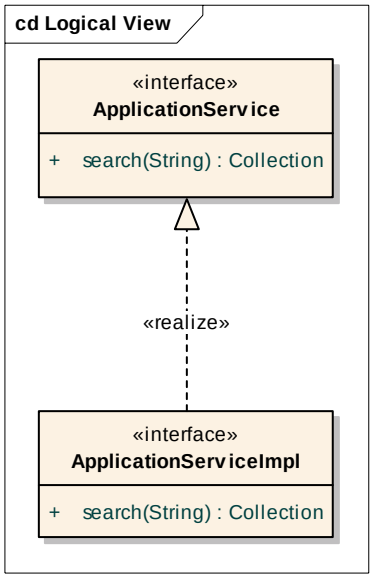


Figure 4-3 Application Service

Extending the Application Service Tier

The default Application Service tier has methods to support an ORM-based system. If additional methods are added in the Application Service tier, then 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 Figure 4-4, a *CustomApplicationService* can extend the ApplicationService interface and the class *CustomApplicationServiceImpl* provides a concrete implementation of the method inside the *CustomApplicationService* interface. 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 ApplicationService.

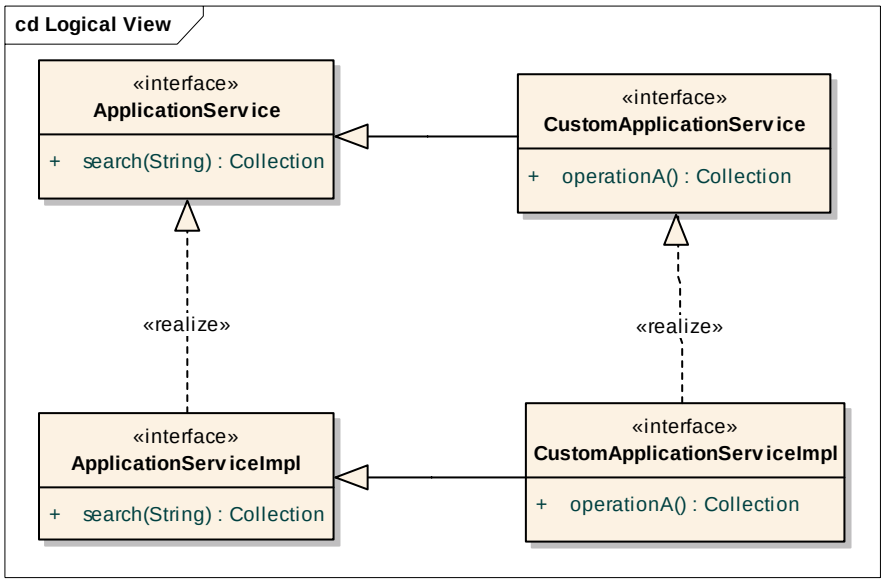


Figure 4-4 Extending Application Service

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 NCICB. 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 *Security* on page 31.

Client Interface Tier

The SDK provides four distinct methods to reach the Application Service Tier.

1. XML-HTTP Interface (browsers, thin clients)
2. Web Services Client
3. Local Java API Client
4. 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. Details about system usage for each method follows.

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 was also known as Happy.jsp interface in the previous release. The 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. When client intends to fetch the data in the form of XML instead of a web page format, he/she 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`).

See *XML-HTTP* Client on page 25 for additional information.

Web Services Client

SDK generated web services runs 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 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. See *Web Services Client* on page 26 for additional information.

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 4-5.

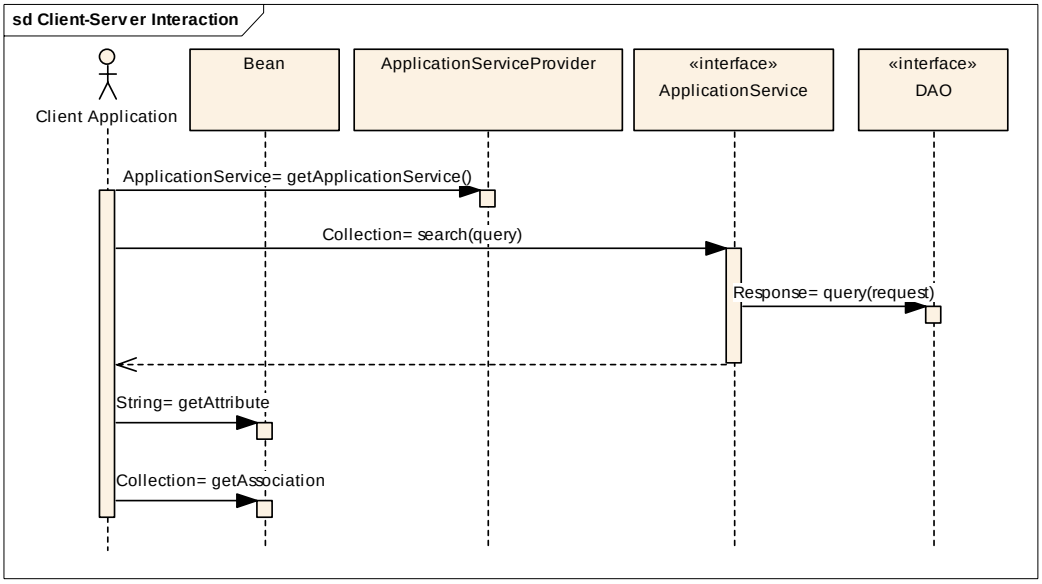


Figure 4-5 Java API communication with an SDK generated application

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 with 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 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 better to isolate the client from knowledge about what type of client is used to fetch the data (that is, local client versus remote client).

Technical Challenges

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, it is least relevant which technology is adopted to solve the problem of the client server communication. Another problem 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 the large result set and 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 the event when the 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 whenever the client application makes an attempt to access the attribute/association of that 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 in Figure 4-6 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

receive ApplicationServiceProxy instead.

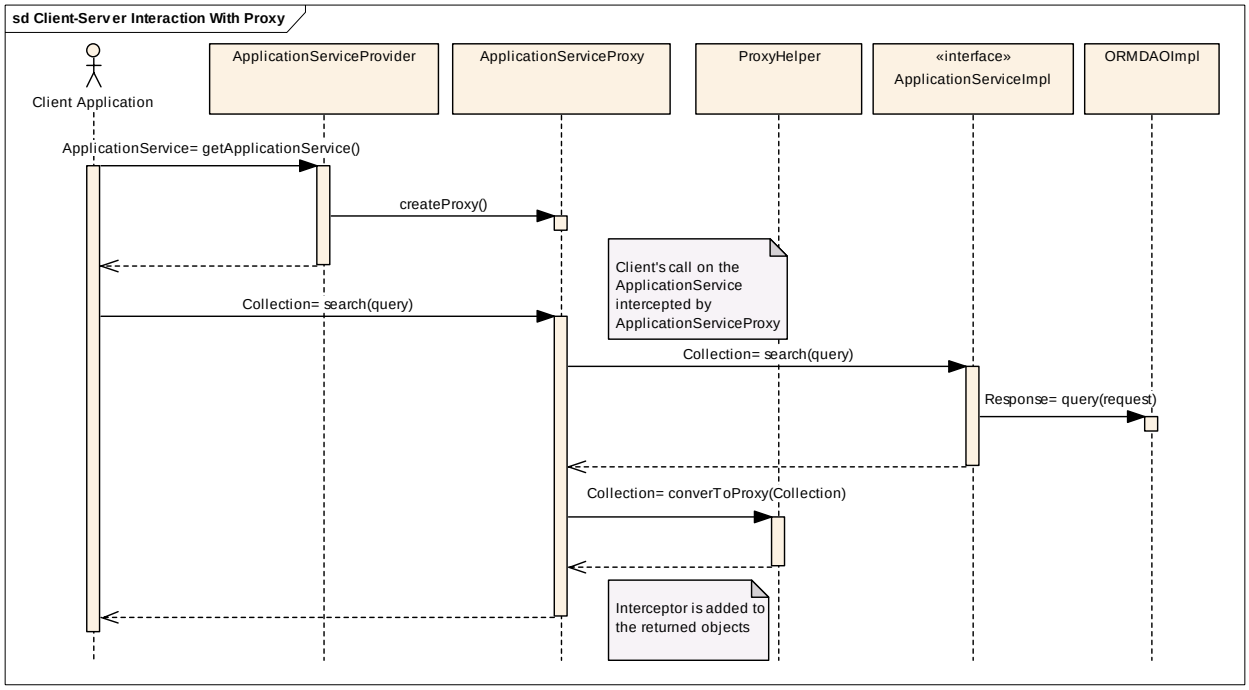


Figure 4-6 Actual Behavior of the SDK Generated Application - 1

When the client application calls the ApplicationService, *ApplicationServiceProxy* intercepts the call and executes it. 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 the 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.

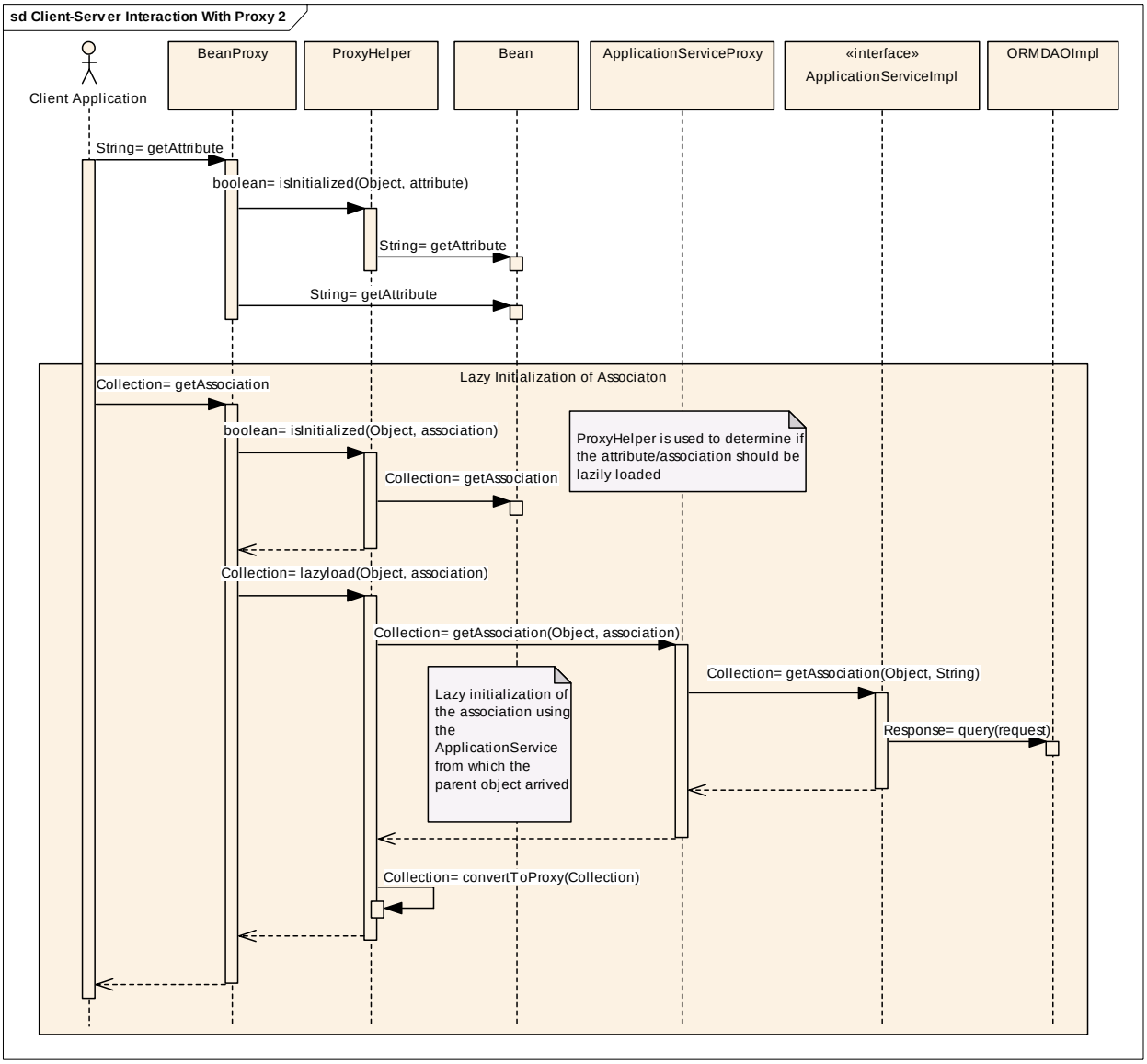


Figure 4-7 Actual Behavior of the SDK Generated Application - 2

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 Application Service.

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 the 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 this feature, 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 modified in the ApplicationService interface then 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.

Security

The caCORE SDK provides an integrated security solution that uses ACEGI and CSM as its underlying technologies with ACEGI as the security interceptor layer (Figure 4-8). Once calls from the user are intercepted with the help of ACEGI, CSM is used to provide security at the core level. By default, security is disabled in the SDK configuration. To enable the security, enable the security flag in the configuration file and generate the system. Security can also be enabled the security in the generated system by modifying multiple configuration files. However, since this process is error prone, it is not recommended to all the users of the SDK.

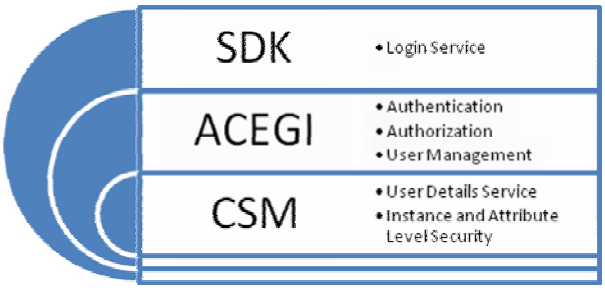


Figure 4-8 Security Layers in caCORE SDK

The SDK can provide security at the following levels (Figure 4-9):

- Unsecured system/No Security – All 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 the data. For example, doctors can view data for their patients whereas administrators cannot.
- Instance level security – Users are allowed to access only data to which they have access. For example, doctors can view data for their patients only and no other patients in the system.
- Attribute level security – Users are allowed to see data for which they have authorization. For example, doctors can view a patient's medical record number but not a patient's social security number.

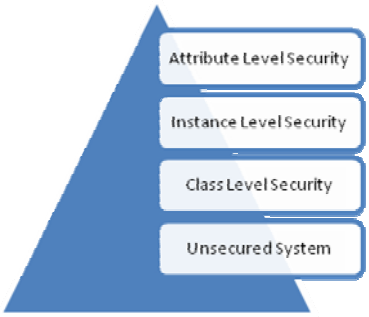


Figure 4-9 Security Levels in caCORE SDK

SDK users can select the appropriate level of security for a system. Security level configuration is enabled and managed through a configuration file at the system generation time. When security is enabled, the system achieves the class level security by default. To provide instance and/or attribute level security, see *Configuring Security* on page 117.

Authentication

SDK generated application provides a mechanism to log user in the application. It takes the user credentials from the client and supplies it to ACEGI framework so that ACEGI framework can validate the user credentials and decide if 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 in the application. If the user is successfully logged in the system, his/her security policy is cached at the application level. In case of unsuccessful login, an exception is thrown back to the user indicating the cause of the error.

Authorization

Authorization in SDK refers to the class level security. Whenever user is trying to execute any operation on the service layer, ACEGI framework intercepts the call to the operation and with the help of a SecurityHelper class it determines what classes the user is trying to access. Subsequent to that, 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 then access denied exception is thrown back to the user.

Instance and Attribute Level Security

If the system is a secured system and instance and/or attribute level security is enabled then SDK utilizes the CSM's services to provide the instance and attribute 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 which restricts user to retrieve only the records to which he/she has access to. In order to use this feature, the CSM security configuration has to be available on the same database schema.

Chapter 5 System Usage

This chapter provides examples to access the generated system's client interfaces by a client application or a user.

Topics in this chapter include:

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 SDK web page contains links to several other pages that facilitate access to domain data. The *Home* page is accessed from the following URL (Figure 5-1):

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

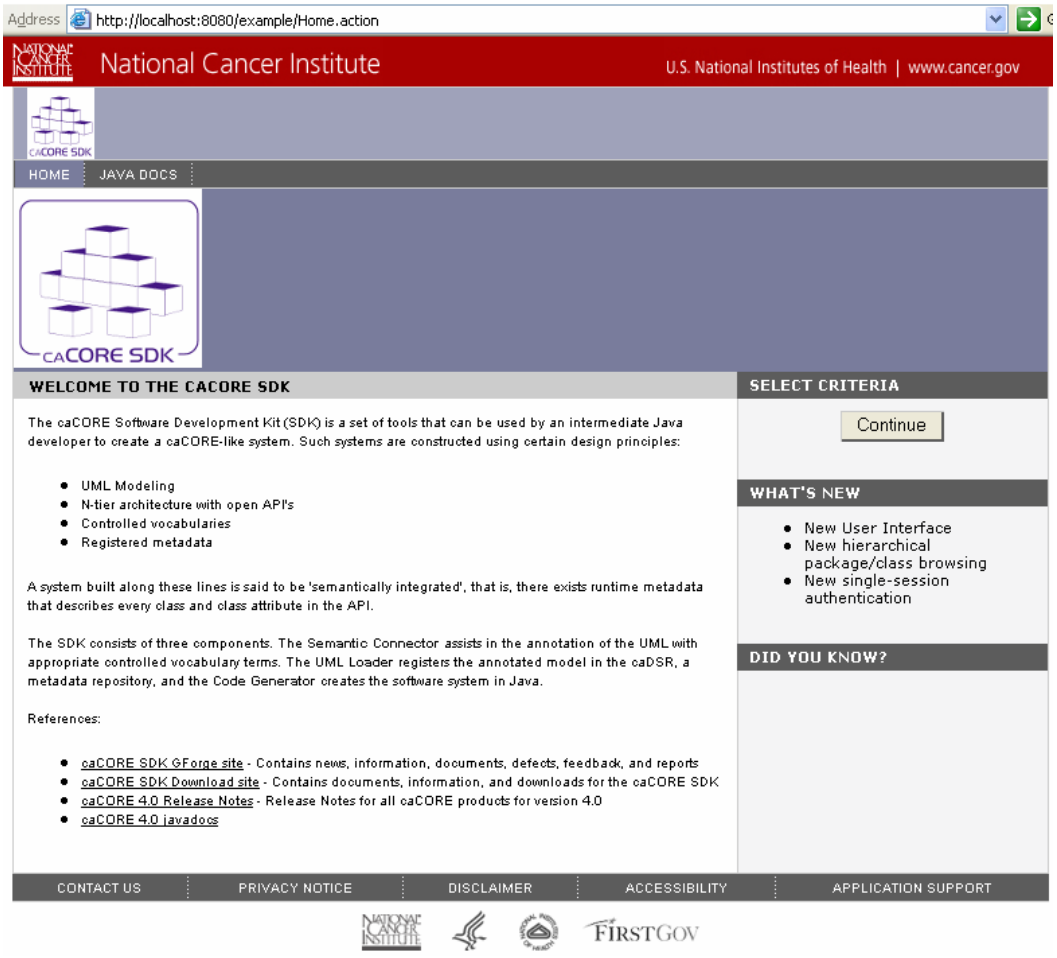


Figure 5-1 SDK Home page

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

Note: When security is disabled (which is the default), a **Continue** button displays on the Home page. If security is enabled, a Login form requesting a User ID and Password displays instead. More about enabling security is provided in *Configuring Security* on page 117.

Click the **Continue** button to display a hierarchical domain package/class browser tree known as the *Content* page, which 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 5-2.

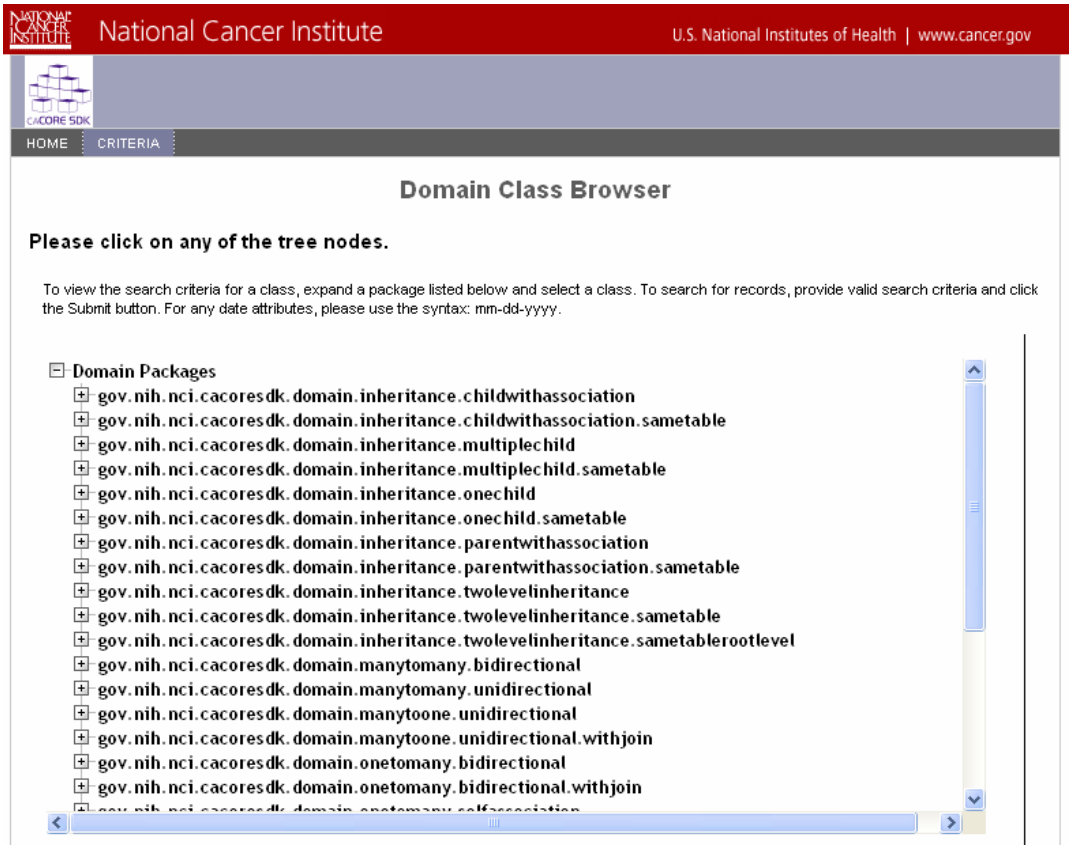


Figure 5-2 Content page

Expand or collapse 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 is displayed to the right of the browser tree.

Figure 5-3 illustrates the *Search Criteria* form for the Professor class of the sample SDK model:

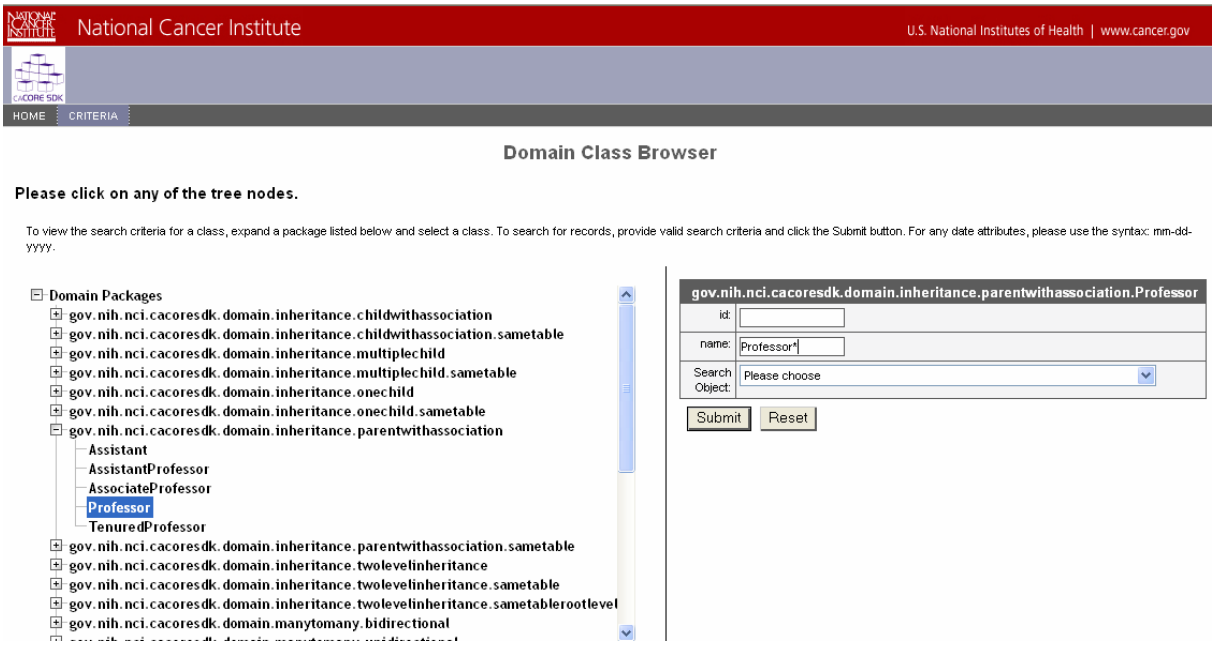
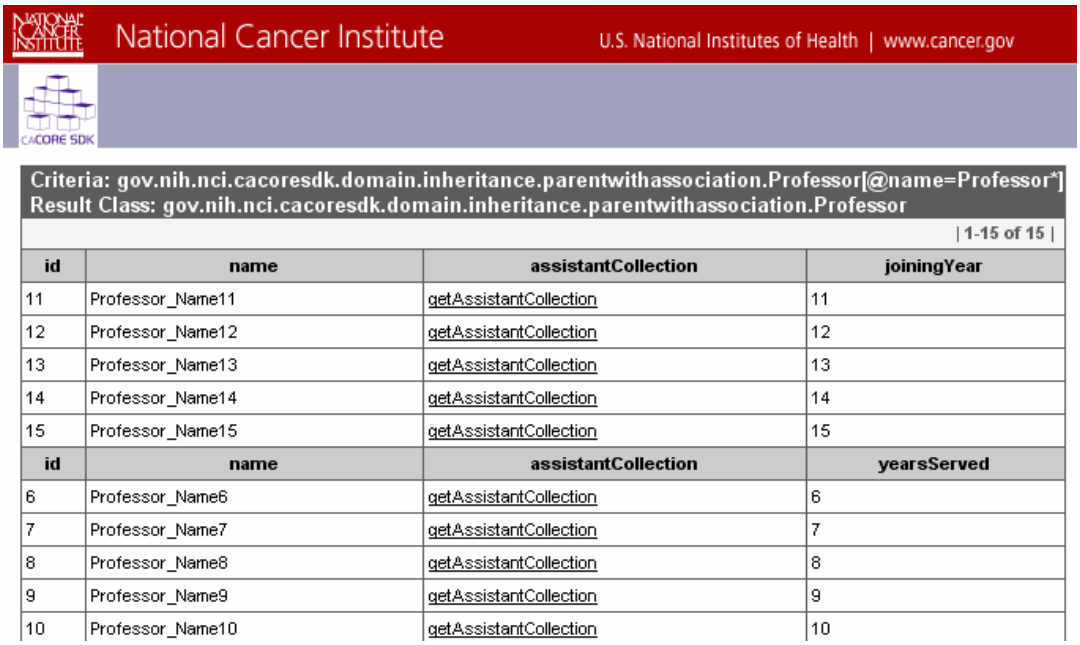


Figure 5-3 Search Criteria Form

Note: 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.

Click the **Submit** button on the *Search Criteria* form to display any matching records on the *Result Data Table* page in a new window (Figure 5-4). Fields from the resulting domain class type are displayed as table columns within the table header on the Results page. A collection of wrappers of primitive object types and field values are displayed as strings within the corresponding table cells. Fields that represent an association to another domain class are displayed as links, which can be clicked to retrieve any associated domain object records.



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 5-4 Result Data Table page

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.

The URL used by this interface uses the following pattern:

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

The following table (Table 5-1) 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

Parameter	Description
	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.
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 5-1 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 5-5 provides a sample of the XML output produced from invoking the *Sample REST URL* 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 5-5 Sample XML output from REST call

Java API Interface

The use of the Java API client involves two primary steps. The first step involves obtaining a reference to the instance of *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 illustrating 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 112.

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)

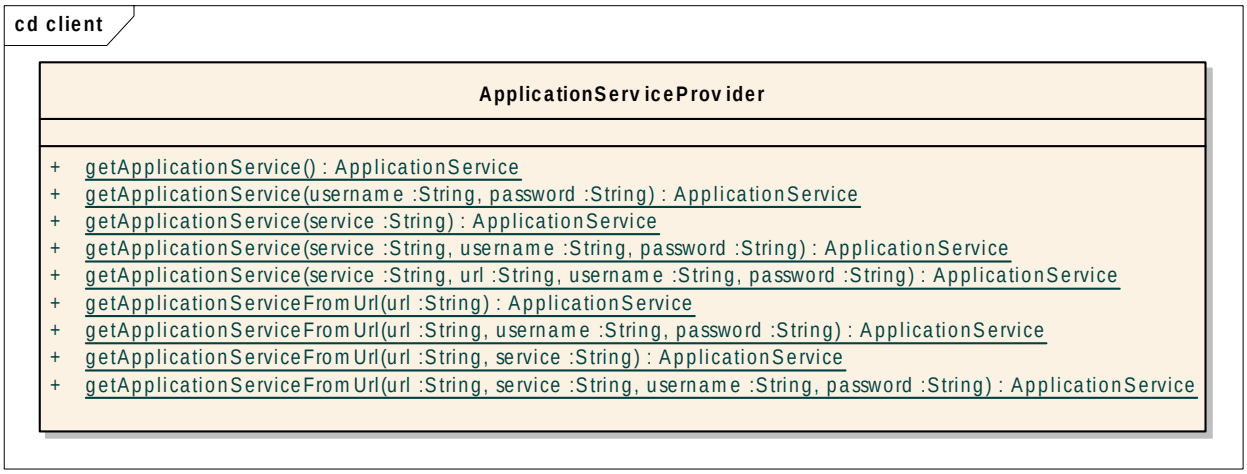


Figure 5-6 ApplicationServiceProvider Methods

The four required parameters required by the methods of the *ApplicationServiceProvider* class are described in the following table (Table 5-2).

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.
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:

ApplicationService Parameter	***Description***
	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 5-2 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 differing 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. A class diagram, which highlights these methods, is shown in Figure 5-7.

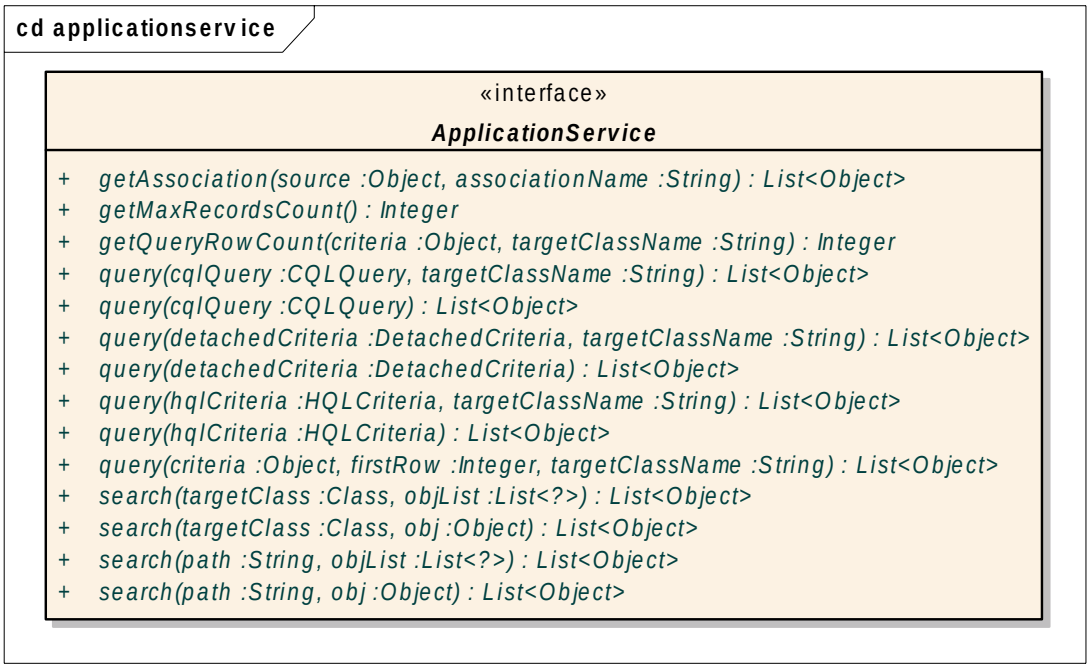


Figure 5-7 ApplicationService Interface Methods

The *ApplicationService* methods are grouped into different categories and are discussed is the following sections.

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 5-3 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 5-3 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 4-8.

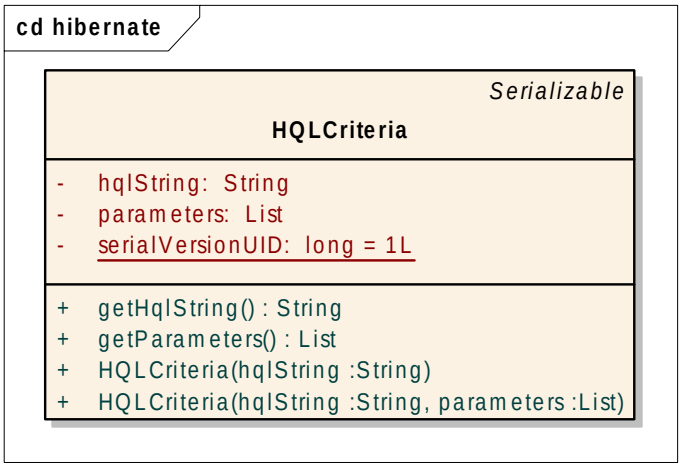


Figure 5-8 HQLCriteria Class Diagram

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

ApplicationService Method	*Description*
	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 5-4 HQL ApplicationService methods

Figure 5-9 shows how an SDK *HQLCriteria* object representing an HQL query might be instantiated and submitted. Figure 5-10 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 5-9 Sample HQL 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 5-10 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 the scope of a session to be executed later using some arbitrary Hibernate Session.

Table 5-5 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 5-5 Detached Criteria related ApplicationService methods

Figure 5-11 shows how a Hibernate *DetachedCriteria* object might be instantiated and the query submitted. Figure 5-12 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 5-11 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 5-12 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. See http://en.wikipedia.org/wiki/Query_by_Example for more information.

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 5-13. The starting object for a CQL query is always *CQLQuery* object in which the user has to specify which object (target object) 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. To create a query like "where zipcode=20852 and name like '%Dav%' ", then CQLGroup can be utilized.

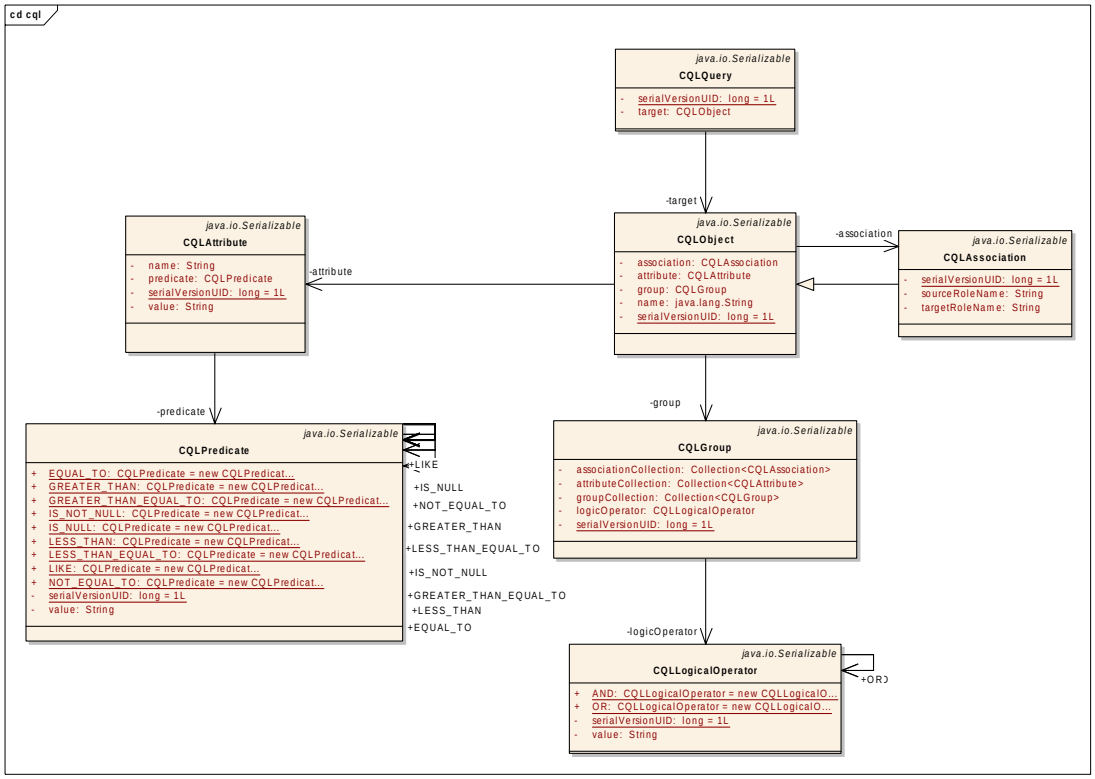


Figure 5-13 CQL Query Association Diagram

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. **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(CQLQuery cqlQuery, String targetClassName)	Deprecated. Internally calls the query(CQLQuery cqlQuery) method without the targetClassName parameter.

Table 5-6 CQLQuery related ApplicationService query methods

The following paragraphs provide an example of how to create and execute a CQL query using the *ApplicationService* interface. Figure 5-14 shows classes from the sample SDK model package *gov.nih.nci.cacoresdk.domain.other.levelassociation*, and is provided as a point of reference.

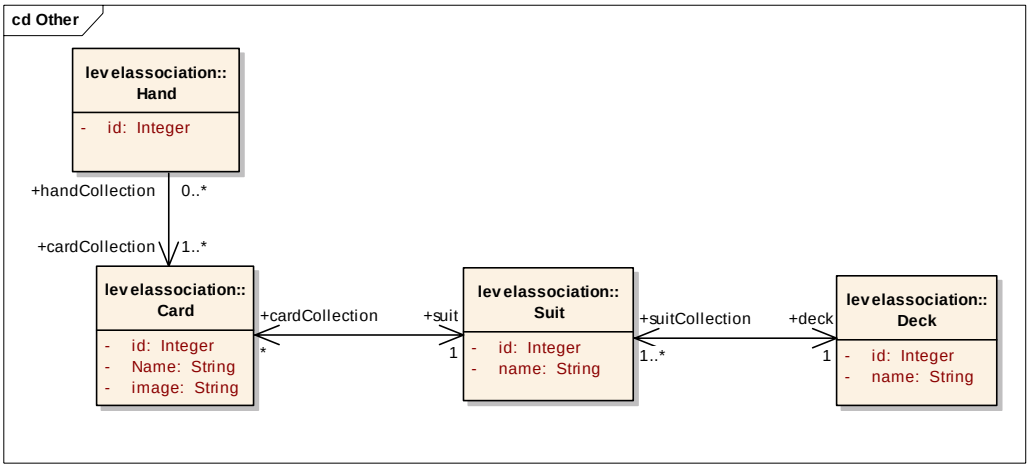


Figure 5-14 Sample Domain Class Diagram

Figure 5-15 shows how an SDK CQL query object might be instantiated and the query submitted as "select * from Suit where id=1".

```
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 5-15 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 5-16 Sample CQL Query without Association Results

Figure 5-17 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)”.

```
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 5-17 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 5-18 Sample CQL Query with Association Results

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 (the fully qualified name of the class) to retrieve from the database. The next item in the comma-separated path is a link in the chain to an element (fully qualified name of the class) that connects the element on its left to the element on its right. 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 object that is desired using a comma separated path.

Table 5-7 highlights the *Nested Search Criteria* related *ApplicationService* methods.

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. **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.
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

ApplicationService Method	Description
	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. **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.
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. **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.
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. **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.

Table 5-7 Nested Search Criteria related ApplicationService methods

Figure 5-19 demonstrates how to use the nested search criteria. In the 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 5-19 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 5-20 Sample Nested Search Criteria Query Results

Web Service Interface

The SDK 4.0 Web Services 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 4.0 Web Services uses the Remote Procedure Call (RPC) Web Service style).

See the Introduction to Web Services Metadata: http://dev2dev.bea.com/pub/a/2004/10/Anil_WServices.html. There are four "styles" of service in Axis. *RPC* services use the SOAP RPC conventions, and also the SOAP "section 5" encoding. *Document* services do not use any encoding (so in particular, you will not see multiref object serialization or SOAP-style arrays on the wire) but DO still do XML<->Java databinding. *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, 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 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 (that is, 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 115.

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 5-21 provides a list of the schema imports found within the WSDL for the *sample SDK model.* It is provided here in order to emphasize 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:

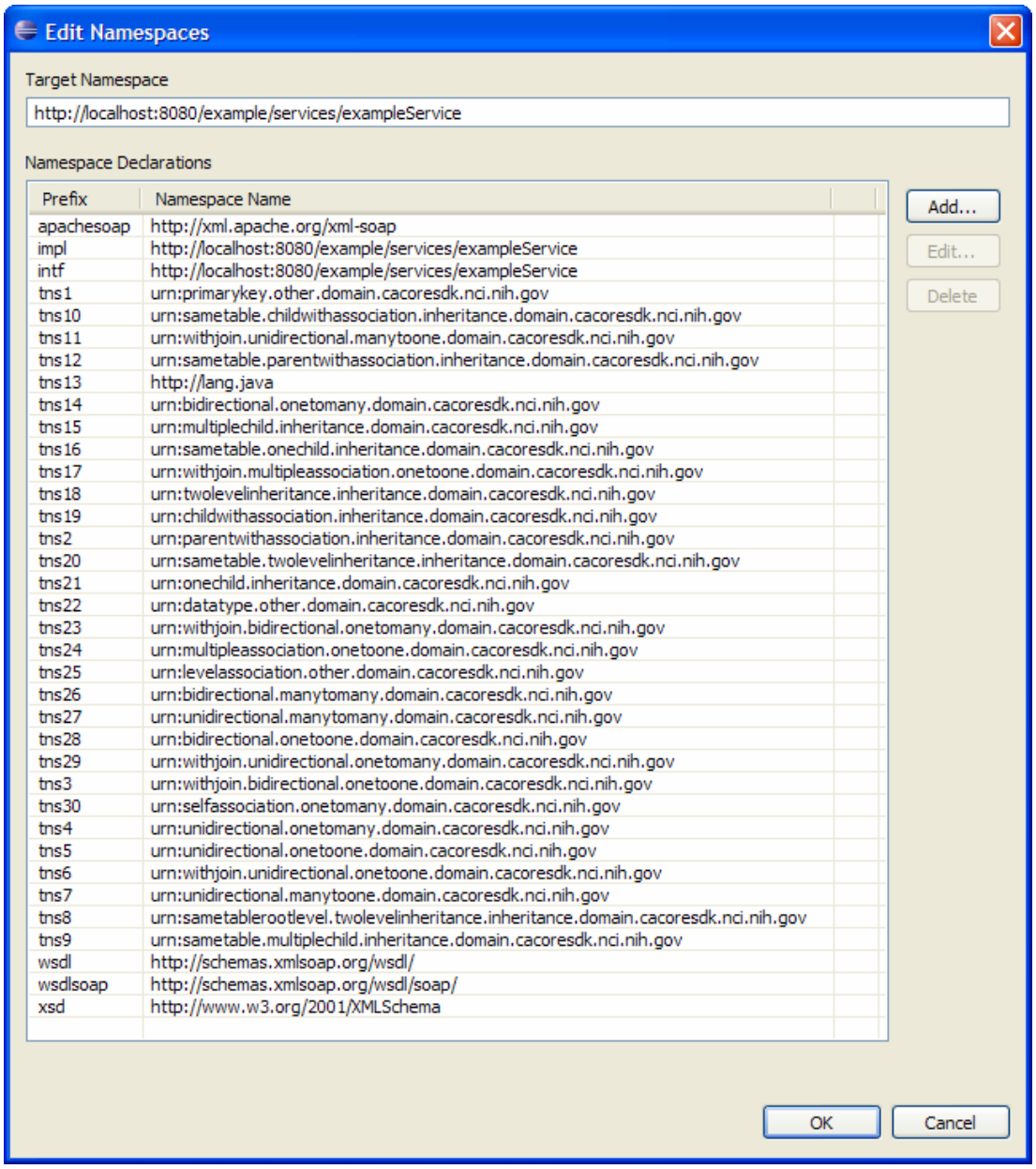


Figure 5-21 Sample WSDL Directives - Schema Imports

WSDL Service Definition

The WSDL defines a Web Service as a collection of network endpoints, or *ports*. Figure 5-22 and Figure 5-23 provide details for the SDK generated Web Service defined within the WSDL, which includes its:

- Name
- Prefix
- Target Namespace, and
- Port Information

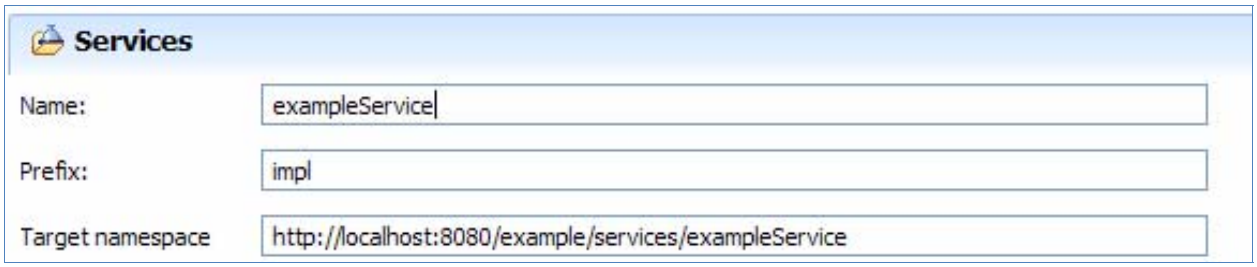


Figure 5-22 Sample WSDL Service Definition

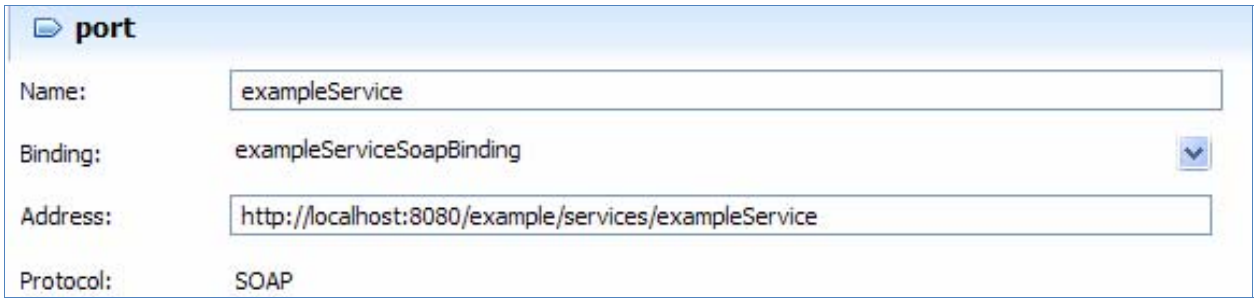


Figure 5-23 Sample WSDL Service Definition – Port

Note: The SDK Code Generator uses the value of the PROJECT_NAME property provided within the `deploy.properties` file (in this case, "example") while generating the WSDL. 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 4-24 provides a summary of the collection of network endpoints (and their messages) that composes any SDK generated Web Service.

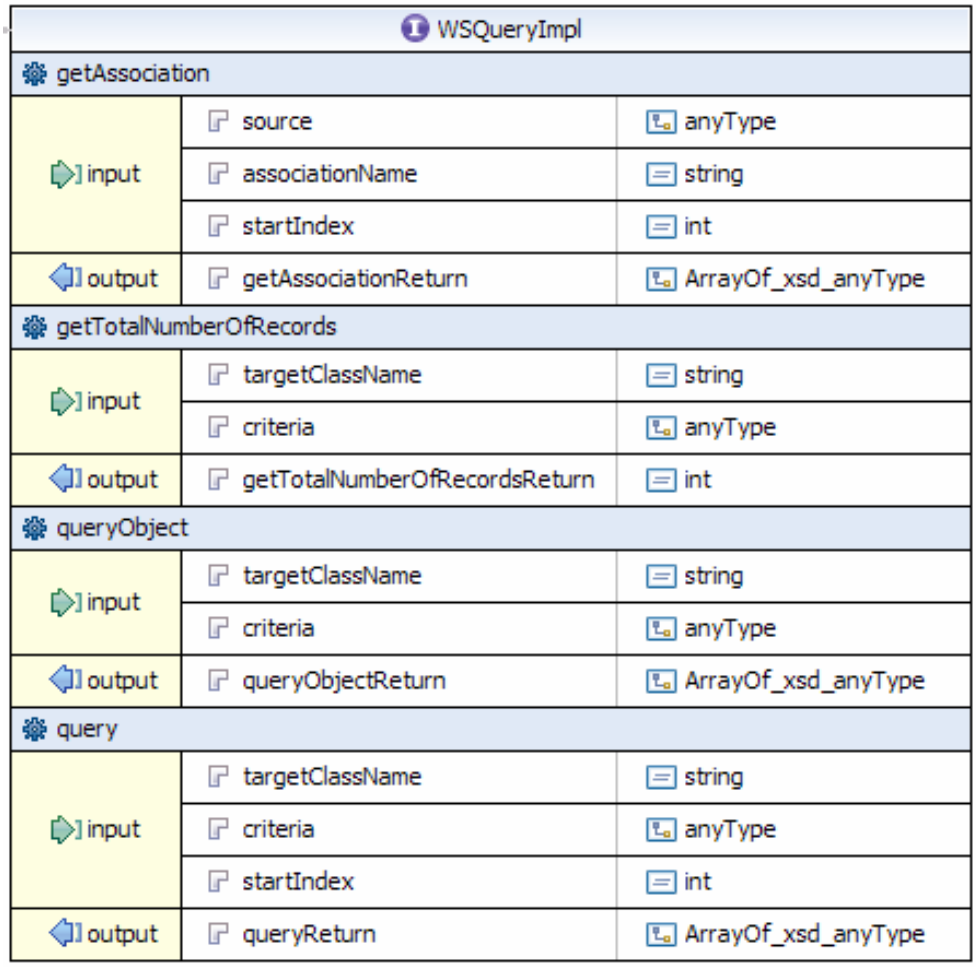


Figure 5-24 WSDL Port Types (Network Endpoints)

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 5-8 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

Message	***Description***
	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 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 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.
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 seem to allow the "overloading" of method

Message	*Description*
	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 5-8 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 cannot fulfill a request from a Web Service client. For example, when a Web Service client sends a *getAssociationRequest* message and 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. An example of that would be interruption in network, 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, if it can be delivered.

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 (Table 5-9).

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 5-9 SOAP Fault Structure Element Descriptions

Chapter 6 System Usage for a Secured System

This chapter describes how to use the SDK generated runtime system when security is enabled.

Topics in this chapter include:

Introduction

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. Depending on the client interface that is used, the login operation varies. Information is provided in this chapter about logging into the individual client interfaces. Once the login operation is complete, querying the system can be done in the same fashion as explained in the previous chapter.

XML-HTTP Interface

The XML-HTTP interface can access data using two types of the clients 1) a web browser to view data in the form of a web page and 2) a thin client to get XML data. Browser-based clients have security configured through a form-based authentication meaning that a user must type in a username and password in the login form provided by the application (Figure 6-1).

Note: The login form appears on the SDK Home page whenever the secured system is generated.

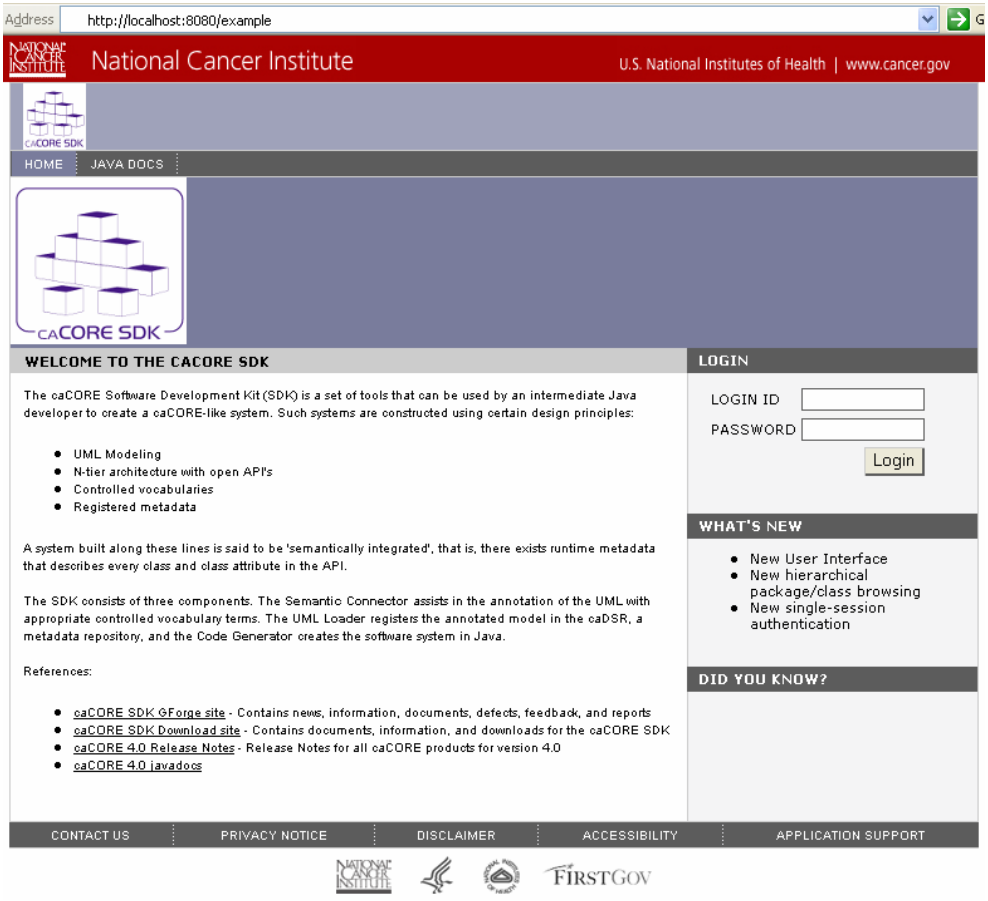


Figure 6-1 Security login form in the web browser

If login is unsuccessful, the client receives an error message shown in Figure 6-2. After three consecutive unsuccessful login attempts, the user's login account is locked out by CSM for 30 minutes.

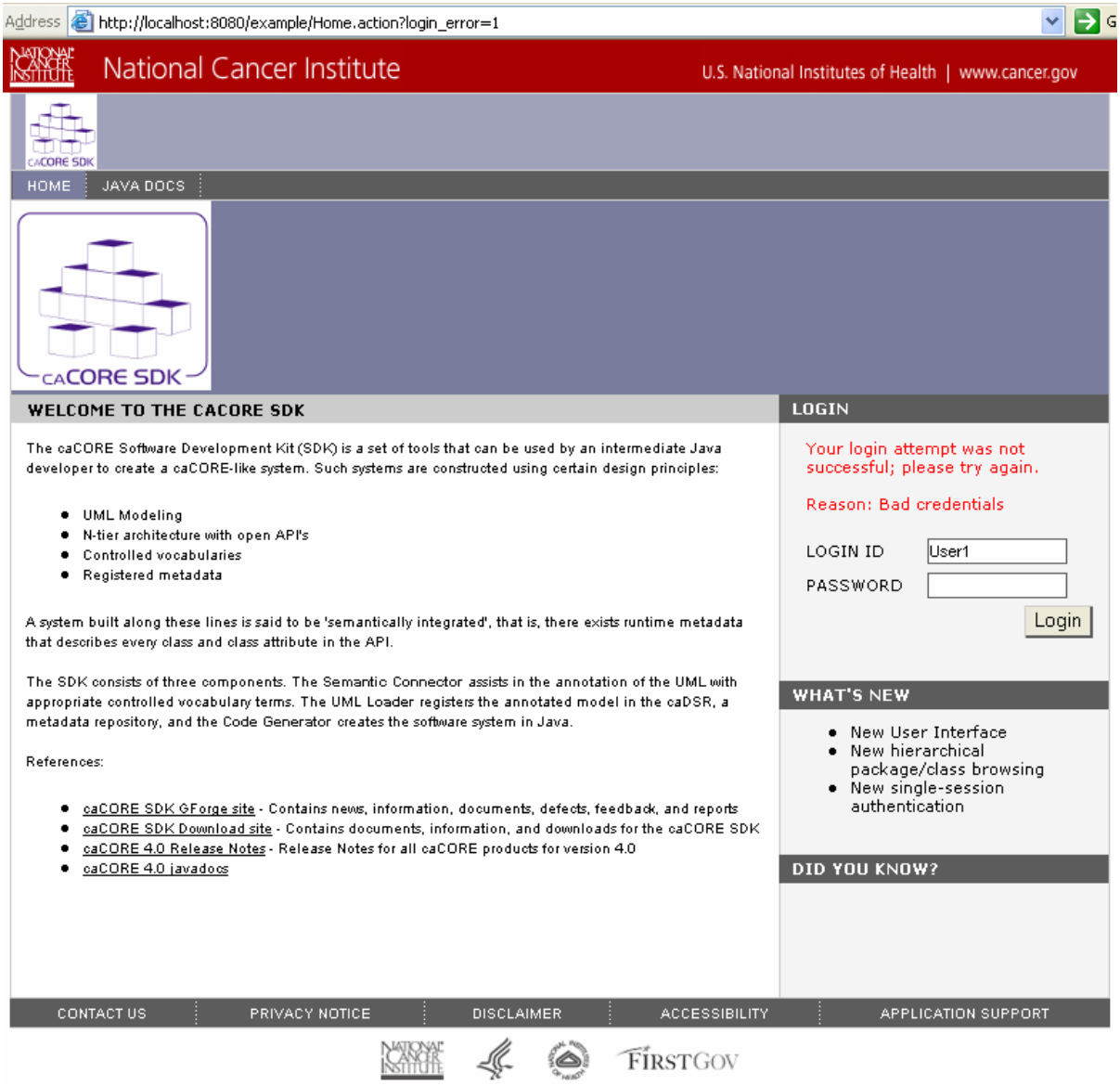


Figure 6-2 Security Login failure in the web browser

The XML-based REST interface communicates with the SDK generated application using a thin client application. 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 in the user in the application. The code snippet in Figure 6-3 demonstrates setting 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 6-3 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 that is made to the server.

Web Services Client

A secured SDK generated 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 in Figure 6-4 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 6-4 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 is making a call to the server.

Java API

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. If the username and password are passed as parameters then the *ApplicationServiceProvider* class validates the username and password against the authentication service and if successful, logs the user in the application (Figure 6-5).

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

Figure 6-5 Security Login in Java API client

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

Chapter 7 Performance Tuning the Java API

The SDK generated Java API provides the powerful feature of creating a data service in a small amount of time. As the SDK is just a tool to generate the API, it cannot understand all the use cases of the 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 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 missing or corrupted due to various reasons including batch data load and recreating 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 of generating 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.

Lazy Loading

Problem: By default, an SDK generated application will only fetch the objects demanded by the query. For example, a query for a *Person* object will result in fetching of only a Person object and none of its associated objects. When the client application invokes the *getter* method to retrieve the associated objects (for example, getAddress() of a Person object),

the SDK generated application seamlessly connects to the server to retrieve the associated object(s). This approach is known as lazy loading and results in a delayed response from the application due to additional calls over the network.

Solution: This default behavior can be overridden in two ways:

1. Disabling lazy loading for certain associations of the object through the O/R mapping. Based on the use case of the system, a user of the SDK code generator can specify a UML tag-value with the key = lazy-load and the value for the key = no on the association which the user intends to fetch eagerly (or not lazily). Currently, this approach works for a unidirectional association only.
2. Using Hibernate's DetachedCriteria for eager loading of associated objects . This approach is the more flexible approach of the two. The user of the SDK generated runtime system can use DetachedCriteria from Hibernate, which allows specifying the eager loading option for some of the associations for the queried object. For more information see, http://www.hibernate.org/hib_docs/reference/en/html/querycriteria.html#querycriteria-detachedqueries.

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 an 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.

Chapter 8 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 (Figure 7-1).

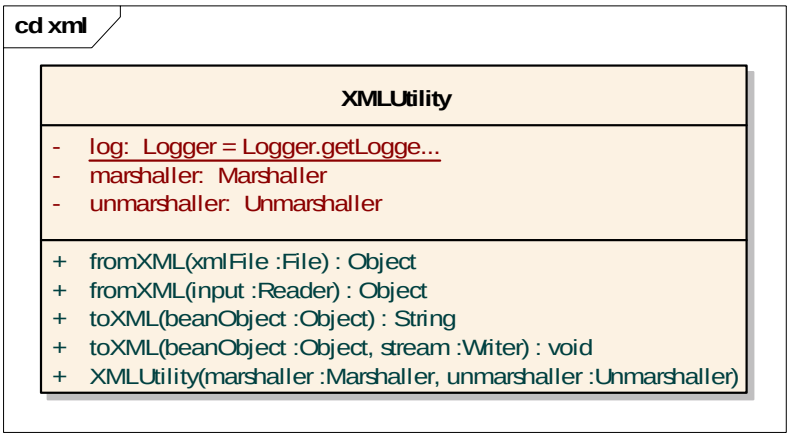


Figure 8-1 XML Utility Class Diagram

As implied by the *XMLUtility Constructor* method, the *XMLUtility* class wraps both an SDK Marshaller and Unmarshaller class, which it is dependent upon in order to perform its work. These collaborating classes and their interfaces are discussed in the following sections.

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 (Figure 7-2).

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

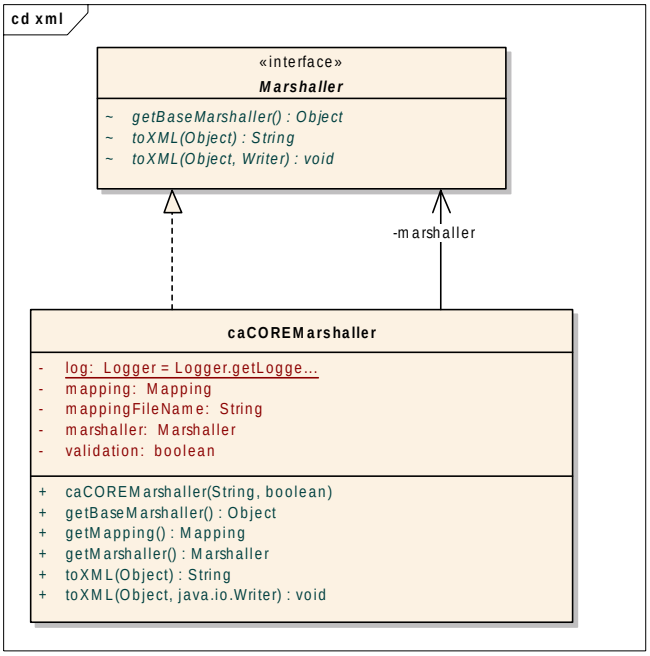


Figure 8-2 Marshaller Class Diagram

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.

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 Figure 7-3.

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

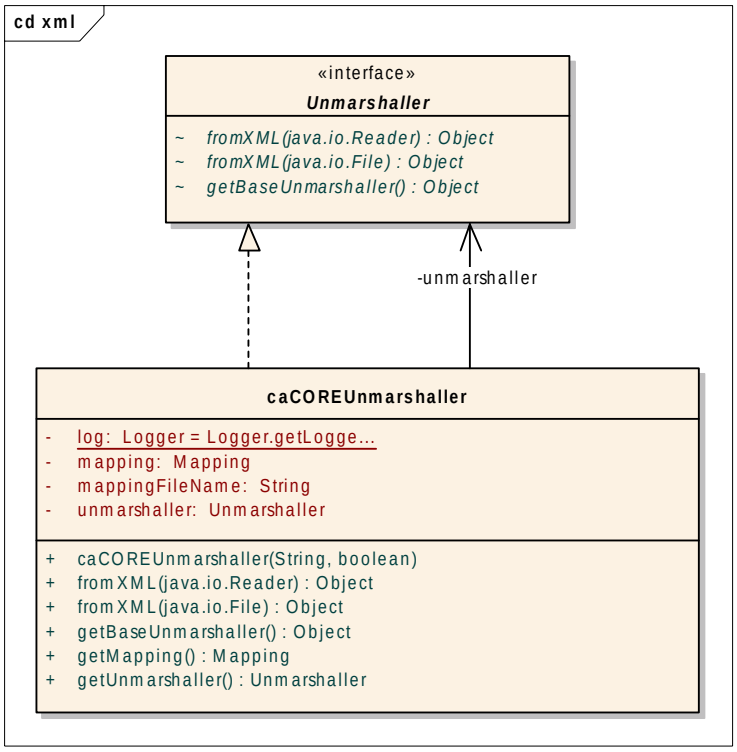


Figure 8-3 Unmarshaller Class Diagram

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.

Marshalling Java Objects to XML

The XMLUtility class provides two wrapper methods for marshaling (serializing) domain Java objects to XML, as described in Table 8-1.

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 8-1 Wrapper methods for marshaling domain Java objects to XML

The code snippet in Figure 8-4 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 8-4 Sample marshaling code

A sample test program, TestXMLClient.java, is also 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 113.

Unmarshalling XML to Java Objects

The XMLUtility class also provides two wrapper methods for unmarshalling (deserializing) XML to domain Java objects, as described in Table 8-2.

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 8-2 Wrapper methods for unmarshalling XML to domain Java objects

The highlighted portion of the code snippet in Figure 8-5 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 8-5 Sample Unmarshalling Code

A sample test program, TestXMLClient.java, is also 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 113.

Chapter 9 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)

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 file is, respectively:

- 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

To create a new object model project file, use the following steps.

1. In EA, 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 9-1).

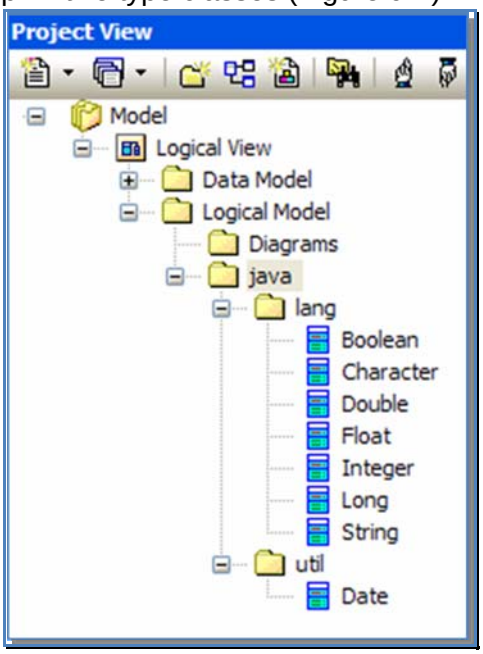


Figure 9-1 EA Project View Browser

In ArgoUML, open the *SDKArgoTemplate.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, classes representing the wrapper Java primitive type classes, *Tag Definitions (TD)* for all the possible Tag Value types, and *DataTypes* (Figure 9-2).

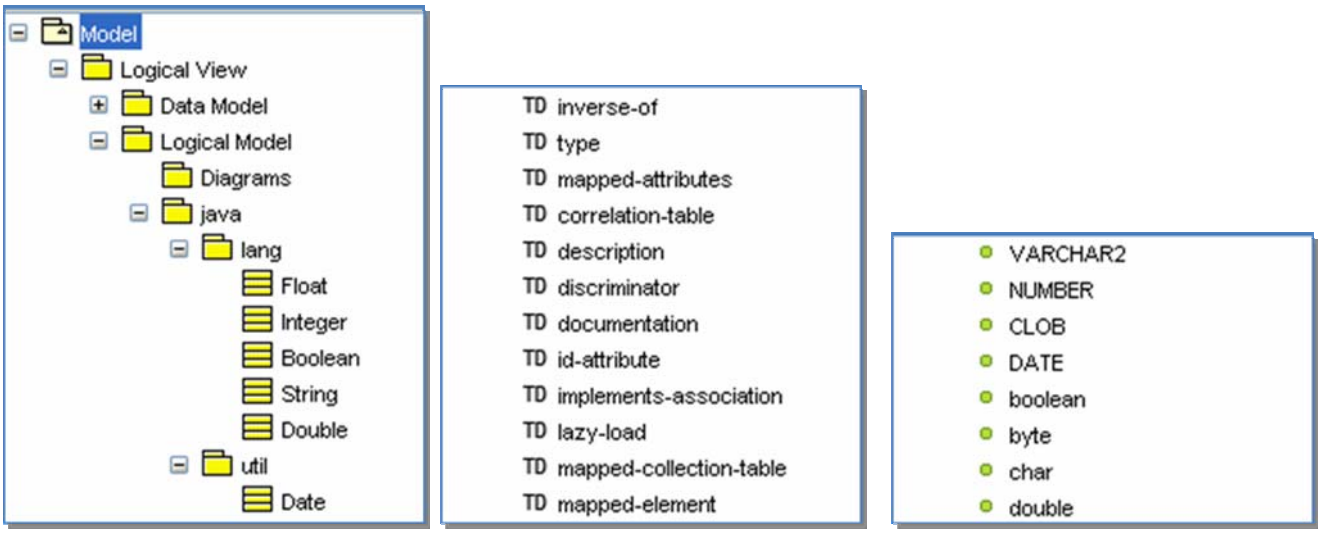


Figure 9-2 ArgoUML Explorer Pane Showing Packages/Classes, Tag Definitions, Data Types

2. Select the **File > Save Project As**.
3. In EA, the **Save Enterprise Architect Project** dialog displays (Figure 9-3).

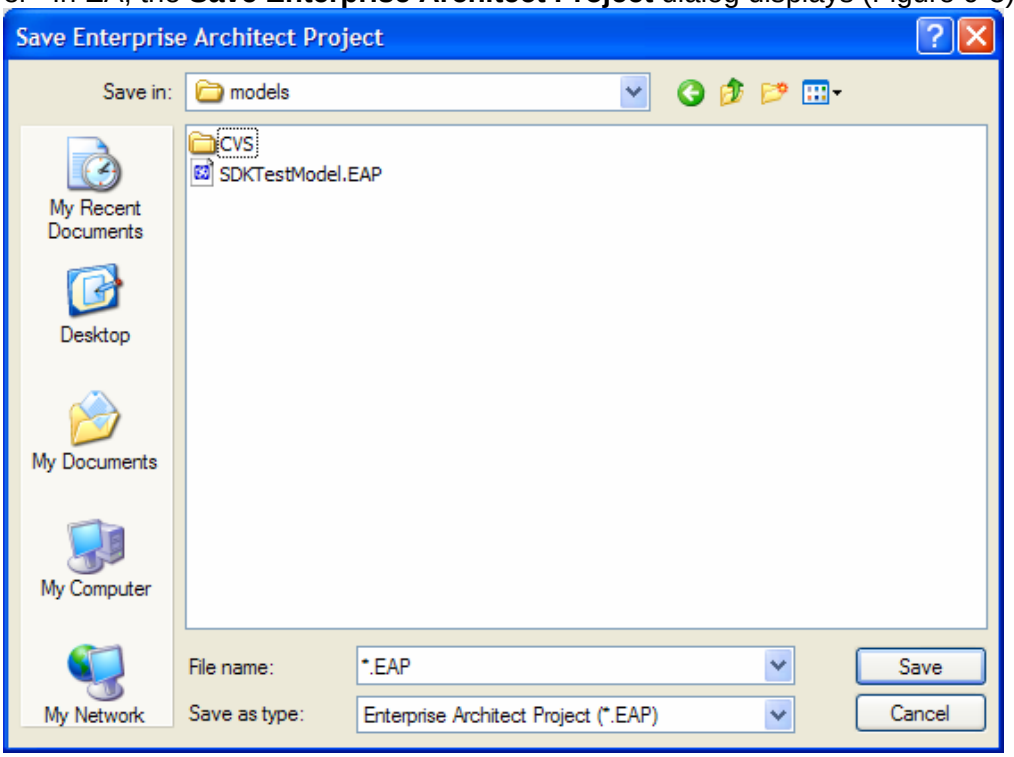


Figure 9-3 EA Save Enterprise Architect Project dialog

In ArgoUML, the **Save Project** dialog displays (Figure 9-4).

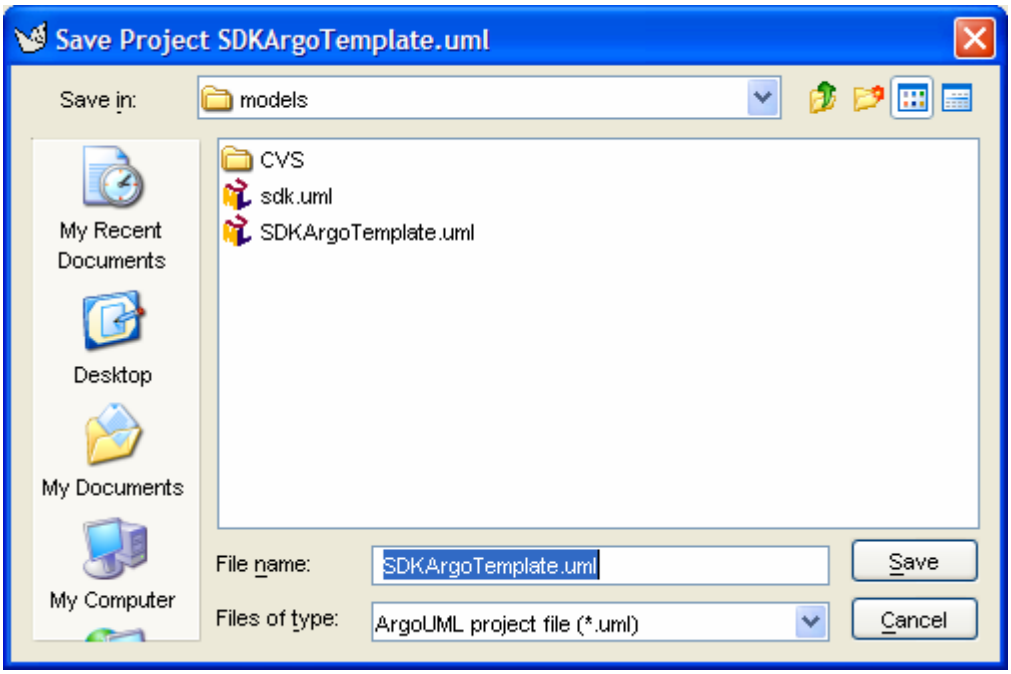


Figure 9-4 ArgoUML Save Project dialog

4. Enter a new project name in the **File name** field.
5. Click the **Save** button.
6. Alternatively, either of the baseline template files, *SDKEATemplate.EAP* or *SDKArgoTemplate.uml*, can be copied and renamed. The new project file is now ready for use in 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

To add a package structure to the *Logical Model*, use the following steps.

1. In EA, select the *Logical Model* package (Figure 9-5).

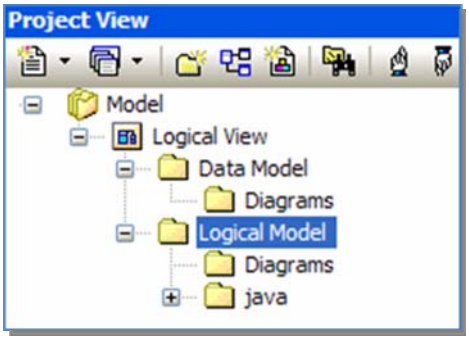


Figure 9-5 EA Project Browser

In ArgoUML, select the *Logical Model* package (Figure 9-6).

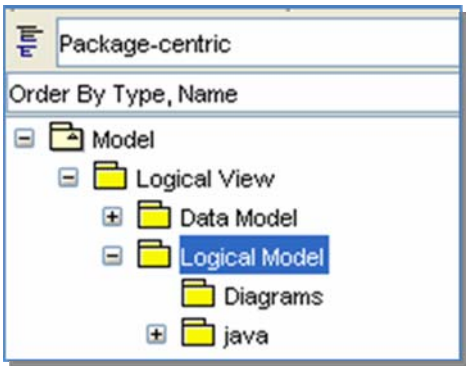


Figure 9-6 ArgoUML Explorer Pane

Note: See http://argouml-stats.tigris.org/documentation/manual-0.24/ch11.html for more information on the ArgoUML Explorer pane.

2. In EA, right click and select **Add > Add Package** or select **Project > Add Package** submenu.
   In ArgoUML, right click and select **Add Package**.
3. In EA, the following dialog displays (Figure 9-7).

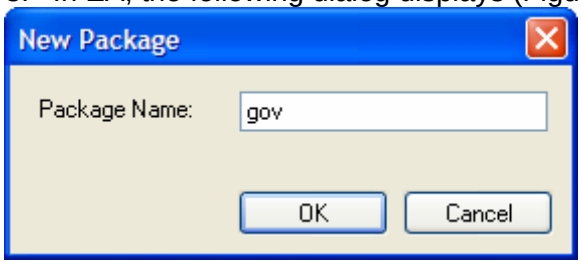


Figure 9-7 EA New Package Dialog

In ArgoUML, the **Properties** tab in the Detail pane becomes active for the new package (Figure 9-8).

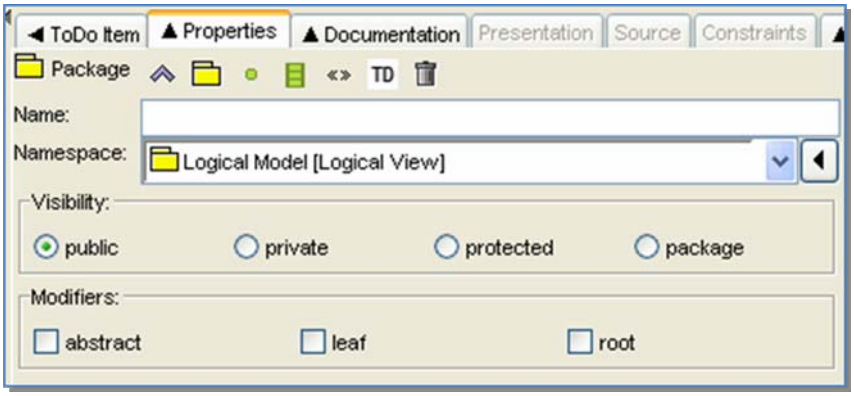


Figure 9-8 ArgoUML Package Detail Pane, Properties Tab

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

4. In EA, enter a package (folder) name, and click **OK**. In ArgoUML, 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.*

   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, that is 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.

5. Repeat steps 2-4 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 2-4 above. The following diagram *(EA only)* shows most of the package hierarchy created in this manner for the SDK sample model:

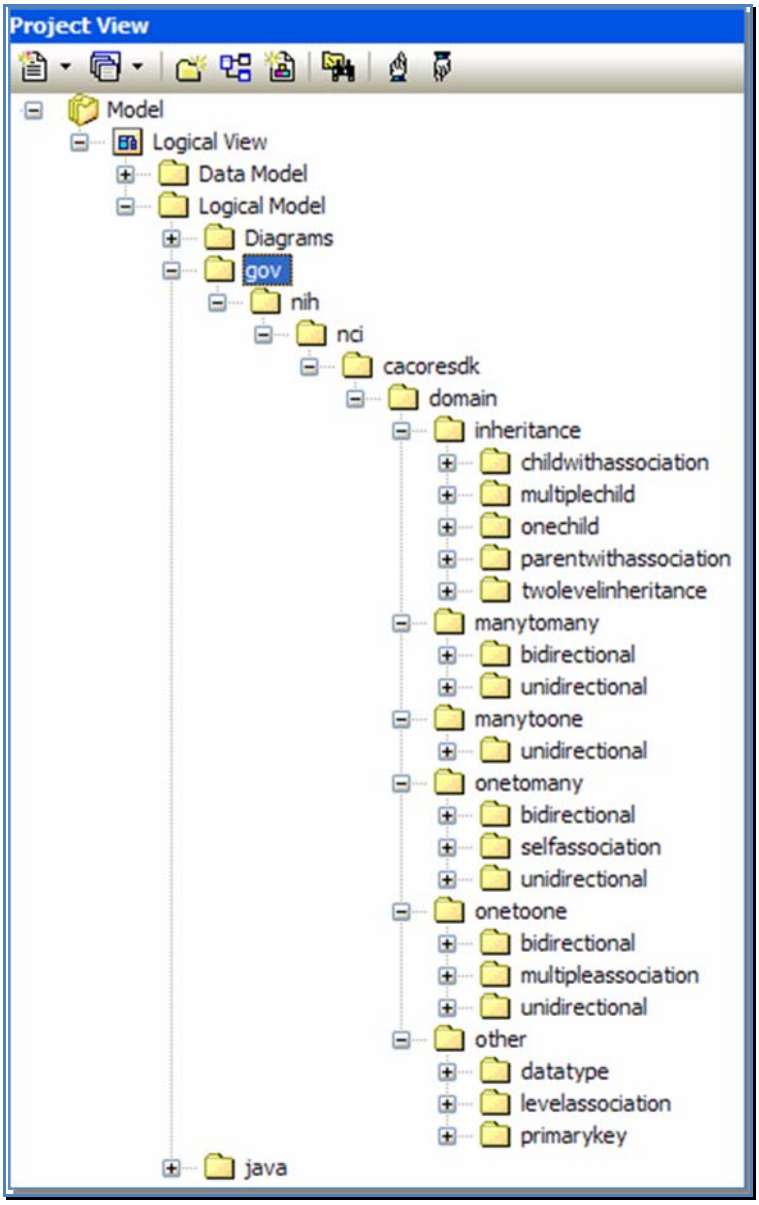


Figure 9-9 EA SDK Sample Model Packages

Creating a Logical (Object) Model Class

To add a Logical Model class to a package, use the following steps:

1. In the EA *Project Browser*, or the ArgoUML *Explorer* pane, select the desired *Logical Model package* to which the class should be added.
2. In EA, right click and select **Add > Add Element** or select **Project > Add Element** submenu.
   In ArgoUML, new classes are added in the context of a class diagram within the selected package. If needed, click on a class diagram within the package to open/activate it, or create a new class diagram within the package if none exists. Select the **New Class** icon

( ) found at the top of the diagram Editing pane and then draw a new class within the diagram. This effectively creates a new class within the selected package. Alternatively, click on an existing class within the selected package. The Property tab of the Detail pane displays the properties for the selected class. It also displays the **New Class** icon. Click on this icon.

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

3. In EA, the Insert New Element dialog displays (Figure 9-10).

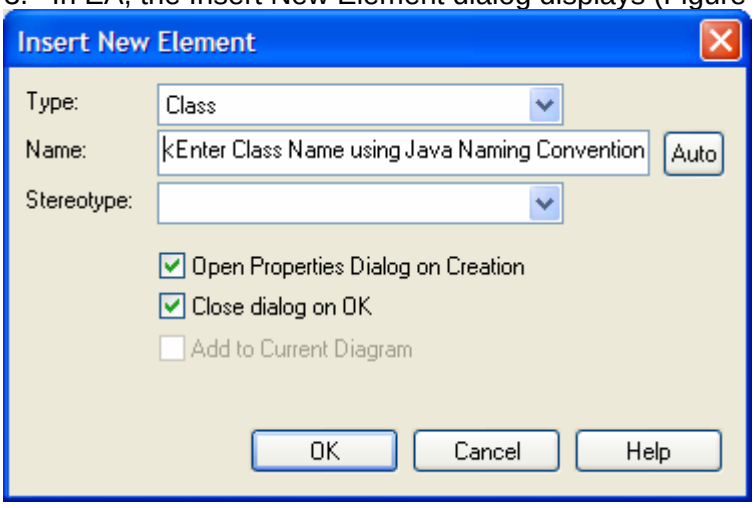


Figure 9-10 EA Insert New Element Dialog

In ArgoUML, the Properties tab in the Detail pane becomes active for the new class (Figure 9-11).

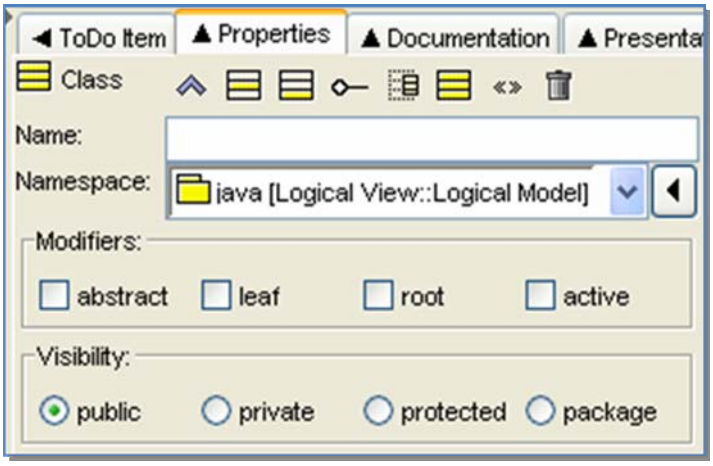


Figure 9-11 ArgoUML Class Detail Pane, Properties Tab

4. 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

Insert New Element Option	*Description*
	embedded words capitalized.
Stereotype	Leave blank for Logical (object) Model classes.
Open Property Dialog	Check the *Open Property Dialog* option if you want the *Property* dialog to open immediately after the class is created.
Close dialog on OK	Uncheck the *Close dialog on OK* 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*.

In ArgoUML, enter a class name in the *Name* field according to the Java class naming conventions; i.e., class names should start with a capital letter, with embedded words capitalized. Next, select the desired package where the class should be created from the *Namespace* drop down.

5. In EA, click **OK**. If the Open Property dialog was checked, the Property dialog opens immediately after the class is created. The Property dialog for the SDK sample *Credit* class is shown Figure 9-12*(EA)*.

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

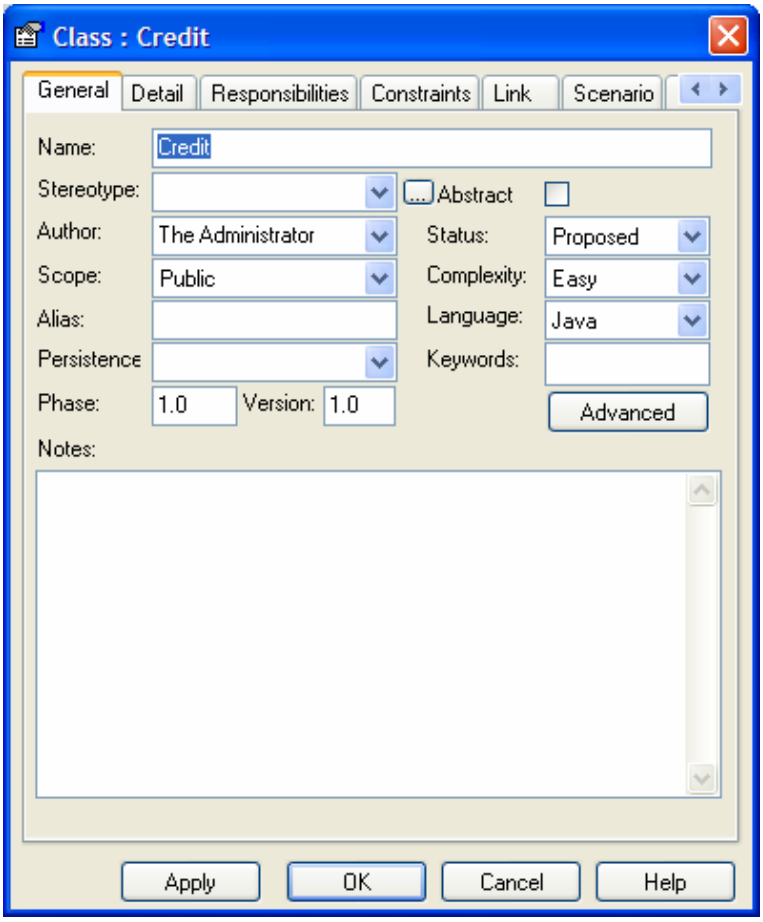


Figure 9-12 EA Class Property Dialog

In ArgoUML, click **Save Project** ( ) or press the CTRL-S.

For instructions on adding attributes to classes, see *Creating Attributes and Data Types* on page 85.

6. Repeat steps 1-5 to add other classes. In EA, 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 repeating steps 4-5. Figure 9-13 shows a series of classes that have been created in the many-to-many *bidirectional* and *unidirectional* packages of the SDK Sample model.

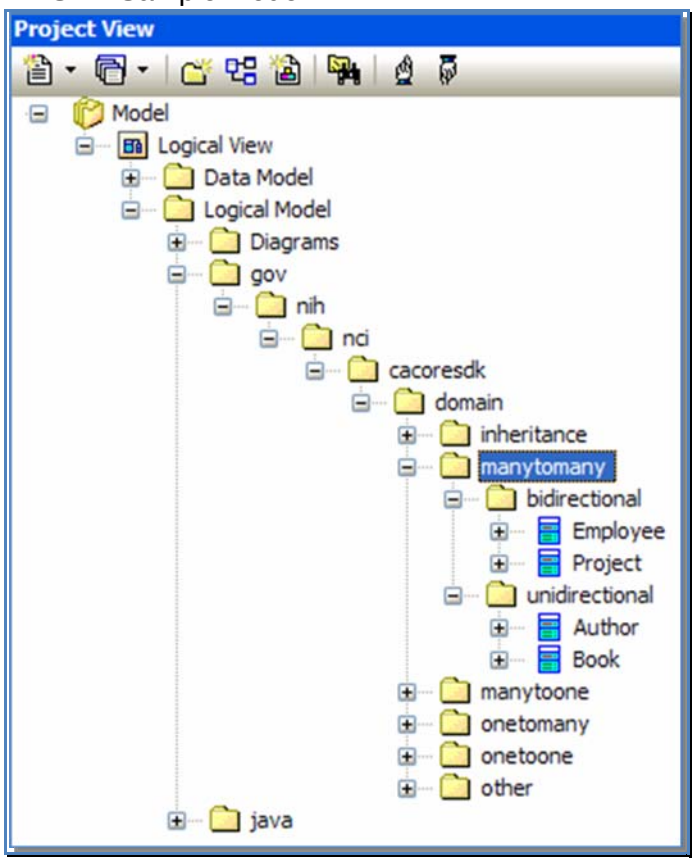


Figure 9-13 EA Project View Browser Showing SDK Sample Classes

Creating a Data Model Table

To add a Data Model (Table) class, use the following steps.

1. In the EA Project Browser, select the **Data Model** package (Figure 9-14).

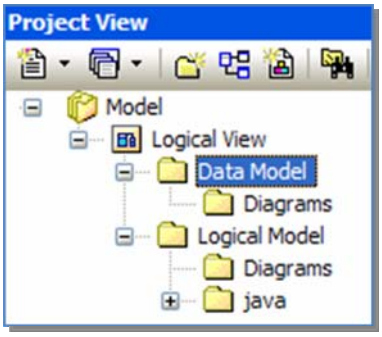


Figure 9-14 EA Data Model Package

In ArgoUML Explorer pane, select the **Data Model** package (Figure 9-15).

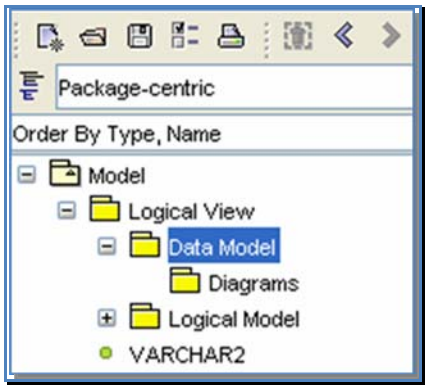


Figure 9-15 ArgoUML Data Model Package

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

2. In EA, right click and select **Add > Add Element** or select **Project > Add Element**.

In ArgoUML, new classes (tables) are added in the context of a class diagram within the selected package. If needed, click on a class diagram within the Data Model package to open/activate it, or create a new class diagram if none exists. Select **New Class** ( ) at the top of the Diagram Editing pane and then draw a new class within the diagram.

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

This effectively creates a new class within the selected package. Alternatively, click on an existing class (table) within the selected package. The Property tab of the Detail pane displays the properties for the selected class. It also displays the **New Class** icon ( ). Click on this icon.

3. In EA, the *Insert New Element* dialog displays (Figure 9-16).

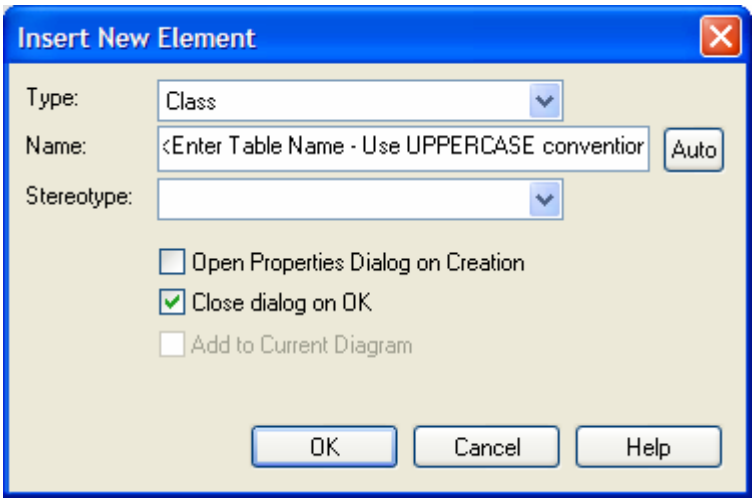


Figure 9-16 EA Insert New Element (Table) Dialog

In ArgoUML, the Properties tab in the Detail pane become actives for the new table (class).

4. 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 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*.

In ArgoUML, enter a class name in the **Name** field of the Properties tab according to the Java class naming conventions; i.e., class names should start with a capital letter, with embedded words capitalized. Next, select the **Data Model** package from the **Namespace** drop down (Figure 9-17).

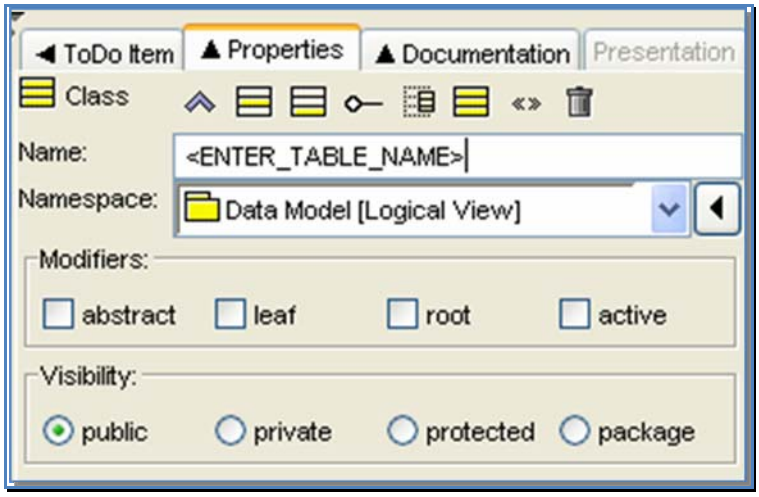


Figure 9-17 ArgoUML Table (Class) Properties Tab

Finally, click on the **Stereotype** tab, select the **table** stereotype, and apply it to the new class by clicking the **>>** icon (Figure 9-18).

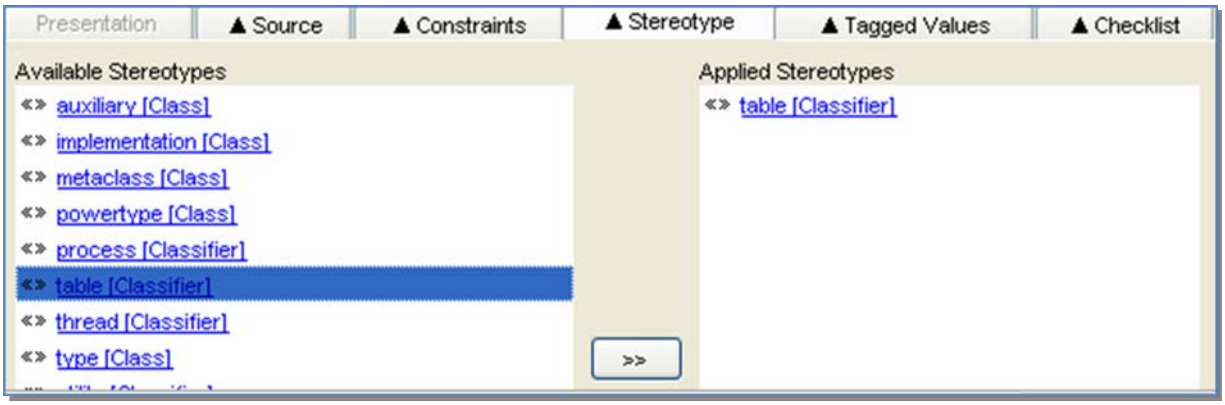


Figure 9-18 ArgoUML Applying a Stereotype to a Table Class

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

5. In EA, click **OK**. If the Open Property dialog was checked, the Property dialog opens immediately after the class is created. The Property dialog for the SDK sample *Credit* table is shown (Figure 9-19). Note that the Stereotype field is set to *table.*

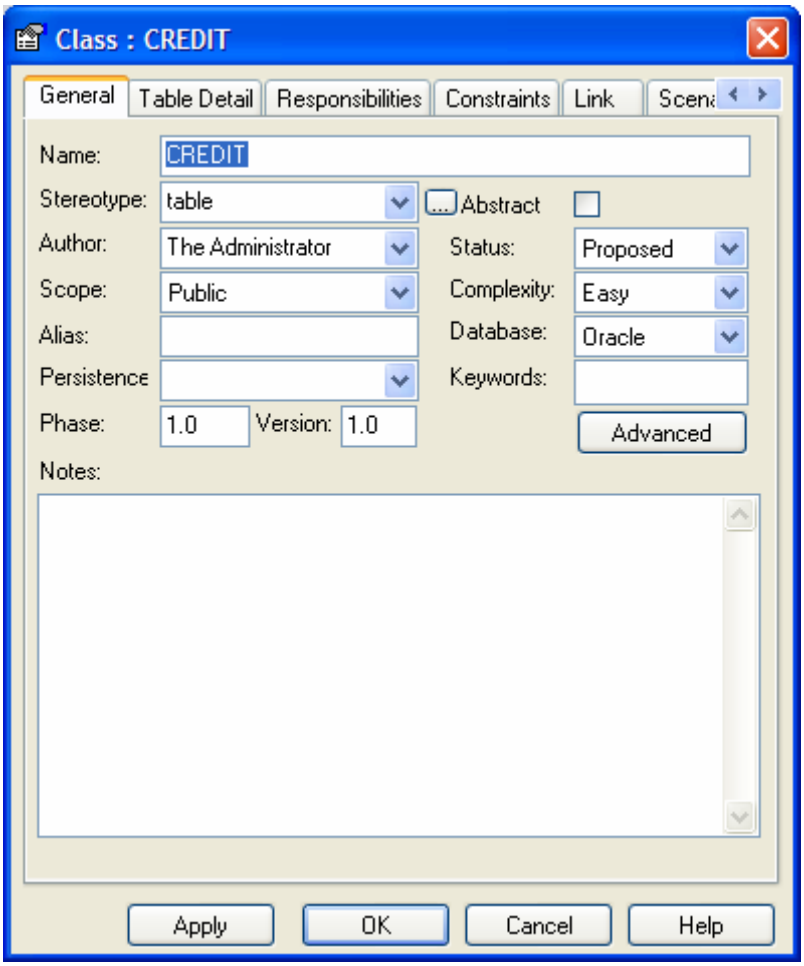


Figure 9-19 Property dialog for the SDK sample Credit

In ArgoUML, click **Save Project** (💾) or CTRL-S.

For instructions on adding attributes (columns) to tables, see *Creating Attributes and Data Types* on page 85.

6. Repeat steps 1-5 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 repeating steps 4-5. Figure 9-20 shows various tables that have been created in the *Data Model* package of the SDK sample model.

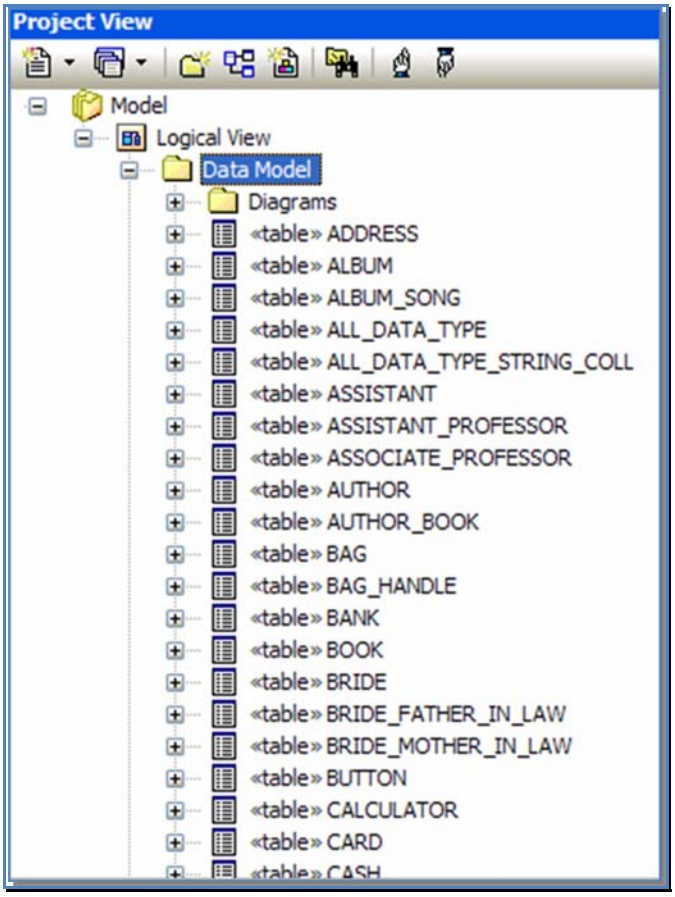


Figure 9-20 EA Various Tables from the SDK Sample Model

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.

To add/modify a class or table attribute, use the following steps.

1. Select the desired **Logical Model** class or **Data Model** table (class) element. Figure 9-21 shows the Logical Model **AllDataType** class from the SDK sample model selected.

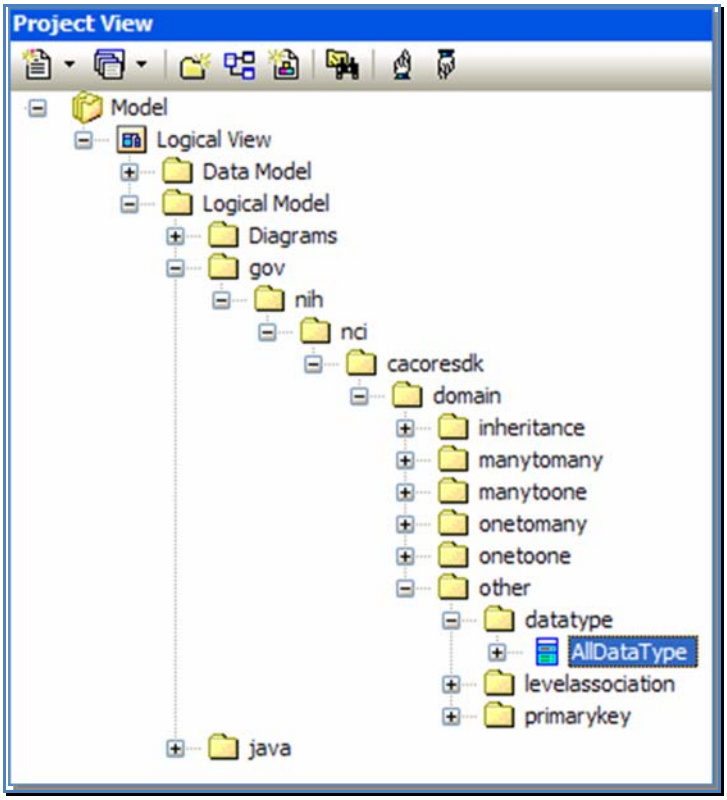


Figure 9-21 Logical Model AllDataType class

2. In EA, right click and select **Attributes** or select **Element > Attributes**.
In ArgoUML, the **Properties** tab in the Detail pane becomes active for the selected class. Click the **New Attribute** ( ) icon.

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

3. In EA, the **Attributes** dialog opens. Figure 9-22 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.

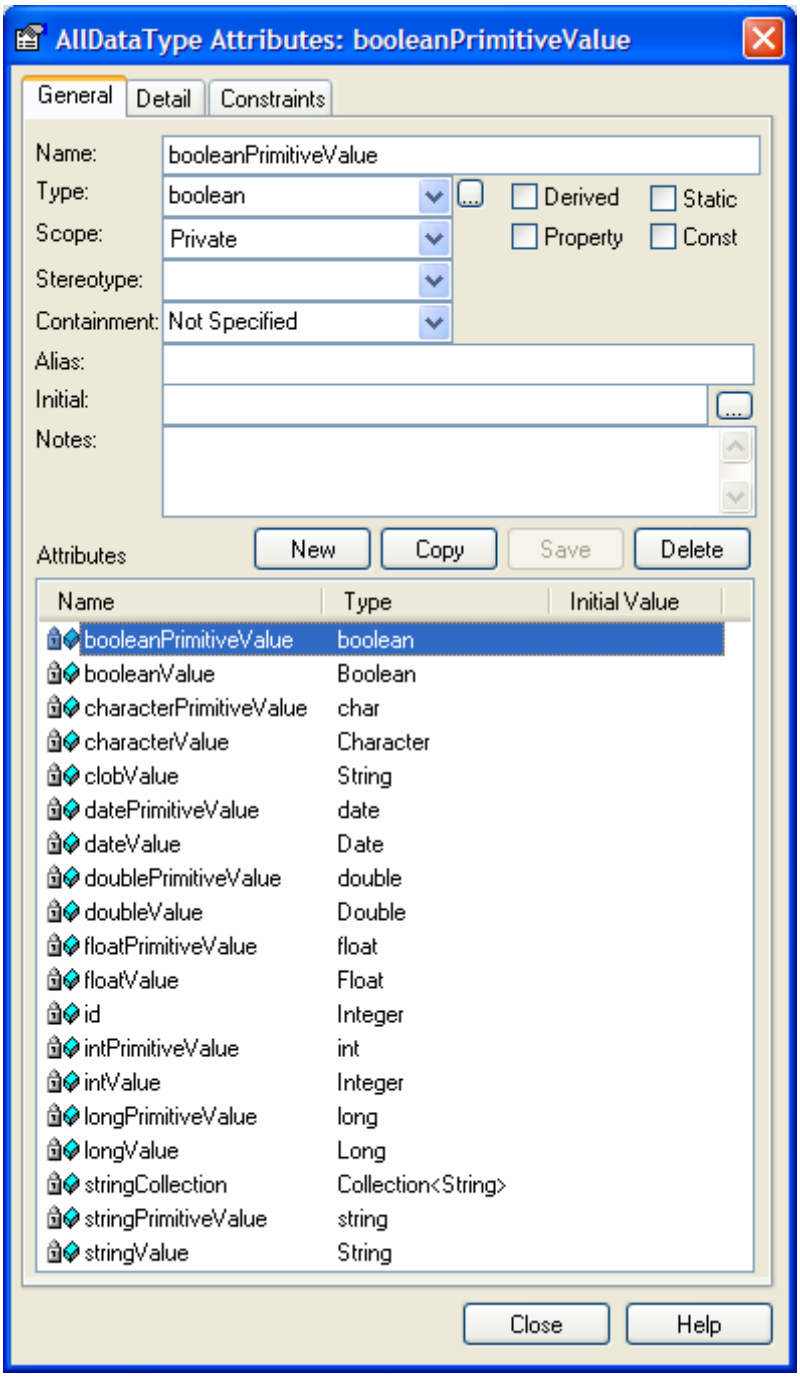


Figure 9-22 Sample AllDataType Class Showing Available Attribute Primitive Data Types

In ArgoUML, the **Attribute** properties tab becomes active in the Detail pane (Figure 9-23).

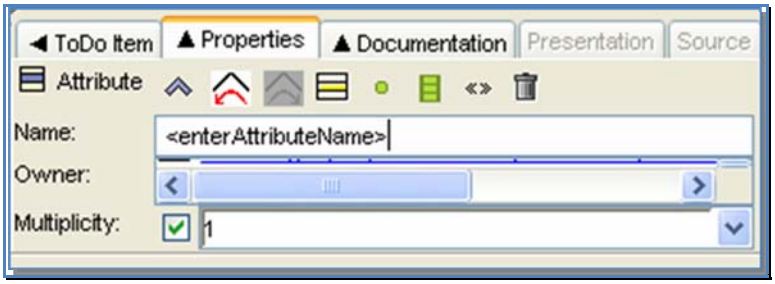


Figure 9-23 ArgoUML Attribute Properties Tab

4. In EA, *add* an attribute by clicking **New** and type an attribute name in the **Name** field. Select a type from the **Type** drop down and then click **Save**.

   Note: 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). In addition, 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, it can be entered (typed) 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.

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

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

   To *modify* an attribute in ArgoUML, select the attribute in the *Explorer* pane by expanding the class to show its attributes, or click on the attribute name within the class in the *Editing* pane, if working with a diagram. The *Attribute* properties tab will become active for the attribute. 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 (*Class*, *Attributes*, and *Associations*) directly. Tag Values, for instance, are used to hold class/attribute documentation (comments and/or descriptions) 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.*

For those who may find that working directly with the Tag Values may be too cumbersome or error prone, please reference the **caAdapter** tool, which, among other capabilities, 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

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

The following diagram (Figure 9-24) from the SDK sample model illustrates an association between the *Employee* and *Project* Logical Model classes.

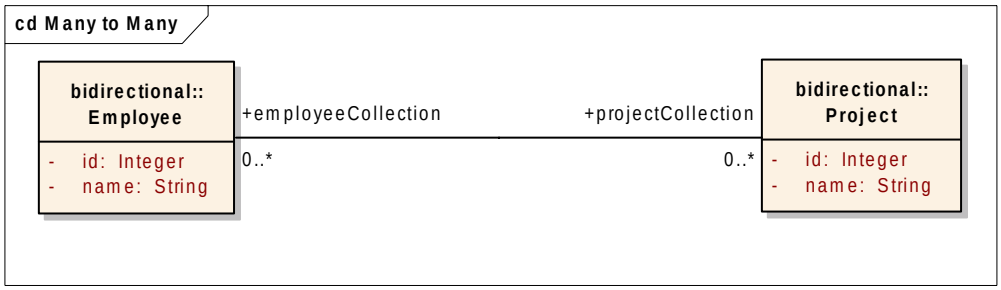


Figure 9-24 Employee-Project Asssociation Diagram

A sample of the Tag Value browser for the *Association* (line) between both classes is shown in (Figure 9-25).

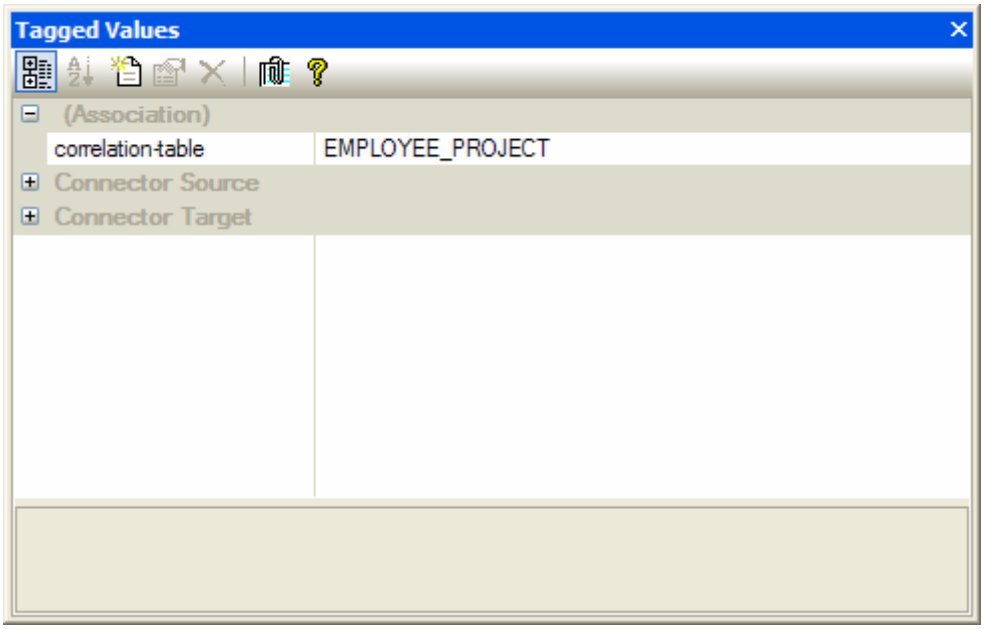


Figure 9-25 EA Tag Values Browser

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 will cause the *Detail* pane to become active for the selected element. Next, click on the *Tagged Values* tab to activate it.

A sample of the Tagged Values tab for the Association between the Sample SDK Employee and Project Logical Model classes is shown in Figure 9-26.

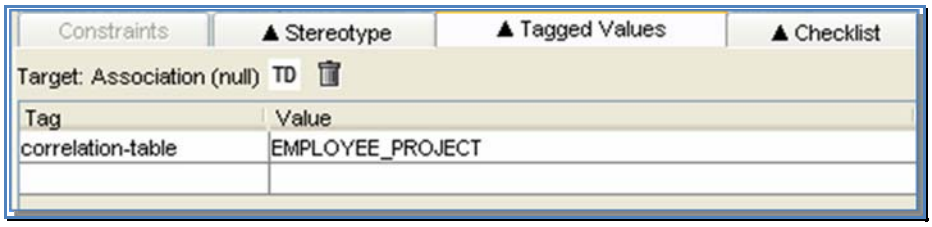


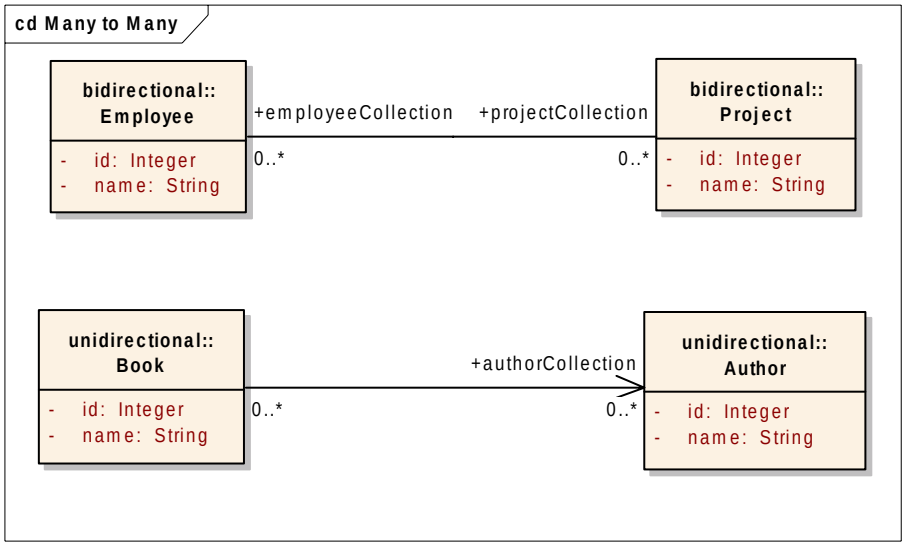
Figure 9-26 ArgoUML Detail Pane, Tagged Values Tab

SDK Custom Tag Value Descriptions

The following table (Table 9-1) lists the tag values recognized by the SDK Code Generator, and also describes when and where to use them.

<table>
<tr><th>Tag Value</th><th>Description</th></tr>
<tr><td>correlation-table</td><td>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:

A couple of corresponding examples from the SDK sample model are provided in the table below:
<table>
<tr><th>Logical Model Class (Source)</th><th>Logical Model Class (Target)</th><th>Tag Value (correlation-table) Note: should be added to the Association (line) element</th></tr>
<tr><td>Employee</td><td>Project</td><td>EMPLOYEE_PROJECT</td></tr>
<tr><td>Book</td><td>Author</td><td>AUTHOR_BOOK</td></tr>
</table>
</td></tr>
<tr><td>description</td><td>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.

Note: The description Tag Value is only used if the documentation tag value for the element is empty or does not exist.</td></tr>
</table>

<table>
<tr><th>Tag Value</th><th>Description</th></tr>
<tr><td>discriminator</td><td>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.

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.

A couple of examples from the SDK sample model is provided in the table below:
<table>
<tr><th>Data Model Class (Table)</th><th>Data Model Attribute (Column)</th><th>Tag Value (discriminator)</th></tr>
<tr><td>SHOES</td><td>DISCRIMINATOR</td><td>gov.nih.nci.cacoresdk.domain.inheritance.childwithassociation.sametable.Shoes</td></tr>
<tr><td>GOVERNMENT</td><td>DEMOCRATIC_ DISCRIMINATOR</td><td>gov.nih.nci.cacoresdk.domain.inheritance.twolevelinheritance.sametable.DemocraticGovt</td></tr>
</table>
</td></tr>
<tr><td>documentation</td><td>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.

See also the documentation Tag Value.</td></tr>
<tr><td>id-attribute</td><td>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.

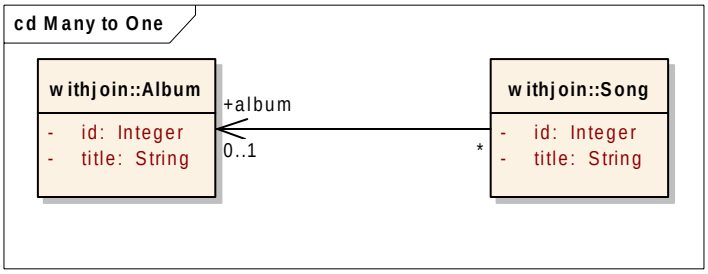
An example from the SDK sample model is provided in the table below:
<table>
<tr><th>Logical Model Class Name</th><th>Logical Model Attribute Name</th><th>Tag Value (id-attribute)</th></tr>
<tr><td>NoIdKey</td><td>myKey</td><td>gov.nih.nci.cacoresdk.domain.other.primarykey.NoIdKey</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Tag Value</th><th>Description</th></tr>
<tr><td>implements-association</td><td>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>.

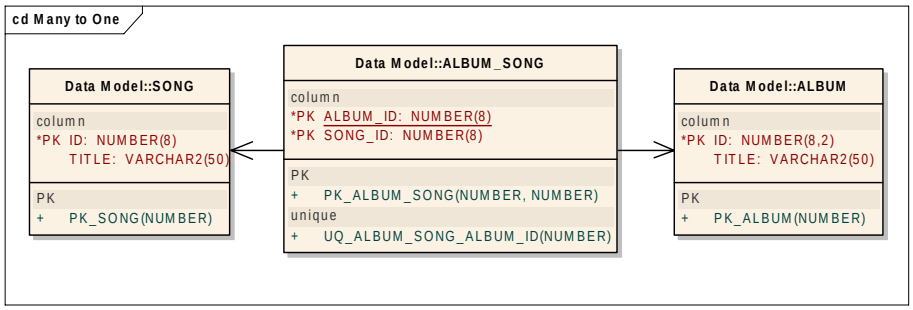
A couple of examples from the SDK sample model are provided in the table below:

<table>
<tr><th>Data Model Class (Table)</th><th>Data Model Attribute (Column)</th><th>Tag Value (implements-association)</th></tr>
<tr><td>CARD</td><td>SUIT_ID</td><td>gov.nih.nci.cacoresdk.domain.other.level association.Card.suit</td></tr>
<tr><td>ASSISTANT</td><td>PROFESSOR _ID</td><td>gov.nih.nci.cacoresdk.domain.inheritance .parentwithassociation.Assistant. professor</td></tr>
</table>
</td></tr>
<tr><td>inverse-of</td><td>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:

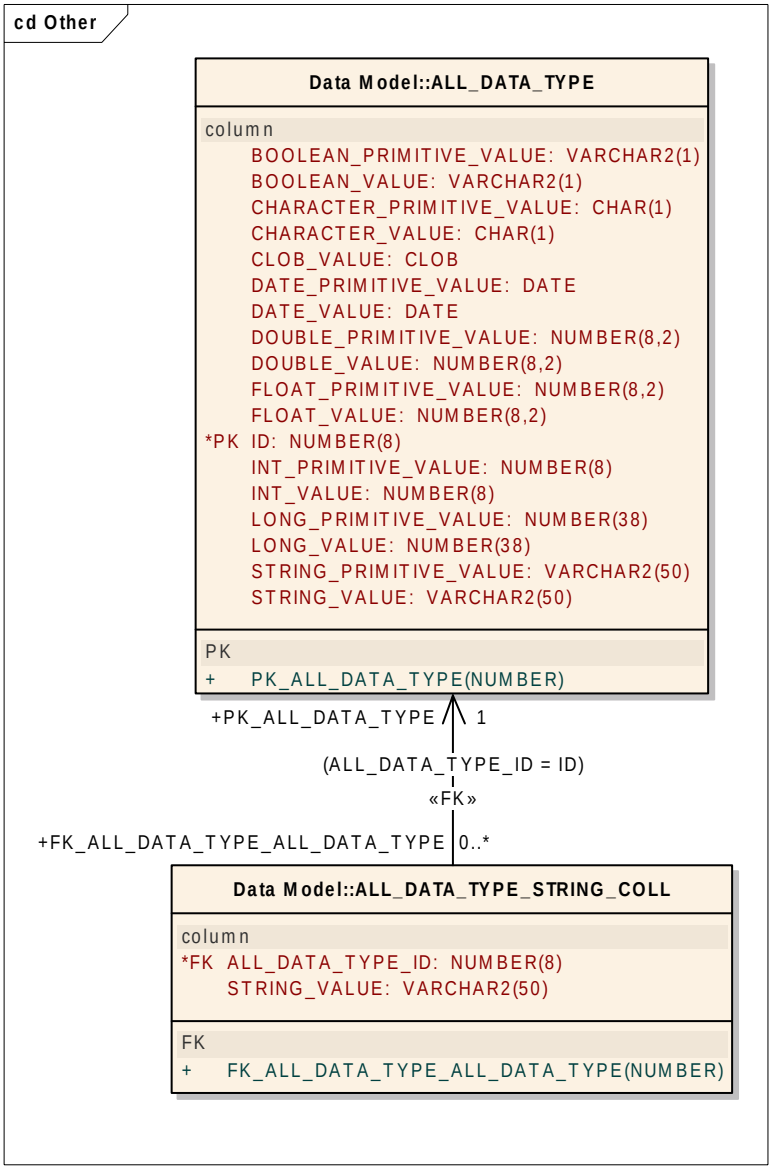
And the corresponding Data Model class diagram:

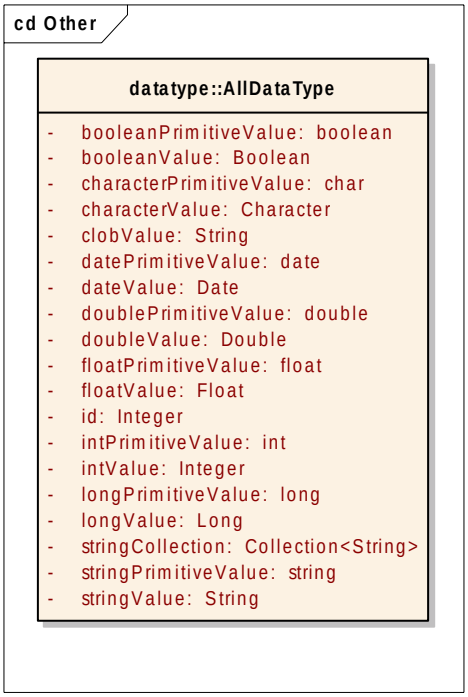
An example inverse-of Tag Value would be:</td></tr>
</table>

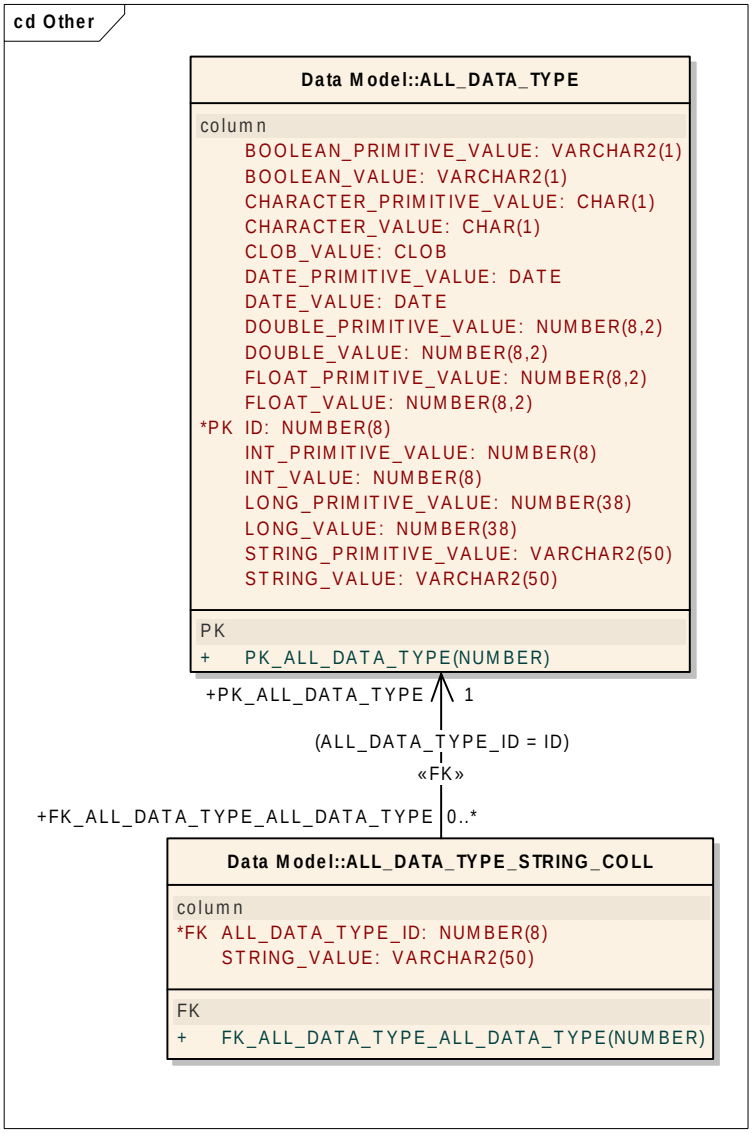
<table>
<tr><th>Tag Value</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Data Model Class (Table)</th><th>Data Model Attribute (column)</th><th>Tag Value (inverse-of)</th></tr>
<tr><td>ALBUM_ SONG</td><td>SONG_ID</td><td>gov.nih.nci.cacoresdk.domain.manytoone.unidirectional.withjoin.Song.album</td></tr>
</table>
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.
</td></tr>
<tr><td>lazy-load</td><td>
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. 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.
</td></tr>
<tr><td>mapped-attributes</td><td>
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:
<table>
<tr><th>Data Model Class (Table)</th><th>Data Model Attribute (column)</th><th>Tag Value (mapped-attributes)</th></tr>
<tr><td>UNDERGRADUATE_ STUDENT</td><td>STUDENT_ID</td><td>gov.nih.nci.cacoresdk.domain.inheritance.multiplechild.UndergraduateStudent.id</td></tr>
<tr><td>SHOES</td><td>ID</td><td>gov.nih.nci.cacoresdk.domain.inheritance.childwithassociation.sametable.Shoes.id</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Tag Value</th><th>Description</th></tr>
<tr><td>mapped-collection-table</td><td>
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:


An example mapped-collection-table Tag Value would be:
<table>
<tr><th>Logical Model Class</th><th>Logical Model Attribute</th><th>Tag Value (mapped-collection-table)</th></tr>
<tr><td>AllDataType</td><td>stringCollection</td><td>ALL_DATA_TYPE_STRING_COLL</td></tr>
</table>
</td></tr>
</table>

Tag Value	*Description*
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:  And the following Data Model class diagram from the SDK sample model:

Tag Value	*Description*
	 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

<table>
<tr><th>Tag Value</th><th>Description</th></tr>
<tr><td>type</td><td>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:
<table>
<tr><th>Data Model Class (Table)</th><th>Data Model Attribute (Column)</th><th>Tag Value (mapped-element)</th></tr>
<tr><td>CHARACTER_ PRIMITIVE_KEY</td><td>ID</td><td>CHAR</td></tr>
<tr><td>CARD</td><td>IMAGE</td><td>CLOB</td></tr>
<tr><td>UNDERGRADUATE_STUDENT</td><td>STUDENT_ID</td><td>NUMBER</td></tr>
<tr><td>SHOES</td><td>COLOR</td><td>VARCHAR2</td></tr>
</table></td></tr>
</table>

Table 9-1 Tag values recognized by the SDK Code Generator

Exporting the UML Model to XMI (EA Only)

Note: This section only applies to EA, as the ArgoUML project is stored in an XML format that the SDK Code Generator can understand and process directly; i.e., the ArgoUML project file does *not* need to be exporting to XMI prior to processing it via the SDK Code Generator.

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

To export a package to XMI, use the following steps.

1. In the EA Project Browser, select the **Logical View** package (Figure 9-27).

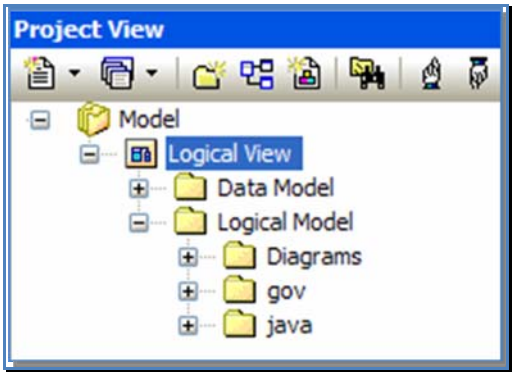


Figure 9-27 EA Logical View Package

2. Right click and select **Import/Export** or select **Project > Import/Export**. Select the **Export Package to XMI**.
3. The Export Package to XMI dialog displays (Figure 9-28).

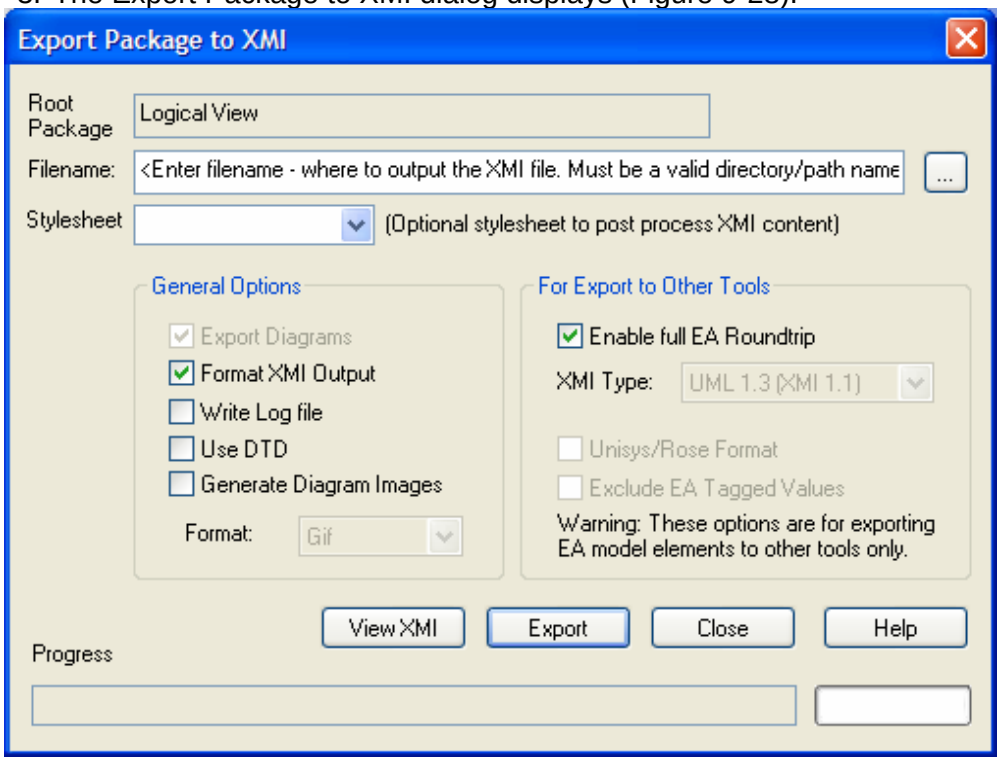


Figure 9-28 EA Exporting Package to XMI

4. Set the Export options according to those in Table 9-2.

Note: The XMI export options that should be selected have changed for SDK 4.0. The new required options are:

- Export Diagrams
- Enable full EA Roundtrip

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.*
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 9-2 EA Export options

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

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.

Importing XMI into the UML Model (EA Only)

Note: This section only applies to EA, as the ArgoUML project is stored in an XML format that the SDK Code Generator can understand and process directly; i.e., the ArgoUML project file does *not* need to be exporting to XMI prior to processing it via the SDK Code Generator.

The SDK now supports the processing of XMI 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.

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, use the following steps.

1. In the EA Project Browser, select the **Logical View** package (Figure 9-29).

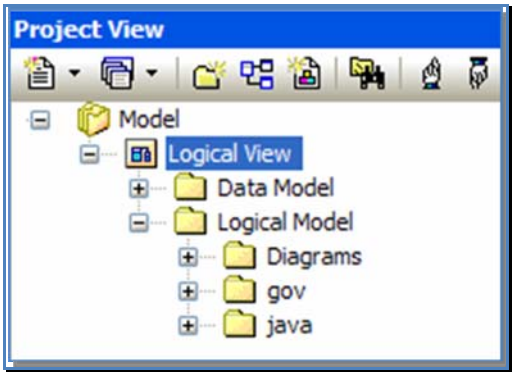


Figure 9-29 EA Logical View Package

2. Right click and select **Import/Export** or select **Project > Import/Export**. Select **Import Package from XMI**.
3. The Import Package from XMI dialog opens (Figure 9-30).

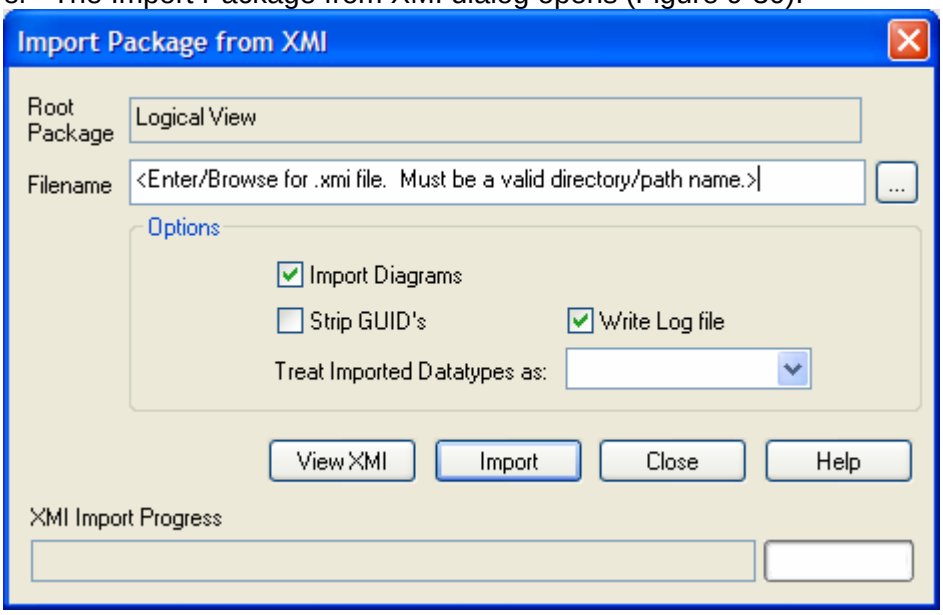


Figure 9-30 EA Import Package from XMI

4. Set the Import options according to those in Table 9-3.

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.*

Import Option	Description
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 9-3 Import options

5. Click **Import**. A confirmation dialog opens (Figure 9-31).

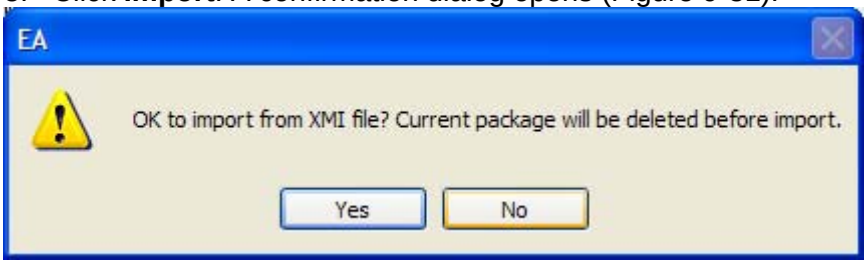


Figure 9-31 EA Confirm XMI File Import Dialog

6. Click **Yes**. The XMI file is imported back into EA and the XMI and UML model are synchronized.

Chapter 10 Configuring and Running the SDK

This chapter describes how to configure the SDK Code Generator and generate the SDK system.

Topics in this chapter include:

SDK Configuration Properties

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

The following table (Table 10-1) describes each of the properties (and their values) found within this file.

Property	*Default Value*	*Description*
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.
SECURITY_ENABLED	False	Used to enable or disable security within the generated system during code generation. This applies to all of the SDK interfaces,

Property	Default Value	Description
		including: • Web Interface (GUI) • Java API Interface (local and remote clients) • Web Service Interface
CSM_PROJECT_NAME	Sdk	Used as a prefix when creating the CSM security configuration file name. CSM configuration should have the same application name configured
INSTANCE_LEVEL _SECURITY	False	Used to enable/disable CSM instance level security. Only relevant if the SECURITY_ENABLED property is set to 'true'
ATTRIBUTE_LEVEL _SECURITY	False	Used to enable/disable attribute level security. Only relevant if the SECURITY_ENABLED property is set to 'true'
WEBSERVICE_NAME	${PROJECT_NAME}Service	The name of the Web Service.
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.
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_FILE	sdk.xmi	The name of the file that contains the object/data model to be processed. SDK users should modify this property to reflect their own model file name. The file is to 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

Property	***Default Value***	***Description***
		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	Used to determine which packages within the model should be included during code generation; i.e., all packages containing this property value, as part of their fully qualified name will be included during code processing.
EXCLUDE_PACKAGE	None	Used to determine which packages within the model should be excluded during code generation; i.e., all packages containing this property value, as part of their fully qualified name will be excluded during code processing. **Note:** All packages are first filtered/constrained by the INCLUDE_PACKAGE property value, and then furthered filtered by the EXCLUDE_PACKAGE value.
EXCLUDE_NAME	None	Used to determine which classes within the model should be excluded during code generation; i.e., all classes containing this property value, as part of their non-fully qualified name will be excluded during code processing.
USE_JNDI_BASED _CONNECTION	No	Indicates whether a JNDI DB Connection should be used for the application database. If USE_JNDI_BASED_CONNECTION=yes, then the DB_JNDI_URL property value is used to obtain the DB connection and retrieve 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	None	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

Property	Default Value	Description
		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
CSM_USE_JNDI_BASED_CONNECTION	No	Indicates whether a JNDI DB connection should be used for the CSM database. If USE_JNDI_BASED_CONNECTION=yes, then the DB_JNDI_URL property value is used to obtain the DB connection and retrieve data.
CSM_DB_JNDI_URL	java:/SDK	The DB JNDI URL value for the CSM database. This property is irrelevant/ignored if CSM_USE_JNDI_BASED_CONNECTION=no.
CSM_DB_CONNECTION_URL CSM_DB_USERNAME CSM_DB_PASSWORD	None	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.
CSM_DB_DIALECT	org.hibernate.dialect.OracleDialect	The Hibernate Database dialect used when connecting to the CSM database. Typical values include: • org.hibernate.dialect.OracleDialect • org.hibernate.dialect.MySQLDialect
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.
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

Property	Default Value	Description
		unmarshalling mapping files.
GENERATE_XSD	True	Used to enable/disable the generation of the XML Schemas (XSDs).
GENERATE_WSDD	True	Used to enable/disable the generation of the Axis Web Service Deployment Descriptor (WSDD) file.
INCLUDE_SEARCH _EVENT_LISTENER	False	Used to toggle whether or not Event Listeners for the Hibernate Search API will be generated in the Hibernate Configuration file (*hibernate.config.xml*).
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.

Table 10-1 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 *SDK Configuration Properties* on page 103 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 *SDK Configuration Properties* on page 103. Please review and update these properties as needed to reflect your environment prior to generating the system.

For those interested in the remaining targets, the table below (Table 10-2) 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

Ant Target	Description
	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
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 • GENERATE_HIBERNATE_MAPPING • GENERATE_BEANS • GENERATE_CASTOR_MAPPING • GENERATE_XSD • GENERATE_WSDD See *SDK Configuration Properties* on page 103 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.
compile-beans	Compiles the generated beans. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
doc	Generates Javadocs for the generated beans. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
init	An internal target that prepares the output directory structure. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
package-codegen-artifacts	Packages (jars) the generated Hibernate ORM and Java bean artifacts. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
package-local-client	An internal target that prepares, packages the local client files. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
package-release-contents	Creates release binary and source packages in a zip file format for distribution. This target is typically run by the SDK team when creating an SDK release for distribution.
package-remote-client	An internal target that prepares, packages the remote client files. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
package-system	Packages the system. Internally runs the following targets: • package-remote-client • package-local-client • package-webapp • package-ws-client This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
package-webapp	An internal target that prepares, packages the Web application Archive (war) file for deployment to a Servlet container such as JBoss or

Ant Target	*Description*
	Tomcat. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
package-ws-client	An internal target that prepares, packages the Web Service client. This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually.
prepare prepare-codegen prepare-condition-codegen prepare-condition-system prepare-system	An internal target that runs the following targets: • prepare-codegen • prepare-system This target is run as part of the 'build-system' process. SDK users should rarely, if ever need to invoke this target individually
show-properties	Dumps a list of all currently set properties.

Table 10-2 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' will cause the component to be generated; conversely, setting it to 'false' will cause the component to be ignored. See *SDK Configuration Properties* on page 103 for more information.

Overview of Generated Packages

During the code generation process, 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 1) the Java API interface 2) the XML marshalling and unmarshalling and 3) 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 a <project_name>.war file generated by the SDK, which 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.

Deploying the Generated System

The Ant build process packages the generated SDK system 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 guarantees are made.

Deploying to JBoss

If the generated system war file is to be deployed to a JBoss server instance, the SERVER_TYPE property found 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 a JBoss server instance, the SERVER_TYPE property found in the file `\conf\deploy.properties` should be set to 'jboss'. This will ensure that the log4j.jar file is excluded from the packaged war file. 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 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 10-1should display.

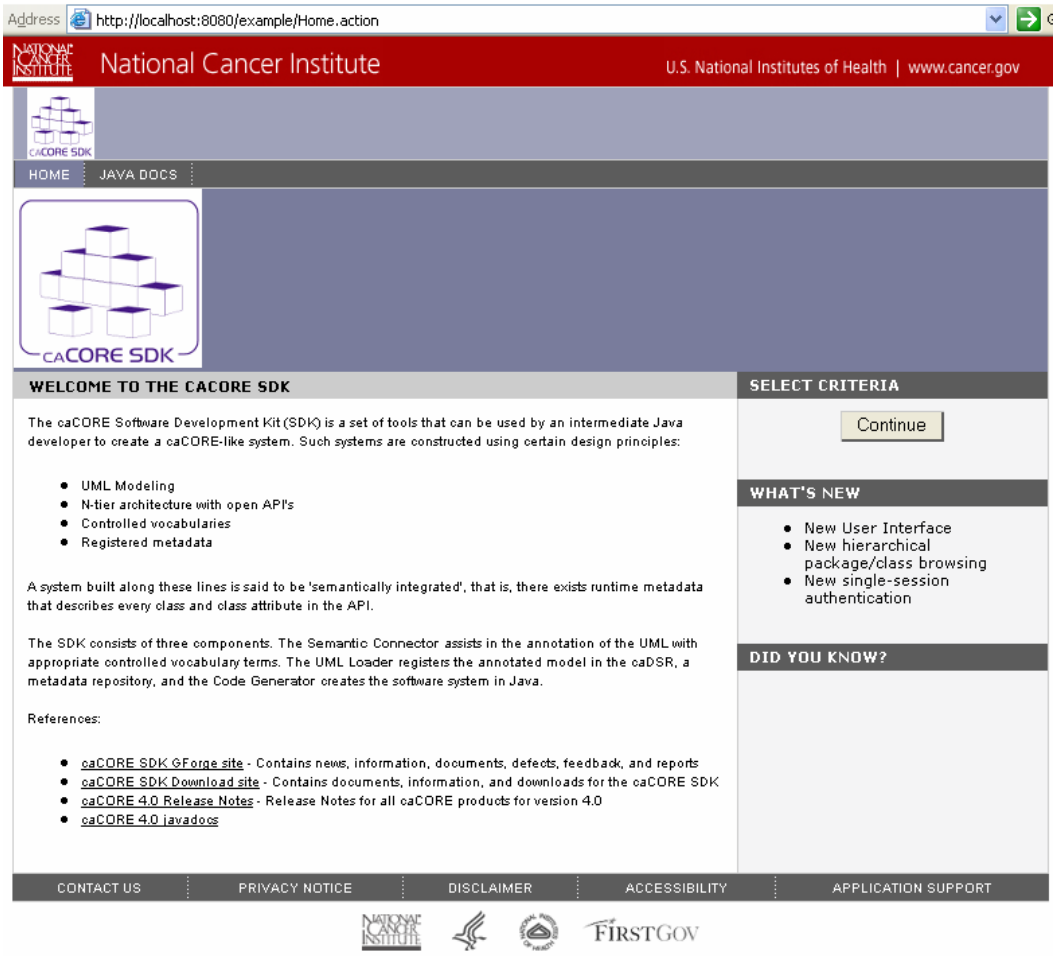


Figure 10-1 Web Interface test page

For more information, see *Accessing Data from a Web Browser* on page 33.

Testing the Java API

The program, *TestClient.java*, is provided with the SDK distribution for testing the Java API. This program is located within the folder `\output\<project_name>\package\remote-client\src\`. To execute the program, run the default target of the Ant script, `build.xml`, located within 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 10-2 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 10-2 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 that, 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, it can be modified to use a different method within the Application Service API, or the return results filtered by adding criteria data to the search object prior to the search.

See *Java API Interface* on page 39 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 10-3 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 10-3 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 in, parses the XML, and validates it against the generated schema. Finally, it unmarshal (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 that, a client for testing the XML Utility. It provides only one example of how the XML Utility marshalling/unmarshalling methods may be invoked. It can be modified to use a different method if so desired.

In addition, the same search algorithm used during the testing of the Java API is reused here. See *Testing the Java API* on page 112 for more information.

Finally, 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 much 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 10-4 illustrates the result of a successful Web Service deployment test.

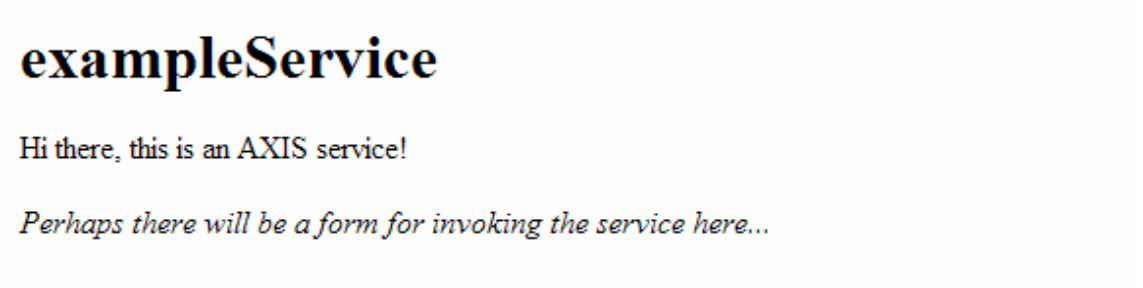


Figure 10-4 Web Service test page

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 in the previous section, entering the Web Service URL in a browser causes a message to be displayed 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 10-5 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 10-5 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 10-6 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 10-6 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 a couple of the Web Service operations defined within the WSDL.

Note: The Web Service program *TestClient.java* is simply that, 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. It can be modified to use a different operation or algorithm if so desired.

In addition, the same search algorithm used during the testing of the Java API's is re-used within the test program. See *Testing the Java API* on page 112 for more information. See also the *Web Service Interface* on page 52 for more information.

Configuring Security

Security in SDK is provided by ACEGI and CSM. Security can be configured in three steps.

1. Configure security related properties by altering the configuration parameters in the file `deploy.properties` before generating the system.
2. Configure the application server for JAAS based authentication configuration.
3. Setup the CSM database configuration for an SDK-based application.

The following table (Table 10-3) shows the properties that must be modified in order to correctly enable the security in SDK.

Property Name	Default Value	Description
SECURITY_ENABLED	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) • Web Service Interface
CSM_PROJECT_NAME	Sdk	Used as a prefix when creating the CSM security configuration file name. CSM configuration should have the same application name configured
INSTANCE_LEVEL _SECURITY	False	Used to enable/disable CSM instance level security. Only relevant if the SECURITY_ENABLED property is set to 'true'
ATTRIBUTE_LEVEL _SECURITY	False	Used to enable/disable attribute level security. Only relevant if the SECURITY_ENABLED property is set to 'true'
CSM_USE_JNDI _BASED_CONNECTION	No	Indicates whether a JNDI DB connection should be used for the CSM database. If USE_JNDI_BASED_CONNECTION=yes, then the DB_JNDI_URL property value is used to obtain the DB connection and retrieve data
CSM_DB_JNDI_URL	java:/SDK	The DB JNDI URL value for the CSM database. This property is irrelevant/ignored if CSM_USE_JNDI_BASED_CONNECTION=no
CSM_DB_CONNECTION _URL CSM_DB_USERNAME CSM_DB_PASSWORD	*None*	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.
CSM_DB_DIALECT	org.hibernate.dialect.OracleDialect	The Hibernate Database dialect used when connecting to the CSM database. Typical values include: • org.hibernate.dialect.OracleDialect • org.hibernate.dialect.MySQLDialect

Table 10-3 Security properties

JAAS-Based Authentication Configuration

Applications dependent on JAAS-based (http://java.sun.com/j2se/1.4.2/docs/guide/security/jaas/tutorials/GeneralAcnOnly.html) login can configure their login procedure in several ways. 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 10-7 provides an example of how to configure JAAS-based authentication in a JBoss server. See the CSM Developer's 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 10-7 Configuring JAAS-based 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.

Configuring CSM

caCORE SDK 4.0 requires CSM 4.0 for security. SDK users must setup a CSM database schema and configure it with CSM's User Provisioning Tool (UPT).

Note: If you are planning to use instance level security then you are required to put CSM tables on the same database schema where the tables for domain classes reside. See the CSM Developer's Guide for more information on installing CSM on a particular database and using the User Provisioning Tool (UPT) for configuring the security schema.

While configuring the security schema with the UPT, a Protection Element is required to be created for each domain object in the SDK generated system. The Protection Element should have the name as fully qualified name of the domain object. The SDK Security implementation uses the name of the domain object as a key to be searched in the CSM configuration to determine access privileges. Figure 10-8 demonstrates the addition of the *StringKey* class as a protection element in the CSM.

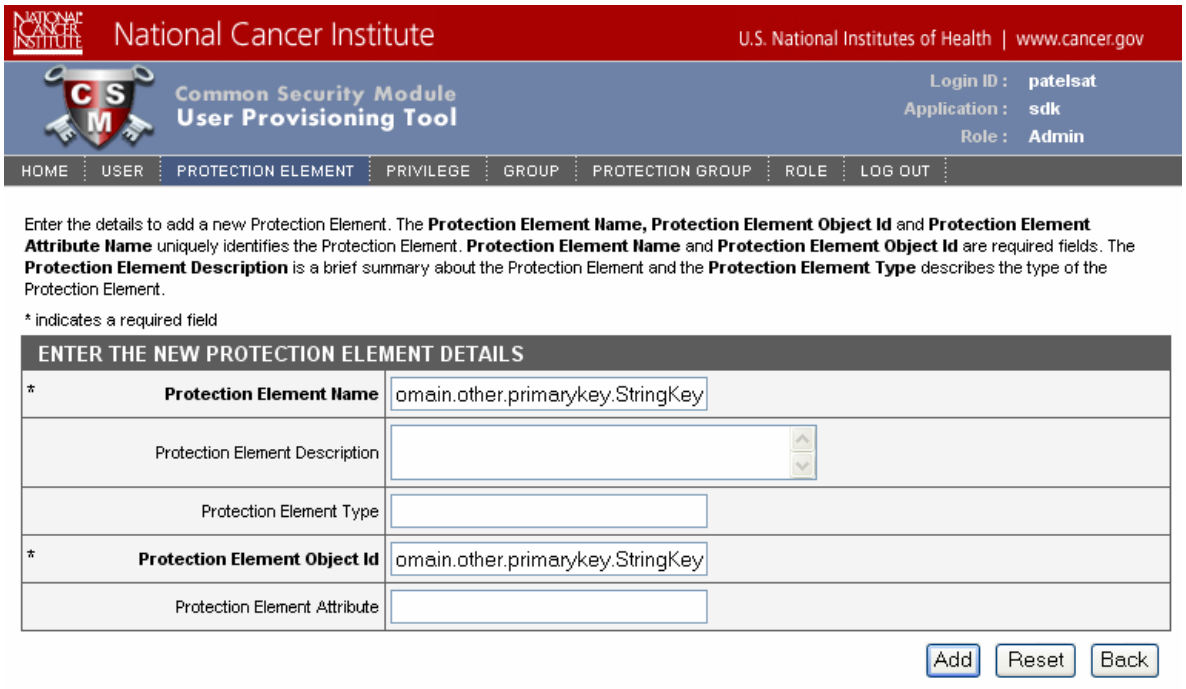


Figure 10-8 StringKey class as protection element in CSM

Once all the protection elements are created, users and user groups can be created and assigned READ privileges to appropriate protection elements based on the security needs of the application.

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. Chapter 7 includes information on optimizing the performance of the Java API.

Appendix B 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.

Grid Integration – The latest release of SDK is not compatible with caGrid 1.0 or 1.1. The SDK team will be working with the caGrid team to integrate the latest release of the SDK with a future version of caGrid.

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 to edit the records

Writable API – The current version of the SDK provides a read only API for the domain model. The previous release of the SDK did include a primitive version of a writable API but is not being released with the new SDK. The SDK team is planning to develop a complete working solution for the writable API, which is applicable to the requirements of most users.

Appendix C 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. The example model is available under the models directory of the release package and is available for Enterprise Architect (SDKTestModel.eap) and ArgoUML (sdk.uml). Users can easily refer to these models, which are organized in a self-explanatory fashion.

The current version of the example model includes the scenarios in the following tables.

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 class

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 section *0 Ant Build Script Targets* 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

Term	Definition
	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.
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

S

T

U

W

X