realMethods Framework

Architecture Guide

realMethods Framework Architecture Guide

http://www.realmethods.com

realMethods Framework Architecture Guide
Copyright © 2001-2017 applied to realMethods, Inc. including this documentation, all demonstrations, and all software. All rights reserved. The document is furnished as is without warranty of any kind, and is not intended for use in production. All warranties on this document are hereby disclaimed including the warranties of usability and for any purpose.

Trademarks
"realMethods Framework" is a trademark of realMethods in the United States and other countries. Microsoft, Windows, Windows NT, the Windows logo, and IIS are trademarks or registered trademarks of Microsoft Corporation in the United States, other countries, or both. UNIX is a registered trademark of The Open Group in the United States and other countries. Sun, Solaris, Java, Enterprise Java Beans, EJB, J2EE, and all Java-based trademarks or logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both. MDA®, Model Driven Architecture® and XMI® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, UML™, Unified Modeling Language™, MOF™, and CWM™ are trademarks of the Object Management Group. IBM and MQ Series are trademarks or registered trademarks of International Business Machines Corp. in the United States and other countries.

Copyright (C) 1999 - The Apache Software Foundation. All rights reserved.
This product includes software developed by the Apache Software Foundation. (http://www.apache.org/).

Disclaimer
The use of the software developed by the Apache Software Foundation is only for the purpose of demonstrating integration with Jakarta Struts, Log4J, Hibernate, and Velocity.

Other company, product, and service names mentioned in this document may be trademarks or service marks of others.

realMethods model driven; pattern based

This document completely describes the architecture related aspects that comprise the realMethods Framework. To better describe certain parts of the framework, UML diagrams are included. The document provides you with a general overview of the MDA and design pattern support provided by the framework, UML considerations, and tier-by-tier descriptions. Use the Development Guide to learn how to generate, configure, build, deploy, and execute your application, as well as deployment and execution of the Proof of Concept application bundled with the install.

Document Naming Conventions

The following words and acronyms are used throughout this document:

Framework – realMethods Framework

RM_HOME – framework installation root directory location

AIB – Application Infrastructure Builder (code generator)

Patterns – refers to the core J2EE Design Patterns

MDA – refers to the OMG specification for Model Driven Architectures

XMI – refers to XML Meta-Data Interchange

UML – refers to Unified Modeling Language

Important Notes:

Note 1: Sun's Core J2EE Design Patterns have changed the naming convention of a business entity from Value Object to Business Object. Although the two can be used interchangeably, the Framework's code and your generated application code use the *BusinessObject* naming. Some of the contents of this document may, however, still refer to the term *VavlueObject* for certain diagrams and examples.

General Overview of the Framework's Architecture 6
The Importance of Supporting MDA 7
Benefits of MDA 7
realMethods Adherence of MDA 7
Core J2EE Design Patterns 8
Code Generation (AIB) 9

Presentation Tier Overview 10
Action Classes 10
Tag Library Usage 10
AJAX 11
Javascript 11
HTML Form Validation 11

Business Tier Overview 12
Business Delegate 12
Business Delegate to DAO Directly 12
Business Delegate to Session Façade via RMI 12
Web Services 14
Business Object Versioning 15
Business Object Notification Service 16

Integration Tier Overview 17
Data Access Objects (DAOs) 17
Mule ESB Integration 19
Security Management Overview 20
Loading and Creating Security Managers 20
Framework JAAS Security Manager 21
Framework ACL Security Manager 23
Log Handlers and Definitions 25
Log Output Destinations 25
Framework Log Handler and the Command/Task Architecture 26
Asynchronous Logging Sequence Diagram 27
Synchronous Logging Sequence Diagram 28
Framework Connection Pooling 30
Mandatory Framework JMS Connections 33
Framework Internally Logging 34
Available Cache Implementations 35
Automatically Purge with a Framework Cache Monitor 36
Using a Framework Cache Monitor 37
Enabling Caching within the DAOs 38

Application Re-factoring Options to Consider 39
Scenario #1 Presentation Tier to Business Delegate to DAOs 39
Scenario #2 Business Delegate to Session Façade to DAO 40

Startup Scenarios 41
Implicitly starting the framework 41
Startup via Framework based Servlet 42
Valid configuration file locations 43
1. WAR file WEB-INF/properties Directory 43
2. Classpath 43
3. -DFRAMEWORK_HOME 43
Specifying the location of the framework.license file 43

Configuration File Loading Process 44

Service Locator Overview 46

Checked Exception Extensions 48

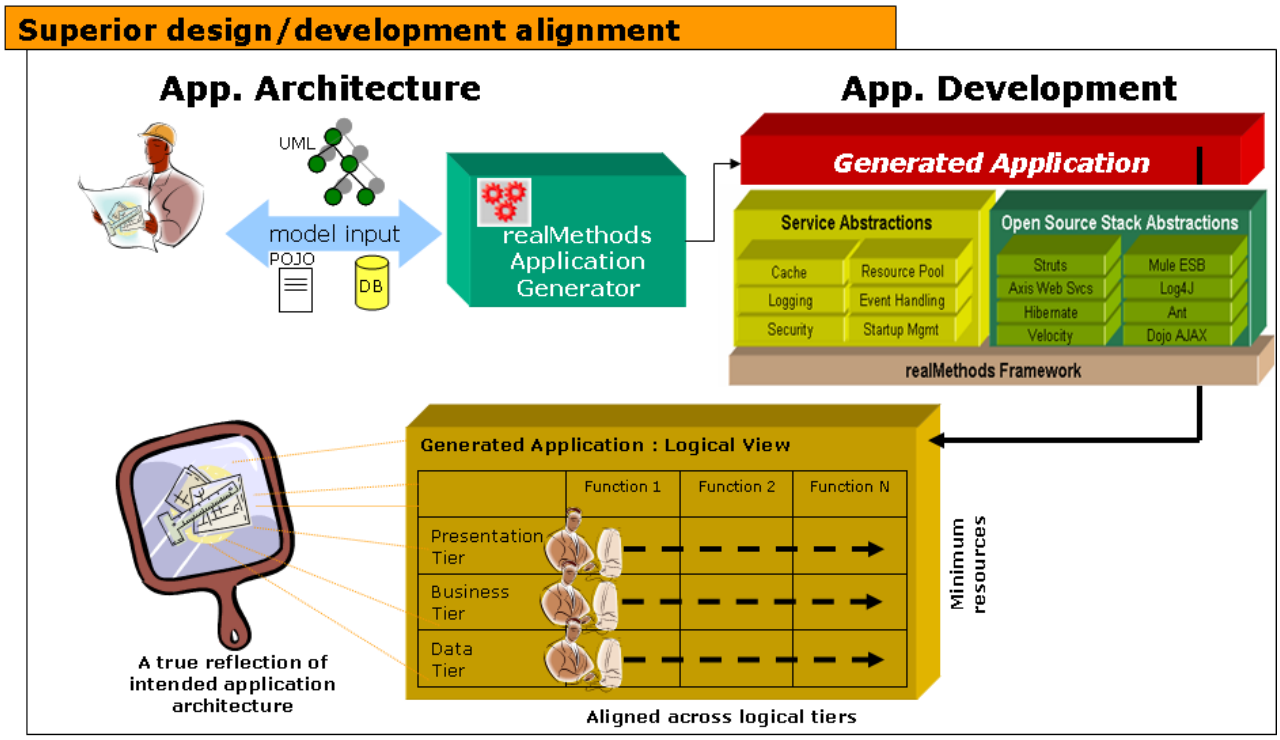
Observing / Configuring an Executing Framework Instance 50
Framework components defined as Dynamic MBeans 51
Registering Framework JMX MBeans 51
Specifying the JMX Server Factory to use 51
How an MBean Server Domain is specified 52
3rd Party JMX Management Console - XtremeJ 52

Unique Identifier Generation 53
Default implementation defined by the framework 53
Defining and assigning a customer UID generator 53

Before using the AIB (code generator) 54
Creating the domain model 54
Declaring attributes in the model 55
Defining associations in the model 57
Inheritance considerations when modeling 59
Defining One or More *findAllBy* Methods 61
Exporting the domain model to an XMI file 62
Exporting the domain model using Rational Rose 62
Exporting the domain model using Together/J 63
Using the Velocity Template Engine with the AIB 64
Existing Templates 64
AIB Velocity Variables 64
Creating New Templates 65
Configuration 65
Changing Default Package Names 66

General Overview of the Framework's Architecture

This section describes the key concepts behind the design and implementation strategies employed within the framework. It begins by explaining the importance of MDA within a framework, continues to define the supported core J2EE patterns, ad concludes with a description of the AIB.



realMethods Aligns Architecture and Development

The Importance of Supporting MDA

MDA is a suite of OMG established standards that provide an open, vendor-neutral approach to the change of business and technology change. The Key standards that make up the MDA suite of standards include Unified Modeling Language (UML); Meta-Object Facility (MOF); XML Meta-Data Interchange (XMI); and Common Warehouse Meta-model (CWM). MDA's goal is to separate business logic from the underlying platform technology.

Benefits of MDA

Forces designers to focus on the business domain model, rather than how to implement that model using J2EE.

- The domain model is no longer to be viewed as a "guideline" in development, but the actual driving force, serving as the basis for development, maintenance and application evolution.
- MDA ensures the generated application model reflects the business requirements defined by the domain model. It also ensures that non-business functional requirements (such as scalability, security, etc.) carry through as well.
- PIM - Platform Independent Models greatly shield an application from upgrades and changes to the targeted technology platform, in this case J2EE.

realMethods Adherence of MDA

To initially engage the framework, development efforts are forced to create an XMI file. This XMI file is the result of constructing a domain model with any popular UML tool, and then exporting to the supported XMI format. This first step is intentional and facilitates the most complete manner to share the business problem with the AIB.

As described by MDA, development is based around a platform independent model (PIM) captured in UML using tools such as Rational Rose, Magic Draw or any other XMI compliant UML modeling tools. Platform specific J2EE patterns and best practices are then used to generate an application according to the targeted tiers.

Core J2EE Design Patterns

In general, a pattern provides a time proven solution to a given problem. The core J2EE patterns provide a common language and set of cohesive ideologies to apply to a business problem, using the J2EE technologies. With respect to an n-tiered J2EE framework, it must provide an implementation within the spirit and intent of the supported patterns.

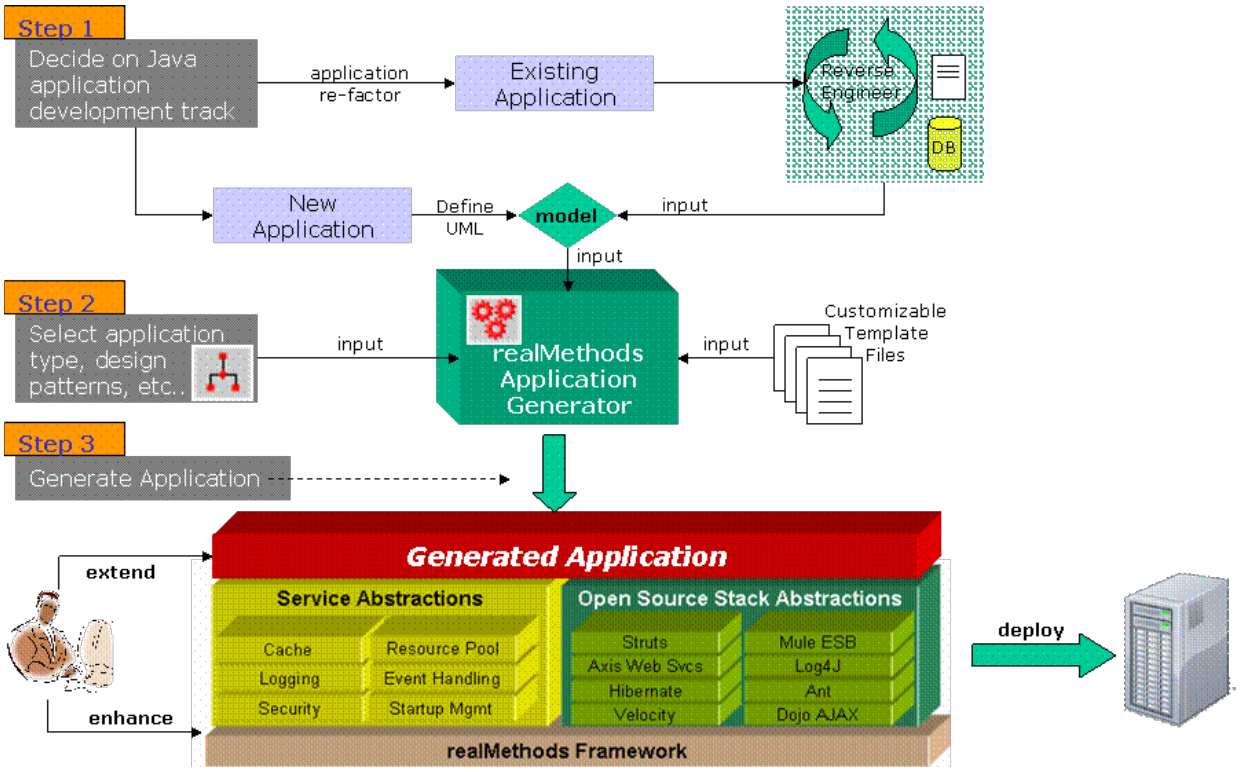
The true purpose of the realMethods Framework is to provide a cohesive implementation of the latest core J2EE Patterns. This ensures that any application that is a reflection of the framework will also implement these patterns.

Design pattern based software, leveraging the framework, is rewarded with:

- Higher levels of Re-usability
- Increased Return on Investment
- Industry standard ideas and experiences

Code Generation (AIB)

The Framework's ability to interpret your domain model into a design pattern based application model is the most important step in building the application. The AIB still allows you load an expression of your domain model as an XMI file. Importantly, however, it now allows you to reverse engineer an existing model from either a set of POJOs (Plain Old Java Objects) or existing database schema. From there, you are able to select the patterns and tiers the application should support. The final step of actually generating the code yields an ANT ready J2EE application, complete with all Java class files, Hibernate schema and property files, JSP files, AJAX content, deployment descriptors, and more.



Presentation Tier Overview

The goal of the generated presentation tier is to provide a robust Web 2.0 like user interface that allows access to the generated business and integration tiers of your application. Importantly, with the use of AJAX and design patterns, your application users will feel at home using a more current look-and-feel.

One of the most important enhancements to v5.0 is the complete integration of Struts 2.0i. What this means is that generated applications now automatically take advantage of many of the new features of Struts 2 including AJAX, improved form validation, as well as an enhanced library of tags.

In order to extend the generated JSP files, you will want to familiarize yourself with the Struts-2 architecture, as well as many of the key JSP tags.

Action Classes

Struts-2 Action classes have been improved and now treated more simply as POJOs. The AIB generates a Struts-2 Action class for each of the business objects defined within your business model. Each class contains all of the complex code necessary to receive and interpret user requests from the UI and then communicate to the business tier in order to execute the necessary business logic. Once you generate an application, spend some time reviewing one of your action classes to get an understanding as to what it does.

Tag Library Usage

Struts-2 contains a number of new and improved Tag Libraries. The AIB generates JSP pages that include the usage of many of these tag libraries. These include:

- s:form
- s:hidden
- s:url
- s:if/s:else
- s:textfield
- s:select
- s:datetimepicker

AJAX

Generated JSP pages make use of Dojo AJAX (0.4.3) either directly or indirect via Struts-2. Some of the Dojo tags used are:

<div dojoType="dialog"…
<table dojoType="SortableTable"…
<div dojoType="contentPane"…
<button dojoType="button"…

Javascript

In order to leverage AJAX completely, generated JSP and resulting HTML pages make heavy usage of Javascript. This adds to the dynamic real-time feel of the UI, allowing the updating of only the necessary pieces of the UI, rather than the entire page.

HTML Form Validation

Instead of server side validation, which tends to be slow, generated JSP pages now include client side validation which is more dynamic. This means user's get more clues as to what fields need data, and what kind of data. Rather than wait for the "Submit" or "OK" button to be clicked, validation takes place while the user is interacting with a form field, or when the user causes the form field to lose focus.

Next, let us breakdown how form validation now works.

```
<s:textfield id="employee.annualSalary"
name="employee.annualSalary"
label="annualSalary" readonly="false"
cssClass="required validate-number"/>
```

This piece of code declares a Struts-2 text field. The validation portion of this code is the "*cssClass*" declaration. The declaration indicates that the field is required and is validated as a number. When the user causes the text field to lose focus or when the associated form is submitted, the data in the field is checked against the validation rule, which in this case is a number.

The actual Javascript that handles the field validation is declared within the *homepage.jsp* file as:

```
<script type="text/javascript" src="./js/validation.js"></script>
```

Business Tier Overview

Business Delegate

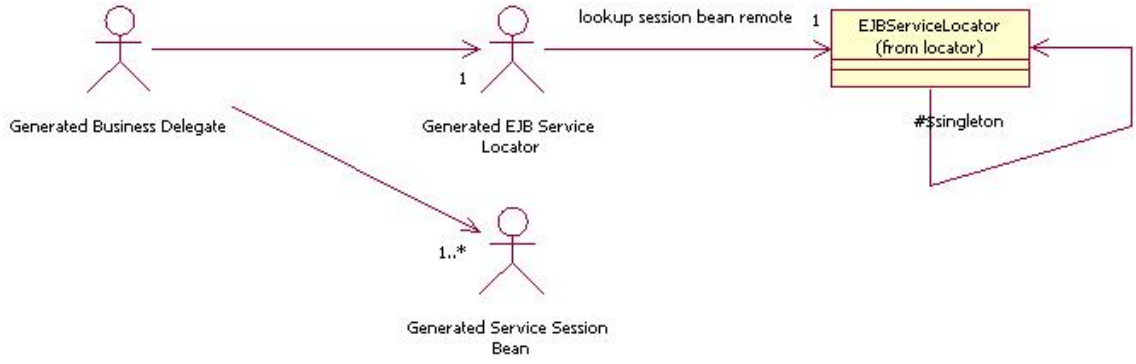
For each business object defined in your model, a business delegate is generated. The purpose of a business delegate is to act as a proxy for any client wishing to access business data or execute business logic. Importantly, this layer isolates the client from changes and versions that might take place within the business tier.

Business Delegate to DAO Directly

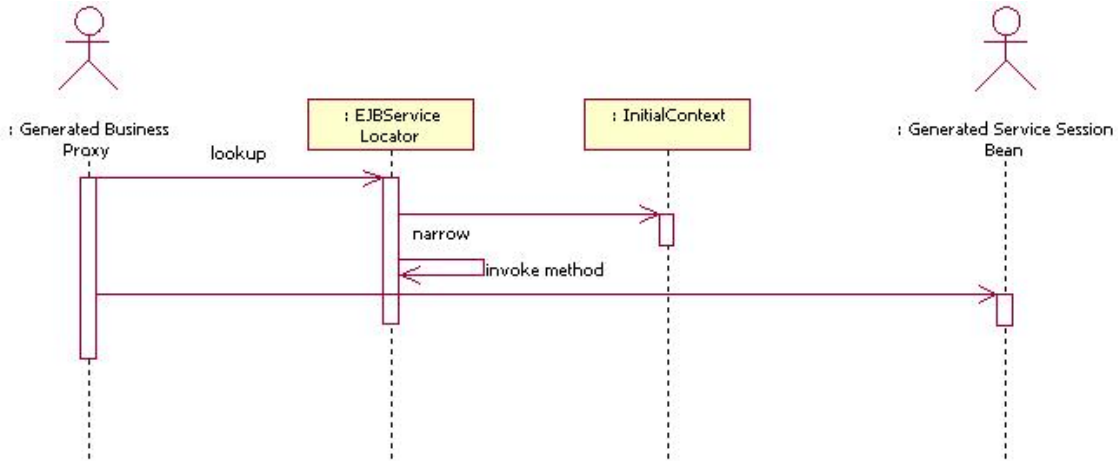
In this scenario, each business delegate access business data by connecting and communicating directly with generated Data Access Objects (DAOs). Although this scenario more tightly couples the business delegate with the DAO layer, many users will find their application simpler to manage since no EJBs are generated. The downside is that deployment options are somewhat more limited since business delegates and DAOs must reside within the same JVM.

Business Delegate to Session Façade via RMI

A client will access the business functionality via one or more generated business delegates. When a session façade is generated, the AIB provides the option to use RMI as the means of a business delegate to communicate with the session façade. The business delegate first uses an EJB service locator to assist in acquiring the remote interface of the require session bean. The remote interface is then used directly, using RMI to communicate with the deployed session bean on the targeted application server.



Business Delegate Accessing Session Façade via RMI Class Diagram



Business Delegate Accessing Session Façade via RMI Sequence Diagram

Web Services

The implementation of Web Service enablement into a generated application is easier than ever with the integration of the Mule ESB, the world's most popular enterprise service bus implementation. The AIB automatically necessary Mule configuration files with the correct declaration to "automagically" provision the generated Business Delegates as accessible as a Web Service. Under the covers, the Mule ESB uses the latest version of Apache AXIS as its SOAP engine. This means Web Service access to a business delegate is simple, easy, and well documentated.

Located within your application directory "*WEB-INF**classes*" is a file called *mule-**name-of-your-app**-config.xml.* This file contains a set of mule-descriptors, one for each business delegate, declared by way example as:

```
<mule-descriptor name="MutualFundBusinessDelegate"
implementation="com.poc.delegate.MutualFundBusinessDelegate"
inboundEndpoint="axis:http://localhost:81/services">
```

The *inboundEndpoint* setting defines the location of the Apache AXIS engine. This location is adjustable within the AIB and can be set before generating your application. It should be modified according to the location of your server and dedicated port.

Business Object Versioning

Motivation

It is often important to distinguish one version of an object instance from another. In this way, one is considered to be the most recent version, while all others are considered stale. This becomes useful in situations where two clients are attempting to store the same object instance. Since one will store before the other, the latter will therefore fail since its version identifier is less than that currently stored.

Framework Implementation

All that is necessary to apply versioning to any business object is to simply apply the *Auditable* stereotype to the entity in your class diagram. This stereotype tells the Framework to apply two date related attributes to the business object.

```
createTimestamp          - assigned during the creation of the BO
lastUpdateTimestamp      - assigned each time the BO is stored
```

The *lastUpdateTimestamp* attribute is applied within the Hibernate schema XML file for the corresponding business object. This tells Hibernate to use it as the differentiator between business object versions. The AIB automatically generates all that is required for this to work successfully with Hibernate.

Writing an Audit Trail System

You can use easily write your own audit trail system when you apply the *Auditable* stereotype to the each applicable entity in your class diagram.

The first step is to subclass the Framework class:

```
com.framework.integration.persistent.FrameworkHibernatorInterceptor
```

The next step is to overload the appropriate methods of this class, most notably:

```
onSave(…), onLoad(…), and onDelete(…)
```

Make sure you delegate back to the parent class within any method you overload.

Finally, and most importantly, apply the fully qualified name of your audit trail class as the value to the following property within the generated framework.xml file:

```
AuditTrailInterceptor="your fully qualified audit trail class name"
```

Business Object Notification Service

Motivation

Any J2EE framework should provide a simple facility to allow any client, external to a business object, the ability to be notified of key events pertaining to that business object. Most importantly, a client should be able to "sign-up" for notification and be notified in an asynchronous fashion.

Framework Implementation

Since the notion of a single base class for all business objects is central to the framework making assumptions of your business object, a subscription and notification infrastructure exists around this base class. There are 3 default types of Business Object related events the framework is interested in.

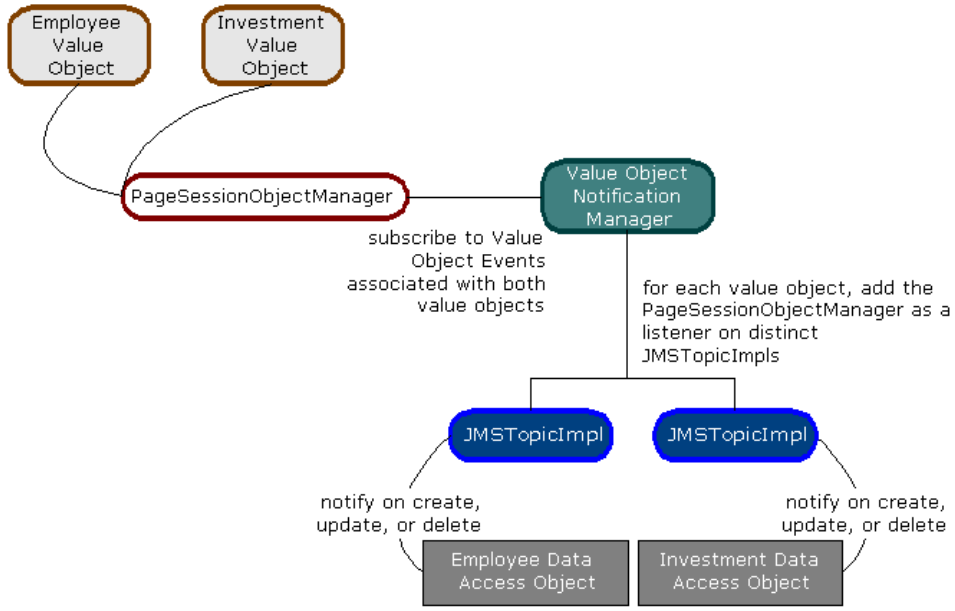
These event types are:

- Create
- Update
- Delete

These can be seen in class

com.framework.integration.notify.ValueObjectNotificationType

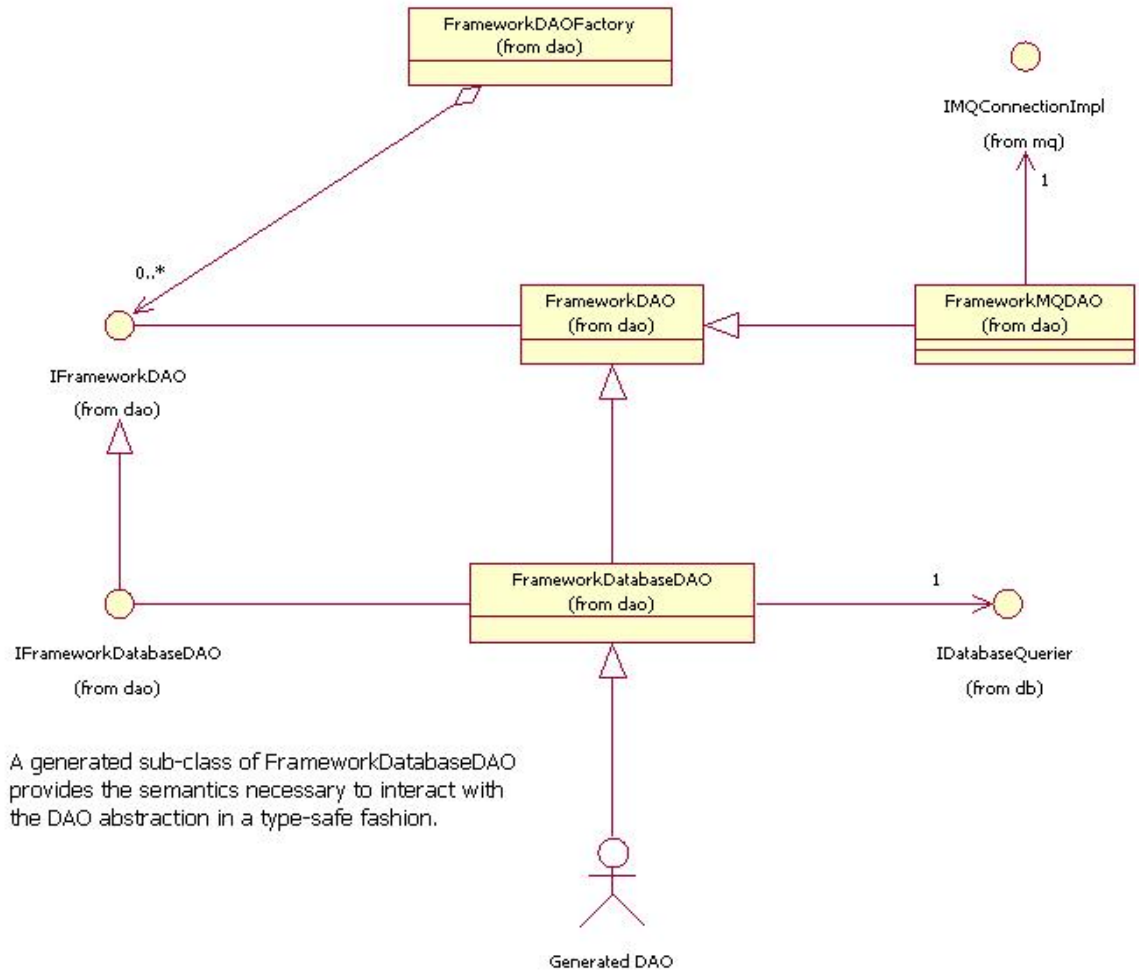
A ValueObjectNotificationEvent is broadcast on a JMS Topic and contains the associated IFrameworkBusinessObject of interest and ValueObjectNotificationType. Use the ValueObjectNotification parameter in the framework.xml configuration file to turn this feature on or off.



Business Object Notification Logical View

Integration Tier Overview

Data Access Objects (DAOs)



Mule ESB Integration

One of the most important new features of v5.0 is the integration of the Mule Enterprise Service Bus (ESB). Mule is the most popular and widely used enterprise service bus implementation available today. The introduction of an ESB provides many benefits to the realMethods framework as well as a generated application.

For the Framework, internal component communication, as well as generated application communication to the framework, is now done via the Mule ESB. Service components such as logging and security are now Mule ESB endpoints, that can be invoked and listen to as any other endpoint on the bus. All of the rewiring that went into making many of the key framework services to be able to use the Mule ESB is transparent to the user. This means the same interfaces and mechanisms previously used are unchanged.

For a Generated Application, the Mule ESB simplifies the provisioning of Web Service capabilities. Since each business delegate is actually a POJO, it is a matter of configuration in order to make a business delegate available as a Web Service. This can be enabled easily within the AIB before generating our application.

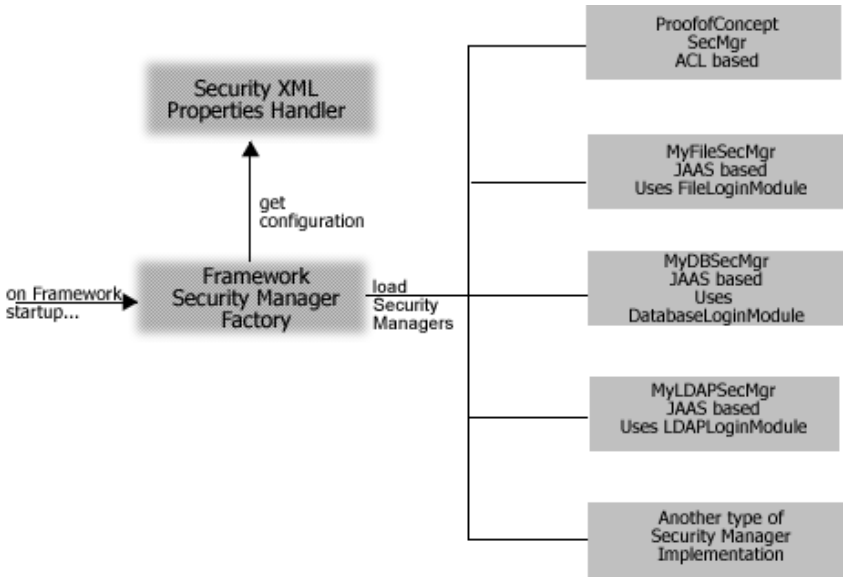
Security Management Overview

The Framework's notion of security is to delegate responsibility to a handful of components to assist in user specific authentication and authorization. Authentication is the process of determining that a user is who they say they are, while authorization is the process of determining if a user can do what they are trying to do.

Loading and Creating Security Managers

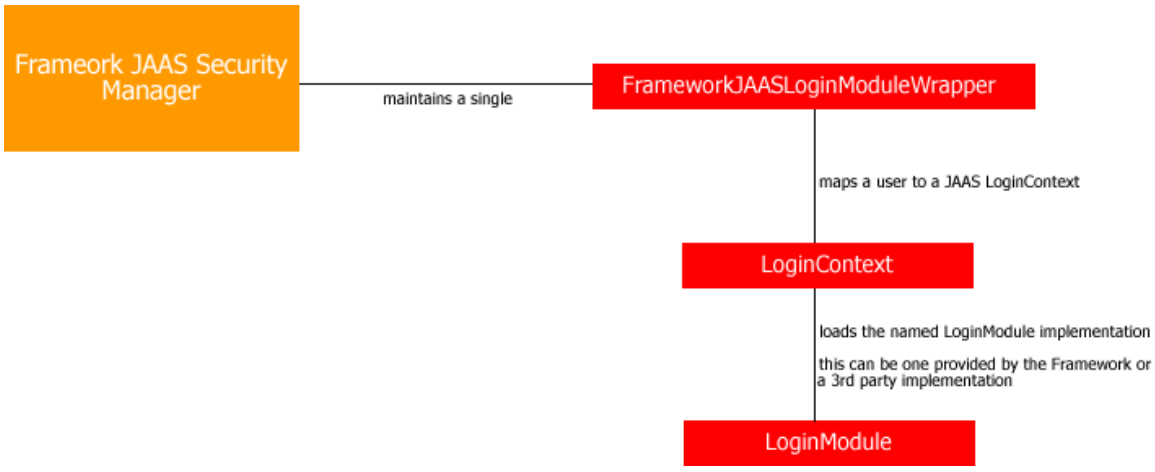
At startup, the Framework's StartupManager creates all Security Managers as specified in the security.xml file, designating one of them as the default. Each Framework Security Manager either supports JAAS or ACL based security.

The realMethods Presentation tier applies the default Security Manager to each HttpSession. Using the actiondefs.xml file, you can specify which actions require authentication and which require authorization. You can access any Security Manager by name in order to apply security to other parts of your application.



Framework Security Manager Loading Logical View

Framework JAAS Security Manager



JAAS-based Authentication

The Framework JAAS Security Manager provides a wrapper to the LoginModule implementation in use. This wrapper acts as an interface between Framework Security Management and JAAS, interpreting login/logout and authorization requests from the application to the LoginModule.

JAAS-based Authorization

The JAAS authorization component supplements the existing Java 2 security framework by providing the means to restrict the executing Java code from performing sensitive tasks, depending on its codesource (as is done in Java 2) and depending on who was authenticated.

For the purpose of the JSP/Servlet Framework, authorization is a function of verifying the user can take on one or more required identities to perform the current action. If you need to provide different restrictions, you should refer to the JAAS API Developer's Guide.

Usage

Using a LoginContext, a Framework JAAS Security Manager will acquire the LoginModule based on the name provided during configuration. Our experience has been that each application server has its own way of applying the JAAS security model, whether through configuration files, Java system properties, etc. Refer to your application server's developer's guide for more information.

Configuration

- See Security Properties→security.xml in the Developer's Guide.

Framework LoginModule Implementations

The Framework provides 3 different JAAS LoginModule interface implementations. Each is only differentiated by where user name, password and allowed identities (roles) are stored.

```
com.framework.integration.security.jaas DatabaseLoginModule
com.framework.integration.security.jaas FileLoginModule
com.framework.integration.security.jaas LDAPLoginModule
```

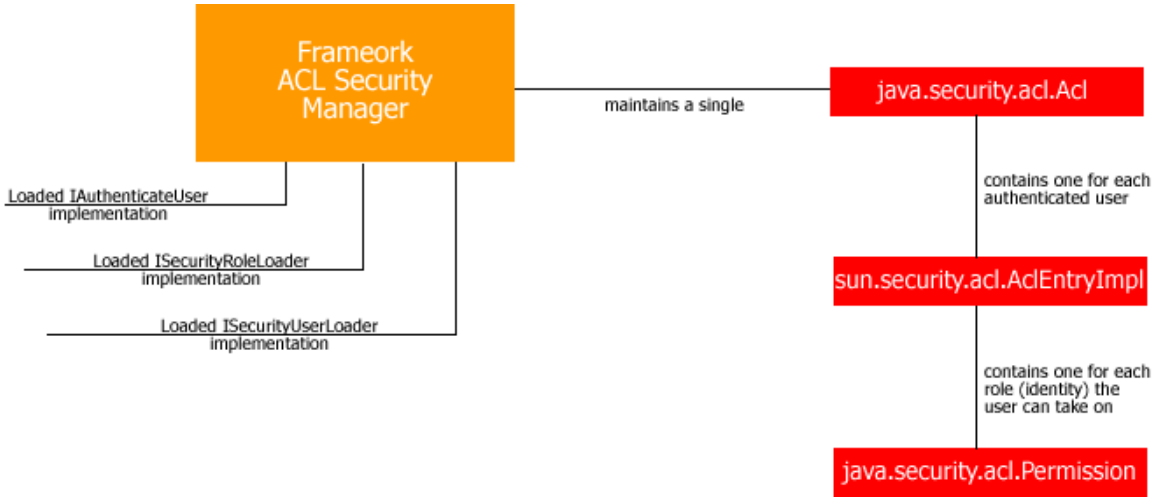
Designer Note:
You have the ability to define a Framework JAAS Security Manager that uses any uniquely named LoginModule interface implementation. Application servers such as Weblogic, Oracle's OC4J, and JBoss provide LoginModule implementations as well.

```
<sec-manager
name="DBSecurityManager"
securityManagerClassName=

"com.framework.integration.security.jaas.FrameworkJAASSecurityManager"
loginModuleName="DBSecurityManager"
/>
```

The name field refers to the name of a LoginModule the JAAS security implementation has been configured to use. This is normally done within a JAAS config file, but you may have to check with the documentation of the JAAS provider.

Framework ACL Security Manager

An ACL (Access Control List) is a data structure used to guard access to resources. An ACL can be thought of as a data structure with multiple ACL entries. Each ACL entry contains a set of permissions associated with a particular java.security.Principal. Additionally, each ACL entry is specified as being either positive or negative. If positive, the permissions are to be granted to the associated principal. If negative, the permissions are to be denied.



ACL-based Authentication

By implementing a single interface

```
com.framework.integration.security.IAuthenticateUser
```

a developer is responsible for assisting the Framework in identifying whether a user id/password combination is valid. This typically takes place at login, but authentication can be conditionally applied on any page.

ACL-based Authorization

Concerning authorization, a Framework Security Manager handles "when" to authorize and "what" to do based on the result of an authorize request, but it cannot handle "how" to authorize.

Configuration

- See Security Properties-security.xml in the Developer's Guide.

Framework LoginModule Implementations

The Framework provides 3 different JAAS LoginModule interface implementations. Each is only differentiated by where user name, password and allowed identities (roles) are stored.

```
com.framework.security.jaas DatabaseLoginModule
com.framework.security.jaas FileLoginModule
com.framework.security.jaas LDAPLoginModule
```

Designer Note:
You have the ability to define a Framework JAAS Security Manager that uses any uniquely named LoginModule interface implementation. Application servers such as Weblogic, Oracle's OC4J, and JBoss provide LoginModule implementations as well.

```
<sec-manager
name="DBSecurityManager"
securityManagerClassName=

"com.framework.integration.security.jaas.FrameworkJAASSecurityManager"
loginModuleName="DBSecurityManager"
/>
```

The name field refers to the name of a LoginModule the JAAS security implementation has been configured to use. This is normally done within a JAAS config file, but you may have to check with the documentation of the JAAS provider.

Log Handlers and Definitions

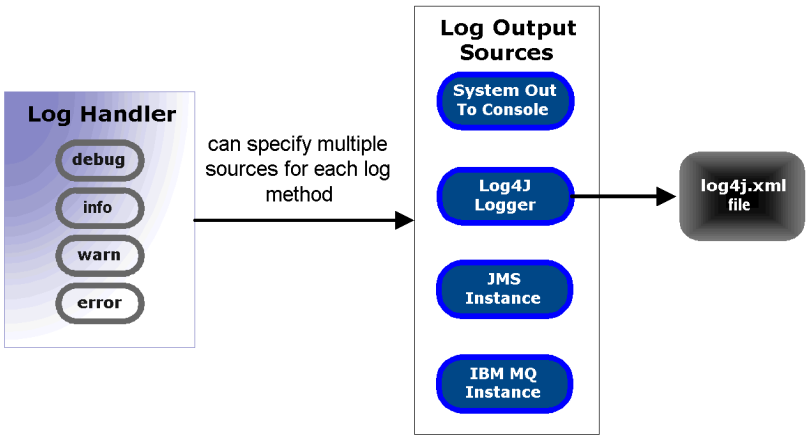
The Framework's Logging Service provides an application with a simple to use, yet flexible and robust mechanism for logging output to multiple destinations. Property driven parameters allow an application to easily modify the semantics of logging without changing any of the applications code.

In order to adhere to the spirit of Jakarta's Log4J implementation, the Framework's Logging Service make available a single programmatic interface to handle debug, info, warn, and error.

Log Output Destinations

Each Log Handler is able to specify any number of pre-determined destinations for output. This means a log message can be output to more than one source. The types of destinations supported are:

System out to the console
JMS Connection - as named within connectionpool.xml
IBM/MQ Connection - as named within connectionpool.xml
Log4J named Logger



Framework's Log Handler Logical View

Framework Log Handler and the Command/Task Architecture

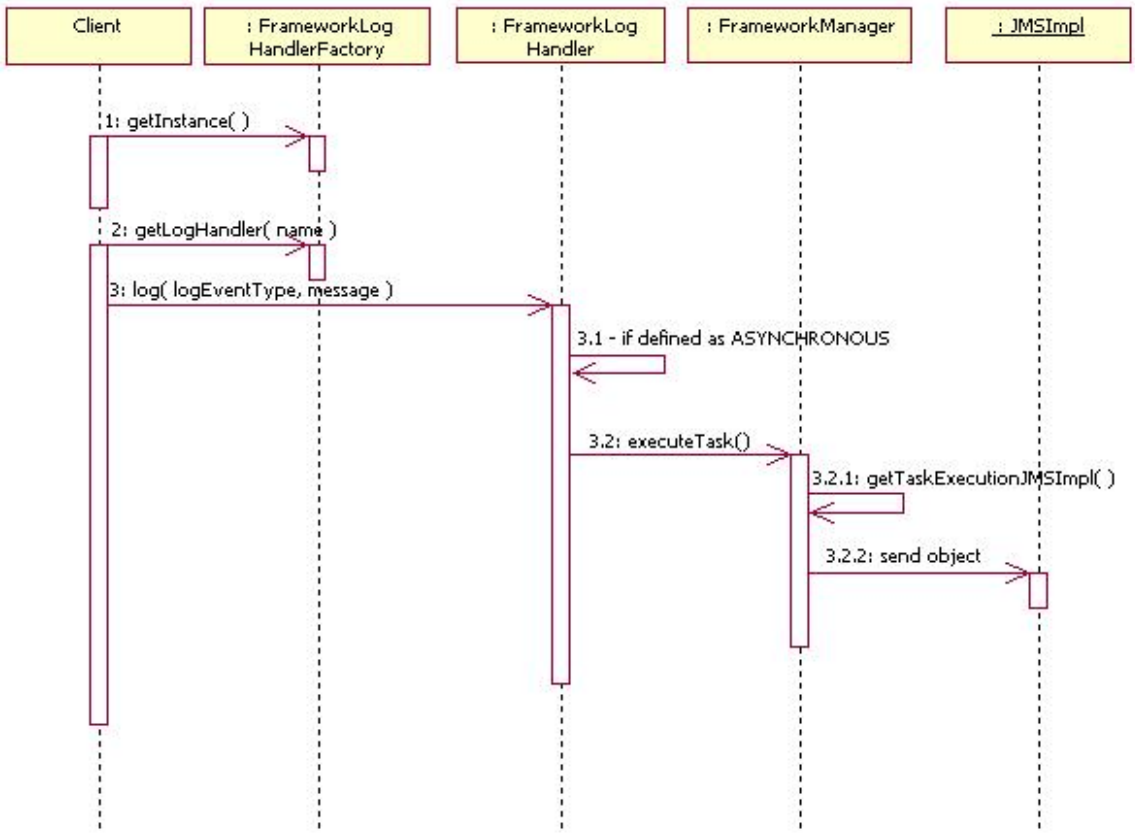
When a framework log handler instance attempts to actually log the client provided message, it has the help of the Command/Task Architecture. The framework requires the following definition within the task.xml configuration file.

```
<tasks>
    <!-- the following definition is used internally by the framework to handle
     application logging -->

  <task name="FrameworkLogTask"
    commands="com.framework.integration.command.FrameworkLogCommand"/>
  />
.
.
.
</tasks>
```

The FrameworkLogCommand contains the logic used to determine which output source(s) to send the client's log message. How a log handler invokes a the FrameworkLogCommand is either synchronously or asynchronously. This is indicated as part of each log handler definition within the loghandlers.xml configuration file.

Asynchronous Logging Sequence Diagram

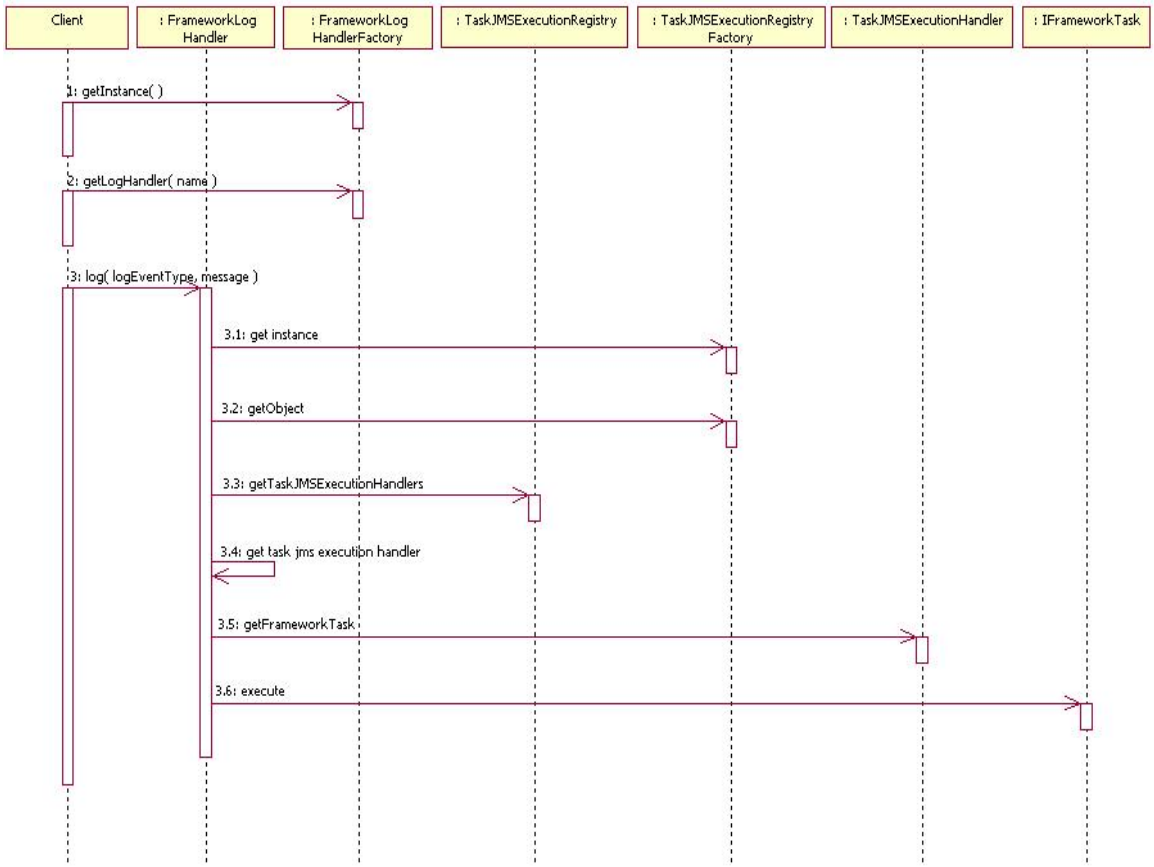
A framework log handler accomplishes asynchronous logging by indirectly invoking FrameworkLogCommand via JMS.



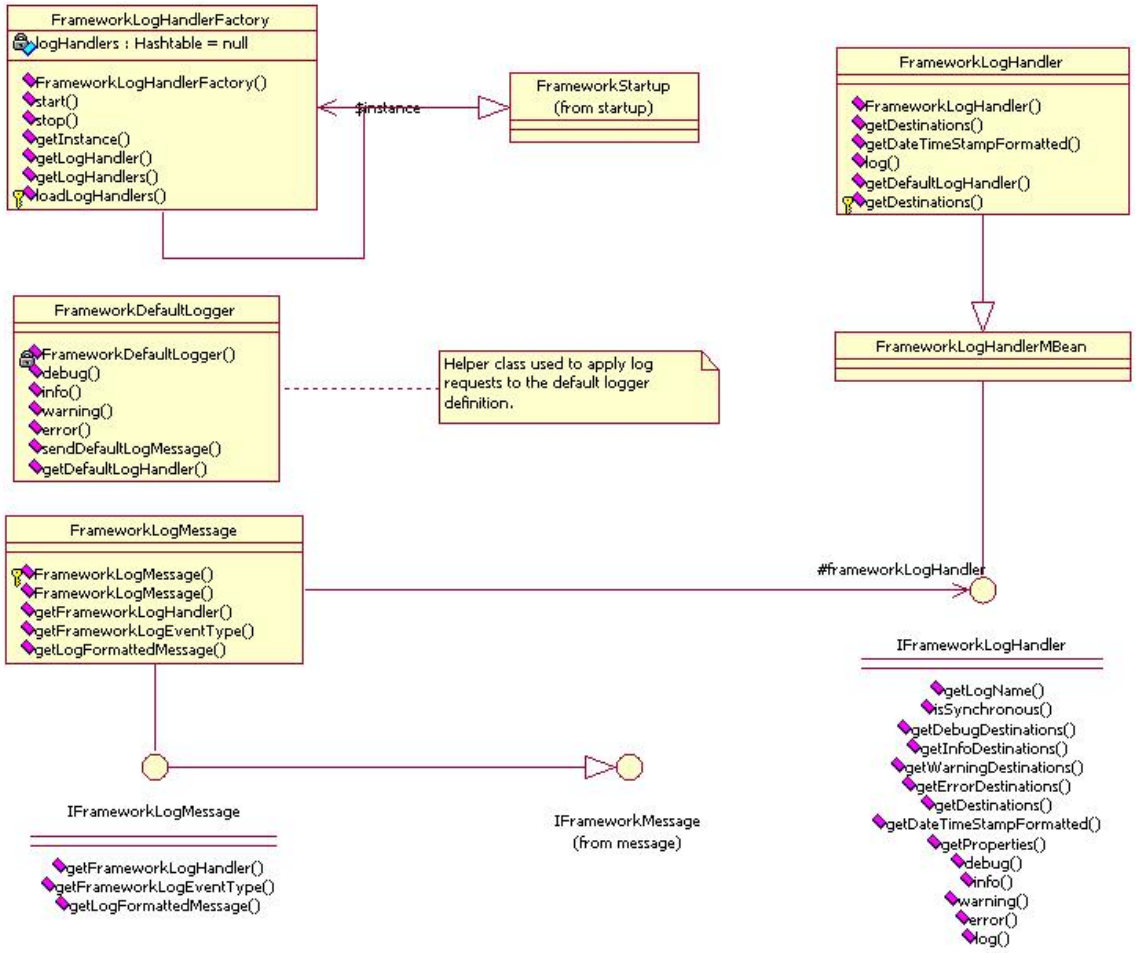
Framework Asynchronous Logging Sequence Diagram, using the Default Log Handler

Synchronous Logging Sequence Diagram

A framework log handler accomplishes synchronous logging by indirectly invoking FrameworkLogCommand by acquiring the named *FrameworkLogTask* from the appropriate TaskJMSExecutionHandler, and calling execute() directly on it.



Framework Synchronous Logging Sequence Diagram, using the Default Log Handler



Framework's Log Handler Class Diagram

Framework Connection Pooling

Note:
As of version 4.0, the Hibernate O/R API is used to handle all non-CMP persistence scenarios. Hibernate has its own database connection pool implementation, as well as the ability to use a data source. If you have a need to define a database connection pool for non-Hibernate queries to one or more databases, you should consider using the Framework's connection pool implementation.

Motivation

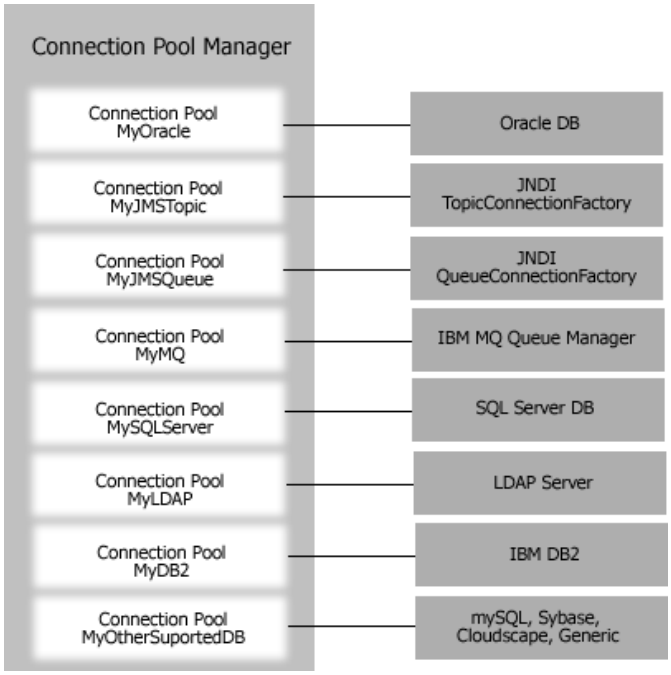
Design and implement a single object that can handle a fundamental set of responsibilities surrounding the pooling of implementations of a well-defined interface. Included in this set of responsibilities is:

1. Manage any number of pools of different types, each of varying sizes with different constraints.
2. Optionally, individual pools must grow and shrink on an as needed basis.
3. Consumers of a pooled entity must be able to access any entity in the pool by name, without regard for any specific entity in the pool.
4. Consumers are responsible for returning an entity to the pool.

The Framework's Implementation

The Framework connection pooling provides a powerful, consistent, and isolated manner of establishing and maintaining connections to multiple sources. The current version of the Framework includes connections to the following sources:

- Database Connection - Oracle, Sybase, MS SQL Server, DB2, MySQL
- Messaging Services - IBM/MQ Series, Java Message
- Directory Services - LDAP



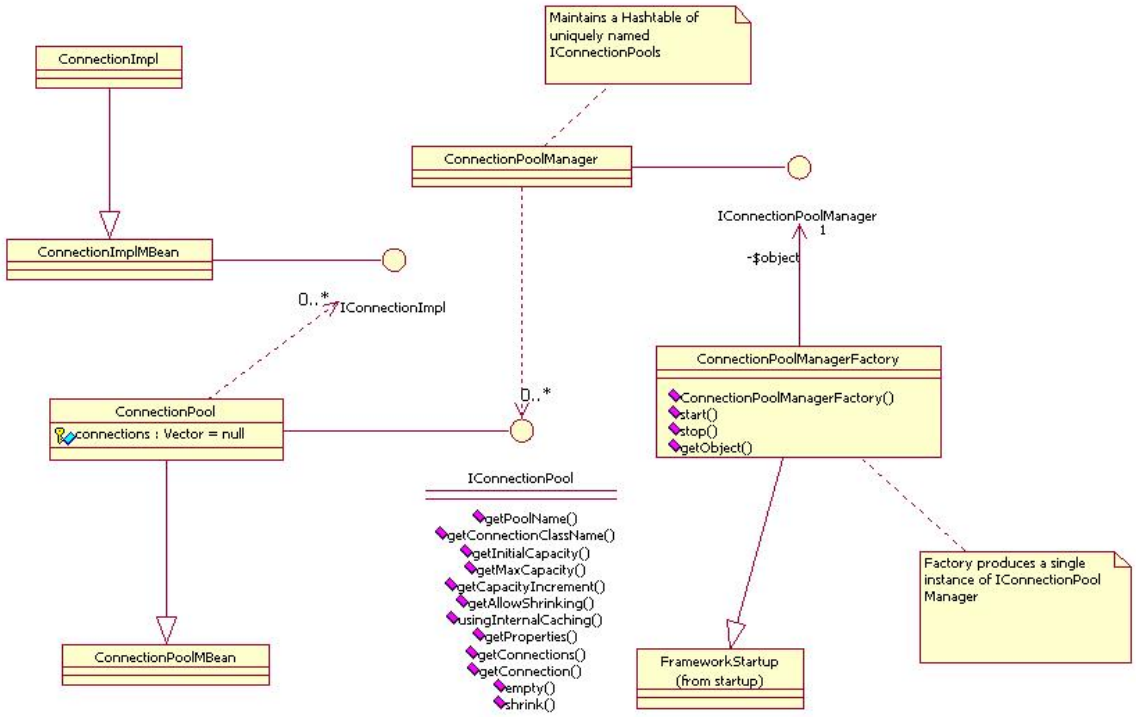
Framework Connection Pooling Logical View

As is evident, connection pooling involves more than just databases. In general, any resource that should be carefully consumed and can have actions taken upon it that may require to be "undone" should be considered for definition as a connection and be pooled using the Connection Pool Manager.

The Framework connection pooling uses a Connection Pool Manager that is responsible the creation of one or more Connection Pools. Each Connection Pool is responsible for the creation and maintenance of its connections, and also contains logic to reuse, grow and shrink itself according to user-defined parameters. The caching and reuse of a connection is important in any resource intensive environment.

The Framework defines a Connection Pool as any object that can support the interface:

```
com..framework.integration.objectpool.IConnectionImpl
```



Framework Connection Pooling Class Diagram

Each connection the application needs to be created and pooled by the Connection Pool Manager is specified within the connectionpool.xml file:

```
<connectionpool
        name="TheOracleDBConnection"
        connectionClassName
        ="com.framework.integration.objectpool.db.OracleConnectionImpl"
        user="scott"
        password="tiger"
        driver="oracle.jdbc.driver.OracleDriver"
        url="jdbc:oracle:oci8:@"
        maxCapacity="10"
        initialCapacity="1"
        capacityIncrement="20"
        allowShrinking="TRUE"/>
```

Design Note:
Concerning database connections, your design may call for the use of a DataSource or PoolableDataSource. In either case, you can still specify the data source in the connectionpool.xml file, and have it used by any client requiring it as a source of persistence, such as the application's generated DAOs.

```
<connectionpool name="OracleDS"
        connectionClassName
            ="com.framework.integration.objectpool.db.OracleConnectionImpl"
        user="poc"
        password="poc"
        usingDataSourcePooling="TRUE"
        JNDIDataSourceName="MyOracleDataSource" />
```

Mandatory Framework JMS Connections

The Framework internally attempts to make use of 3 default JMS Connections, each already defined in:

RM_HOME\home\connectionpool.xml

connection name	purpose
FrameworkEventJMS	Used to broadcast Framework related events
FrameworkTaskExecutionJMS	Used to broadcast and receive task execution requests.
FrameworkValueObjectNotificationJMS	Use by the Framework's internal Business Object notification mechanism. If turned on framework.xml, listeners on this JMS Topic Connection are notified when a Business Object is created, updated, or deleted. A generated ApplicationUSOM is such a listener.

Framework Internally Logging

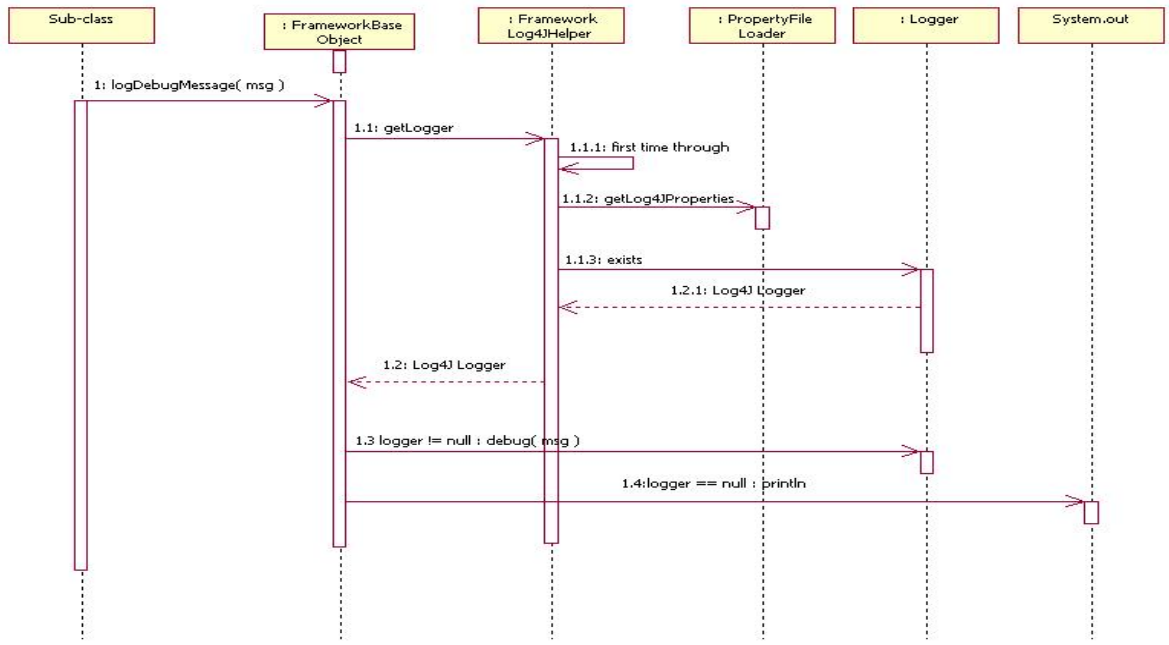
Framework internal logging is how and where the code specific to the framework actually logs output messages. Framework internal logging is handled differently than application logging, although the end result may be that both end up logging to the same source.

The framework uses Log4J to handle logging, according to the definition of a logger named *'realMethods'* found in the log4j.xml configuration file. The location of this configuration file is specified within the config.xml file. The framework outputs debug, warning, error, and informational messages. You can control the actual output of the message types by modifying the value attribute of the level element for the realMethods logger element.

```
<logger name="realMethods">
   <level value="debug"/>
   <appender-ref ref="poc"/>
</logger>
```

Design Note:
If you desire to have the application log its output to the same output source as the Framework, simply provide the logger name 'realMethods' as the output source for the desired log handler in the loghandlers.xml configuration file.

```
<logHandler name="MySharedLogHandler"
synchronous="true"
debug="realMethods"
dateTimeStampFormat="yyyy.mm.dd-hh:mm:ss"/>
```

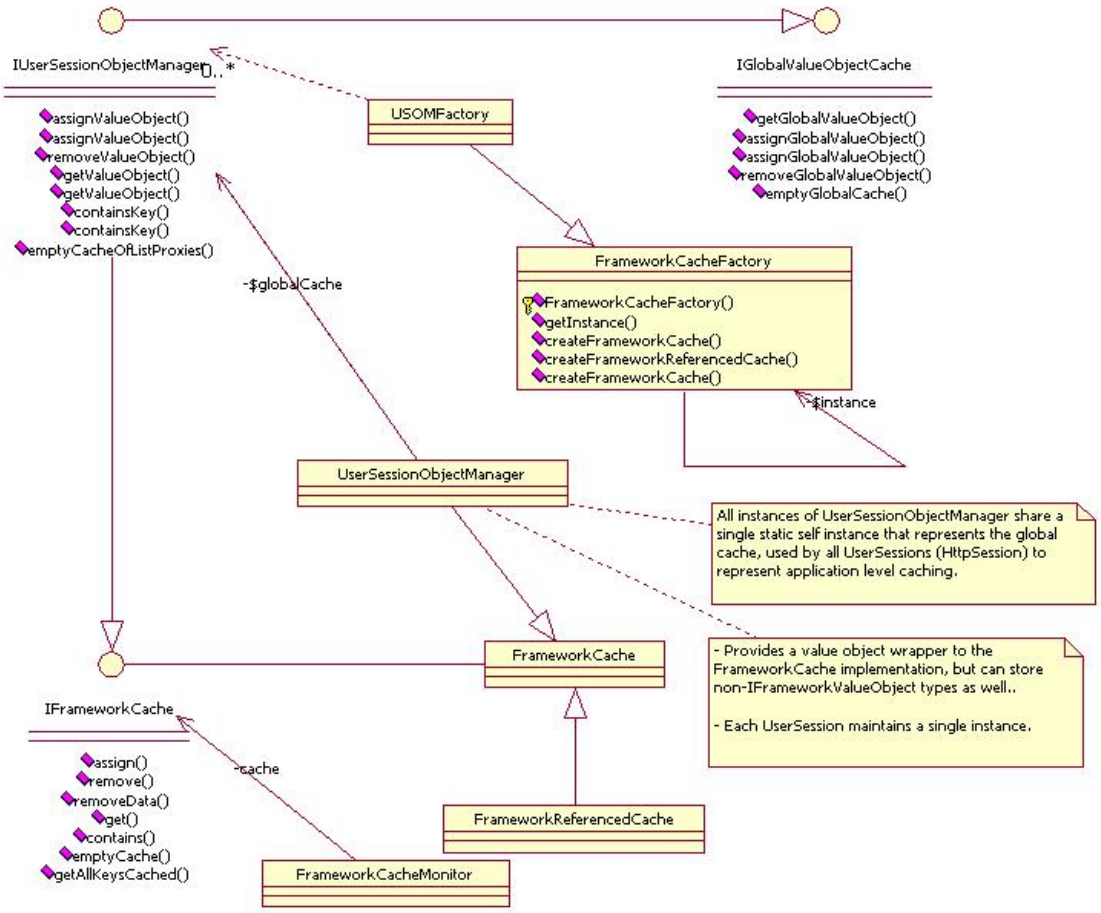


Framework internal logging Sequence Diagram

Available Cache Implementations

An application often requires caching mechanics that incorporate more logic and functionality beyond the standard java.util.Collection implementations. The framework defines 2 types of cache, both of which have FrameworkCache as its base class. The first is called UserSessionObjectManager, and it implements a Business Object facade to the underlying FrameworkCache access methods. The second sub-class is FrameworkReferencedCache, and it stores an object to be cached as a java.lang.ref.SoftReference. This allows an un-referrenced object in the cache to be purged by the JVM.

Designer Note:
The DAO layer can be configured to use an internal **FrameworkReferencedCache**, to store Business Objects that are pushed and pulled from the data store. This can expedite the read related options of a Business Object. However, this option is only useful in situations where the DAO layer is not in a clustered environment, and is the only means of persisting data. There is more on this below.



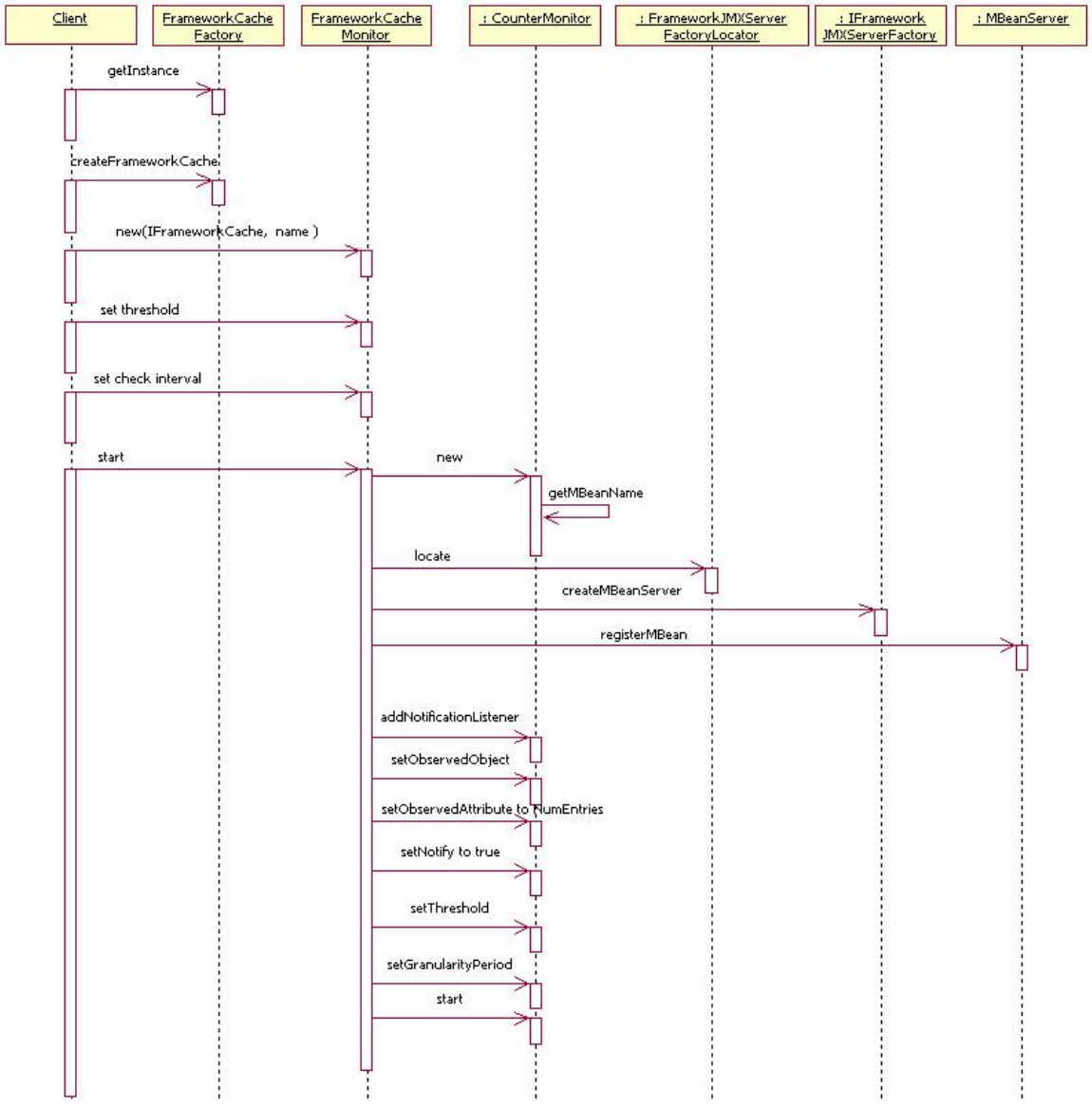
Framework Cache Class Diagram

Automatically Purge with a Framework Cache Monitor

The framework provides a convenient means of applying a JMX Counter Monitor to an IFrameworkCache implementation. The purpose of this monitor is to automatically purge a designated cache once a specified capacity maximum has been reached. When created and bound with the IFrameworkCache to monitor, the client of the FrameworkCacheMonitor provides 2 monitoring details:

Threshold: Indicates how many objects the cache can contain before purging.

Check Interval: How often the cache threshold should be checked, in milliseconds.



Using a Framework Cache Monitor

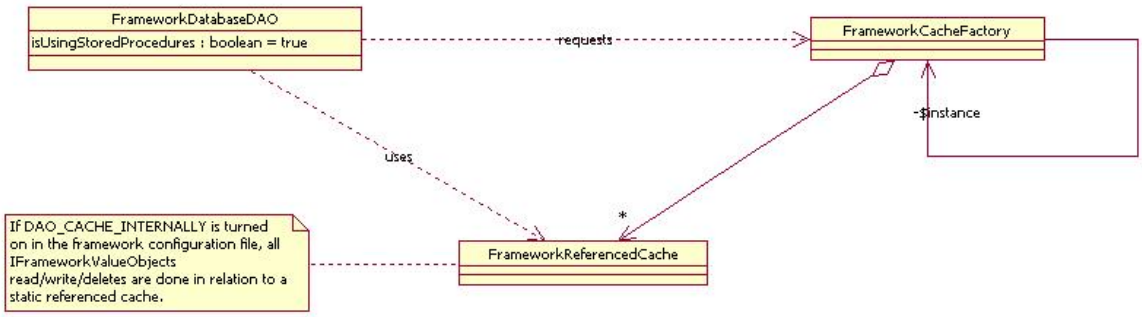
If the framework.xml config file property USOM_MAX is set to a value greater than –1, each UserSession automatically creates a FrameworkCacheMonitor, and applies its UserSessionObjectManager to it. The parameters for the monitors threshold and check-interval are designated by the framework.xml configuration file properties:

USOM_MAX
USOM_MAX_CHECK_PERIOD_IN_MILLIS

Designer Notes:
Leverage this capability in situations where you wish to discourage the possibility of bloat within an HttpSession. A USOM_MAX value of –1 indicates that the cache isn't to be purged, therefore a FrameworkCacheMonitor is not necessary.

Enabling Caching within the DAOs

The framework provides the ability to automatically cache the data that flows through the DAO layer.

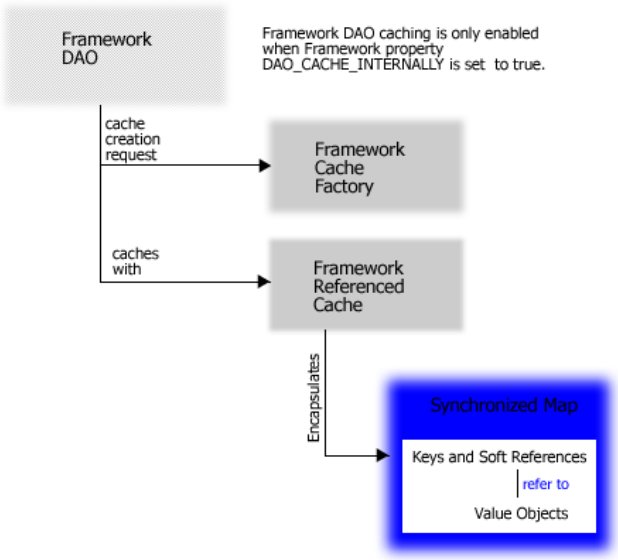


Framework Database DAO Cache Class Diagram

The following matrix explains how the read/write operations of Business Objects to the data source are handled in relation to the contained IFrameworkCache

Scenario	Cache Action
Create	Create to data store, cache locally
Read	Get on local cache; if null, go to data store
Update	Update to data store, cache locally, meaning replace if exists
Delete	Remove
Read a List	Cache all

Designer Note:
This option is only useful in situations where the DAO layer is not in a clustered environment, and is the only means of persisting data. If enabled in such conditions, the likely hood of serving up a stale Business Object is extremely high.

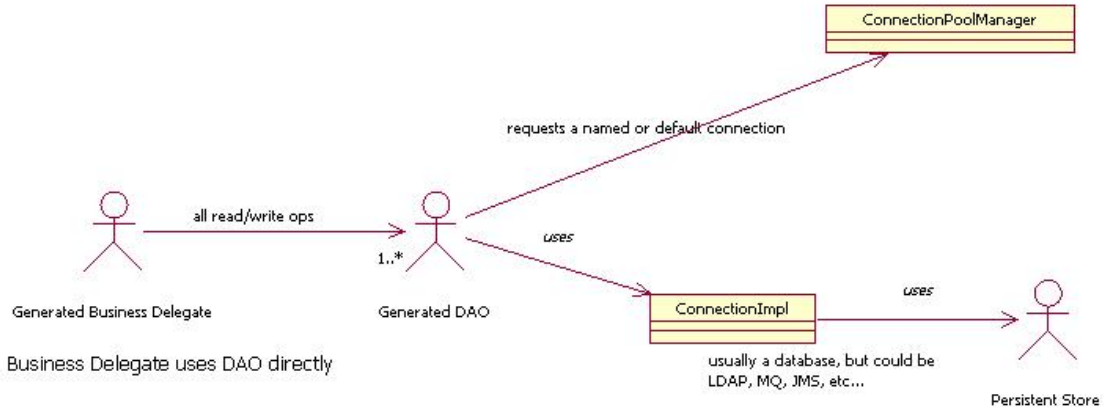


Application Re-factoring Options to Consider

Since no two single domain problems are exactly alike, no single implementation strategy will suffice. A smaller application may not require EJBs while a Session Façade fronted distributed read-only application may not require entity beans. Whatever the case, the AIB provides you the flexibility to distribute your application model as you see fit, but in a manner consistent with the core J2EE design patterns.

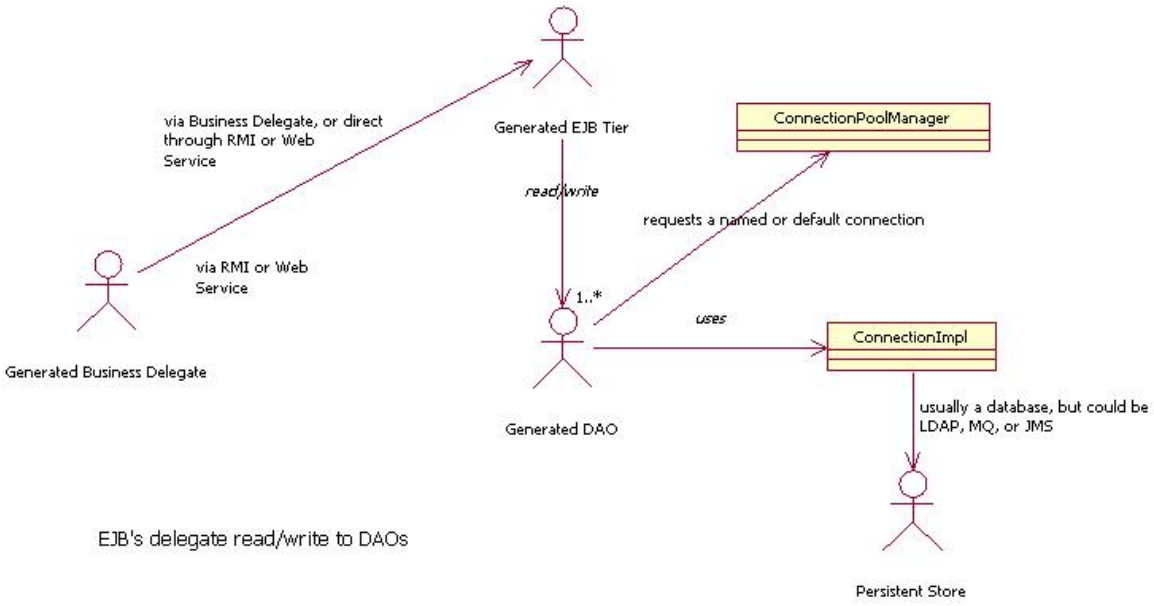
Scenario #1
Presentation Tier to Business Delegate to DAOs

An application model that reflects the components of this scenario normally is the result of a simple, less complex domain model. In it, the value of keeping business logic out of the presentation tier is not lost. However, the generated Business Delegates communicate directly to the generated DAOs for all read/write operations to the persistent store, instead of to a remote Session Façade via RMI or SOAP. The DAO delegates all persistence responsibilities through the Hibernate API.



Scenario #2
Business Delegate to Session Façade to DAO

Many read-only applications, requiring distributive access to coarse-grained and fine-grained business logic can take advantage of this scenario. With it, a Business Delegate, using a Service Locator, accesses the Session Façade. In turn, the Session Façade, without the aid of entity beans, uses a set of Data Access Objects (DAOs) to handle all persistence.



Startup Scenarios

Important to executing your application is the starting of the framework. During the startup phase, the framework:

- Locate and validate the framework.license file
- Locate, load, and parse the config.xml file
- Locate, load, and parse the configuration files referenced within the config.xml file.

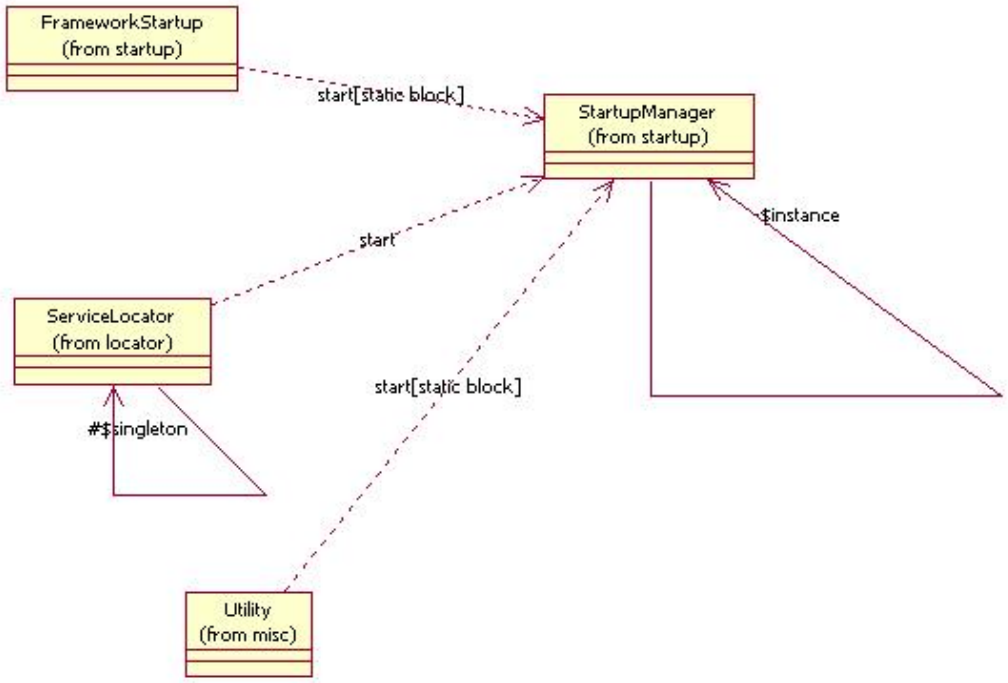
Implicitly starting the framework

The framework is able to start itself, without application interaction. There are three distinct framework points where this takes place:

1. Initial access to any class that is a sub-class of com.framework.common.startup.FrameworkStartup

2. Initial access to the Service Locator, covers scenarios where a presentation tier isn't present, or the EJB tier is accessed first. com.framework.common.locator.ServiceLocator

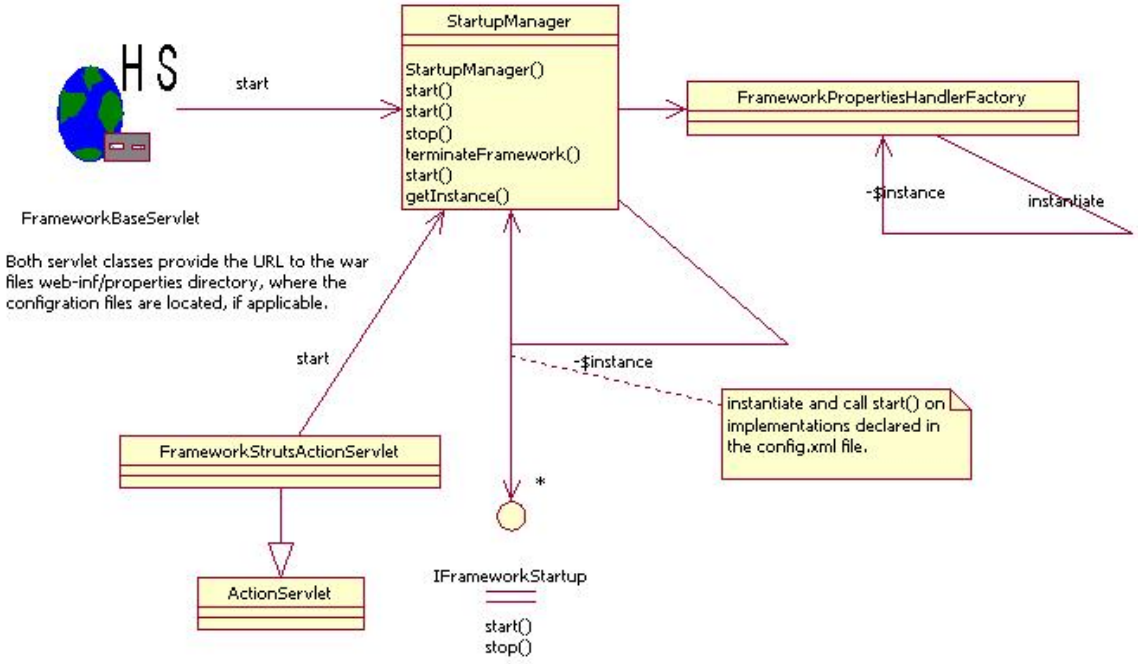
3. Initial access to the Utility class com.framework.common.misc.Utility



3 Framework points of automatically starting on behalf of client.

Startup via Framework based Servlet

An AIB generated application that includes a presentation tier (Struts or realMethods) will include a servlet entry in the web.xml. This servlet will be able to automatically start the framework when initialized by the servlet engine.



Valid configuration file locations

On startup, the Framework's *StartupManager* attempts to locate the config.xml file. This file contains named references to the configuration files to be loaded and parsed by the framework. The location of the config.xml is provided to the *StartupManager* in one of three manners:

1. WAR file WEB-INF/properties Directory

Any web-based application generated by the AIB contains the necessary configuration files in the .war files web-inf/properties directory. On startup, depending on which presentation framework is in use, either class

```
com.framework.presentation.servlet.FrameworkFrontControllerServlet
```

or

```
com.framework.presentation.struts.FrameworkStrutsActionServlet
```

will call *start* on the *StartupManager* and provide the location of the web-inf/properties directory.

Note:
There is a known bug in Weblogic whereby files are unable to be loaded when contained and referenced within a .WAR file. Refer to the next two options when deploying on Weblogic.

2. Classpath

The next place the framework attempts to locate the config.xml is in the classpath.

3. -DFRAMEWORK_HOME

If the *StartupManager* is signaled to start without being provided the location of the configuration files, it will attempt to next locate it in the directory location specified by the Java system property FRAMEWORK_HOME. This parameter will be provided for the purpose of indicating the location of the framework.license file.

Specifying the location of the framework.license file

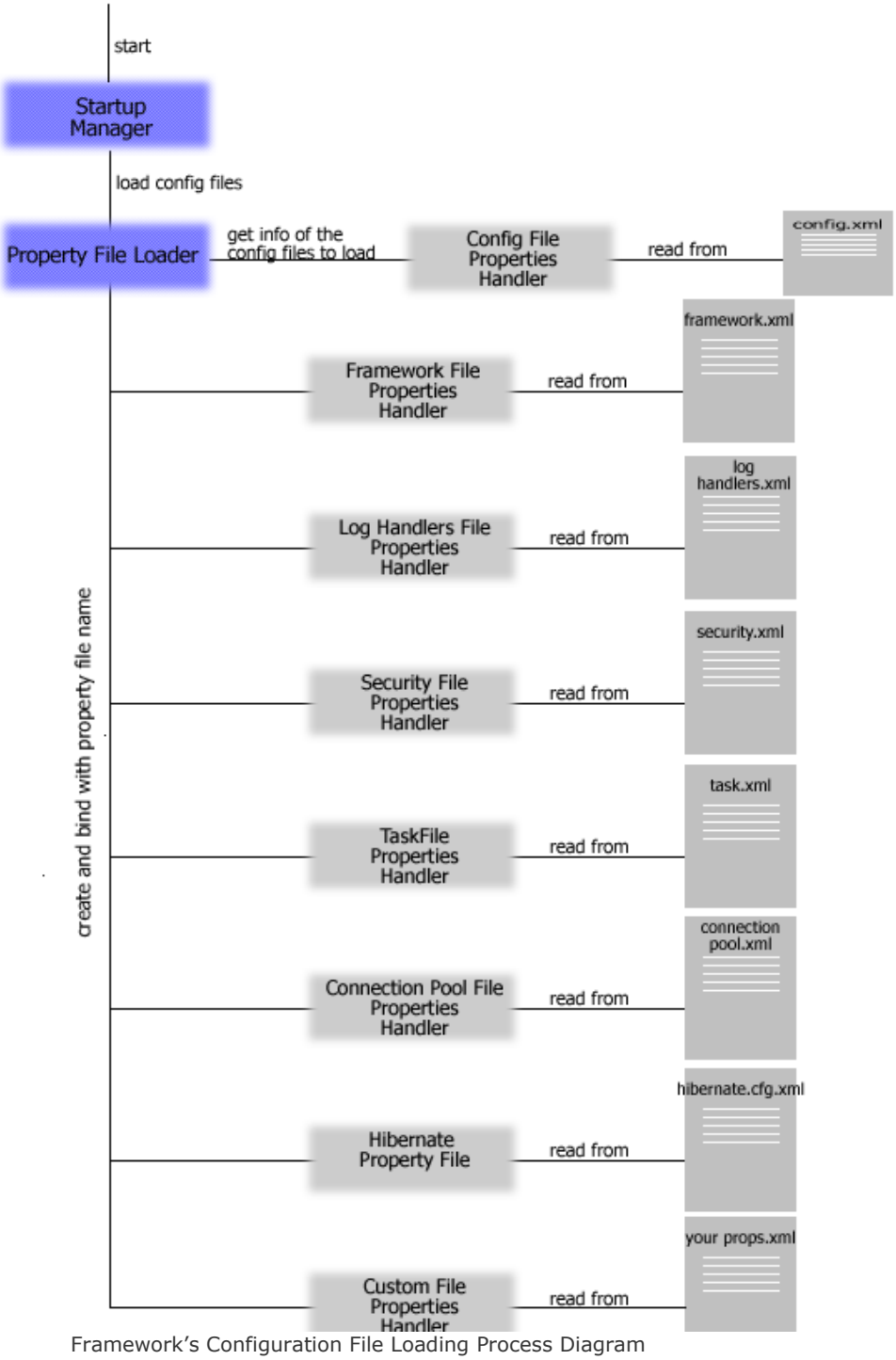
The first step the framework takes during the startup phase is to locate and load the framework.license file, in order to validate your license to use the framework. In order to locate the license file, you must provide the following Java system property:

```
-DFRAMEWORK_HOME=directory-location-of-the-framework-license-file
```

A –D parameter is normally specified on the system execution line used to start your application server.

Configuration File Loading Process

In order to understand what takes during the startup of your application, you should have an understanding as to how the Framework's configuration files are loaded during this phase.



Framework's Configuration File Loading Process Diagram

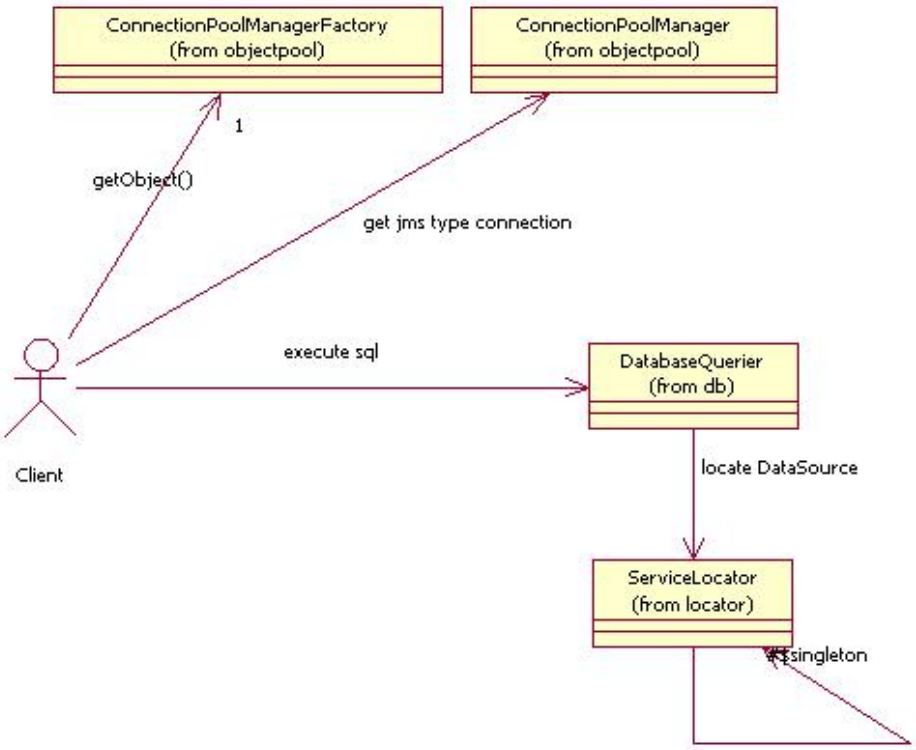
Service Locator Overview

The framework provides a convenient abstraction that encapsulates the task of acquiring JNDI based components and services. The service locators generated by the AIB make use of this service in order to lookup EJB home interfaces.

Connecting to JNDI
The ServiceLocator is a singleton, and upon its initial creation of the InitialContext, it uses the value assigned to the JNDI_ARGS property of the framework.xml configuration file. This property must be set correctly based on the parameters required of the InitialContext provider.

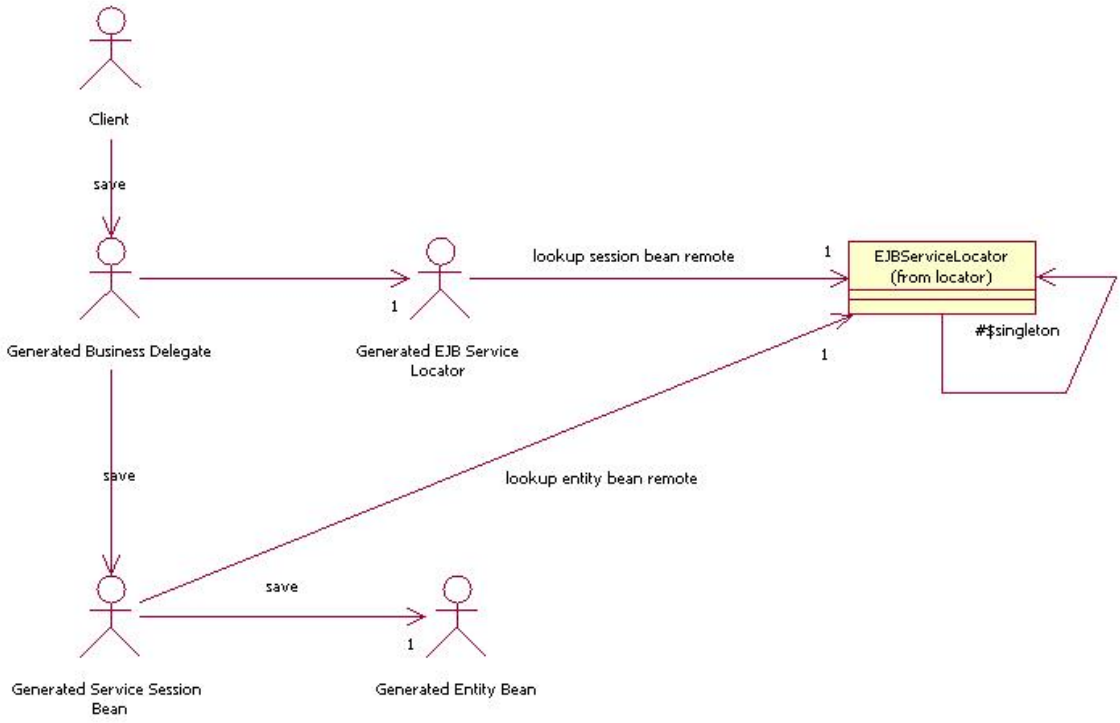
Caching of Retrieved JNDI Objects
The ServiceLocator contains a synchronized HashMap for the purpose of storing an Object initially retrieved from JNDI. Subsequent requests for the object by name will use the HashMap instead of JNDI.

Caching of Retrieved Data Sources
For the sake of efficiency, a DatabaseQuerier instance uses the ServiceLocator to retrieve a data source from JNDI. The name of the data source is provided as the *jndiName* property for a data source definition within the connectionpool.xml configuration file. The data source itself is cached locally wiithin the ServiceLocator, for future access.



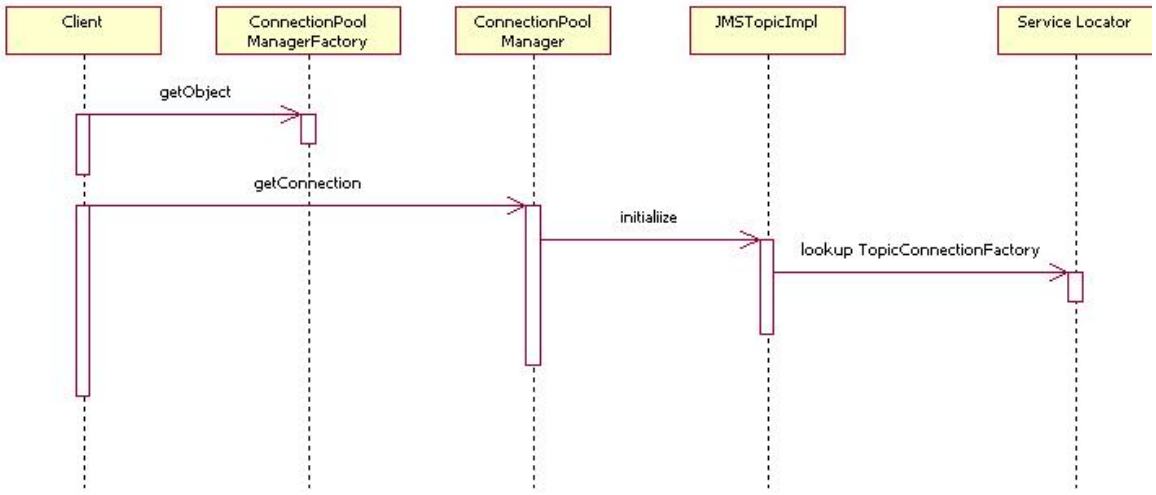
Caching of Retrieved EJB Home Interfaces

For the sake of efficiency, an initially retrieved EJB home interface is cached locally, for future access.



Caching of JMS Connection Factories

Framework JMS Queue and Topic Connection Pool implementations leverage the ServiceLocator to lookup the appropriate JMS Connection Factory.



Checked Exception Extensions

Framework exception handling has been extended so that all exception related classes (found in the com.framework.common.exception package) extend base class:

```
com.framework.common.exception.FrameworkCheckedException
```

This class replaces com.framework.common.exception.BaseException, which still exists and derives from FrameworkCheckedException, but will soon be deprecated.

Most framework exceptions thrown by the Framework make use of the following constructor of FrameworkCheckedException:

```
public FrameworkCheckedException(String msg, Throwable exc )
```

This constructor allows a FrameworkCheckedException to store the Throwable that was caught and caused the FrameworkCheckedException type class to be thrown from its source.

Functionality

Chained Exceptions

FrameworkCheckedException contains method

```
public Throwable getChainedException()
```

which allows you to return the original exception which caused this instance to be created, if applicable. If you would like to search for the existence of a specific exception within the chain of exceptions that led to the creation of the current instance, use method

```
FrameworkCheckedException
containsBaseExceptionTypeInChain(String)
```

Unique Exception ID

Each newly created FrameworkCheckedException is assigned a unique identifier. This is accomplished by calling the public static method

```
com.framework.common.misc.Utility.generateUID()
```

This call will cause the application specific Framework UID Generator to be invoked.

Host name

The toString() method of FrameworkCheckedException now includes the name of the host as part of the exception message format. This is useful in situations where your application is distributed across multiple machines.

Added static method

```
static public String getHostName()
```

to com.framework.common.misc.Utility to help with this.

Observing / Configuring an Executing Framework Instance

The Java Management extensions (JMX) provide a standard for the instrumentation of manageable resources and developing dynamic agents. JMX provides developers of Java technology-based applications across all industries with the means to instrument Java platform code, create smart agents and managers in the Java programming language, implement distributed management middle-ware, and smoothly integrate these solutions into existing management systems. The JMX architecture is divided into three levels:

Instrumentation level - gives instant manageability to any Java technology-based object. This level is aimed at the entire developer community utilizing Java technology. This level provides management of Java technology that is standard across all industries.

Agent level - provides management agents. JMX agents are containers that provide core management services which can be dynamically extended by adding JMX resources. This level is aimed at the management solutions development community and provides management through Java technology.

Manager level - provides management components that can operate as a manager or agent for distribution and consolidation of management services. This level is aimed at the management solutions development community and completes the management through Java technology provided by the Agent level.

In order to assist you in the external administration of a Framework-based application, the Framework has done its part by implementing key components as JMX Dynamic MBeans. These MBeans, along with an external MBean Server and JMX Manager, expose important actions and attributes to allow you to monitor and manipulate the Framework aspects of your application.

JMX White Paper - refer to this document to learn more about JMX and how to make use of registered MBeans.

Framework components defined as Dynamic MBeans

```
ActionDefMBean
ConnectionImplMBean
ConnectionPoolMBean
FrameworkCacheMonitor
FrameworkFrontControllerServletMBean
FrameworkLogHandlerMBean
HTTPRequestHandlerMBean
TaskJMSExecutionHandlerMBean
TaskJMSExecutionRegistryMBean
```

Registering Framework JMX MBeans

In order to access a Framework JMX MBean, it must be registered with a JMX Server Factory. The Framework supports two means of registering its MBeans:

Self-Registration
See attribute JMXSelfRegistration in the framework.xml configuration file, as well as the following section titled *Specifying the JMX Server Factory to Use*.

Programmatic Registration
Each of the Framework JMX services is of type

com.framework.common.jmx.FrameworkDynamicMBean.

This class provides the following method for MBean registration with the registered JMX Server.

```
public void registerMBeanWithServer();
```

Specifying the JMX Server Factory to use

In order to register the Framework JMX MBeans, you must specify a JMX MBean Server provider using the attribute JMX_MBEAN_SERVER_IMPLEMENTATION located in the framework.xml configuration file.

This provider must implement interface:

com.framework.common.jmx.IFrameworkJMXServerFactory

and provide an implementation for method

public MBeanServer createMBeanServer()

The Framework provides a default implementation for this interface in class:

com.framework.common.jmx.StandardJMXServerFactory

For Weblogic 6.1 or later, you can use the following class instead:

com.realmethods.jmx.WeblogicJMXServerFactory

This class file is located in RM_HOME\realmethods\lib\rmthirdparty.jar

How an MBean Server Domain is specified

The Framework uses the ApplicationID element found in the framework.xml as the unique identifier during the MBean registration of a Framework JMX MBeans. By having a unique name for each of your applications, this will ensure that multiple Framework-based applications having their MBeans registered to a single MBean Server will be able to be managed independently.

3rd Party JMX Management Console - XtremeJ

Although you can programmatically access the Framework Dynamic MBeans, it is often simpler to do so through a GUI-based application. realMethods recommends you download the XtremeJ Management Console for this task.

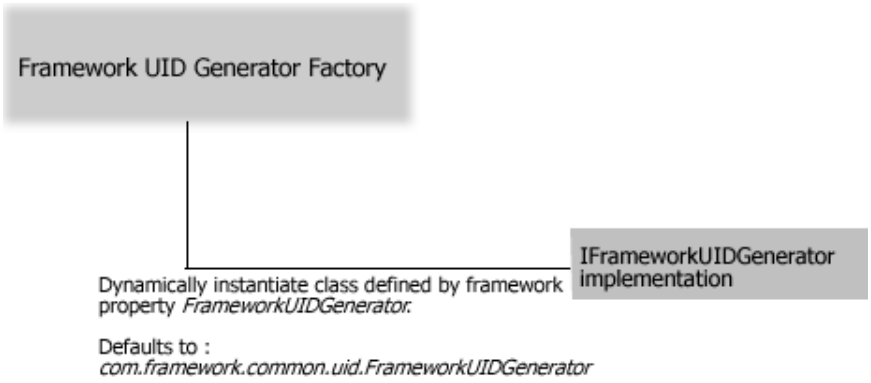
http://www.xtremej.com

By providing a set of Eclipse plug-ins (Eclipse is the open source IDE project from IBM) XtremeJ provides an implementation for the following vendors:

JBoss
Weblogic
MX4J

Unique Identifier Generation

Certain functionality of the framework and your application require the generation of a unique identifier. The framework provides a factory to produce an instance of an object capable of generating a unique identifier.



Default implementation defined by the framework

The framework defines a default implementation of the interface:

`com.framework.common.uid.IFrameworkUIDGenerator`

by the class

`com.framework.common.uid.FrameworkUIDGenerator`

This class full package name is the default value provided for the element

`FrameworkUIDGenerator=`

in the framework.xml file.

Defining and assigning a customer UID generator

Step #1: Define a Java class that implements interface:

`com.framework.common.uid.IFrameworkUIDGenerator`

Step #2: Provide an implementation for method:

```
public String generateUID( Object key )
```

Step #3: Provide the full class name as the value for element

`FrameworkUIDGenerator=`

in the framework.xml file.

Before using the AIB (code generator)

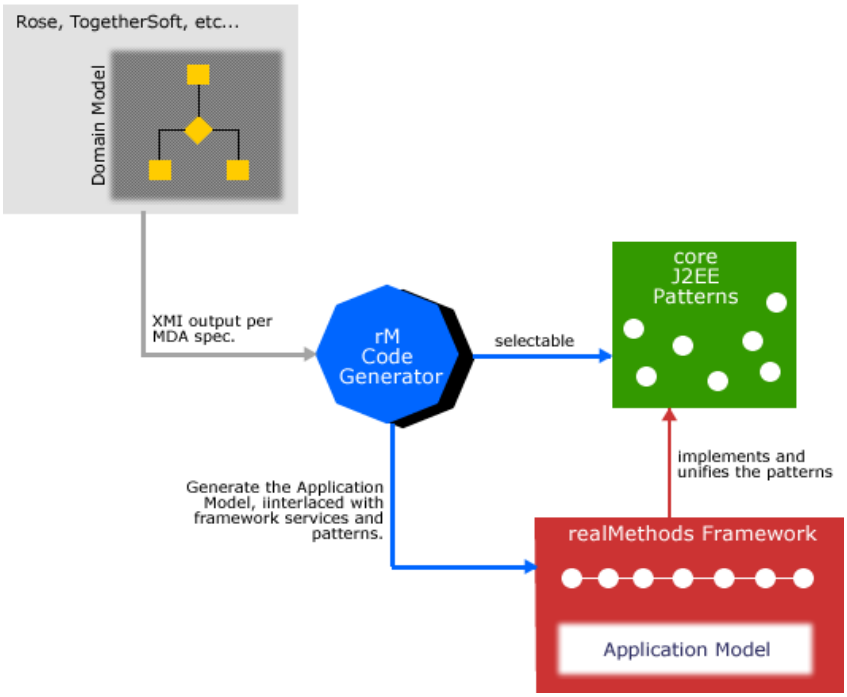
It is important to review the following steps before you consider using the AIB. In order for the AIB to accurately generate an application model that represents the needs of your domain, and assuming you are not reverse engineering from an existing set of POJOs or database schema, you must produce an appropriate and accurate domain model.

Creating the domain model

The resulting application model will only reflect the needs of the domain if they are specified correctly within the domain model. Per the MDA, the creation of a Platform Independent Model will greatly shield the application from upgrades and changes to the targeted technology platform, in this case J2EE. The framework only requires that the model you create be a reflection of the business requirements, void of:

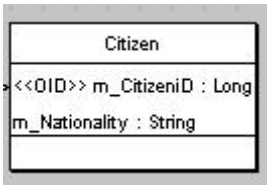
- J2EE as the underlying technology
- Core J2EE Design patterns to be reflected

The framework and the AIB are responsible for applying the J2EE artifacts and design patterns to your domain model, the result of which is your application model.



Framework's Application of J2EE Artifacts and Patterns

Declaring attributes in the model



Naming
A business object, by definition, only contains access methods and member data. The AIB uses the name of a member attribute to generate a "get" and "set" method. This makes the naming of the attribute very important in terms of readability.

Type
When declaring an attribute of a certain class type, make sure to include the fully qualified class name. This is not true of any class type from the "*java.lang*" package, which is automatically found by the compiler. This does hold true, however, for classes like *java.sql.Date*, as well as user defined classes.

- *Date: any field declared of type java.util.Date or java.util.Calendar will be replaced with java.sql.Date. This is for consistency in persistence amongst the difference JDBC vendors.*

- *java.lang Class types over intrinsic types. Always use Long, Integer, Boolean (etc..) in favor of their intrinsic counterparts (long, int, boolean, etc..).*

Primary Key Fields
In order for the AIB to recognize an attribute as a primary key field, you must assign the attribute with a stereotype called *OID*. During XMI parsing, the AIB treats such a stereotyped attribute differently than the others. More than one field of a class declaration may be tagged as a primary key field.

Order
The order in which you define the attributes within a class will be the order they are exported to the XMI file and the order they will be generated within the code. You should consider defining your primary key attributes (those declared with the *OID* stereotype) first.

Designer Note Concerning Associations
Do not apply the key field attributes of a child to a parent. Instead, use UML to denote a relationship between the two. The AIB will discover the relationship between business objects via an association definition, and generate code accordingly.

Designer Note Concerning Primary Key Fields
The AIB will generate a Primary Key class for each business object that has at least one attribute declared with an OID stereotype. In almost every case, a single attribute of Long type is sufficient to uniquely identify an object. The Framework is able to handle multiple key fields, as well as different types other than java.lang.Long, but compound primary keys should only be considered in the most extreme cases over a simple primary key of type Long.

Defining associations in the model

The attention to detail in defining associations between business objects within the model cannot be overstated. The AIB code generation techniques are directly related to the associations it discovers within the exported XMI from your UML tool. The following represent the keys to appropriately constructing the associations within the object model.

End Points
An association is made of two end points. The AIB mandates that each end point have an explicit cardinality associated with it. If an end point does not have a cardinality specified it is assumed to be 0-to-N, and not 0. The AIB also mandates that at least one end of the association be named. An exception will be thrown while parsing the XMI file if this "naming criteria" is not met.

Naming
As previously mentioned, each association end point with non-zero cardinality must have a name associated with it. The AIB will use this name, applied to the association, to generate one of the following, depending on the cardinality of the association end:

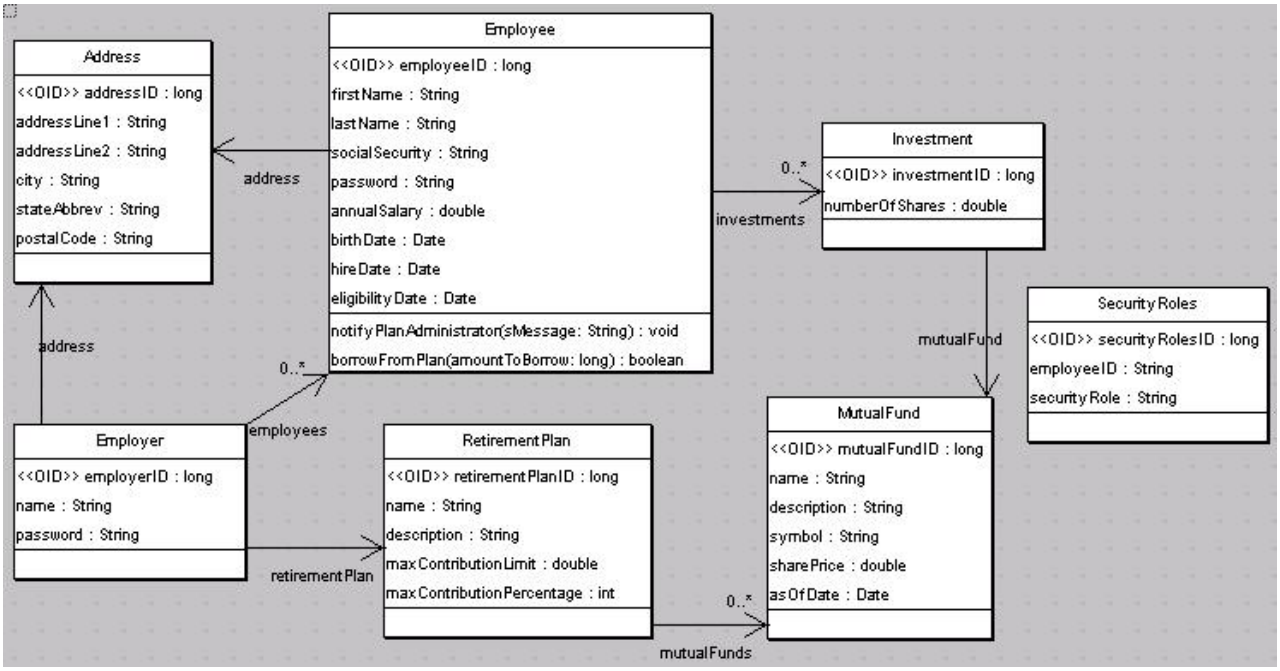
- *Cardinality of 1*

  This will cause the AIB to generate an attribute on the parent Business Object of the child class type, using the role name as the name of the attribute. This means a business object cannot be the parent of two associations with the same name.

  Reminder
  Attributes as associations are deduced and generated by the AIB and should not be explicitly declared as an attribute within the class diagram, but as an association instead.

- *Cardinality of m to n*
    - This cardinality causes the AIB to generate an attribute of type java.util.List

Let's examine such a relationship found in the Proof of Concept:



This diagram indicates that there is an association, named "employees", between an Employer and one or more Employees. It is important to not only name the association ends, but to name them in a descriptive manner, since these names are part of the naming within methods generated by the AIB.

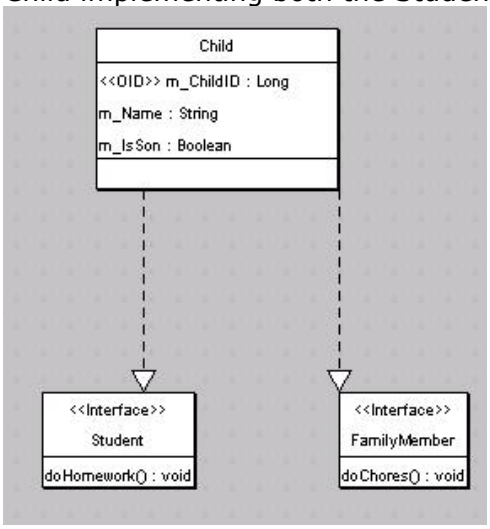
What will get generated based on this type of relationship is as follows:

```
EmployerProxy:
static public Collection getEmployees( EmployerPrimaryKey key )

EmployeeServiceSessionBean, EmployeeSessionRemote:
public Collection findAllEmployees( EmployerPrimaryKey key )

EmployeeDAO:
public Collection findAllEmployees( EmployerPrimaryKey key )
```

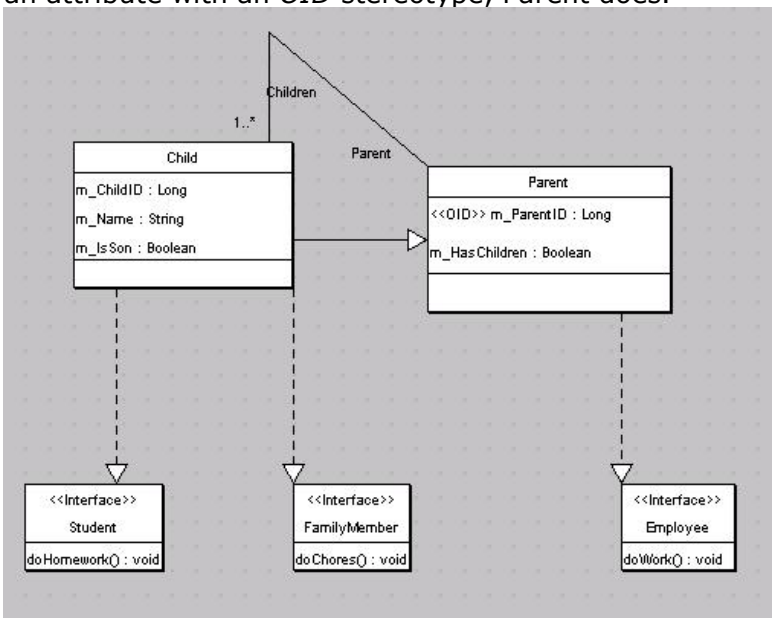
Inheritance considerations when modeling

The following considerations for modeling inheritance and interface support should be observed to effectively generate code using the AIB.

Consideration #1
A class may realize more than one interface. This is illustrated by the class Child implementing both the Student and FamilyMember interfaces.

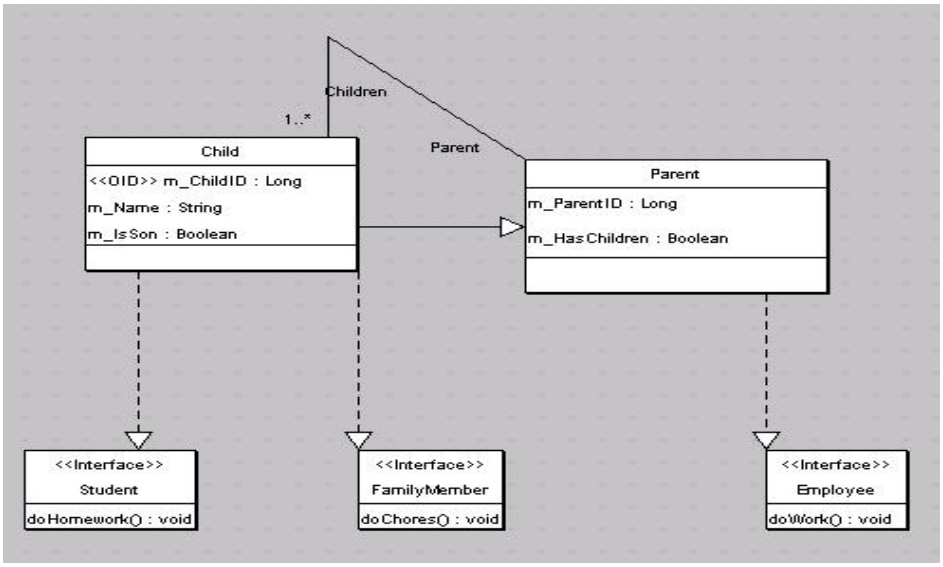


Consideration #2
A class with out a primary key attribute (stereotype of *OID*) can generalize (sub-class) another class that has one or more primary keys. The sub-class will be considered a persistent class, as will every other class in the same hierarchy. Notice in the following example that although Child doesn't contain an attribute with an *OID* stereotype, Parent does.

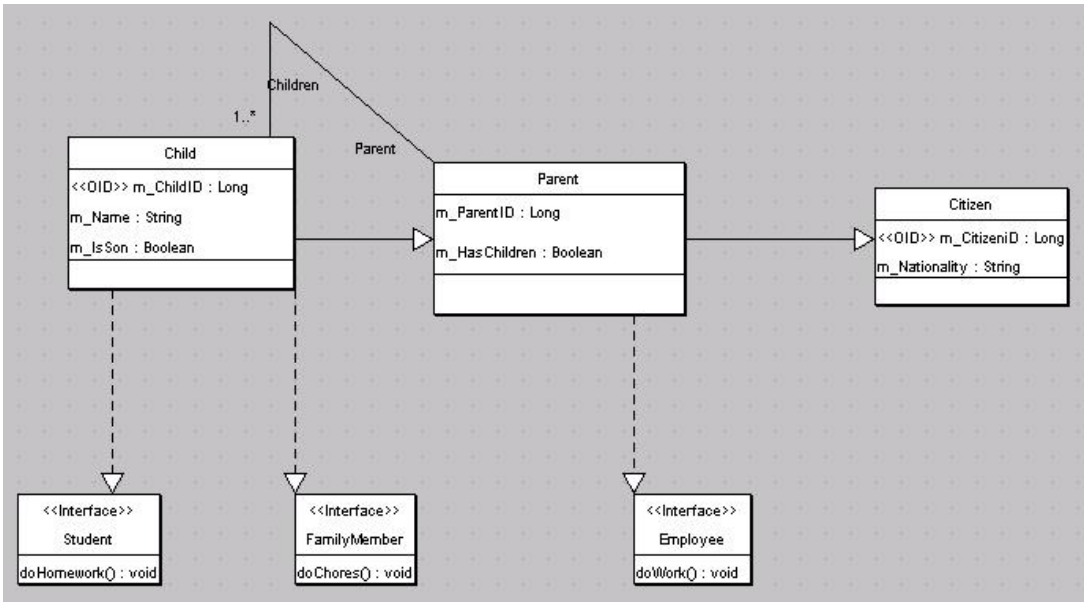


Consideration #3

A class with one or more primary key attributes can sub-class another class only if that class either contains at least one primary key, or is itself the sub-class of a class that contains at least one primary key. The point here is that the top most class in any hierarchy must contain at least one primary key attribute. The following inheritance relationship will be considered <u>invalid</u> by the AIB.



The problem here is that Child contains an attribute with the OID stereotype, but Parent does not. But, consider the following diagram.

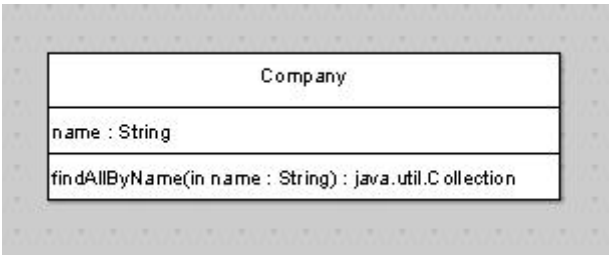


This inheritance chain is valid because the top most class (Citizen) contains at least one attribute with an *OID* stereotype. Child will be considered to contain a compound primary key, made up of attributes m_ChildID and m_CitizenID.

Defining One or More *findAllBy* Methods

The AIB allows you to define multiple "*findAllBy*" methods on a single Class. The importance of this is to generate all the code necessary to allow your application to return one or more of a single Class type based on one or more none primary key attributes.

For instance, you could define a single method on Company called *findAllByName*, as follows



Each "*findAllBy*" class you define must meet the following criteria:

1. Each input argument to the method must have the same name and type of an attribute found in the class, or its parent. Otherwise, the generated Hibernate query in the corresponding DAO will fail. Obviously your method can contain one or more input arguments

2. The method signature must return a java.util.Collection. If it doesn't, the AIB will make the correction during the import of the XMI file.

Note: You can also define "findAllBy" methods within your AIB project.

Exporting the domain model to an XMI file

Note about the Eclipse Modeling Framework (EMF)

If you are using Eclipse, you should take advantage of the Eclipse Modeling Framework (EMF). This modeling framework allows you to either create the required UML diagrams within Eclipse, or has the ability to import a Rational Rose MDL file. The resulting file(s) (of type .ecore) are able to imported directly into the AIB.

Most of the major UML tool vendors support the EMF as an Eclipse plug-in.

Learn more about EMF.

Exporting the domain model using Rational Rose

Note:
If your version of Rose doesn't support the exporting of a model to a UML 1.3 formatted XMI file, you will need to download the following plug-in:

http://www.realmethods/thirdparty/RoseXMLTools1.3.6.01.zip

Step 1 - Apply unique options for key field Attributes

Take the following steps to denote within the model that a particular attribute is a unique key field within the class.

a.) Right click the class that contains the attribute of interest and select 'Open Specification...'

b.) Select the attribute tab, and double click the attribute in the list.

c.) Type the word OID in the Stereotype combo-box.

d.) Click OK, and then OK again.

Step 2 - Generating an XMI file

a.) From the Tools menu option, select Export Model to UML

b.) De-select the *Export Diagrams* option to keep the XMI file size down.

c.) Select the XMI 1.0 format

Note:
There is a problem with Rose in exporting certain artifacts for XMI 1.1.
http://www.realmethods/thirdparty/RoseXMLTools1.3.6.01.zip

Exporting the domain model using Together/J

Step 1 - Apply unique options for key field Attributes

Take the following steps to denote within the model that a particular attribute is a unique key field within the class.

a.) Select Attribute Properties

b.) In the Properties tab, type OID in the stereotype text-box

c.) Click outside the text-box and close the dialog

Step 2a - TCC Version 6.x - Generating an XMI file

a.) From the File menu option, select Export->Export to XMI

b.) Choose XMI 1.0 (or XMI 1.1) for UML 1.3

c.) Select 'ASCII' as the encoding type

d.) Select the directory where the XMI file will be stored and click on Select

e.) Name the file is <project_name>.xmi

OR

Step 2b - TCC Version 5.5 - Generating an XMI file

a.) From the Tool menu option, select Export->Export to XMI

b.) Choose option UML 1.3 Unisys XMI (recommended for Rose)

c.) Click OK and select the directory where the XMI file will be stored

Using the Velocity Template Engine with the AIB

Velocity is an Apache/Jakarta based open source project which offers a Java-based template engine. The AIB code generator leverages this template engine to maximize the flexibility and extensibility in generating a realMethods based application.

Existing Templates

Each application file generated by the AIB is represented by a single Velocity template, located in an appropriately directory under the /aib/CodeTemplate directory. Each template file is a mix of code and Velocity macro calls. A Velocity macro is a function written to generate code in place of the macro call itself. All of the macros used by the shipped Velocity template files are contained in individual macro files (which themselves are Velocity template files) contained in the /aib/CodeTemplate/macros directory. Database related macro files are located in the /aib/CodeTempate/macros/db directory

AIB Velocity Variables

When the AIB executes the Velocity Template Engine, it makes available a set of variables that used within the template and macro files. The public methods of each of these variables are accessible within the template and macro file, just as though they were plain old java objects.

- **$aib** - com.framework.aib.codetemplate.AIBVelocityHelper
  Contains a set of important methods used to provide many different aspects of the loaded domain model.

- **$Utils** - com.framework.aib.helpers.Utils
  Act as a common helper class for String formatting, etc.

- **$classObject** - com.framework.aib.codetemplate.ClassObject
  For each business entity, a new *ClassObject* instance is applied to the Velocity Engine as the AIB loops through the Collection of business entities to generate on behalf of.

To learn more about the APIs of each of these object types, review the AIB JavaDocs in the /javadoc/aib directory, or view a handful of the delivered template and macro files

Creating New Templates

To actually create a new template file, you should refer to the Apache Jakarta Velocity documentation. The template files shipped with the AIB are good examples of what your template files will look like.

What's actually important is how to get the AIB to acknowledge your template file, generate a new file, and put it where you want it. This is where the aib-velocity.xml configuration file comes in. There is more on this file in the *Configuration* section below.

```
<custom-templates>
   <custom-template
      template-file-name="/action/new-action-template.vm"
      output-path="/action"
      output-file-name="NewAction.java"
      for-each-entity=”true” persistent-only=”true”
      prefix-the-package=”true”/>
</custom-templates>
```

The *for-each-class* attribute tells the AIB to apply the template file for each entity in the model. The name of the entity will be prefixed to the file name. The *persistent-only* attribute tells the AIB to apply the template file for only persistent entities (meaning at least one attribute in its hierarchy has a stereotype of *OID*). The *prefix-the-package* attribute will cause the AIB to place the generated file in a directory named by appending the user provided package name to the *output-path*.

Configuration

The \aib\aib-velocity.xml file contains the configuration parameters used to control the macros loaded into the Velocity Template Engine at runtime. If you would like to make your own custom macro file(s) available to the Templates (and other macros) at runtime, simply add the name of the macro file as a <macro> entry in the <macros> element.

```
<macros>
        <macro>macros/application-usom-macros.vm</macro>
        .
        .
</macros>
```

Importantly, your macro file must be specified relative to the element:

```
<codetemplate root-dir="/CodeTemplate" />
```

Changing Default Package Names

To change the name of any default package name suffix, simply edit the names declared in the CodeTemplate\macros\package-macros.vm file. Each package name is found as a macro within this file.

Note:
Package name changes are not applied towards files already generated, but only to new files generated.