

Corticon Foundation User Guide

Notices

Copyright agreement

© 2014 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

Business Making Progress, Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, Fathom, Making Software Work Together, OpenEdge, Powered by Progress, Progress, Progress Control Tower, Progress OpenEdge, Progress RPM, Progress Software Business Making Progress, Progress Software Developers Network, Rollbase, RulesCloud, RulesWorld, SequeLink, SpeedScript, Stylus Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. AccelEvent, AppsAlive, AppServer, BusinessEdge, Progress EasyI, DataDirect Spy, DataDirect SupportLink, EasyI, Future Proof, High Performance Integration, Modulus, OpenAccess, Pacific, ProDataSet, Progress Arcade, Progress Pacific, Progress Profiles, Progress Results, Progress RFID, Progress Responsive Process Management, Progress Software, ProVision, PSE Pro, SectorAlliance, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, WebClient, and Who Makes Progress are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

Please refer to the Release Notes applicable to the particular Progress product release for any third-party acknowledgements required to be provided in the documentation associated with the Progress product.

Table of Contents

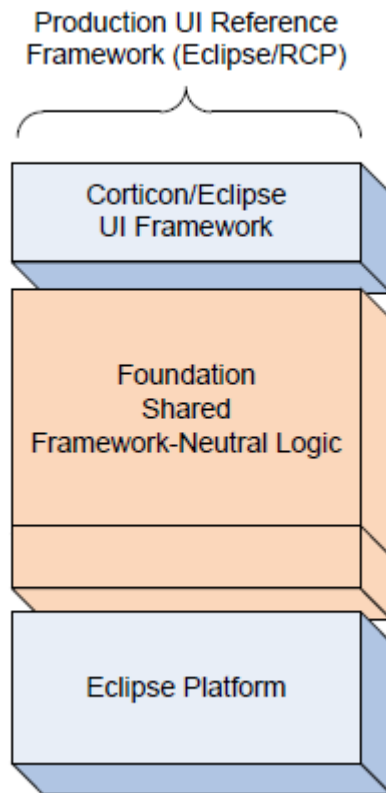
Chapter 1: Overview.....	7
Chapter 2: Installing and setting up Corticon Foundation.....	9
Chapter 3: Foundation APIs.....	15
Chapter 4: Technical Details.....	21

Overview

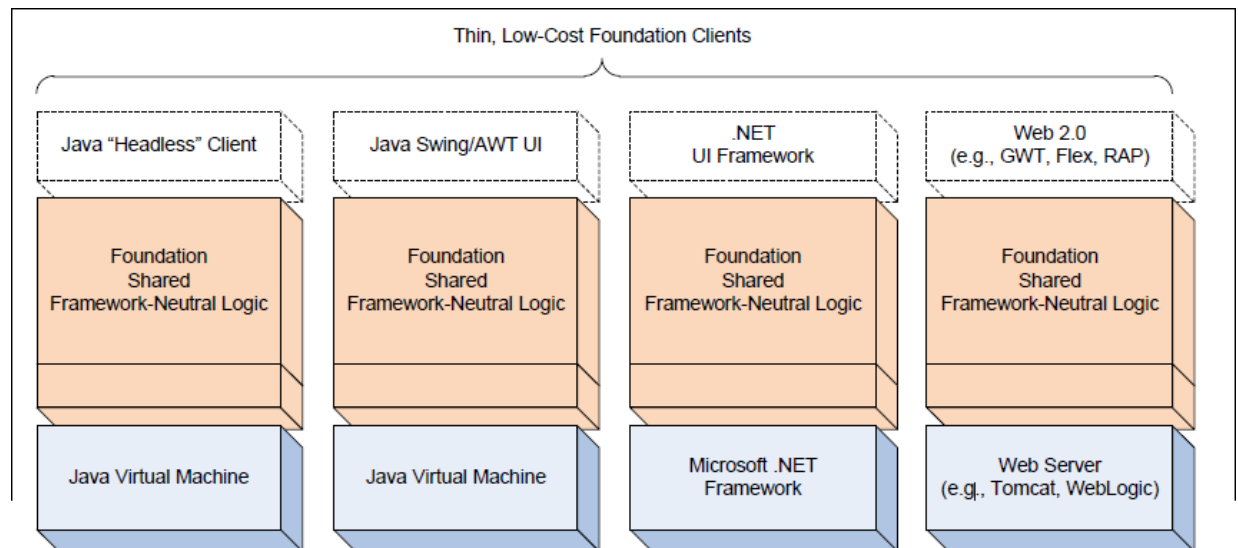
Corticon Foundation is a set of framework-neutral services you can use in your own applications. With Foundation, you can programmatically perform functions as simple as changing a single cell of a Rulesheet and as complex as submitting a test to the rule engine for execution. For example, you can:

- Create and update Vocabulary, Rulesheet, Ruleflow and Ruletest assets
- Execute rule analysis functions (such as ambiguity check, completeness check, expand, collapse)
- Execute tests, and compare output to expected results
- Generate reports from assets
- Analyze assets to gather metrics
- Create a simplified web-based UI to allow users to change rules in a limited way
- Create a comprehensive UI that delivers the entire rule authoring experience, comparable to Corticon Studio, to your users

In fact, using Foundation, your applications can do anything that Corticon Studio can do. That's because Corticon Studio itself uses Foundation services:



We've designed Foundation to maximize code reuse by packing as much logic into the services as possible, reducing UI framework coding to a minimum. This thin-UI philosophy allows you to implement new Foundation clients at low cost. Consider these possibilities:



Installing and setting up Corticon Foundation

Supported Configurations

Software Component	Version	Certified on Version
JDK	1.6.0 or higher	JDK 1.7.0_05
Eclipse (32-bit and 64-bit)	4.3.1 or later	4.3.1

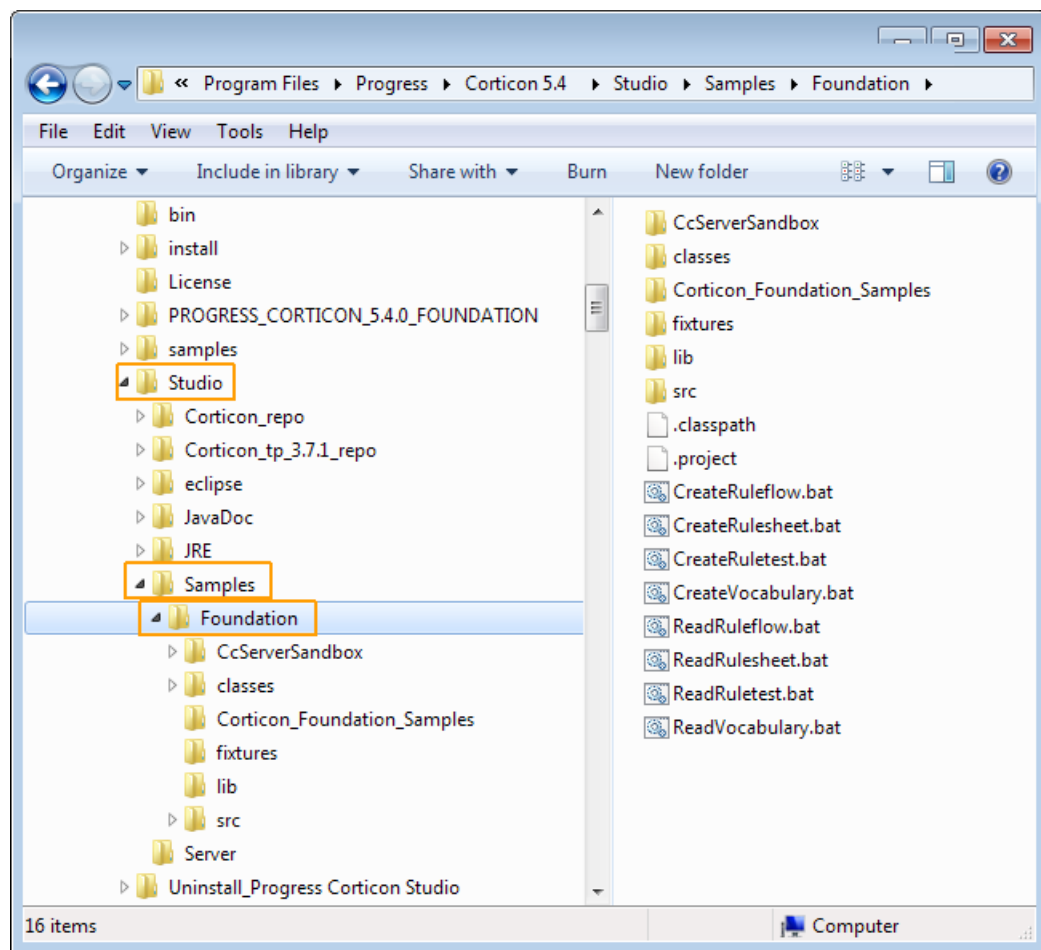
Unpacking the Corticon Foundation download

Download and install Corticon Studio 5.4 and any related service pack. Download the corresponding version's Foundation, such as `PROGRESS_CORTICON_5.4.0_FOUNDATION.zip`, and unpack to the root of the Studio installation where it sets up a folder with the same name. The package contains the following files:

File	Description
<code>Corticon_Foundation_Samples.zip</code>	Foundation samples
<code>Corticon_Foundation_API.jar</code>	Foundation Java classes
<code>Corticon_Foundation_I18n.jar</code>	Foundation resource bundles

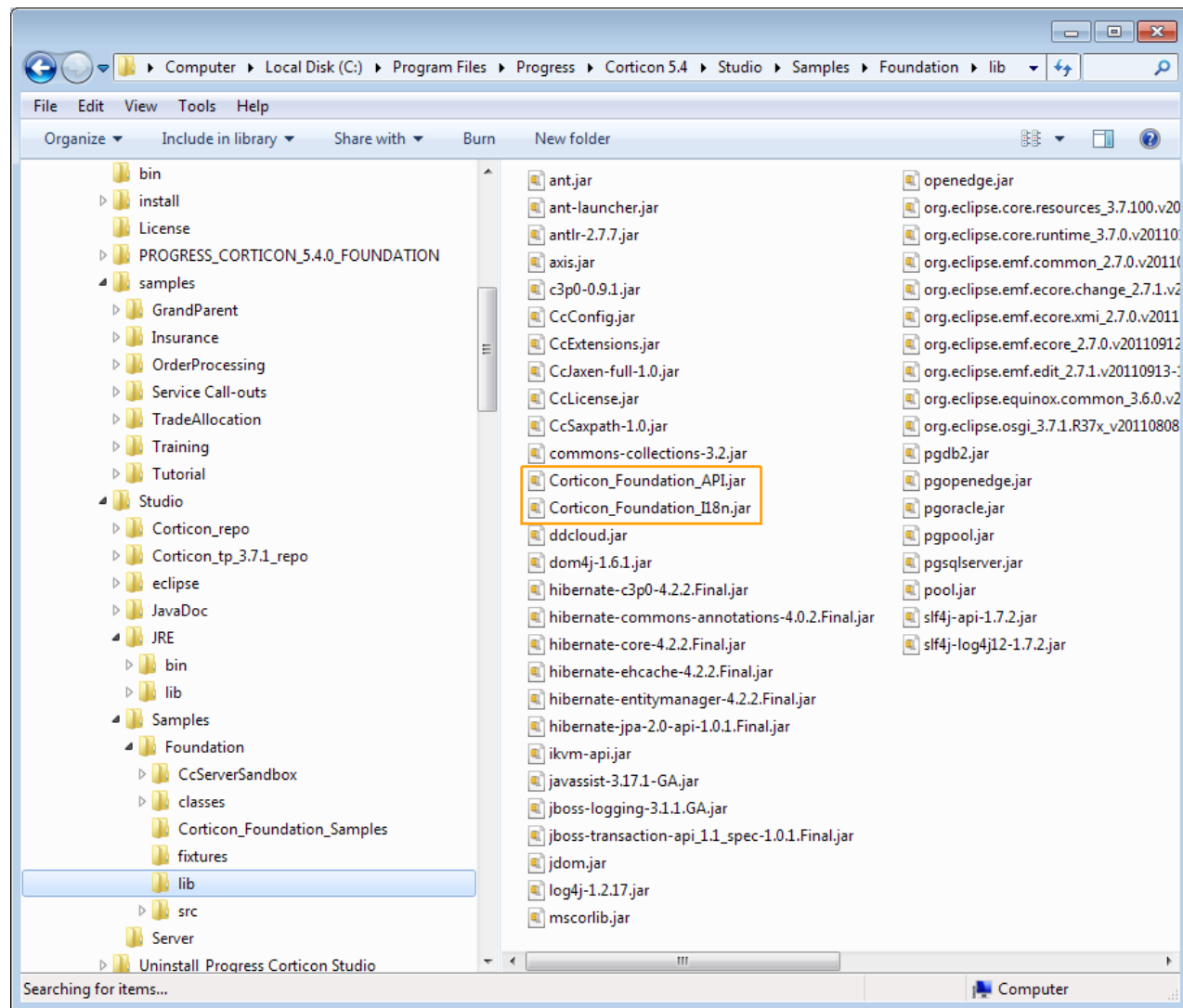
File	Description
Corticon_Foundation_NonPluginJars.zip	Third-party jars required by Foundation
Corticon Foundation User Guide.pdf	This guide with instructions for Foundation users
Corticon_JavaDoc_Foundation.zip	JavaDoc generated from the Foundation APIs
Corticon_Foundation_models	If you have access to Rational Rose or a tool that reads its files, models and class diagrams in that format.

Create a `Samples` directory under the `Studio` directory as shown, and then unpack the `Corticon_Foundation_Samples.zip` into that location so that its root is `Foundation` as shown:



Installing Foundation Libraries

Copy the loose JARs in the download, `Corticon_Foundation_API.jar` and `Corticon_Foundation_I18N.jar`, into the `Foundation/lib` directory, and then unzip the `Corticon_Foundation_NonPluginJars.zip` into that directory so that the JARs are all adjacent in that directory, as shown:



Installed Materials

The remaining materials in the unpacked download provide the documentation: this guide and the JavaDoc for the Foundation API. You can unpack the Javadoc right where it is.

Within the `Samples` directory, are files you will use in creating custom Foundation solutions:

- Java source files in `<InstallDir>\Samples\Foundation\src`
- Java class files in `<InstallDir>\Samples\Foundation\classes`
- Corticon assets in `<InstallDir>\Samples\Foundation\fixtures`
- Batch scripts in `<InstallDir>\Samples\Foundation`

In addition, the <InstallDir>\Samples\Foundation folder contains Eclipse .project and .classpath files, so that it can be imported into Eclipse IDE as a Project via **File > Import... -> General -> Existing Projects into Workspace**. There, you can use Eclipse to compile, execute and debug your programs.

The batch scripts execute samples in a Windows Command Prompt.

Batch Script	Java Source	Description
CreateRuleflow.bat	CreateRuleflow.java	Creates a new Ruleflow asset consisting of three selected Rulesheet assets. Deploys and executes the resulting Ruleflow using the Corticon In-Process Server.
CreateRulesheet.bat	CreateRulesheet.java	Creates a new Rulesheet asset.
CreateRuletest.bat	CreateRuletest.java	Creates a new Ruletest asset, executes the created test case then copies the output tree to the expected tree.
CreateVocabulary.bat	CreateVocabulary.java	Creates a new Vocabulary asset.
ReadRuleflow.bat	ReadRuleflow.java	Loads a Ruleflow and iterates over the Rulesheet shapes. For each Rulesheet, instantiates the Rulesheet Presentation API to look up the number of rule columns. Displays results on console.
ReadRulesheet.bat	ReadRulesheet.java	Loads a Rulesheet and displays the Scope, Conditions, Actions and Rule Statements on the console.
ReadRuletest.bat	ReadRuletest.java	Loads a Ruletest and displays the Input tree, Output tree, Expected tree and the Rule Messages on the console.
ReadVocabulary.bat	ReadVocabulary.java	Loads a Vocabulary and displays the Vocabulary tree on the console.

These examples can help get you started. The source code is straightforward and heavily-commented; every Foundation call has an accompanying explanation. The examples illustrate “best practices” for creating, loading and processing assets.

Verifying the Foundation Setup

To verify that everything works as intended, launch the script `CreateRuleflow.bat`, and enter the input 1, 2, and 3. The processing displays similar to the following:

```
Administrator: C:\Windows\system32\cmd.exe

C:\_53x_install_dir\home\Studio\Studio\Samples\Foundation>CreateRuleflow.bat
Foundation Example - CreateRuleflow
Choose DerivePrice Rulesheet [1]=Matinee [2]=Normal [3]=IMAX 3D:1
Choose DeriveTotal Rulesheet [1]=Simple Sum [2]=10 Percent Discount [3]=Family Discount:2
Choose PromoSnacks Rulesheet [1]=Free Coke [2]=Free Popcorn [3]=Free Hershey's Kisses:3
java version "1.6.0_37"
Java(TM) SE Runtime Environment (build 1.6.0_37-b06)
Java HotSpot(TM) 64-Bit Server VM (build 20.12-b01, mixed mode)
Decision service name DerivePrice1-DeriveTotal2-PromoSnacks3
Starting Progress Corticon Server : 5.3.4.0 -b5724
Progress Corticon Server log level : UIOLATION
Progress Corticon Server log path : C:\_53x_install_dir\home\Studio\Studio\Samples\Foundation\logs
Progress Corticon Server sandbox location : C:\_53x_install_dir\home\Studio\Studio\Samples\Foundation\CcServerSand
Creating Ruleflow file:C:\_53x_install_dir\home\Studio\Studio\Samples\Foundation\fixtures\DerivePrice1-DeriveTotal2-PromoSnacks3
```

The run generated a request and submitted it to the test server built into Studio:

```
Administrator: C:\Windows\system32\cmd.exe

-----INPUT-----
<?xml version="1.0" encoding="UTF-8"?>
<CorticonRequest xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="DerivePrice1-DeriveTotal2-PromoSnacks3">
  <WorkDocuments>
    <Family id="Family_id_1">
      <total xsi:nil="true" />
      <person id="Person_id_1">
        <name>Barbara</name>
        <ticketPrice xsi:nil="true" />
        <ticketType>Senior</ticketType>
      </person>
      <person id="Person_id_2">
        <name>Kenner</name>
        <ticketPrice xsi:nil="true" />
        <ticketType>Child</ticketType>
      </person>
      <person id="Person_id_3">
        <name>Dave</name>
        <ticketPrice xsi:nil="true" />
        <ticketType>Adult</ticketType>
      </person>
      <person id="Person_id_4">
        <name>Heather</name>
        <ticketPrice xsi:nil="true" />
        <ticketType>Adult</ticketType>
      </person>
    </Family>
    <Snack id="Snack_id_1">
      <name>Coke</name>
    </Snack>
    <Snack id="Snack_id_2">
      <name>Popcorn</name>
    </Snack>
    <Snack id="Snack_id_3">
      <name>Hershey's Kisses</name>
    </Snack>
  </WorkDocuments>
</CorticonRequest>
```

The server processed the request, and then returned the following reply.



```

Administrator: C:\Windows\system32\cmd.exe

-----OUTPUT-----
<?xml version="1.0" encoding="UTF-8"?>
<CorticonResponse xmlns="urn:Corticon" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
decisionServiceName="DerivePrice1-DeriveTotal2-PromoSnacks3">
  <WorkDocuments>
    <Family id="Family_id_1">
      <total>10.800000</total>
      <person id="Person_id_1">
        <name>Barbara</name>
        <ticketPrice>3.000000</ticketPrice>
        <ticketType>Senior</ticketType>
      </person>
      <person id="Person_id_2">
        <name>Kenner</name>
        <ticketPrice>3.000000</ticketPrice>
        <ticketType>Child</ticketType>
      </person>
      <person id="Person_id_3">
        <name>Dave</name>
        <ticketPrice>3.000000</ticketPrice>
        <ticketType>Adult</ticketType>
      </person>
      <person id="Person_id_4">
        <name>Heather</name>
        <ticketPrice>3.000000</ticketPrice>
        <ticketType>Adult</ticketType>
      </person>
    </Family>
    <Snack id="Snack_id_1">
      <name>Coke</name>
    </Snack>
    <Snack id="Snack_id_2">
      <name>Popcorn</name>
    </Snack>
    <Snack id="Snack_id_3">
      <name>Hershey's Kisses</name>
    </Snack>
  </WorkDocuments>
  <Messages version="1.0">
    <Message>
      <severity>Info</severity>
      <text>Child ticket price for Kenner is 3.000000</text>
      <entityReference href="#Person_id_2" />
    </Message>
    <Message>
      <severity>Info</severity>
      <text>Adult ticket price for Dave is 3.000000</text>
      <entityReference href="#Person_id_3" />
    </Message>
    <Message>
      <severity>Info</severity>
      <text>Adult ticket price for Heather is 3.000000</text>
      <entityReference href="#Person_id_4" />
    </Message>
    <Message>
      <severity>Info</severity>
      <text>Senior ticket price for Barbara is 3.000000</text>
      <entityReference href="#Person_id_1" />
    </Message>
    <Message>
      <severity>Info</severity>
      <text>Total for family is the sum of the prices of all tickets 12.000000</text>
      <entityReference href="#Family_id_1" />
    </Message>
    <Message>
      <severity>Info</severity>
      <text>Family gets unconditional 10% discount yielding 10.800000</text>
      <entityReference href="#Family_id_1" />
    </Message>
  </Messages>
</CorticonResponse>

```

That test proved that our setup was correct. Now we can start exploring Corticon Foundation.

Foundation APIs

Foundation APIs are implemented as Java classes. For Eclipse applications, the classes are contained in a set of Eclipse plug-ins. For non-Eclipse applications, the classes are contained in JARs.

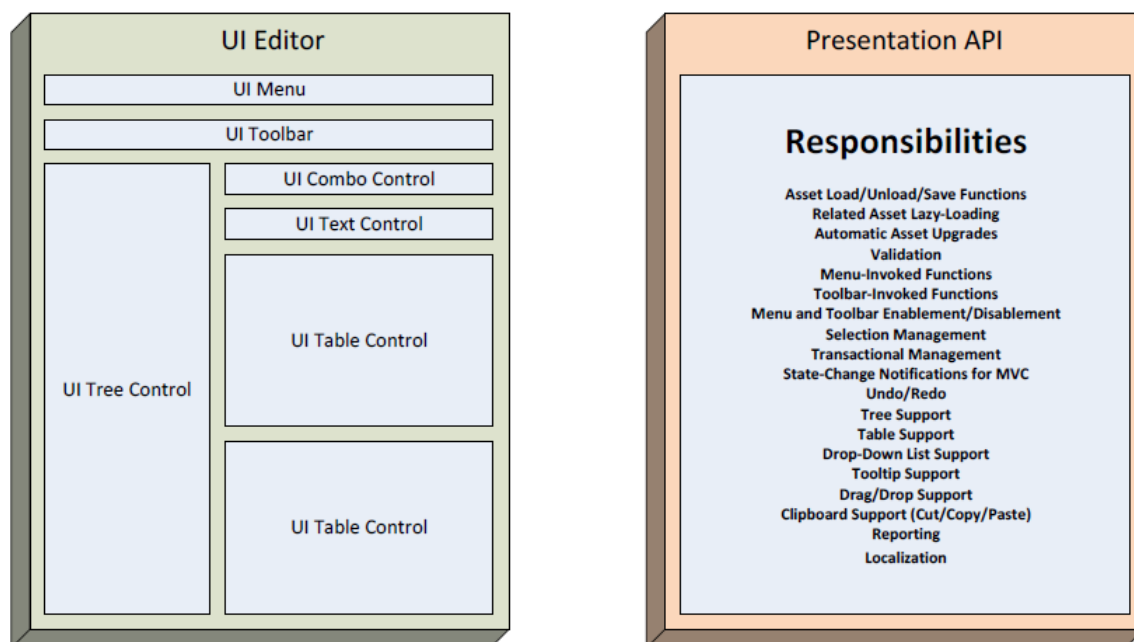
If you plan to invoke Foundation services from your own Eclipse plug-ins, you refer to API classes by declaring plug-in dependencies in your plug-in manifests. If you're writing traditional Java applications outside of Eclipse, you add Foundation JARs to your classpath.

There are several types of Foundation APIs.

Foundation API Type	Purpose
Presentation API	Contains services needed to retrieve information from an asset or to implement an editor.
Companion API	Augments Presentation API with additional functions.
Dialog "Smarts" API	Contains services needed to implement a UI modal dialog or wizard.
Canonical API	Contains services needed to maintain asset's canonical elements (currently for internal use only).

Presentation APIs

Each type of asset has at least one Presentation API. A Presentation API can be regarded as a service to support an editor; consequently, the API anticipates a particular UI presentation (that is, set of UI controls and their expected mechanical behaviors).



For example, a Rulesheet asset is presented to the user as a decision table. The decision table presentation, and the resulting user experience, requires a certain arrangement of table and tree UI controls. The Rulesheet Presentation API (interface `IRulesheettableModelAPI`) has methods to fully support that presentation.

If you use a Presentation API to create your own editor, your UI controls can often delegate to tailor-made Foundation methods, resulting in a thin UI layer. For example:

- Your menu and toolbar actions can delegate directly to corresponding Foundation methods with as little as one instruction.
- Your UI table and tree content providers and controllers can delegate to the API in order to retrieve table cell values and retrieve tree node children.
- Your table and tree cell editors can delegate to API methods to implement in-place cell editing.

In most cases, your editor can mechanically exchange string data with the Presentation API without having to transform it. This minimizes client-side logic, creating a loosely-coupled relationship that can reduce development costs.

A Presentation API has all of the functions you need to implement a full-featured editor; however, you might need to use only a handful of the available functions. For example, your application might need to retrieve information from existing assets, but not to create or update. In such cases, you'll be able to code just a few instructions to load an asset into a Presentation API, then call functions to retrieve just the information you need.

Presentation APIs are essential for developing most Foundation client applications.

Presentation API	Purpose
IVocabularytreeModelAPI	Allows you to create and update Vocabulary assets using a tree view and an accompanying properties table.
IRulesheettableModelAPI	Allows you to create and update Rulesheet assets using a decision table presentation.
IRuleflowDiagramModelAPI	Allows you to create and update Ruleflow assets using a diagram presentation.
IRuletestTabFolderModelAPI	Allows you to create and update Ruletest assets using a tab folder (each tab contains a Testsheet).

Companion APIs

A Presentation API may have Companion APIs when:

- There is optional functionality that may or may not be installed due to Corticon product licensing terms or other conditions.
- There is a group of related functions that can be moved to the Companion API to improve organization and clarity.

For example, the Rulesheet Presentation API has several Companion APIs that handle analysis functions. The Vocabulary Presentation API has several Companion APIs that handle Java Object Messaging and Database Access Functions.

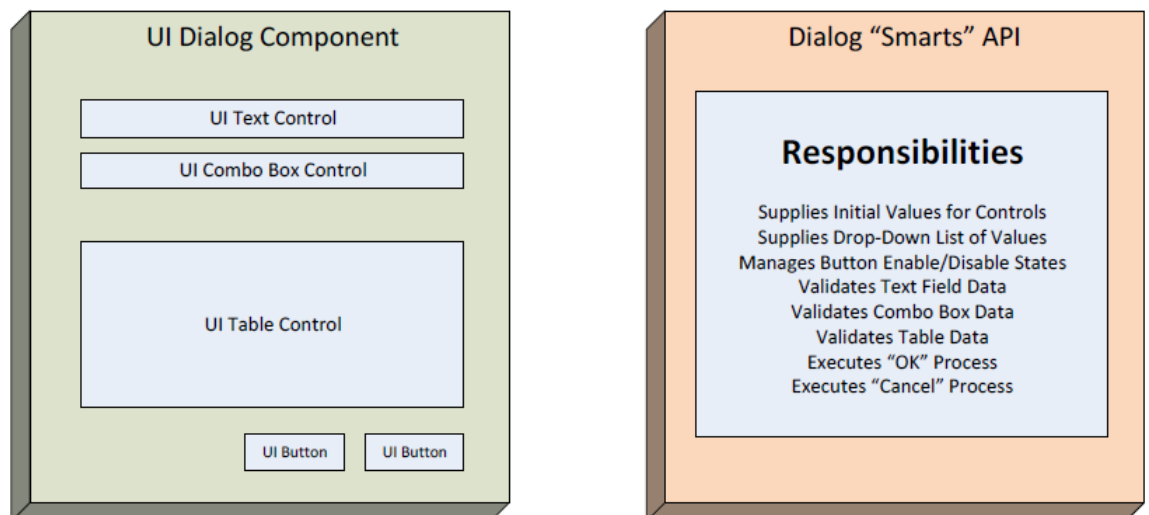
Companion API	Purpose
IAmbiguityCheckAPI	Checks a Rulesheet for ambiguities.
ICompletenessCheckAPI	Checks a Rulesheet for completeness and adds missing rules.
IExpandCollapseAPI	Expands/Collapses/Compress rules.
IDatabaseAccessAPI	Adds database access functions to the system.
IDatabaseMappingAPI	Handles mappings between Vocabulary elements and database metadata.
IDatabaseMetadataAPI	Handles database metadata.

Companion API	Purpose
IJavaMappingAPI	Handles mappings between Vocabulary elements and end-user java objects.
IJavaMetadataAPI	Handles end-user Java Object metadata.
IOperationsModelAPI	Provides access to metadata regarding Corticon rule operators and end-user extensions.
ITestsheettreeSetModelAPI	An extension of the Ruletest Presentation API, this API offers methods that operate at the Testsheet level.

Dialog “Smarts” APIs

Corticon Studio includes a number of UI dialogs and wizards. To minimize code redundancy, we have segregated the service logic needed to support these dialogs and wizards into Dialog “Smarts” APIs.

Generally, there is a one-to-one correspondence between a dialog or wizard and a Dialog “Smarts” API:



Similarly to a Presentation API, a Dialog “Smarts” API anticipates a particular UI control set.

Dialog “Smarts” API	Supported Dialog or Wizard
IImportAPI	Version 4 Asset Import Wizard
IVocabularyDialogAPI	New Vocabulary Asset Wizard

Dialog “Smarts” API	Supported Dialog or Wizard
IRuleflowDiagramDialogAPI	New Ruleflow Asset Wizard
IRulesheetdialogAPI	New Rulesheet Asset Wizard
IRuletestdialogAPI	New Ruletest Asset Wizard
IAssociationDialogAPI	Vocabulary Editor Association Dialog
IJavaMetadataImportdialogAPI	Vocabulary JOM Metadata Import Dialog
IRuletestrenameDialogAPI	Testsheet Rename Dialog
IRuletestTabFolderDialogAPI	Change Test Subject Dialog
IAssociationHREFDialogAPI	Ruletest Editor HREF-Style Association Dialog

Canonical APIs

In addition to a Presentation API, each type of asset has a Canonical API which contains methods that update the canonical (that is, fundamental) contents of each asset.

For example, at the canonical level, a Rulesheet asset contains rules, conditions and actions. These canonical elements are always the same in any presentation. The Rulesheet Canonical API (IRulesheetModelAPI) has methods that maintain these canonical elements.

Internally, Presentation APIs delegate to Canonical APIs to update canonical elements.

Foundation general design anticipates that in the future these canonical APIs may be exposed to Foundation client applications; however, at this time they are reserved for internal use only.

Canonical API	Purpose
IVocabularyModelAPI	Vocabulary canonical API
IRulesheetModelAPI	Rulesheet canonical API
IRuleflowModelAPI	Ruleflow canonical API
IRuletestModelAPI	Ruletest canonical API
ITestsheetModelAPI	Testsheet canonical API

Technical Details

Foundation Classpath Requirements

To use Foundation services for traditional (that is, non-Eclipse) development, you must add a set of JARs to your classpath. Corticon supplies the necessary JARs in three files:

File	Description
<code>Corticon_Foundation_API.jar</code>	Contains all Foundation API classes and interfaces.
<code>Corticon_Foundation_I18n.jar</code>	Contains internationalization bundles.
<code>Corticon_Foundation_NonPluginJars.zip</code>	Contains additional third-party JARs (must be unzipped).

Once these JARs are added to your classpath, you can start developing your application.

Eclipse Modeling Framework (EMF)

Foundation APIs use Eclipse Modeling Framework (EMF) for asset management. EMF is a crucial component of the Foundation technology stack, so if you're facing an ambitious Foundation project, you'll want to visit <http://www.eclipse.org/emf> to learn more.

EMF is a toolkit that supports MDA (Model-Driven Architecture). It allows software engineers to define a domain model using UML class diagrams.

Here, Rational Rose was used to create class diagrams that represent our Vocabulary, Rulesheet, Ruleflow and Ruletest assets; then, we used EMF tools to transform those class diagrams into generated Java classes.

Foundation APIs contain instances of those generated EMF classes. When you invoke Foundation API state-changing functions, Foundation APIs will create, update and remove EMF model objects.

Generally, you don't need to know much about our domain model to use Foundation; nevertheless, our Rational Rose models are available to Foundation users in

`Corticon_Foundation_Models.zip`.

Asset URIs

Foundation APIs use EMF URIs to uniquely identify assets. Resource-oriented methods that create or load assets require you to supply EMF URIs (class `org.eclipse.emf.common.util.URI`) as parameters.

EMF URIs can refer to a wide range of storage systems (such as file, http, RDBMS), but in practice, your application will typically use either File URIs or Platform URIs.

File URIs refer to assets in your file system. You create a File URI by calling EMF URI static method `createFileURI`, supplying a String that specifies the path name of your asset:

```
URI myURI = URI.createFileURI("C:/Fixtures/MyRulesheet.ers");
```

Platform URIs refer to resources in an Eclipse workspace. You create a Platform URI by calling EMF URI static method `createPlatformResourceURI`, supplying an Eclipse workspace-relative path name of the asset:

```
URI myURI = URI.createPlatformResourceURI("/Project/MyRulesheet.ers", true);
```

Note that you only use Platform URIs if you're developing Eclipse plug-ins. traditional Java applications typically use File URIs.

Instantiating and Disposing of APIs

You instantiate an API via its factory class. The factory class name is simply the name of the API with the "Factory" added to the end and without the letter "I" at the beginning.

Presentation API	Factory Class Name
IVocabularytreeModelAPI	VocabularytreeModelAPIFactory
IRulesheettableModelAPI	RulesheettableModelAPIFactory
IRuleflowDiagramModelAPI	RuleflowDiagramModelAPIFactory
IRuletestTabFolderModelAPI	RuletestTabFolderModelAPIFactory
ITestsheettreeSetModelAPI	TestsheettreeSetModelAPIFactory

To instantiate the Rulesheet Presentation API, you code:

```
IRulesheetTableModelAPI myAPI = RulesheetTableModelAPIFactory.getInstance();
```

If you're developing an editor, you will typically instantiate an API at the beginning of an editing session, and maintain a reference to that API instance until the session is ended. When an API is no longer necessary, it is a good idea to call the dispose method to immediately free resources:

```
myAPI.dispose();
```

Load/Update/Save Cycle

Foundation APIs are stateful in that they contain an in-memory copy of an asset. They facilitate a load/update/save cycle that is typical of editors.

You call `loadResource` to load an asset into the API. Then, you can call API accessor methods (such as `get<foo>` and `find<foo>`) to retrieve information from the asset. You can call API state-change methods (such as `set<foo>`, `add<foo>` and `remove<foo>`) methods to change the in-memory asset state. Finally, you can call `saveResource` to save the in-memory asset back to disk.

```
IRulesheetTableModelAPI myAPI = RulesheetTableModelAPIFactory.getInstance();
Resource myResource =
myAPI.loadResource(URI.createFileURI("C:/Fixtures/MyRulesheet.ers"));

String myExpression = (String)myAPI.getCellValue
    (IRulesheetTableModelAPI.MATRIX_ID_CONDITIONS,
     IRulesheetTableModelAPI.COLUMN_INDEX_CONDITION_EXPRESSION, 0);
System.out.println("Original condition expression "+myExpression);
myAPI.setCellValue(IRulesheetTableModelAPI.MATRIX_ID_CONDITIONS,
    IRulesheetTableModelAPI.COLUMN_INDEX_CONDITION_EXPRESSION, 0, "Person.name
= 'Mike'");
System.out.println("Condition updated to Person.name = 'Mike'");
myAPI.saveResource(myResource);
myAPI.dispose();
```

Creating a New Asset

You can use a Dialog “Smarts” API to create a new asset.

```
IRulesheetTableModelAPI myAPI = RulesheetTableModelAPIFactory.getInstance();
URI myRulesheetURI = URI.createFileURI("C:/Fixtures/NewRulesheet.ers");
URI myVocabularyURI = URI.createFileURI("C:/Fixtures/MovieTheater.ecore");
IRulesheetDialogAPI myRulesheetDialogAPI =
RulesheetDialogAPIFactory.getInstance();
myRulesheetDialogAPI.createRulesheet(myAPI, myRulesheetURI, myVocabularyURI);
myAPI.dispose();
```

`IRulesheetdialogAPI` is the Dialog “Smarts” API used by the Corticon Studio New Rulesheet Wizard.

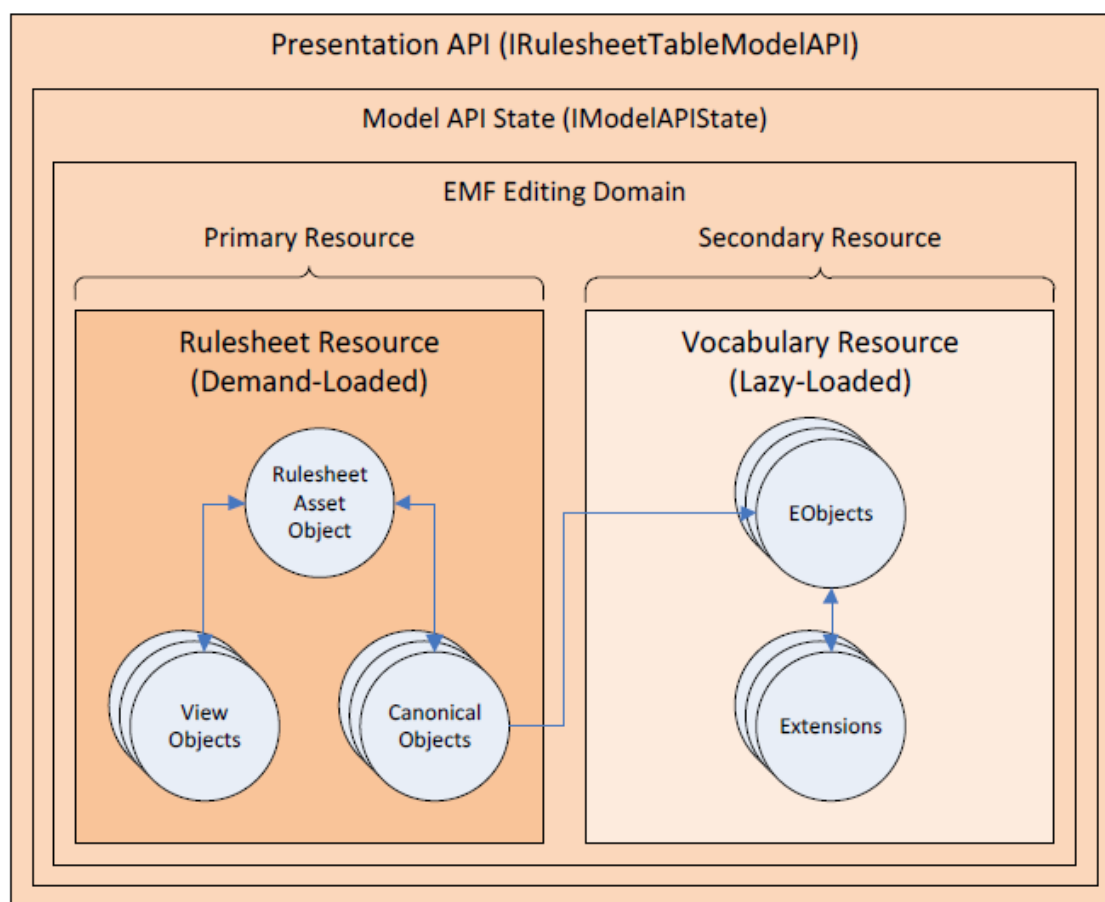
Method `createRulesheet` requires three parameters:

- An instance of the Presentation API (`IRulesheetTableModelAPI`)
- The URI of the new Rulesheet asset to be created
- The URI of the Vocabulary to be associated with the new Rulesheet asset

The Dialog “Smarts” API will create and save `C:/Fixtures/NewRulesheet.ers` in the file system.

Model API State

A Foundation API is a wrapper for a Model API State (`IModelAPIState`). The Model API State contains an EMF Editing Domain. The Editing Domain contains zero or many Resource instances. Each Resource instance represents a single asset, such as a Rulesheet. A Resource in turn contains EMF model objects. Those model objects are the in-memory representation of the loaded assets.



A Foundation API will automatically create a Model API State as it is needed. You can also explicitly create your own Model API State and then share it among multiple Foundation API instances.

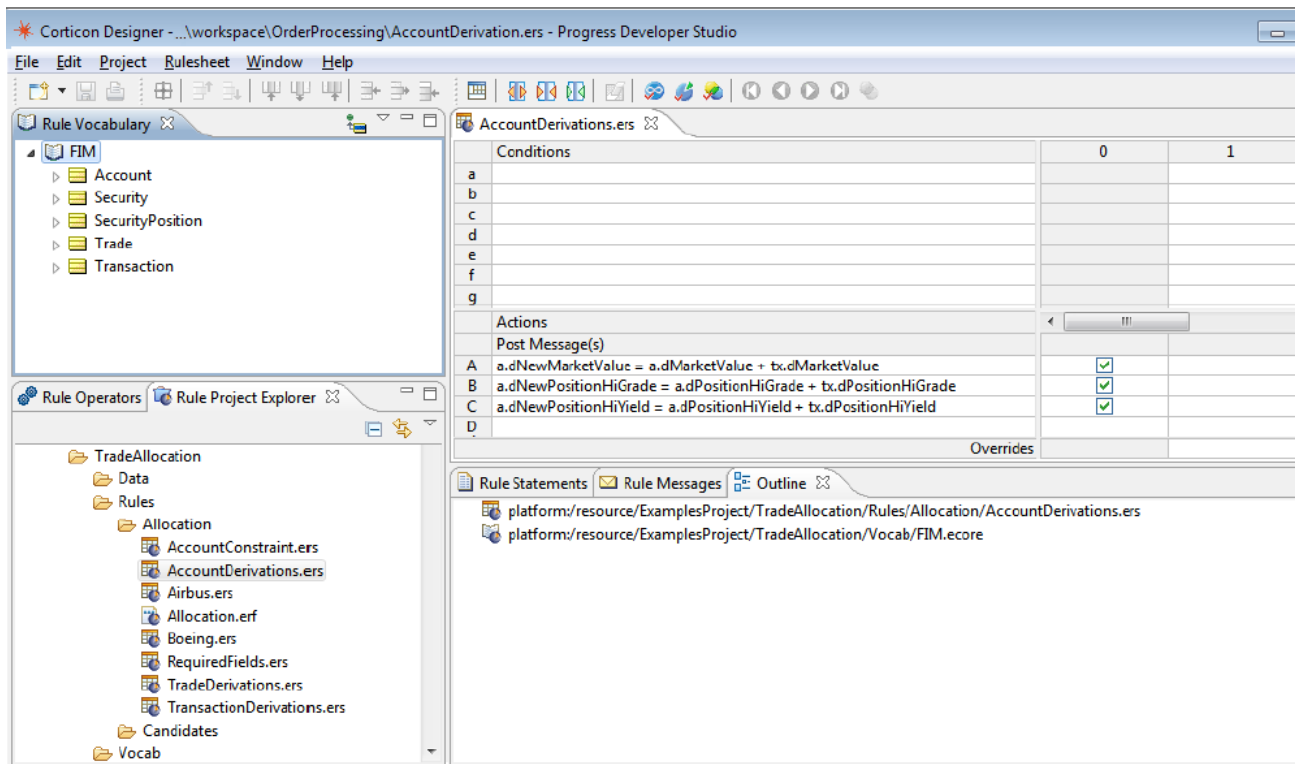
Demand-Loading and Lazy-Loading

EMF (and hence Foundation) allows assets to “point” to one another, on disk as cross-document references, and in memory as Java object references. For example, a Rulesheet asset “points” to its Vocabulary asset.

When you demand-load a Rulesheet asset via the `loadResource` method, EMF will automatically lazy-load the corresponding Vocabulary asset. This lazy-loading will occur the moment any information in the Vocabulary is referenced.

To get an intuitive feel for lazy-loading, you can experiment with Corticon Studio and display the Outline View (**Window -> Show View -> Other -> General -> Outline**). This view will show all Resources currently loaded into the shared Model API State.

In the screen image below, the Rulesheet Editor has explicitly demand-loaded Rulesheet `AccountDerivations.ers`, and Foundation has lazy-loaded the accompanying Vocabulary `FIM.ecore` which is visible in the Rule Vocabulary View.



As you open and close editors, note how the Outline View changes to reflect the set of the Resources that have been loaded into the shared Model API State.

Foundation will never load more than one copy of the same asset. If you attempt to demand-load a Resource that has already been loaded, the API will return control to your application without affecting the in-memory asset.

Primary and Secondary Resources

An API has one Primary Resource. This is a special designation given to a Resource that has been demand-loaded via the `loadResource` method. You can also explicitly set the Primary Resource by calling the API `setPrimaryResource` method.

Due to lazy-loading, your API will typically contain at least one Secondary Resource (such as the associated Vocabulary). Furthermore, when you demand-load Ruleflow and Ruletest assets, you may indirectly lazy-load any number of additional Secondary Resources, because a Ruleflow may reference any number of Rulesheets, and a Ruletest may reference multiple Ruleflow and Rulesheet test subjects.

Note that the Primary Resource defines the context of the API. Most API functions implicitly operate on the Primary Resource.

Sharing Model API State

Any number of Foundation APIs can share a single Model API State instance. In fact, Corticon Studio uses a Singleton shared Model API State for all editors and views.

To make two APIs share the same state, you use the `setAPIState` method immediately after the second API is instantiated.

```
IRulesheetTableModelAPI myAPI1 = RulesheetTableModelAPIFactory.getInstance();
Resource myResource =
myAPI1.loadResource(URI.createFileURI("C:/Fixtures/MyRulesheet.ers"));
IExpandCollapseAPI myAPI2 = ExpandCollapseAPIFactory.getInstance();
myAPI2.setAPIState(myAPI1);
myAPI2.setPrimaryResource(myResource);
```

Method `setAPIState` sets the state of `myAPI2` to that of `myAPI1`, effectively sharing the state. Note that when you share Model API State in this manner, you must also explicitly call `setPrimaryResource` on the second API. This will ensure that both APIs operate on the same Resource.

Alternatively, you can create a Model API State independently then share it with any number of APIs via method `setModelAPIState`.

```
IModelAPIState myState = new ModelAPIStateImpl();
IRulesheetTableModelAPI myAPI1 = RulesheetTableModelAPIFactory.getInstance();
IRulesheetTableModelAPI myAPI2 = RulesheetTableModelAPIFactory.getInstance();
myAPI1.setModelAPIState(myState);
myAPI2.setModelAPIState(myState);
myAPI1.loadResource(URI.createFileURI("C:/Fixtures/DerivePricel.ers"));
myAPI2.loadResource(URI.createFileURI("C:/Fixtures/PromoSnacks1.ers"));
.
.
.
myAPI1.dispose();
myAPI2.dispose();
myState.dispose();
```

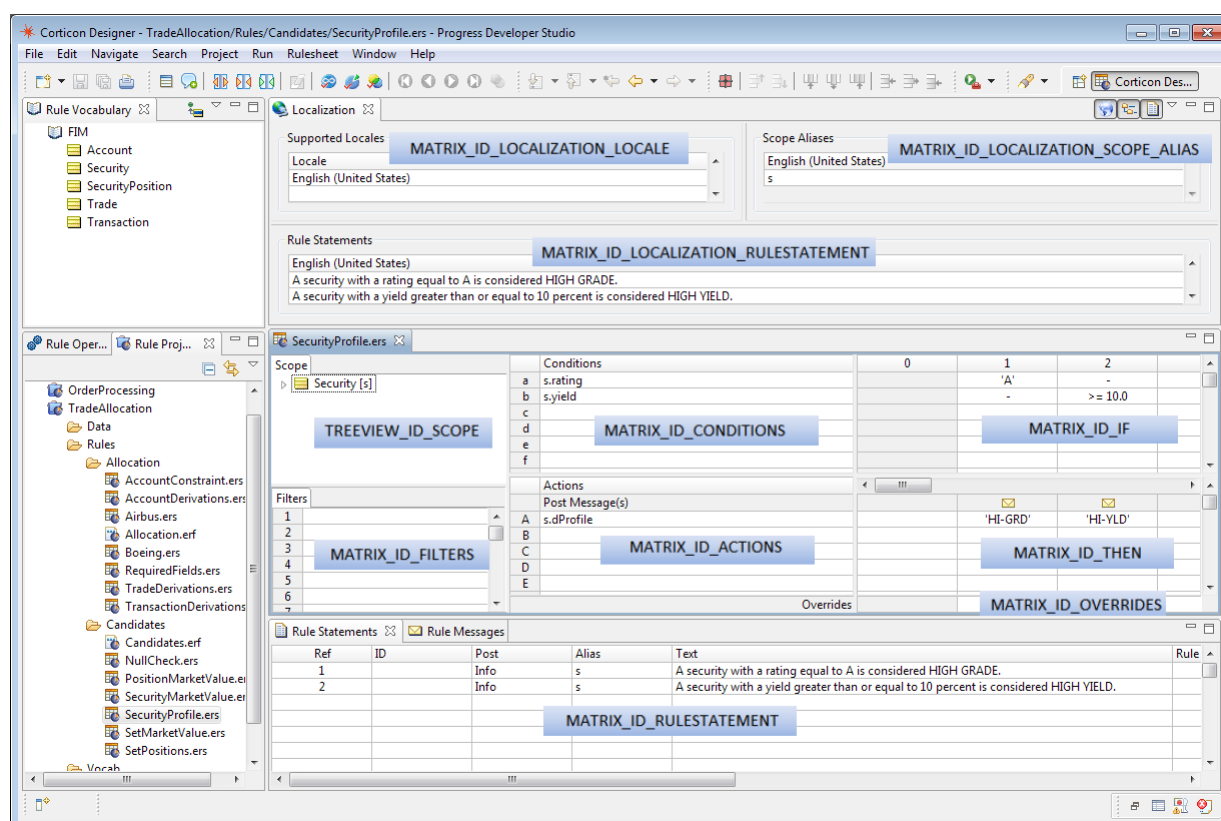
Associated APIs

Foundation APIs offer methods that return associated APIs to facilitate convenient access to Secondary Resources. For example, the Rulesheet Presentation API has method `getVocabularyModelAPI` which returns associated API `IVocabularyModelAPI` whose Primary Resource is set to the Vocabulary. Using these methods, you can traverse from one API to another to access information you need. For example, to find entity `Person` in the Vocabulary associated with `myAPI` (`IRulesheettableModelAPI`), you code the following:

```
IEntity myPerson = myAPI.getVocabularyModelAPI().findEntity("Person");
```

Control IDs

A Control ID is an integer number that identifies a UI control. Consider the Corticon Studio Rulesheet Editor which uses 10 table controls and one tree control. These controls are assigned unique Control IDs, which are declared as constants in `IRulesheettableModelAPI`:



Many Presentation API methods require you to specify a control ID in order to retrieve or update data associated with the control.

Accessing Tabular Data

Presentation APIs offer standardized methods to allow you to retrieve and update tabular information. You can use these methods to do anything from retrieving a single cell of data to implementing custom UI table controls, complete with in-place cell editing.

Method	Description
<code>getColumnCountActual(aMatrixID)</code>	Returns the actual column count (that is, the number of columns in the table that contain data).
<code>getColumnCountDisplayed(aMatrixID)</code>	Returns the displayed column count (that is, the number of columns in the table that contain data plus empty padding rows).
<code>getrowCountActual(aMatrixID)</code>	Returns the actual row count (that is, the number of rows in the table that contain data).
<code>getrowCountDisplayed(aMatrixID)</code>	Returns the displayed row count (that is, the number of rows in the table that contain data plus empty padding rows).

Method	Description
<code>getCellValue(aMatrixID, aColumn, aRow)</code>	Returns the localized value of the cell as a string.
<code>getListCellItems(aMatrixID, aColumn, aRow)</code>	Only applicable to <code>CELL_EDITOR_COMBO</code> or <code>CELL_EDITOR_MULTISELECT_COMBO</code> , this function returns a localized list of strings you use to populate your cell editor drop-down or combo box control.
<code>setCellValue(aMatrixID, aColumn, aRow, aValue)</code>	Sets the value of a cell.
<code>getColumnHorizontalAlignment(aMatrixID, aColumn)</code>	Returns a constant that specifies the horizontal alignment that your UI should use to render data in the column: <code>Constants.COLUMN_HORIZONTAL_ALIGNMENT_LEFT</code> <code>Constants.COLUMN_HORIZONTAL_ALIGNMENT_CENTER</code> <code>Constants.COLUMN_HORIZONTAL_ALIGNMENT_RIGHT</code>
<code>getCellEditorType(aMatrixID, aColumn, aRow)</code>	Returns a constant that specifies the type of cell editor that your UI should use for in-place editing: <code>Constants.CELL_EDITOR_TEXT</code> <code>Constants.CELL_EDITOR_COMBO</code> <code>Constants.CELL_EDITOR_MULTISELECT_COMBO</code> <code>Constants.CELL_EDITOR_CHECKBOX</code> <code>Constants.CELL_EDITOR_MULTILINE_TEXT</code>
<code>isCellEditable(aMatrixID, aColumn, aRow)</code>	Only applicable to <code>CELL_EDITOR_COMBO</code> or <code>CELL_EDITOR_MULTISELECT_COMBO</code> , this function specifies whether your UI should allow direct text editing or should restrict the user to select items in the drop-down list.
<code>isCellValid(aMatrixID, aColumn, aRow)</code>	Indicates whether the cell is valid. This allows your UI to render invalid cells with a different font color (such as red).

Method	Description
<code>hasErrors(aMatrixID, aColumn, aRow)</code>	Indicates whether the cell has any errors.
<code>hasWarnings(aMatrixID, aColumn, aRow)</code>	Indicates whether the cell has any warnings.

Although you can use these table-oriented methods for any purpose, they are especially well-suited for implementing your own UI table controls.

Typically, a UI table control, such as Swing `Jtable` or SWT `JFace tableViewer`, will have a table model or content provider that provides data to the table widget from a domain model. A content provider is an adapter that allows any data to be presented as a two-dimensional grid.

When a table cell is about to be rendered, the table widget will interrogate the content provider to fetch the text string to display in that cell. The widget will supply the coordinates of the cell, and the content provider will return the string that should be displayed.

Method `getCellValue` allows you to retrieve the value of a cell from the API. You supply a control ID, column index and row index; `getCellValue` will return a string that contains the external representation of the cell, namely the value that should be presented to the user. The API will automatically localize the returned string based on the API locale setting.

In many cases, your content provider can be reduced to a single instruction. For example, here is the actual code in our Corticon Studio Rulesheet Conditions table content provider:

```
public Object getContentAt(int aiColumnIndex, int aiRowIndex)
{
    return
    CcUtil.getEmptyString(iIRulesheetTableModelAPI.getCellValue(getMatrixID(),
        aiColumnIndex, aiRowIndex));
}
```

This code returns cells from the Rulesheet Conditions table. Note that:

- Method `getMatrixID()` returns constant `IRulesheettableModelAPI.MATRIX_ID_CONDITIONS`
- Method `CcUtil.getEmptyString` converts `null` to empty string as required by our table widget

Corticon Studio uses tabular data extensively, so methods `getCellValue` and `setCellValue` will play a prominent role in many Foundation client applications. Although these methods have been designed to support table content providers, they can be used to access tabular data for any purpose.

Accessing tree-Structured Data

Corticon Vocabulary, Rulesheet and Ruletest assets all contain tree-structured data. For example, the Rulesheet asset contains the Rulesheet Scope, which is internally represented as a hierarchy of `ScopeNode` object instances. Presentation APIs offer methods to help you access and potentially display tree-structured data.

Tree controls, such as the Swing `Jtree` and the SWT `JFace treeViewer`, will have a tree model or content provider that supplies data to the tree widget from a domain model.

The tree widget designates one particular object as the root node. The root node has a collection of child nodes, each of which may in turn have additional child nodes. Also, each node typically has a reference to its parent node.

When a tree widget is drawn or refreshed, the UI tree control will recursively process each visible node and will interrogate the content provider to determine:

- The list of child nodes belonging to the current node
- The parent node of the current node
- The icon to use to represent the current node
- The icon decoration to apply to the current node (error or warning)
- The text to be displayed adjacent to the icon

Presentation APIs offer methods to make it easier to deal with these tree-related issues. Although each API has unique tree-oriented method names (and in some cases special parameters), the functions behave similarly and follow standard conventions.

Vocabulary tree (IVocabularyModelAPI)

Method	Description
<code>getVocabularyElementChildren(aIElement, abExpandAssociations, abIncludeInherited, abSort)</code>	Returns a List of <code>IElement</code> instances that are the children of <code>aIElement</code> . This method offers various parameters to control the children returned and the sort order.
<code>getVocabularyElementParent(aIElement)</code>	Returns the parent of <code>aIElement</code> .
<code>getVocabularyElementIcon(aIElement)</code>	Returns an integer number, declared as a constant in <code>IVocabularyModelAPI</code> , that specifies the icon to be displayed (such as <code>IVocabularyModelAPI.ICON_VOCABULARY_ENTITY</code>).
<code>hasErrors(aIElement, abRecursive)</code> <code>hasWarnings(aIElement, abRecursive)</code>	Returns a boolean value that indicates whether the icon should be decorated with an error or warning symbol.
<code>getVocabularyElementText(aIElement, abEditMode)</code>	Returns the localized text to be displayed adjacent to the icon in the tree, taking into account the API Locale.

Rulesheet Scope tree (IRulesheettableModelAPI)

Method	Description
<code>getScopeNodeChildren (aScopeNode)</code>	Returns a List of <code>ScopeNode</code> instances that are the children of <code>aScopeNode</code> . The list is ready for presentation and is sorted taking into account the API Locale.
<code>getScopeNodeParent (aScopeNode)</code>	Returns the parent of <code>aScopeNode</code> .
<code>getScopeNodeIcon (aScopeNode)</code>	Returns an integer number, declared as a constant in <code>IRulesheettableModelAPI</code> , that specifies the icon to be displayed (such as <code>IRulesheettableModelAPI.ICON_SCOPE_ENTITY</code>).
<code>hasErrors (aScopeNode, abRecursive)</code> <code>hasWarnings (aScopeNode, abRecursive)</code>	Returns a boolean value that indicates whether the icon should be decorated with an error or warning symbol.
<code>getScopeNodeText (aScopeNode)</code>	Returns the localized text to be displayed adjacent to the icon in the tree, taking into account the API Locale.

Testsheet Input/Output/Expected trees (ITestsheettreeSetModelAPI)

Method	Description
<code>getTestNodeChildren (aTestNode)</code>	Returns a List of <code>TestNode</code> instances that are the children of <code>aTestNode</code> . The list is ready for presentation and is sorted taking into account the API Locale.
<code>getTestNodeParent (aTestNode)</code>	Returns the parent of <code>aTestNode</code> .
<code>getTestNodeIcon (aTestNode)</code>	Returns an integer number, declared as a constant in <code>ITestsheettreeSetModelAPI</code> , that specifies the icon to be displayed (such as <code>ITestsheettreeSetModelAPI.ICON_TEST_ENTITY</code>).

Method	Description
<code>hasErrors(aTestNode, abRecursive)</code> <code>hasWarnings(aTestNode, abRecursive)</code>	Returns a boolean value that indicates whether the icon should be decorated with an error or warning symbol.
<code>getTestNodeText(aTestNode)</code>	Returns the localized text to be display adjacent to the icon in the tree, taking into account the API Locale.

Example Usage (Scope tree)

Using Presentation API methods, your content provider methods can often be reduced to a single instruction. Here are some code excerpts from Corticon Studio Rulesheet Scope tree Content and Label Providers:

```

{
    return
    iIRulesheetTableModelAPI.getScopeNodeChildren((ScopeNode) aObject).toArray();
}

public Object[] getElements(Object aObject)
{
    return getChildren(aObject);
}

public boolean hasChildren(Object aObject)
{
    return getChildren(aObject).length > 0;
}

public Object getParent(Object aObject)
{
    return iIRulesheetTableModelAPI.getScopeNodeParent((ScopeNode) aObject);
}

public String getText(Object aObject)
{
    return iIRulesheetTableModelAPI.getScopeNodeText((ScopeNode) aObject);
}

public Image getImage(Object aObject)
{
    return
    iImageArray[iIRulesheetTableModelAPI.getScopeNodeIcon((ScopeNode) aObject)];
}

```

Selection

Presentation APIs anticipate that UI widgets will allow users to select a rectangular area of table cells, a set of tree nodes or a set of diagram shapes. Some Presentation API functions (such as cut, copy, delete) are designed to operate upon that which is selected.

UI selection machinery is platform-specific, so Foundation declares a set of platform-neutral selection classes you can use to communicate your UI selection state to the API in a simplified, universal form.

Class Name	Role
<code>com.corticon.eclipse.core.util.Selection</code>	Ancestor of all selection classes.
<code>com.corticon.eclipse.core.util.MatrixSelection</code>	Selection for table controls that identifies a Matrix ID, anchor and lead coordinates that delineates a rectangular range of cells.
<code>com.corticon.eclipse.core.util.treeviewSelection</code>	Selection for tree controls that identifies a treeview ID and a list of object instances.
<code>com.corticon.eclipse.core.util.DiagramSelection</code>	Selection for diagrams that identifies a Diagram ID and a list of object instances.

Your UI creates an instance of one of the above classes (depending on the type of UI widget) then invokes Presentation API `setSelection` method.

`MatrixSelection` requires some additional explanation because it supports three selection modes:

Selection Mode	Purpose
<code>MatrixSelection.SELECTION_MODE_NORMAL</code>	Specifies that a rectangular range of cells has been selected. The selected rectangle starts at anchor coordinates and extends to lead coordinates contained in the <code>MatrixSelection</code> object.
<code>MatrixSelection.SELECTION_MODE_ROW</code>	Specifies that a range of full rows is selected. In this mode, all of the cells in each selected row are implicitly selected, so column coordinates are immaterial.
<code>MatrixSelection.SELECTION_MODE_COLUMN</code>	Specifies that a range of full columns is selected. In this mode, all of the cells in each selected column are implicitly selected, so row coordinates are immaterial.

You instantiate a `MatrixSelection` object that specifies the Matrix ID, selection mode, anchor and lead coordinates; then, you call the `setSelection` method to notify the API.

```
MatrixSelection mySelection = new
MatrixSelection(IRulesheetTableModelAPI.MATRIX_ID_IF,
    MatrixSelection.SELECTION_MODE_NORMAL, anchorCol, anchorRow, leadCol,
    leadRow);
myAPI.setSelection(mySelection);
```

If you are implementing a UI editor, you'll need a strategy to keep the Presentation API informed of your UI selection state. Most UI frameworks provide a means to subscribe as a selection listener, so one effective strategy is to create a UI selection listener that notifies the API whenever the selection state changes.

Here's a code snippet from the Corticon Studio Ruletest Editor, which uses a selection listener to keep the API informed of changes in the Messages View table selection state.

```
private class TableViewSelectionListener implements SelectionListener
{
    public void widgetSelected(SelectionEvent e)
    {
        int[] liSelectedIndices =
iDynamicTableView.getTable().getSelectionIndices();
        if (liSelectedIndices.length == 0)
            return;

        int liLowestRowNumber = getLowestSelectedIndex(liSelectedIndices);
        int liHighestRowNumber = getHighestSelectedIndex(liSelectedIndices);

        MatrixSelection lMatrixSelection = new MatrixSelection
        (ITestsheetTreeSetModelAPI.MATRIX_ID_MESSAGES_OUTPUT,
        MatrixSelection.SELECTION_MODE_ROW, 0, liLowestRowNumber,
        0, liHighestRowNumber);

        iITestsheetTreeSetModelAPI.setSelection(lMatrixSelection);
    }
}
```

Drop-Down Lists

Presentation APIs have methods that return lists of strings you can use to populate your drop-down and combo-box controls. The method names take the form `getList<foo>` where `<foo>` is a self-documenting name. Here are some example methods declared in the Rulesheet Presentation API:

Method	Description
<code>getListIfCell(aColumn, aRow)</code>	Returns list of values that UI should use to populate the drop down list for the IF cell at the specified row and column.
<code>getListThenCell(aColumn, aRow)</code>	Returns list of values that UI should use to populate the drop down list for the THEN cell at the specified row and column.

Method	Description
<code>getListThenCell(aColumn)</code>	Returns list of values that UI should use to populate the drop down list for the Overrides cell at the specified column.
<code>getListruleStatementPostSeverity()</code>	Returns list of values that UI should use to populate the drop down list for the Rule Statement Post cell.
<code>getListruleStatementAlias()</code>	Returns list of values that UI should use to populate the drop down list for the Rule Statement Alias cell.

Note that these methods return localized values, and the system will automatically supply a blank (or unspecified) value to the list if it is applicable.

Your UI can use these localized lists to populate your drop-down controls. When the user chooses a value, your UI can simply call `setCellValue`, supplying the localized string value to the API. The API will automatically convert the localized value into base form.

Menu and Toolbar Actions

All Corticon Studio menu and toolbar actions delegate to Foundation APIs. For each Corticon Studio function, there will typically be two Foundation API methods:

- 1. An `isEnabled` method that returns true or false depending on whether the function should be enabled based on the UI selection state and other factors
- 2. A functional method that contains the logic to be performed

By convention, menu or toolbar action names tend to match the API method names, although there are some exceptions. Here are some examples in `IRulesheettableModelAPI`:

Menu or Toolbar Action Name	<code>isEnabled</code> Method Name	Functional Method Name
Insert Rows	<code>isEnabledInsertrow</code>	<code>insertrows</code>
Delete Rows	<code>isEnabledDeleteRow</code>	<code>deleteRows</code>
Renumber Rules	<code>isEnabledRenumberRules</code>	<code>renumberRules</code>

These conventions make it easier to code your own menu and toolbar actions. In most cases your actions will require only two methods, and both methods will delegate to Foundation.

```
package com.corticon.eclipse.studio.vocabulary.tree.ui.actions;
import com.corticon.eclipse.studio.vocabulary.tree.ui.Messages;
```

```
public class ActionClearJavaClassMetadata extends VocabularyTreeAction
{
    public ActionClearJavaClassMetadata()
    {
        super(Messages.getString("ActionClearJavaClassMetadata.MenuText"));
    }

    public boolean canRun()
    {
        return iIVocabularyTreeModelAPI.isEnabledClearJavaClassMetadata();
    }

    public void run()
    {
        iIVocabularyTreeModelAPI.clearJavaClassMetadata();
    }
}
```

Note that all Corticon Studio actions invoke `canRun` whenever the user changes the UI selection state; the action will enable or disable itself based on the return value. The system will invoke the `run` method when the user selects the action's menu item or presses the action's toolbar button.

Many actions can control their enable/disable states based solely on changes in the UI selection state; however, in some cases it may be helpful to register your actions as Batched Listeners.

Validation

As your client invokes state-change methods, Foundation will automatically validate the asset. If the API detects a problem, such as a syntax error in a Rulesheet expression, it will create one or more validation messages which are also stored in the asset. Corticon Studio displays these validation messages in the Problems View.

Validation messages may be either warnings or errors. If any error-level validation messages are present in the asset, the system will treat the asset as invalid and will prevent it from being deployed until the error is corrected.

Foundation APIs offer methods to allow you to check the validity of in-memory assets. For example, you can call the `isValid` method to determine whether your asset is completely free of any error-level validation messages:

```
boolean assetValid = myAPI.isValid();
```

Alternatively, you can check for errors or warnings:

```
boolean assetErrors = myAPI.hasErrors();
boolean assetWarnings = myAPI.hasWarnings();
```

Your application can call `getValidationMessages` to get all of the messages:

```
List<InternationalValidationMessage> myList = myAPI.getValidationMessages();
for (InternationalValidationMessage myMessage : myList)
    System.out.println(myMessage.getTextKey());
```

Note that these messages are international; so they consist of a resource bundle Text Key plus variables.

For some applications, such as large-scale batch updates, you can improve performance by disabling real-time validation via `setSuppressValidation`:

```
myAPI.setSuppressValidation(true);
```

When validation is suppressed in this manner, you can explicitly validate the entire asset by calling `validate`:

```
myAPI.validate();
```

Transaction Management

Foundation API state-change methods are either transactional or non-transactional.

When you invoke a transactional state-change method, the API will automatically begin a transaction, make a series of EMF object state changes, and finally commit those changes. If the transaction is successful, the API will record the change history in the command stack as necessary to support undo. If an error occurs during a transaction, the API will roll back any interim changes, effectively leaving the in-memory asset unchanged.

When you invoke a non-transactional state-change method, the API will immediately update the asset without beginning a transaction. Non-transactional methods do not affect the command stack and cannot be undone. Foundation uses a non-transactional approach for setting transient attributes such as the API selection state that should not be subject to undo/redo.

Transaction management is normally automatic; however, you can control it directly by calling Foundation API methods `beginTransaction` and `endTransaction`.

```
IRulesheetTableModelAPI myAPI = RulesheetTableModelAPIFactory.getInstance();
Resource myResource =
myAPI.loadResource(URI.createFileURI("C:/Fixtures/MyRulesheet.ers"));
myAPI.beginTransaction();
myAPI.setCellValue(IRulesheetTableModelAPI.MATRIX_ID_CONDITIONS,
    IRulesheetTableModelAPI.COLUMN_INDEX_CONDITION_EXPRESSION, 0, "Person.name
    = 'Mike'");
myAPI.setCellValue(IRulesheetTableModelAPI.MATRIX_ID_CONDITIONS,
    IRulesheetTableModelAPI.COLUMN_INDEX_CONDITION_EXPRESSION, 1, "Person.name
    = 'Dave'");
myAPI.setCellValue(IRulesheetTableModelAPI.MATRIX_ID_CONDITIONS,
    IRulesheetTableModelAPI.COLUMN_INDEX_CONDITION_EXPRESSION, 2, "Person.name
    = 'Heather'");
myAPI.endTransaction();
myAPI.saveResource(myResource);
myAPI.dispose();
```

For large-scale batch operations, you can improve performance by disabling transaction management via `setDisableTransactions`:

```
myAPI.getModelAPIState().setDisableTransactions(true);
```

For example, the Corticon Studio Version 4 Import Wizard disables transaction management to improve import performance.

Undo/Redo

Foundation APIs support undo/redo. Each API contains a command stack that captures transactional change history. If your application calls the `undo` method, the API will automatically reverse the effects of the prior transaction:

```
myAPI.undo();
```

Then, if your application calls the `redo` method, the API will re-execute the transaction:

```
myAPI.redo();
```

This returns the API to its original state.

State-Change Notifications

To facilitate the MVC design pattern, Foundation APIs allow your applications to subscribe (that is, listen) for model state-change notifications in several ways. You can register your UI controls as listeners so that they can refresh themselves to match the state of in-memory assets as they are updated.

Traditional Subscription

Traditional Subscription notifies your application of every object state change the moment it occurs.

```
myAPI.addModelListener(new MyTraditionalListener());
.
.
.
private static class MyTraditionalListener implements INotifyChangeListener
{
    public void notifyChanged(Notification aNotification)
    {
        System.out.println("Asset is changing = " + aNotification);
    }
}
```

Batched Subscription

Batched Subscription delivers a single batch of notifications to your application at the end of every completed transaction.

```
myAPI.addBatchedModelListener(new MyBatchedListener());
.
.
.
private static class MyBatchedListener implements IBatchedChangeListener
{
    public void notifyChanged(NotificationBatch aNotificationBatch)
    {
        List<Notification> myList = (List)aNotificationBatch;
        for (Notification myNotification : myList)
            System.out.println("Batched notification = "+myNotification);
    }
}
```

Resource Subscription

Resource Subscription notifies your application of Resource-level events such as asset loading, unloading and saving.

```
myAPI.addResourceListener(new MyResourceListener());  
.  
.  
.  
private static class MyResourceListener implements IResourceListener  
{  
    public void notifyChanged(int aEvent, Resource aResource)  
    {  
        switch (aEvent)  
        {  
            case IResourceListener.LOAD:  
                System.out.println("Resource "+aResource.getURI()+" loaded.");  
                break;  
            case IResourceListener.SAVE:  
                System.out.println("Resource "+aResource.getURI()+" saved.");  
                break;  
        }  
    }  
}
```

Localization

Presentation APIs support localization. API-based localization handles the translation of static text (such as asset validation messages) and asset-based localization (Corticon Studio lets users localize their assets, particularly the Vocabulary).

You set the locale of an API by calling the `setLocale` method supplying a Java Locale instance. For example, to set the API Locale to Japan:

```
myAPI.setLocale(Locale.JAPAN);
```

The API locale controls:

- Static text messages returned by the API
- International validation messages
- Decimal formats (that is, decimal point or comma)
- Set separator (that is, comma or semicolon)
- Asset-based localizations

Presentation API methods such as `getCellValue` will automatically localize strings based on the Locale you specify. Conversely, when you call state-change methods such as `setCellValue`, the Presentation API will reverse the translation process, converting localized strings into locale-neutral base form, which is stored in the asset.

Delegates

Foundation APIs allow you to register optional delegates that can allow APIs to call back into your UI framework. For example, you can code a Message Box Delegate, which is technically an adapter for your UI message box widget; then, Foundation can use your delegate to display message boxes.

A delegate must implement a particular Java interface. Foundation APIs remain neutral by referring only to the delegate interfaces.

Delegate Interface	Description
<code>IPersistenceServicesDelegate</code>	Performs platform-specific functions needed for resource creation, loading, unloading, saving and deletion.
<code>IPreferencesDelegate</code>	Allows Foundation APIs to access platform-specific user preference specifications.
<code>IMessageBoxDelegate</code>	Allows Presentation APIs to display a message box.
<code>IScrollableMessageDelegate</code>	Allows Presentation APIs to display a dialog with a potentially large message in a scrollable text box.
<code>IUserInfoDelegate</code>	Allows Presentation APIs to access platform-specific information about the UI, particularly font metrics.

Each type of delegate has a default implementation supplied with Foundation. In most cases, the default implementation doesn't do anything useful, but merely serves as a placeholder if no custom delegate is registered.

Corticon Studio comes with an Eclipse implementation for each type of delegate. The Corticon Studio UI registers the Eclipse delegates with Foundation during editor initialization, allowing Foundation APIs to call back into Eclipse services.

Delegates should be registered with Foundation APIs after they are instantiated but before the APIs are used.

```
public IModelAPI getModelAPI()
{
    IModelAPI lIModelAPI = VocabularyTreeModelAPIFactory.getInstance();
    lIModelAPI.setPreferencesDelegate(new EclipsePreferencesDelegateImpl());
    lIModelAPI.setMessageBoxDelegate(new EclipseMessageBoxDelegateImpl());
    lIModelAPI.setScrollableMessageDelegate(new
EclipseScrollableMessageDelegateImpl());
    lIModelAPI.setUserInfoDelegate(new
EclipseUserInfoDelegateImpl());
    return lIModelAPI;
}
```

Generating Reports

Presentation APIs offer methods you can use to generate reports that consist of information contained in Corticon assets.

Method `createDefaultXmlReport` returns a JDOM Document instance that contains selected information extracted from the asset.

```
Document myReport = myAPI.createDefaultXmlReport(true);  
System.out.println(CcXmlIO.fromJDomToString(myReport));
```

You can use the returned Document for a variety of applications. For example, the Corticon Studio reporting subsystem uses XSLT processor to transform the report Document into HTML, then presents the results to the user in a browser.

