# IBM Rational

# Team API Programmer's Guide (version 7.0.1 Preview)

**May, 2007**

**Legal Notices**

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patentapplications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation North Castle Drive
Armonk, NY 10504-1785
U.S.A.
For license inquiries regarding double-byte (DBCS) information,
contact the IBM Intellectual Property Department in your country or send inquiries,
in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law**.

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow
disclaimer of express or implied warranties in certain transactions, therefore,
this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Department BCFB
20 Maguire Road
Lexington, MA 02421
U.S.A.
Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs

conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

**Trademarks**
http://www.ibm.com/legal/copytrade.shtml

# Rational Team API introduction

The IBM Rational Team API is a unified Java API through which you can access Rational Team products (including ClearCase, ClearQuest, and RequisitePro for this release). The Rational Team API extends the WVCM (Workspace Versioning and Configuration Management) API, which is a standard Java API for configuration management (see http://www.jcp.org/en/jsr/detail?id=147).

IBM Rational software products provide a comprehensive set of integrated tools that facilitate software engineering best practices and span the entire software development lifecycle. Traditionally, each individual Rational product has had its own API that provides access to its product-specific repository. The Rational Team API provides one unified API for access to all Rational Team products.

With the Rational Team API you can build client applications that access Rational Team product applications, and build new integrations to these products. The client application can be an Eclipse plug-in or other Java client application. You can use the Rational Team API to build client applications that:

- Perform ClearCase checkout and checkin operations from your Java application.
- Identify the ClearCase Web views on a target server machine, and browse the hierarchy of ClearCase elements to view them.
- Store persistent references to ClearCase objects (that is, elements or versions) in a database and later retrieve those objects, or find where those objects are loaded into a ClearCase view.
- Perform common ClearQuest functions such as retrieving and updating change requests and other record types.
- Change the state of a change request record in a database and programmatically do other common functions.
- Execute ClearQuest queries and browse the ClearQuest records in the result set.
- Retrieve, update, and create new RequisitePro requirements and other requirements management artifacts, such as documents, views and packages.
- Browse the Requirement types in a RequisitePro project or repository.
- Store persistent references to objects in a ClearQuest or RequisitePro database and later retrieve those objects, or find where those objects are located.

For an introduction to the programming model for the Rational Team API, see Rational Team API Object Model.

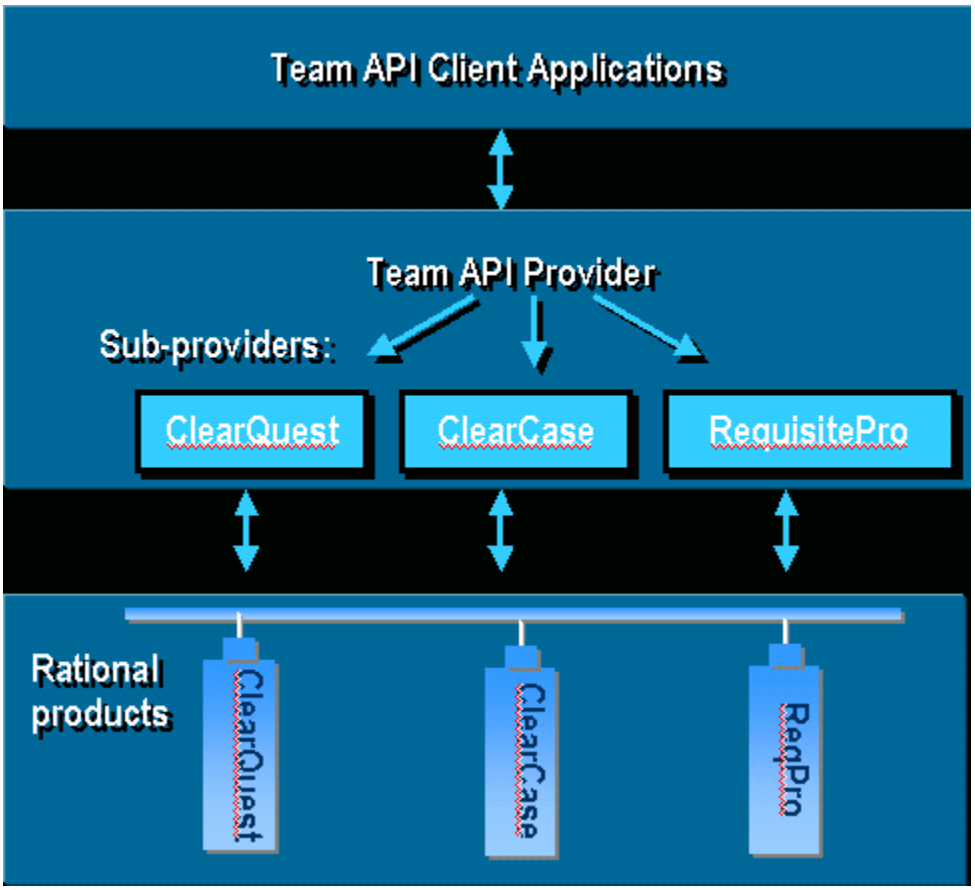
# Rational Team API provider

The Rational Team API is implemented by a Rational Team API provider. The provider is the collection of Java packages with which clients can interact with requirements, change and configuration management services. A provider receives requests from

Rational Team API clients and interacts with the repositories for the given products to process the requests.

A sub-provider is a component of the Rational Team API that provides product-specific functionality. Each sub-provider package maps a product-specific object model to the Rational Team API object model and thus makes the product-specific objects available to Rational Team API client applications.

The Team API Provider dispatches requests to product-specific sub-providers, as shown in the following architecture diagram:




As the figure illustrates:

- A Rational Team API client application makes Rational Team API calls to the Rational Team API provider.
- The Rational Team API provider dispatches the Rational Team API calls to the appropriate sub-provider.
- The Rational Team API sub-providers map the Rational Team API calls to the underlying Rational Team products.

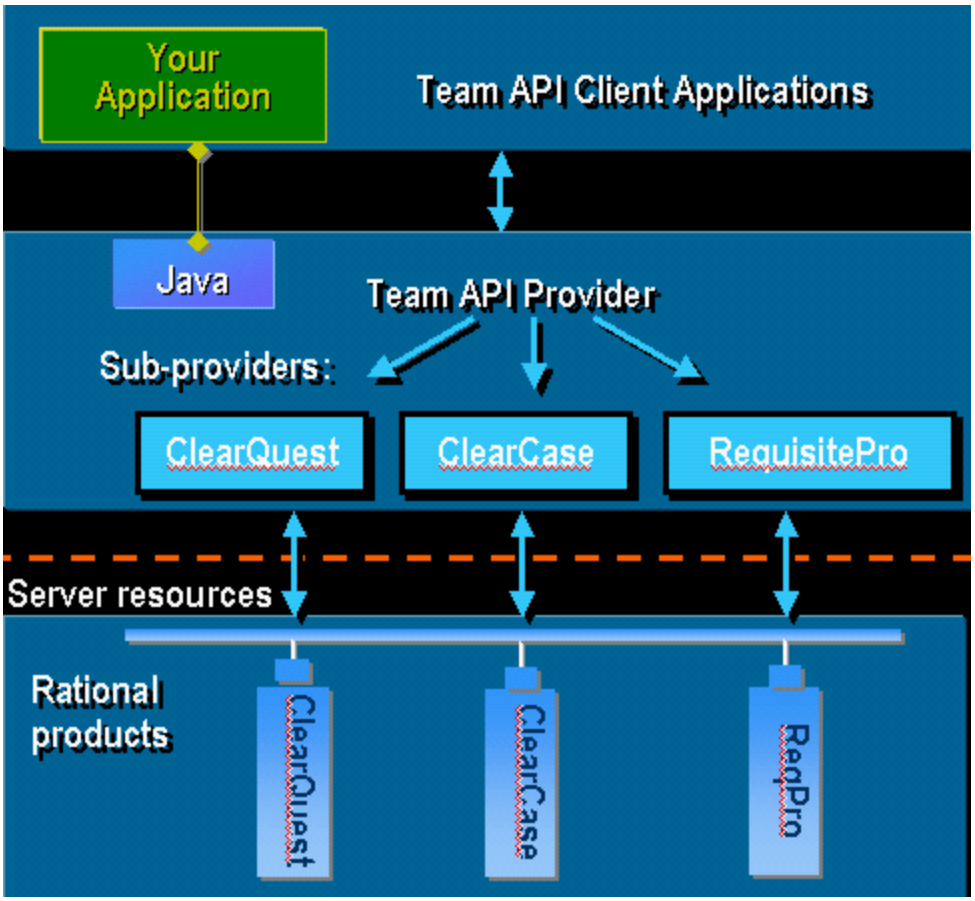
# Rational Team API clients

The Rational Team API defines a client/server processing model, where the client makes explicit requests to a server to obtain information about resources on the server and to make changes to these resources. The client uses proxy objects to marshal data to and from the server through the Rational Team API. Each proxy class defined in the Rational Team API maps to a specific type of Team product resource on the server.

By defining proxies that map to Team product resources, the Team API client/server programming model helps distinguish client-side processing and server-side processing. There is a well-defined naming convention for all methods to help identify methods that may make calls to a server. For example, to read data from a Team product, a client application must first make an explicit request for the object or data to be read from the server and into a proxy before the value can be read from the proxy. Clients must call a ***do*** method (for example, `Resource.doReadProperties`) to request that specified values be read from a product server resource. The client application specifies the properties to be read (or written) by creating the appropriate proxy type that contains the names for each property to be read or written.

You can create client applications to read, modify, create and delete data from any product that has a Rational Team API sub-provider. The Rational Team API provides the interfaces to perform product-specific operations available in many Rational software products.

The following figure shows a client application, which could be a integration between an existing application and Rational Team products, or a tool or utility that performs operations on data in Rational Team product repositories. With the Rational Team API, client applications have access to data in any of the Rational products through the Rational Team API sub-providers.




For example, if users have an application to access and work on some set of source files in ClearCase or records in a ClearQuest or RequisitePro database, the Team API could be used to create an integration between the existing application and the Rational Team product involved. If the files that users are updating are under ClearCase source control, then the integration could enable users to check out and check in their files from their application. The Rational Team API could be used to both perform source file updates and associate the update with a ClearQuest change request record.

The Rational Team API provides developers of client applications with a:

- Single data access API to Rational Team products. As more applications become Rational Team API-enabled, Rational Team API clients have access to the additional application data.
- Consistent mechanism for relating objects within and across applications.
- Tight integration with Rational Team products.

# Rational Team API sub-providers

A sub-provider is a Rational Team API extension package that is created for a supported product. The sub-providers connect to the Rational Team API provider and represent

defined product-specific resources stored in each integrated product. Each sub-provider maps an integrated product object hierarchy to the Rational Team API object model hierarchy.

While the top-level Rational Team API packages enable you to retrieve data from the product-specific repositories in a generic way, the Rational Team API domain-specific packages provide additional interfaces for performing product-specific tasks.

- ClearQuest change management capabilities supported by Rational Team API (in the com.ibm.rational.wvcm.stp.cq package) include the ability to:

    - create, modify, and delete database records (for example, create a defect)
    - apply an action to a record to change its state
    - create and execute a query
    - modify fields in a record

- ClearCase configuration and asset management capabilities supported by the Rational Team API (in the com.ibm.rational.wvcm.stp.cc package) include the ability to:

    - create, populate, and delete ClearCase Web views
    - operate on elements within Web views (such as, checkout, checkin, and hijack operations)
    - navigate various VOB-object hierarchies and request properties of those ClearCase objects

    Note: The Rational Team API supports Web views but does not currently support ClearCase dynamic or snapshot views.

- RequisitePro requirements management capabilities supported by Rational Team API (in the com.ibm.rational.wvcm.stp.rp package) include the ability to:

    - open a project and add packages to the project
    - view requirement types and their attributes
    - create, retrieve, modify requirements, requirement attributes, and traceability
    - create, retrieve, modify documents, views and queries

With the Team API the client has the ability to configure, integrate, and synchronize repositories or resources (for example, between ClearQuest and RequisitePro).

# Packaging

The Rational Team API is comprised of the following packages:

- WVCM - javax.wvcm

The JSR-147 defined interface. The Workspace Versioning and Configuration Management package (WVCM) is the subset of team functionality that has been accepted by the standards body.

The WVCM interfaces form the basis of Rational Team API and provide a well-defined object model for expressing the configuration management operations and functions.

WVCM is expressed as a set of Java interfaces with associated Javadoc comments. The interfaces define the form of the object model, and the comments describe the expected semantics of the operations.

- Rational Team package - com.ibm.rational.wvcm.stp

  The Rational software team package is an extension of the WVCM package. This package contains the interfaces of the Rational Team API and provides the common object model for Rational product resources. This package is independent of product-specific repository or implementation boundaries. It includes the common interfaces from which product-specific interfaces can be derived.

  The Rational Team API extends WVCM into the realm of non-versioned resources, specifying a rigorous editing paradigm, a common query interface, and support for schema-defined resources. General mechanisms of WVCM are enhanced with the introduction of meta-properties, an extended property request mechanism, and support for multiple types of repositories. Additionally, the Rational Team API defines a common syntax for location strings.

- Product specific packages - com.ibm.rational.wvcm.stp.*

  The following packages are product-specific extensions that provide access to specific product repositories, each containing product-specific resources and properties. These packages contain functions that provide fuller (product-specific) access to the functionality of the respective repository type and its underlying resources.

  - com.ibm.rational.wvcm.stp.cq

    Contains extensions to the STP package that provide access to ClearQuest resources.

  - com.ibm.rational.wvcm.stp.cc

    Contains extensions to the WVCM and STP packages that provide interfaces specific to ClearCase resources

  - com.ibm.rational.wvcm.stp.rp

Contains extensions to the STP package that provide access to RequisitePro resources.

The names of the interfaces and classes in each package have a prefix added to the wvcm base class name (for example, Resource, StpResource, CcResource, CqResource, and RpResource).

# Installation and setup requirements

Each individual product installation includes that product's Rational Team API sub-provider interfaces and the required Rational Team API component. For example, the ClearQuest product installation includes the ClearQuest Team API sub-provider. The sub-provider layer ensures that the Team API component infrastructure is installed. Thus, depending on the combination of products installed, systems may have all or a subset of the following JAR files:

- Rational Team API component infrastructure JAR files
- Rational Team API sub-provider JAR file for ClearCase
- Rational Team API sub-provider JAR file for ClearQuest
- Rational Team API sub-provider JAR file for RequisitePro

The Rational Team API infrastructure is designed to function whether or not all of the sub-providers are present. The provider interface allows sub-provider JAR files to be at different release levels.

The Rational Team API component has a multipart version number associated with it. The installation of one sub-provider will overwrite the infrastructure component installed by a previous sub-provider installation, but only if the infrastructure component is a newer version than the one already installed.

The Rational Team API JAR file and other required JAR files are installed by default in the following locations. *install-dir* represents the directory into which the Rational product files have been installed. By default, this directory is `/opt/rational` on UNIX systems and Linux systems and `C:\Program Files\Rational` on Windows systems.

- On Windows systems:
  - <*install-dir*>/Common/stpwvcm.jar
    The Rational Team API interface JAR file

  - <*install-dir*>/Common/stpcmmn.jar
    A common implementation JAR file

  - <*install-dir*>/clearcase/web/teamapi/stpcc.jar
    Extension for the ClearCase product
    Also required are <*install-dir*>/clearcase/web/teamapi/remote_core.jar

<*install-dir*>/clearcase/web/teamapi/commons-logging-1.0.4.jar
<*install-dir*>/clearcase/web/teamapi/commons-httpclient-3.0-rc3.jar
<*install-dir*>/clearcase/web/teamapi/commons-codec-1.3.jar

- <*install-dir*>/ClearQuest/stpcq.jar
  Extension for the ClearQuest product
  Also required is, <install-dir>/ClearQuest/cqjni.jar

- <*install-dir*>/RequisitePro/lib/stprp.jar
  Extension for the RequisitePro product
  Also required are <*install-dir*>/common/RJCB.jar
  and <*install-dir*>/RequisitePro/lib/proxies.jar

- On UNIX and Linux systems:

  - <*install-dir*>/common/stpwvcm.jar
  - <*install-dir*>/common/stpcmmn.jar
  - <*install-dir*>/clearcase/web/teamapi/stpcc.jar
  - <*install-dir*>/clearcase/web/teamapi/remote_core.jar
  - <*install-dir*>/clearcase/web/teamapi/commons-logging-1.0.4.jar
  - <*install-dir*>/clearcase/web/teamapi/commons-httpclient-3.0-rc3.jar
  - <*install-dir*>/clearcase/web/teamapi/commons-codec-1.3.jar
  - <*install-dir*>/clearquest/cqweb/lib/stpcq.jar
  - <*install-dir*>/clearquest/cqweb/lib/cqjni.jar

  Note: stprp.jar, RJCB.jar, and proxies.jar are not installed on UNIX platforms.

To use the Rational Team API JAR files in these default installed locations, you must add `stpwvcm.jar` to the Java class path or Eclipse project. If you move the JAR files to a different location, you must add the new locations of all the JAR files to your class path.

**Accessing the Rational Team API from an Eclipse plug-in**

You can create an Eclipse plug-in using the files packaged in the com.ibm.rational.stp.teamapi.zip file and other installed files (listed in the previous section) to support Rational Team API access from other plug-ins in an Eclipse runtime environment.

To add the Rational Team API plug-in to your runtime configuration you can copy the plug-in into your Eclipse instance or create a new extension install site. This creates a new directory for the Rational Team API (for example, C:\eclipse\plug-ins\com.ibm.rational.stp.teamapi). This new plug-in is a simple self-contained Eclipse plug-in consisting of the Rational Team API JAR files plus the product-specific (ClearCase, RequisitePro, and ClearQuest) JAR files. Note that the .zip file does not contain the actual product JAR files. After creating the plug-in directory, each installed JAR file must be copied from its installed location into the new plug-in directory. Each

sub-provider JAR file requires the Rational Team API JAR file. The plug-in is available for use the next time you start Eclipse.

To use the Rational Team API from your plug-in you must identify com.ibm.rational.stp.teamapi as a dependency. For introductory information on creating a plug-in, see http://help.eclipse.org/help30/topic/org.eclipse.platform.doc.isv/guide/firstplugin.htm.

Note: You must have a licensed and installed version of each Rational Team product in order to use the Rational Team API sub-provider for that product. If you install the sub-provider JAR files but do not have the corresponding product installed, calls to that Java package will fail.

**Accessing the Rational Team API from a Java client application**

For a client application to make requests to the RequisitePro and ClearQuest sub-providers, the RequisitePro and ClearQuest products must be installed on the same machine as the client program invoking Rational Team API.

The ClearCase sub-provider supports Web views and remote access through the ClearCase CCRC server. The current version of Rational Team API does not support ClearCase dynamic or snapshot views. The ClearCase-specific jar files must be copied from the CCRC server to the client machine, either to the client install location or to the plugin directory, depending on how the Rational Team API is being used.

# Summary

The Rational Team API provides unified access to any supported Rational Team product. Rational Team API client applications can gain access to and perform operations on data that is available in any integrated product through the Rational Team API sub-provider packages.

The Rational Team API provides Java programming support for creating:

- Java client applications that access one or more of the Rational Team API enabled products:
    - ClearQuest change management.
    - ClearCase configuration management
    - RequisitePro requirements management
- Integrations between Rational Team products and test management, functional testing, or use case management products.
- Client access to Rational Team product (such as ClearCase) functionality from other products (tools or applications).

This document describes the features of the Rational Team API currently available. This first offering provides access to Rational Team services through the ClearQuest, ClearCase, and RequisitePro client applications installed on the same machine.

# Rational Team API object model

This section provides an overview of the Rational Team API object model and includes code examples that illustrate how to use the API.

The Rational Team API common object model maps the objects of each supported Rational Team product to a resource hierarchy, based on the WVCM resource and property model. This common data model enables Rational Team API client applications to retrieve data from any integrated product through one set of interfaces, as WVCM resources. A file, a Versioned Object Base (VOB), a ClearQuest user database or query, and a RequisitePro project are all examples of WVCM resources.

The product-specific mappings between a product resource and the Rational Team API are defined in each product-specific package. For example, a requirement in RequisitePro is mapped to an instance of the Rational Team API Requirement interface. The Rational Team API Requirement interface is an extension (defined in com.ibm.rational.wvcm.stp.rp) of the Resource interface (defined in com.ibm.rational.wvcm.stp), which, in turn, is an extension of the WVCM Resource interface. The Resource interface provides the standard mechanisms to retrieve the properties (for example, Name and Description) and content of a resource including its relationships to other resource types (for example, from a RequsitePro Requirement to a Document).

## Resources and proxies

The Rational Team API consists of objects that are proxies for the persistent resources stored in the different repositories maintained by Rational Team products. A proxy is an object on the client that represents a resource in a product-specific repository (on a server or on the client system). A *proxy* object represents a resource during a Team API Provider session. Each type of resource is represented by a subclass of the Resource proxy class.

A client can only access a resource by first creating a proxy object. All proxy objects are obtained either by invoking a method on the Provider or invoking a method on another proxy object. The Provider builds the proxy with the requested properties and returns the proxy to the client. The client can then use the methods available through the proxy objects to access specific Rational Team product resources. Proxy objects are client-side objects returned by the Provider and not the actual server resources.

### Proxy method naming conventions

Each interface derived from the Resource interface has a well-defined set of properties than can be examined and modified using the Rational Team API (through the corresponding ***get*** and ***set*** methods). Each interface also has a well-defined set of

operations that can be invoked on the proxy to cause something to happen to the underlying resource (through the corresponding ***do*** methods).

- ***get*** methods return property values from the Provider, for a product resource.
- ***set*** methods specify values in an existing proxy but do not update the actual product resource.
- ***do*** methods are operations that may require a Provider to connect to a server and access the product repository that contains the actual resource.

Property getters and setters do not interact with the resource to get or set property values. The setters store their argument values in the proxy and the getter methods retrieve property values already stored in the proxy. Values are read from a repository using a ***do*** method such as `Resource.doReadProperties()` and are written to a repository using a do method such as `doWriteProperties()`.

The proxy methods that begin with the prefix *do* cause the Rational Team API Provider to perform operations on the resource. While *do* methods do not always indicate that an action is being performed on a server, they do indicate that an action is being performed on the persistent resource rather than just on the proxy in memory. In most cases, this causes the Provider to interact with a server. See Additional information on proxy methods for more information.

## Getting a provider

A client must first get a Provider object before it can access resources and get Resource proxies.

A Provider is a temporary object that represents a single identity within a single client process interacting with one or more repositories through the Rational Team API. The lifetime of a Provider is under the control of the client. The lifetime of some server resources is tied to the lifetime of a Provider.

The following code example creates a Provider object for a session on a server by calling `ProviderFactory.createProvider()`.

```
Provider getProvider()
throws WvcmException
{
        Provider provider = null;

        // set up the parameters for instantiating a provider.
        // the provider name is the fully-qualified class name of the provider.
        String providerName = StpProvider.CLIENT_HOSTED_PROVIDER;

        // a callback provides authentication information to the provider
        Callback callback = new MyCallback();

        // the provider factory class instantiates a provider
        provider = ProviderFactory.createProvider(providerName, callback);
        return provider;
}
```

A ProviderFactory.Callback is an interface through which user credentials are requested from the client by the Rational Team API Provider when needed to access a product repository. The following MyCallback() class supplies pre-configured user authentication information (domain, user login name and password) and returns to the provider.

```
// Callback class, needed to create a provider.
private static class MyCallback implements Callback
{
     // Get a WVCM Authentication object.
     // This implementation of the authentication
     // callback returns the specified username and password.

     // The Provider calls getAuthentication to authenticate the current user.
     public Authentication getAuthentication(final String realm, int retryCount)
          {
          if (retryCount>0)
              throw UnsupportedOperationException("Bad credentials");
          return new Authentication()
               {
               public String loginName()  { return "<the_domain>\\<the_username>"; }
               public String password() { return "<the_password>"; }
               };
          }
}
```

Each Provider instance is given one Provider.Callback object that is to be used to obtain credentials for any repository accessed by the client through that Provider instance.

In this example, the `realm` and `retryCount` arguments are not used. However, client applications should limit retries to a small number of attempts, since a provider will repeatedly try to get authentication after a failure unless the getAuthentication method throws an exception. The `realm` argument is a string identifying the context for which authentication is being requested (for example, a server URL or repository name). The format for the string varies from sub-provider to sub-provider and is intended for display to the user as a mnemonic.

Note: The domain may be part of a username. A ClearCase login requires a domain as part of the username.

In a client application, the authentication callback could open a login dialog to collect the login-name and password from the user. The `realm` argument could be presented to the user in the dialog, to display what product repository the user is logging into (for example, a ClearCase server-URL, or an indication of a given ClearQuest database or RequisitePro repository). This can be helpful if users have different usernames and passwords for different product repositories.

Note: The Callback will be called for each different realm that the client makes requests to while using the Provider. See the Javadoc for the stp.Provider class for more details about the requirements on the Callback passed to a Team API Provider.

Once the Provider is instantiated, the client application can make requests to the Provider for Resource proxies.

## Getting resource proxies

The Provider class builds proxies in response to client requests. The client can then invoke methods on the proxy to work with the represented resource. Clients can get a proxy for a resource at a specific location by requesting that the Provider build and return a proxy for that location. For example, each of the following examples creates a proxy for the location identified:

```
Resource my_resource = provider.resource(the_location);
CcActivity my_activity = provider.activity(location);
```

See Location objects for more information.

## Resources

A resource is a named collection of properties that exists in a repository. Some resources, such as files, have content as well as properties. Some resources can exist only on a server. Some exist solely in a client file area. A resource cannot exist in two different locations, but two resources may be so tightly linked that they give that impression. For example, a file in a file area and the corresponding file on the server are two different resources. They are related, but each has its own unique location, content, and properties.

A proxy object may be used to create, modify, and ultimately destroy resources. After a Resource is created, and until it is destroyed, it persists in its repository between invocations of the Provider that modify it.

In the Rational Team API, the Resource class is the base class for all Rational Team API Resource types. Examples of Resources are:

- Files in a file system
- ClearCase objects in a versioned object base (VOB)
- Defects in a ClearQuest database
- Requirements in a RequisitePro project

The Resource proxy interfaces are all part of a common class hierarchy. The root of the hierarchy is javax.wvcm.Resource. The name of each class and interface in each Rational Team API package is unique, and includes a prefix that identifies the package that contains it. For example, some of the classes that inherit from the WVCM Resource interface include:

- StpResource is derived from Resource.
- CqResource, CcResource, and RpResource are derived from StpResource.
- The `stp` package includes StpAction, StpFolder, StpProject, StpPropertyDefinition, StpQuery, and StpRepository interfaces.
- The `cc` package includes CcActivity, CcAttributeType, CcBaseline, CcBranchType, CcComponent, CcControllableFolder, CcControllableResource, CcElement, CcElementType, CcFolder, CcFolderVersion, CcProject, CcProjectFolder, CcVersion, CcView, CcVob, and CcVobResource interfaces.

- The `cq` package includes CqAction, CqAttachment, CqAttachmentFolder, CqDbSet, CqFieldDefinition, CqForm, CqGroup, CqHook, CqProjectMember, CqQuery, CqQueryFolder, CqQueryFolderItem, CqRecord, CqRecordType, CqReport, and CqUserDb interfaces.
- The `rp` package includes RpAttributeDefinition, RpDiscussion, RpDocument, RpDocumentType, RpFolder, RpGroup, RpProject, RpQuery, RpRelationship, RpRequirement, RpRequirementType, RpRevision, RpUser, and RpView interfaces.

### Controllable Resources

Resource types can be versioned or non-versioned objects.

- Versioned Objects are represented in the Rational Team API as ControllableResource proxies. A file element in a VOB and a workspace are examples of controllable resources.
- Non-Versioned Objects are resources. Non-versioned examples include a Query, Document, Record, Attachment, and a View. Non-versioned resource types are represented as ActionableResources.

See Additional information on Resources for more detail.

# Location objects

Each resource has a location that uniquely identifies the resource in its repository. The Rational Team API Location object represents the location of a resource and such an object is required to construct a proxy for the resource. A Location object is constructed by the Provider from a string representation of the location. The syntax for specifying resource locations in this string representation is defined by the Rational Team API in the Javadoc for StpLocation. The string argument to the WVCM-defined Provider.location(...) operation must conform to the syntax specified in StpLocation.

For example, the location string for the location of a client-side controllable resource (a file on a client machine) is the file pathname. This format is used in the following code fragment to checkout a file named `sample_file.txt`.

```
//  Given a CcProvider object, m_provider…which must first be instantiated,
//  create a Location object from a unique file pathname.
//  Use "C:\\sample_view\\sample_dir\\sample_file.txt" on Windows,
//  or "/sample_view/sample_dir/sample_file.txt" on UNIX.


StpLocation fileLoc =
m_provider.stpLocation("C:\\sample_view\\sample_dir\\sample_file.txt");


//  Create the ControllableResource proxy for the client - a versioned file
//  is a ControllableResource
CcControllableResource my_ctresource = m_provider.ccControllableResource(fileLoc);

//  Use the proxy to work with the controllable resource.
//  For example, check out the file:
my_ctresource.doCheckout();
```

As this example illustrates, the `provider.stpLocation()` method returns a Location object corresponding to a given location specification. The Location object (fileLoc) is subsequently passed to `Provider.ccControllableResource()` to construct a proxy for the resource at that location.

- The following example specifies the location of a server-side resource (for example, an activity in a ClearCase VOB):

```
StpLocation activityLoc = provider.stpLocation("cc.repo/activity:my_fix_a_bug@
vobs/projects");
CCActivity act = provider.ccActivity(activityLoc);
```

- The following example specifies the location of a server-side resource (for example, a record in a ClearQuest database):

```
Location loc =
provider.stpLocation("cq.repo/record:Defect/SAMPL00000234@2003.06.00/SAMPL");
CqRecord record = ((CqProvider)provider.cqRecord(loc);
```

The Location interface also provides methods for parsing and composing strings containing location specifications.

A Location object is available from each proxy, which corresponds to the location of the object referenced by that proxy. Location is extended to provide the Provider from which a Location object originated. For more information about Location objects and location specification syntax, see Additional information on Location objects.

# Properties and meta-properties

Resources have properties. Each property has a name, a type, and a value, and may have other meta-properties associated with it (such as access rights or ownership). The value of a property is of a specific type, such as integer, string, date, time, or reference-to-resource. The property type depends on the property name and the resource class. The name of a property is represented in the Rational Team API by a PropertyNameList.PropertyName object. Some properties are defined by WVCM, others are defined by this API as extensions to WVCM, and some may also be defined by the server, and the client application.

In the Rational Team API, meta-properties are identified by a MetaPropertyName object. The MetaPropertyName may be used to access the corresponding meta-property once it has been read from the server. The MetaPropertyNames are defined in the StpProperty class and its subclasses.

The PROPERTY_NAME and VALUE meta-properties of a property are distinguished meta-properties. The PROPERTY_NAME value is used to request and access the property and any of its meta-properties. The VALUE is the meta-property requested if only the property-name is used in the request.

A set of property names is defined for each type of resource defined by the API. These property names are used to request properties from the server and to access the properties once they have been obtained from the server.

All of the PropertyName fields defined in the Team API are named by an uppercase identifier in which words are separated by underscores (for example, CONTENT_LENGTH).

Examples of property names are Resource.COMMENT, Resource.DISPLAY_NAME, Resource.CREATION_DATE, and Resource.CONTENT_LENGTH. Property names are defined in the Resource class and its subclasses. Properties defined in a class are appropriate for the class and all of its subclasses. For example, the StpQuery class has StpQuery.DISPLAY_FIELDS, StpQuery.DYNAMIC_FILTERS, StpQuery.USER_FRIENDLY_LOCATION, and StpQuery.STABLE_LOCATION property names, this last property having been inherited from the Resource class. Each Resource proxy subclass defines PropertyName fields that name and identify the properties associated with resources of the type represented by the proxy.

For more information on Properties, see Additional information on Properties and Meta-Properties.

## Additional resource properties

Additional properties values not defined by the Rational Team API may be available for RequisitePro and ClearQuest resources using the following Rational Team API interfaces.

- StpDefinedResource

  is an StpActionableResource (as a CqRecord or an RpRequirement) for which the Rational Team product server defines a set of properties not statically defined by the Rational Team API. The server-defined properties can be obtained from the StpResourceDefinition associated with the resource.

- StpResourceDefinition

  is a form of StpActionableType (as a CqRecordType or RpRequirementType) resource that contains a specification for each schema-defined property of a defined-resource.

- StpPropertyDefinition

  is a proxy for a property definition (as an RpAttributeDefinition or a CqFieldDefinition), which provides static information about server-defined properties.

### RpAttributeDefinition and requirement type attributes

The properties of a RequisitePro requirement that are specified by the requirement type are called *attributes*. Each attribute is specified by an attribute definition resource that is tied to the requirement's requirement type. The attribute definition specifies the static traits of an attribute, such as its name, its value type, and perhaps a list of its legal values.

The Rational Team API RpAttributeDefinition object is a proxy class for an attribute definition resource, which defines the set of user defined properties which are applied to requirements of a certain requirement type.

### CqFieldDefinition and record type fields

The properties of a ClearQuest CqRecord that are specified by the record type are called *fields*. Each field is specified by a field definition resource that is tied to the record's record type. The field definition specifies the static traits of a field, such as its name, its value type, and perhaps a list of its legal values. Some traits of a field depend on the state of the record or the action being performed on the record at a given time. These traits are specified as meta-properties of the field property.

The Rational Team API CqFieldDefinition object is a proxy class for a field definition resource.

## Setting up a property name list

A Rational Team API client application must first get a proxy to a resource before it can read or update properties. And before a client can access properties from a proxy it needs to read those properties from the resource into a proxy. The client application must specify the wanted properties by name in a property name list when reading them from the resource into a proxy. For example:

```
// Create a PropertyNameList - specify the names of
// properties wanted from the resource.
PropertyNameList myPropList1 =
        new PropertyNameList(
            new PropertyName[] {
                Resource.COMMENT,
                Resource.CONTENT_LENGTH,
                Resource.CONTENT_TYPE,
                Resource.CREATOR_DISPLAY_NAME,
                Resource.DISPLAY_NAME});
```

For a given resource subclass, you can specify properties defined in the class itself or any of its superclasses. For example, for the Query class you can specify properties that are specific to Query and properties defined in the Resource superclass:

```
PropertyNameList myPropListr =
        new PropertyNameList(
            new PropertyName[] {
                Resource.COMMENT,
```

```
                Resource.DISPLAY_NAME,
                // include properties specific to the Query
                CqQuery.DISPLAY_FIELDS,
                CqQuery.DYNAMIC_FILTERS});
```

After you have specified the PropertyNames in a PropertyNameList, you can then pass this list to the `doReadProperties()` method of the Resource proxy to read the specified properties.

## Reading properties

To read properties from a resource, the client must create a list of property names that identifies the properties to be read and pass the list as an argument to the Resource proxy `doReadProperties()` method. This method passes the resource location specified by the proxy and the list of desired properties to the repository. For example:

```
PropertyName[] wantedPropNames = { Resource.DISPLAY_NAME,
Resource.COMMENT};

// create a PropertyNameList proxy for retrieving the specified
// property names in wantedPropNames
PropertyNameList wantedProps = new PropertyNameList(wantedPropNames);

// you must call doReadProperties to retrieve the properties
// through the proxy
my_resource = (Resource) my_resource.doReadProperties(wantedProps);

//  work with the properties
//  for example, get and set values for these properties
// ...
```

The response from the server is returned through the Rational Team API back to the client application as a new proxy that contains the requested properties. The property values obtained by the doReadProperties method are stored in the proxy it returns.

Once a proxy is populated with properties, the value of these properties can be extracted from the proxy using either a PropertyName object or by using the access method defined by the proxy class specifically for the property. The Rational Team API provides get<*PropertyName*> methods for each statically-defined property, to get the value of that property from a proxy. See Naming convention for Get and Set property value methods for more information.

## Writing properties

You can set a new value for a property using the property-specific setter method to set the property value in the proxy object. (Properties that may not be set using the Rational Team API do not have setter methods.) When specifying new values in a set method, the values are stored in the proxy. The values are not written to the actual resource in its repository until the client application calls a **do** method such as the `doWriteProperties` method on the proxy object.

You must call the doWriteProperties method to update the underlying resource in the product repository. The doWriteProperties method writes the updated properties in the proxy to the product resource all at once, as one transaction. Failures do not occur when property values are set in the proxy, but may occur when doWriteProperties() is called. At that time, an exception may be thrown. Note that all **do** methods will write any new property values set in the proxy to the resource before apply the operation indicated by the method to the resource.

The following example appends some text to the comment property of a resource:

```
PropertyNameList wantComment =
   new PropertyNameList(new PropertyName[] { Resource.COMMENT });
Location location = myProvider.location(…);
Resource myResource = myProvider.resource(location);
myResource = (Resource) myResource.doReadProperties(wantComment);
String comment = myResource.getComment();
myResource.setComment(comment + "addition to comment");
myResource.doWriteProperties();
```

It is not necessary to call doReadProperties() before calling doWriteProperties() if you know what property value to write without first reading the current value. In the following example, the Owner field of Defect SAMPL00000005 in the ClearQuest sample database is set to the user = admin.

The location string syntax for a ClearQuest record location is,

**cq.record:**<record-type>/<record-id>@<db-set-name>/<database-name>

so, in the example the record location string is,

"cq.record:Defect/SAMPL00000005@7.0.0/SAMPL"

where,

- *<record-type>* is Defect
- *<record-id>* is SAMPL00000005
- *<db-set-name>* is 7.0.0
- *<database-name>* is SAMPL

```
CqRecord myRecord =
myCqProvider.cqRecord((Location)myProvider.stpLocation("cq.record:Defec
t/SAMPL00000005@7.0.0/SAMPL"));
PropertyName OWNER = myRecord.fieldPropertyName("Owner");

myRecord.setProperty(OWNER, "cq.record:users/admin@7.0.0/SAMPL");
myRecord.doWriteProperties();
```

See Additional information on Location objects for more information.

## Nested properties

The value of many properties is a reference to another resource.

If the value of a property is a reference to a resource, the PropertyNameList may contain a NestedPropertyName object in place of the PropertyName object for the desired property. The NestedPropertyName object can be used to retrieve properties of the resource referenced by the property of the targeted resource.

In addition to specifying the name of the property, a NestedPropertyName also includes its own PropertyNameList. This nested PropertyNameList specifies the properties of the resource referenced by the property of the original resource whose values are to be obtained from the referenced resource.

For example, the following code fragment creates a property name list that identifies the CREATOR_DISPLAY_NAME, CHECKED_IN, and LAST_MODIFIED properties, as well as the VERSION_NAME and CREATION_DATE of the value of the CHECKED_IN property:

```
PropertyNameList my_prop_name_list = new PropertyNameList(new
PropertyName[] {
     ControllableResource.CREATOR_DISPLAY_NAME,
     ControllableResource.CHECKED_IN.nest(
          new PropertyName[] {
                  Version.VERSION_NAME,
                  Version.CREATION_DATE}).
     ControllableResource.LAST_MODIFIED});
```

After specifying the nested properties, you can then call the doReadProperties method and then access the nested properties. For example,

```
resource =
   (ControllableResource) resource.doReadProperties(my_prop_name_list);
String versionName = resource.getCheckIn().getVersionName();
// work with the properties ...
```

In a NestedPropertyName, the PropertyNameList designating the properties wanted from the server can be augmented with MetaPropertyName elements, which allow the client to request specific meta-properties of a property (instead of, or in addition to, its VALUE meta-property).

Additionally, NestedMetaPropertyName elements can be included in a PropertyNameList. A NestedMetaPropertyName object is used to request a property of a resource referenced by a meta-property, or a meta-property of a property referenced by a meta-property. For example,

```
Record r = (Record) p.cqRecord(p.location("…"));
PropertyName OWNER = r.fieldPropertyName("Owner");
PropertyName NAME = r.fieldPropertyName("Name");
PropertyNameList pnl =
   new PropertyNameList(new PropertyName[] {
       OWNER.nest(new PropertyName[] {
           StpProperty.TYPE,
           CqFieldValue.REQUIREDNESS,
```

```
            StpProperty.VALUE.nest(new PropertyName[]{
                  NAME
            })})});

CqRecord rec = (CqRecord)r.doReadProperties(pnl);
CqFieldValue v = (CqFieldValue)rec.getMetaProperties(OWNER);
String name = (String)v.getValue().getProperty(NAME);


CqRecord r = p.cqRecord(p.stpLocation("…"));
PropertyNameList pnl =
   new PropertyNameList(new PropertyName[] {
     CqRecord.FIELDS.nest(new PropertyName[] {
               StpProperty.VALUE.nest(new PropertyName[]{
                  StpProperty.NAME,
                  StpProperty.TYPE,
                  StpProperty.VALUE
            })})});
Iterator fields =
((CqRecord)r).doReadProperties(pnl).getFields().iterator();
while(fields.hasNext()){
    CqFieldValue field = (CqFieldValue)fields.next();
    System.out.println("field " + p.getName()
                       ": " + p.getType()
                       " = " + p.getValue());
```

The PropertyNameList nested within a NestedPropertyName may, itself, contain additional NestedPropertyName objects. So, in one interaction with the server, it is possible to retrieve an arbitrary number of related resources and their properties.

## Naming convention for get and set property value methods

For each resource type the Rational Team API provides specific methods to get and possibly set each property value defined in the Resource subclass. For example, for the ClearQuest CqRecord class, in addition to get and set methods inherited by Resource in both the WVCM and the Rational Team API packages there are also get and set methods for properties that are specific to a ClearQuest record, such as getHasDuplicates, getFieldsUpdatedThisSetValue, and getAllFieldValues.

The Rational Team API uses the following naming convention for PropertyName fields and the corresponding getter and setter methods for the property value. For a given PropertyName *XXXXXX_YYY_ZZZZZ* (for example, DISPLAY_NAME):

- The getter method for the property is getXxxxxYyyZzzzz (for example, getDisplayName or getComment).
- The setter method for the property is setXxxxxYyyZzzzz (for example, setDisplayName or setComment).

For example,

```
String  DisplayName =  my_resource.getDisplayName();
String  DisplayFields =  my_resource.getComment();
```

If a setter method is not defined, then the property cannot be set directly with the Rational Team API.

You can also get and set property values with the getProperty() and setProperty() methods.

- getXxxxxYyyZzzzz() is equivalent to getProperty(XXXXXX_YYY_ZZZZZ). For example, getDisplayName is equivalent to getProperty(DISPLAY_NAME)
- setXxxxxYyyZzzzz(*val*) is equivalent to setProperty(XXXXXX_YYY_ZZZZZ, *val*). For example, setDisplayName(*val*) is equivalent to setProperty(DISPLAY_NAME, *val*)

For more information, see Additional information on properties and meta-properties.

### Request lists

An StpRequestList is a structure for requesting and retrieving specific properties from multiple resources in one server interaction. StpRequestList extends the PropertyNameList wanted-properties-list mechanism to provide a way to request the properties of multiple resources indirectly through some relationship with the resource targeted by the server operation.

A number of server-contact methods accept an optional StpRequestList object, to which the client has added one or more requests for properties from resources on the server. When the operation completes, the StpRequestList object is populated with resource proxies containing the property values retrieved in response to the requests. For example,

```
    // Write properties from the current dialog tab
    // and fetch the properties for the "newTab"
    StpRequestList request =
        new StpRequestList(new PropertyRequest[] {
            new ModifiedProperties(DISPLAY_PROPERTIES_FOR_SIDE_EFFECTS),
            new TargetProperties(computePropertiesNeededForTab(newTab))});
    record.doWriteProperties(request);

    // Update display to reflect side-effects of the operation
    RefreshDisplay(request.getRequest(ModifiedProperties.class).getResponse());
    if (isEmpty(record.updatedPropertyNameList())) {
        // All fields were written, so proceed to next tab...
        record =
(Record)request.getRequest(TargetProperties.class).getResponse();
        // setup the new tab...
    } else {
        // Report failures and stay on the old tab
    }
```

# Additional resource proxies

This section describes some of the important proxy interfaces in the Rational Team API. These objects can be used to support working with collections of resources and lists of their corresponding Resource proxies. For more information, refer to the Javadoc that is available with each Rational Team API package.

- Folder

  A folder is a resource that contains a set of named mappings to other resources, called the "child resources" of that folder. The "children" of a folder are all of the child resources of the folder.

  The CHILD_ LIST property contains the list of children of a folder, while the CHILP_MAP property contains a mapping from a simple name to the child resource for each child in the folder.

  - Resources may be contained in a Folder
  - ControllableResources may be contained in a ControllableFolder

  Examples of folders:

  - ClearCase directories (ControllableFolder class, which is a subclass of Folder)
  - ClearQuest database set (DbSet class)

- StpRepository

  A repository is a container of the resources for a given product. Each persistent resource accessed through the Rational Team API is contained in one repository. However, a server may provide access to multiple repositories.

  Each Provider instance maintains a mapping from each repository it encounters to a set of user credentials for that repository. The Rational Team API handles the authorization to each repository through the Callback mechanism.

  Examples of repositories:

  - A ClearCase VOB (CcVob class, which is a subclass of StpRepository). The Rational Team API CcVob object contains objects such as CcVersion, CcVersionHistory, CcProject, CcStream, CcActivity, and CcComponent.
  - A ClearQuest user database (CqUserDb class, which is a subclass of StpRepository).

- StpProject

  An StpProject is a logical set of related artifacts treated as a unit. An StpProject is associated with one StpRepository, while an StpRepository may contain 0 or more projects.

  Resources used for change management tasks are contained by project resources. Multiple projects may be accessible from a given server, but not all projects can contain resources of all types.

A RequisitePro project (RpProject class) is a kind of project. In RequisitePro, a single database can store multiple projects. From a project, discussions, documents, groups, related projects, users and views can be obtained. A project also has a set of document types and requirement types. Security can be enabled or disabled at the project level.

There are currently two types of projects that are supported by the Rational Team API: Requirements projects and Defect Tracking projects.

Requirements projects contain the following resource types:

- Requirements
- Requirement types
- Relationships
- Attribute definitions
- Folders
- Discussions and Responses
- Documents and Document types
- Views
- Queries
- Users and Groups

From a RequisitePro project, a client may obtain the:

- Document types defined in the project
- Requirement types defined in the project
- Discussions defined in the project
- Documents defined in the project
- Views defined in the project
- Root of the project's package folder hierarchy
- External projects of this project
- Full UNC file path of the project
- Prefix used in cross-project traceability
- Auto-suspect state of the project
- Security-enabled flag for the project
- Queries of various types and visibility in the project

Defect-tracking projects (user databases) contain the following resource types:

- Records
- Attachments
- Record types
- Field definitions
- Actions
- Dynamic choice lists
- Queries
- Users and Groups

From a ClearQuest defect-tracking project, a client can find the following information:

- the record types defined in the project
- the record type that is to be used by default when creating records and when finding records by id
- the root of the user's personal query folder hierarchy
- whether or not a particular site has mastership of the project
- the root of the project's public query folder hierarchy

- Workspace

  A Workspace extends ControllableFolder and is a container for versioned resources (such as ClearCase elements in a view). If the Workspace is a Web view, it has a client-side component (the copy-area on the client machine), and a server-side component (the underlying ClearCase view). Such a workspace may be described either by its client-side copy-area file location, or by a server-side workspace location containing the view-tag as its identification. See the Javadoc for the Rational Team API ClearCase package (com.ibm.rational.wvcm.stp.cc.jar) for more information.

# Collections

Several Rational Team API methods return collections of resources. The resource collections may be returned as a ResourceList or as a ResourceList.ResponseIterator().

The value of many properties is a list of references to resources. The value of such properties is represented by a ResourceList object, which is a collection of proxy objects with a number of additional methods for performing specific operations on the members of the list. ResourceList-valued properties can use NestedPropertyNames to request properties from the resources in the list.

The ResourceList provides a number of methods for performing specific operations on the members of the list. A ResourceList may contain proxies of any Resource subclass. The proxies in a collection may be all the same proxy class or mixed depending on the generator of the list. A new ResourceList is created by the Provider.resourceList() method.

The ResponseIterator represents a stream of proxy information coming directly from the server, one proxy at a time, as the client moves through the items of the ResponseIterator. Until it is explicitly released (via ResourceList.ResponseIterator().release()) or its end is reached, the ResponseIterator holds open a communication channel with the server. For optimal performance, clients should examine the items in the iterator as quickly as possible and release the iterator as soon as it is no longer needed.

For example, to find available repositories using the Provider.userDbFolderList() method and specifying the type of repository you want (such as ResourceType.CQ_DB_SET or ResourceType.CC_VOB),

```
try {
    Provider provider = getProvider();

    // Request a list of the CQ databases known to the provider
    ResourceList databases =
        ((CqProvider)provider).userDbFolderList(DB_PROPS);

    // List the returned information
    for (Iterator dbs = databases.iterator(); dbs.hasNext(); ) {
        CqUserDb userDb = (CqUserDb)dbs.next();
        System.out.println (userDb.getDbSet().getDisplayName() + "/"
                            + userDb.getDisplayName()
                            + ": " + userDb.getComment());
    }
} catch(Throwable ex) {
    ex.printStackTrace();
} finally {
    System.exit(0);

}
```

See the Rational Team API Javadoc for more information.

# Additional information on resources

Proxies are not designed to be long-lived caches of information about a resource on the server. Their purpose is to marshal the data needed to perform a server operation prior to initiating it and to provide a container for returning the results of such an operation to a client. In a client/server application, it is preferable not to hold data on the client too long or it may get out of synch with the server. For this reason, the Rational Team API always returns a new proxy on each ***do*** method operation.

As a client, to access a resource on the server you have to have a Provider reference and know the location of the resource. Given a Provider and location, a proxy object for that resource can be obtained. Methods on the proxy allow the client to create or delete a resource at the location indicated by the proxy, read property values from the resource at that location, write properties to the resource at that location, or perform any number of other operations on the resource.

If the client does not have the location for a resource, it must browse the resource hierarchy for it, perhaps with the interactive help of the user. Non-versioned resources are found in projects and versioned resources are found in workspaces. So, if the client wants to work on non-versioned objects, it begins its browsing by obtaining a list of project folder proxies from the provider instance. (For versioned objects, it begins its browsing in a list of workspace folders.) The proxies in this list represent locations on the server where projects (or workspaces) may be found or constructed. See the various *Xx*`Provider.`*xxx*`FolderList` methods in the Rational Team API Javadoc for more information.

If the client already has one resource and wishes to create or locate another resource that is or will be related to it, the client can obtain from the resource in hand a list of folders where related resources of a given type can be found or created.

Using a method on the folder proxies in the returned list, the client can obtain a list of proxies for the members of that folder. By repeated application of this method, the client can form a hierarchical list of all the projects or workspaces visible to the user. From this hierarchy, the user can select an appropriate project for the task at hand or select a folder in which to create a new project.

## Resource type

A resource has a resource type, a unique location (in the form of a Location selector string), and a display name. Each type of resource has a unique interface by which it is accessed.

Any resource proxy returned by the Rational Team API as the result of a server contact implements the interface unique to the type of resource. The proxies constructed from Location objects by the Provider implement the interface specified for the construction method used.

## Creating a proxy for an existing resource

You can create a proxy for an existing resource, given a Location. In the following example, an Activity proxy is constructed for a ClearCase activity (named “cc.activity: developerName1_fix_a_bug@\projects”).

```
Activity my_activity =
provider.activity(provider.location("cc.activity:developerName1_fix_a_bug@\projects"));
```

The "my_activity" proxy is the client-side object that represents the activity resource. You can then perform operations on the activity through methods of the Activity proxy. For example:

```
// read properties of the activity
my_activity.doReadProperties(...)

// write properties
my_activity.doWriteProperties(...)

//delete the activity
my_activity.doUnbind()
```

## Creating a new resource

When the client wants to create a new resource of a given type, the client can specify a location where to create the resource. The client may first need to ask the provider (or one of its resource proxies) for a list of folders where resources of that type are to be created and then select one from the list, perhaps with the help of the user. The resulting folder will always be in a repository that supports resources of the given type, but the client may have to descend into the folder hierarchy to find a folder where creation of that resource is actually allowed. That is, a top-level folder in the returned list may not support creation of the resource, but one of its nested folders definitely will.

A new resource is created by providing a location as an argument to a type-specific creation method. The Activity proxy (`a2`, in the following example) is constructed in advance of the server-side object existing. For example:

```
CcStream stream = . . .;
CcActivity a2 =
provider.ccActivity(provider.stpLocation("cc.activity:a_new_activity@\projects"
));

// set the headline
a2.setHeadline("The new task");

// Set the stream
a2.setStream(stream);

// create the activity
a2.doCreateResource();
```

Any properties that may be needed for the creation, such as the stream for the activity, must be set in the proxy before calling the create method. You cannot create a new resource with empty or invalid values for required properties. The failure occurs (as a WvcmException) when you call the `doCreateResource()` or `doCreateGeneratedResource()` method.

A new resource (including a file-area private ControllableResource or ControllableFolder) is constructed in the following steps:

1. Determine the desired address for the new resource.

2. Construct a Location object for that address using one of the Stp`Provider.stpLocation()` methods. Create a new location for the resource by adding a child segment to the selected folder location.
3. Obtain a proxy for that location from the provider. Construct a proxy whose object class matches the desired type of the new resource using the appropriate Provider proxy factory method.
4. Populate the proxy with any property values needed or desired for the new resource. Establish the initial values for settable resource properties using the setters on the new proxy.
5. Invoke the create-resource operation (`doCreateResource` method) on the proxy, which returns a new proxy for the newly created resource containing any property values requested in the create-resource operation.

### Creating a versioned resource

Since a controllable resource or controllable folder in a ClearCase VOB must be controlled, creation of such resources follows a pattern similar to creating a Resource except that the client must use the `ControllableResource.doControl` method rather than `doCreateResource`.

## Change contexts and actionable resources

A change context is a workspace or container for editing actionable resources, which are non-versioned resources such as ClearQuest records and queries and RequisitePro requirements. To edit such a resource the client starts an action, which copies the resource to a change context. Changes are made to the copy within the change context and then, when all changes have been made the modified resource is delivered back to its permanent location in the repository.

Any number of resources can be added to and subsequently edited within the same change context simultaneously, but all modified resources are delivered from a given change context all at the same time. The delivery process empties the change context of all modified resources and activates any triggers or hooks associated with the modifications.

Prior to delivery, the modified resources within a change context are visible only to the user who initiated the modifications. Each change context is associated with a Provider and access to the contents of the change context requires the use of a proxy obtained directly or indirectly from that Provider.

### Actionable resources

An StpActionableResource is a type of resource that must be edited using the Action paradigm. An StpAction object represents a method to be applied to an actionable resource.

Most non-versioned artifacts implement the Stp.StpActionableResource interface. Some of the StpActionableResource types are:

- The `stp` package includes StpActionableResource, StpDefinedResource and StpQuery interfaces.
- The `cq` package includes CqAttachment, CqDynamicChoiceList, CqQuery, CqRecord interfaces.
- The `rp` package includes RpAttributeDefinition, RpDiscussion, RpDocument, RpDocumentType, RpFolder, RpGroup, RpProject, RpQuery, RpRelationship, RpRequirement, RpRequirementType, RpResponse, RpRevision, RpUser, RpView interfaces.

The process of modifying actionable resources involves multiple steps:

1. Initiate: the client specifies the action to be used in the modification, thus declaring the business rules to be followed in making the modifications. The proxy used to initiate the modification determines the change context for the modification.
2. Modify: The modifications are made to the resources and verified according to the business rules.
3. Deliver: When all resources have been modified, all of the changed resources in the change context are delivered back to their respective repositories to make them permanent.

This modification process allows the client to work with its user to make coordinated changes to multiple resources, with the option of altering or abandoning at any time changes to any of the resources involved.

Once a modification has been initiated by a client for a user, changes made to the resources involved are not visible to other users or clients until the modifications are delivered back to their respective projects. The changes are confined to the change context used and visible only through proxies obtained from the Provider of that change context.

The precise locking semantics of this modification process are repository-dependent. The only guarantee is that if an initiate-modify-deliver sequence is successfully completed by a client, the resources modified did not change in the repository while they were being modified by that client.

When the modification of a resource is initiated, a writable version of the resource is created in the change context associated with the proxy used. Unless the resource is being created, the properties of the original resource are subsequently copied to this new version. Subsequent operations targeting the original resource through a proxy from the same change context will be redirected to operate on the version cached by the change context. Only those proxies obtained directly or indirectly from the provider for that specific change context will see the changes before they are delivered.

When modifications are delivered, all of the changes in the change context are delivered together. For changes within a single repository, the delivery of all the changes fails if the delivery of any one of the changes fails. For multiple repositories in one change context, failure to deliver one repository does *not* guarantee that the others will also fail.

If the client wishes to maintain two or more ongoing but independent modification tasks for the user, it must use a different change context for each task. For example, if the user is editing a resource and decides to compose, execute, and save a query before committing the edited resource to the server, the client must construct a new change context in which the user can do the query work so that the query can be saved without committing the resource.

The changes made to resources within a change context may be discarded by restoring the resource to its original state. This deletes the version of the resource from the change context.

**Creating a new actionable resource**

A new StpActionableResource is constructed by the following steps:

1. Create a new location for the resource by adding a child segment to the selected folder location.
2. Obtain a proxy for that location from the provider in whose change context you want to perform the modifications.
3. Populate the proxy with any property values needed or desired for the new resource.
4. Invoke the `doCreateGeneratedResource()` operation on the proxy, which returns a new proxy for the newly created resource containing any property values requested in the create-resource operation.
5. For StpActionableResources, deliver the newly-created resource to a project.
   The final deliver step is needed to make the new resource accessible to other users of the system. It is also the main trigger for business logic running on the server.

**Note:** The doCreateResource methods contact the repository (which could be a server or a file area) referenced in the proxy's Location to construct a new resource and then write the property values in the proxy to the newly-created resource. The creation of a new StpActionableResource follows a similar pattern except that the client should use Stp`ActionableResource.doCreateGeneratedResource` rather than `doCreateResource`.

## Additional information on change contexts

The StpChangeContext resource contains the server state of a change context, that is, the modified copies of the resources that have been changed in the change context but have not yet been delivered back to their repository.

The modified resources contained by a StpChangeContext cannot be accessed using a StpChangeContext proxy. To access the modified resources, the saved context must first be opened and associated with a provider (see Stp`Provider.doOpenContext`). Then the modified resources can be accessed using proxies obtained from that provider. (See Stp`Provider.doGetModifiedResources` in the Javadoc.)

A change context resource exists on each repository that contains modified resources of a given Provider's change context. The first change context resource created for a change context is designated the primary location for the change context. Subsequently constructed change context resources are designated secondary locations.

A StpChangeContext proxy may be used to retrieve or set properties of the primary change context location, such as its DISPLAY_NAME or its DESCRIPTION. These may then be used for identification of the change context.

Operations that modify a non-versioned resource cause the resource to be copied from its permanent location in the repository to a change context, where it is actually modified. Which change context maintains the copy is determined by the proxy used to do the modification. The modified copy of the resource hides the corresponding resource in the repository until the change context is delivered or deleted or the modified copy is removed from the change context.

An edit in progress may be abandoned using the `doClearContext()` method.

The `doGetUnopenedContexts()` method of Provider allows a client to obtain from the specified repositories any change contexts (for the user) that have been persisted. One of the returned StpChangeContext proxies may be passed to the `doOpenContext()` method to recover the modified resources and continue with the editing process.

A change context resource is created automatically by the server the first time it is asked to start an action. A client may also explicitly create the change context using the `doCreateGeneratedResource()` method of the StpChangeContext class. The form of a change context selector is

`<`*`type`*`>.context:<`*`uuid`*`>@<`*`repo`*`>`

The <*uuid*> is a unique identifier generated by the server that created the change context.

The content of a change context is allowed to span multiple repositories, multiple repository types, and multiple servers.

Rather than copy resources between repositories, each change context is distributed across the repositories of the resources that it contains. In each repository, there is a change context resource for the change context. Each change context resource of a given change context is identified by the same <*uuid*>—hence a change context's <*uuid*> must truly be universally unique.

**Note:** The terms *change context* and *change context resource* refer to different things. A change context is an aggregate object made up of individual change context resources. The change context is manifested by the StpProvider class, whereas the change context resource is manifested by the StpChangeContext class.

The repository where the first change context resource for the change context was created is designated the change context's *primary location*. The remaining repositories are designated *secondary locations*.

Once established, the resource at the primary location exists for the lifetime of the change context as determined by the client. Each secondary change context resource, on the other hand, exists on its server only as long as it contains modified resources.

Every change context resource defines the following properties:

- PRIMARY_LOCATION: the location of the change context at the primary location. Servers can use this value to determine if they are working with the primary location or a secondary location.
- IS_PERSISTENT: determines what happens to the change context when a session using that change context terminates.
- MODIFIED_RESOURCES: enumerates the resources on the given repository that are contained in the change context.
- IS_EMPTY: indicates whether or not the change context resource has modified resources.

All other properties of a change context defined in the Stp`ChangeContext` interface are stored only in the primary location.

Each instance also contains the modified local resources and some way to map the original location of each resource into the location for the modified copy of the resource.

# Additional information on proxy methods

If a Resource proxy addresses a file area resource, a server will be contacted only if the operation cannot be completed by interacting with the file area. For example, calling `doWriteContent()` or `doReadContent()` of a controllable resource whose content is stored locally on the client are not executed on the server but do access the resource in the file area.

Many *do* methods take an optional ProperyNameList or StpRequestList parameter in which the client can request which properties are to be read from the resource as part of executing that method. All *do* methods write to the actual resource any property values that have been set in the proxy since the last server interaction.

- The names of the properties that are currently stored in a given proxy are returned by `Resource.propertyNameList()`.

- The names of the properties whose values have been modified in the proxy but not yet written to the resource are returned by `Resource.updatedPropertyNameList()`.
- The names of the properties requested by the last server interaction are returned by `Resource.wantedPropertyNameList()`.

The `Resource.doGetPropertyNameList` method contacts the server and returns the property names defined by the server for the given resource.

## Additional information on ControllableResource proxy methods

The Resource class offers the `doCopy()`, `doRebind()`, and `doUnbind()` methods for copying, renaming, and removing the resource. The Resource class itself does not supply a method for creating the underlying resource, because not all resources are creatable by the end user, using a Rational Team API client application. Note the distinction between creating the proxy, which is just a matter of instantiating a Resource object, and actually creating the resource, which needs to be done via `doCreateResource()` or `doCreateGeneratedResource()`.

A ControllableResource represents a file in the file system that is under source control, or can be put under source control. A file in a file-area is represented by a ControllableResource. A ControllableResource is the proxy through which ClearCase operations on elements can be performed. For example, the ControllableResource class includes the following methods:

`doControl()` - similar to make element
`doCheckout()`
`doCheckin()`
`doUncheckout()`
`doMerge()`
`doRefresh()` - updates, refreshes the client resource from the server

WVCM distinguishes between the ControllableResource, which is the file in a file-area through which the above operations are performed, and the underlying element (VersionHistory) it is associated with. ClearCase elements are represented by VersionHistory objects in WVCM. An element is a history of versions.

The `doControl()` method of ControllableResource creates a VersionHistory object and associates it with the ControllableResource. It also creates the initial Version object, having the same contents as the ControllableResource, and puts that Version in the VersionHistory. Subsequent `doCheckin()` operations create new Version objects and include them in the VersionHistory. Version objects have predecessor(s) and successor(s).

In WVCM, all relationships are represented using properties. For example,

- the VERSION_HISTORY property of a ControllableResource object has as its value the VersionHistory object (element) that is associated with the ControllableResource.
- a Version object has a SUCCESSOR_LIST property whose value is a list of Version objects that are successors (in the version graph) to the given version.

As illustrated by these examples, properties can have arbitrary values, including other objects or lists of objects.

The following code fragment builds a ControllableResource proxy for a known file in a file-area, and checks out that file. The IS_CHECKED_OUT property is checked before and after the checkout.

```
// Get provider. For example,
provider = ProviderFactory.createProvider(providerName, callback,
hash);

// Create the PropertyNameList proxy. The IS_CHECKED_OUT property is
// the one property name on the list of wanted property names.
PropertyName[] wantedPropNames = {
ControllableResource.IS_CHECKED_OUT };
PropertyNameList wantedProps = new
PropertyNameList(wantedPropNames);

// Create a ControllableResource proxy. The checkoutPath value
// is a string whose value is a
// selector string that specifies the copy-area-file –
// “C:\my_views\testview\avob\file1”
Location loc = provider.location(checkoutPath);
ControllableResource res1 = provider.controllableResource(loc);

// First check if IS_CHECKED_OUT is false
res1 = (ControllableResource)res1.doReadProperties(wantedProps);
boolean  isCheckedOut = res1.getIsCheckedOut();
System.out.println("before doCheckout(): IS_CHECKED_OUT=" +
isCheckedOut);
myAssert(isCheckedOut == false);

// Check it out
res1.doCheckout(true, null, false, true);

// Check if IS_CHECKED_OUT is true, to verify that the
// property was updated in the ‘res1’ proxy and in the newly-
constructed ‘res2’ proxy.
res1 = (ControllableResource)res1.doReadProperties(wantedProps);
myAssert(res1.getIsCheckedOut() == true);
ControllableResource res2 = provider.controllableResource(loc);
res2 = (ControllableResource)res2.doReadProperties(wantedProps);
isCheckedOut = res2.getIsCheckedOut();
myAssert(res2.getIsCheckedOut() == true);
```

# Additional information on properties and meta-properties

While some properties are specific to a given resource, many properties are common to all resources. From any resource, a client may obtain the following information.

- A location of the resource.
- A string representing the unique and persistent location of the resource.
- A string containing a user-oriented name for the resource intended solely for display purposes.
- A string containing a user-oriented description of the creator of the resource.
- The time and date the resource was created and last modified.
- A string containing a user-oriented comment about the resource.
- If the resource has content (in addition to properties), pertinent information about the size and form of that content.
- The resources that directly or indirectly contain the resource.
- A shallow or deep list of the resources that the resource contains.

Note: Although these properties are defined for all resources, some resources may return null or empty values for them or provide a `NOT_SUPPORTED` exception for them.

All properties (except the location) must be explicitly requested from the server before they are available from a proxy.

The Resource class provides generic methods for accessing the property values defined by a proxy using the PropertyName object for each property. WVCM defines the methods `Resource.getProperty(PropertyNameList.PropertyName)` and `Resource.setProperty(PropertyNameList.PropertyName, Object)`. The getProperty method throws the exception PropertyException if the proxy does not contain a valid value for the property identified by the given PropertyName object.

As an extension to WVCM, the Rational Team API defines Resource.getPropertyValue(). If the property value is defined, getPropertyValue returns the same Object as getProperty. If the property value is undefined, `getPropertyValue()` returns the same exception that would be thrown by getProperty.

Note: The `setProperty`, `getProperty`, and `getPropertyValue` methods do not verify that the given PropertyName is defined by the proxy class in hand. Any proxy can be used to interact with any type of resource. Such interactions will fail only when they attempt to write or retrieve values for properties that are not defined for the resource addressed by the proxy and the failures will occur only when the API Provider attempts to transfer such property values to or from the resource.

# Additional information on Location objects

Each Resource has a location, which uniquely identifies the resource at a given point in time. For a file-based resource, the location is expressed as a file pathname. For a server-side resource, the location contains the information needed to find the object from the server side, (for example a database ID in a VOB).

The location of the resource is required when reading or writing resource content, or reading or writing properties on a server.

Location objects:

- can be mapped to and constructed from strings
- support hierarchical operations (such as parent, child, and lastSegment)
- are composed of name segments as in a typical file path name and impart a hierarchical structure to the namespace for resources in a repository

The StpLocation object represents a resource address and, as such, can be used to construct a resource proxy (by using one of the resource proxy factory methods of Provider). As a general rule, the Resource proxy constructed from a Location must be the same type as the resource addressed by the Location. However, the type of resource addressed by a Location cannot always be determined from the address specification alone, so this rule cannot always be enforced at the time the proxy is constructed.

All resources have a stable form of location that may be safely used to store resource identities on the client between client sessions. This stable location might not be the location used to create the resource, however. The server creates this stable location for the user at resource creation time and is always available as an unchanging property of the resource. The Rational Team API StpLocation object provides methods for obtaining the string representation for a location and for parsing that string back into a location.

There is a common location specification syntax for locations of all resource types in the Rational Team API. A location specification is a:

- string format for identifying Rational Team API objects
- string representation of a WVCM Location object

An StpLocation instance represents a location specification that has been parsed into its various component fields. A number of different formats or *schemes* are used to express the location of various resources as a string. These schemes consist of one or more of the following fields: *domain*, *repository name*, *namespace*, and *object name*. It is the namespace field that determines the scheme being used.

The StpLocation interface provides methods for parsing a location specification into its constituent parts (*domain*, *repository name*, *namespace*, *object name*). Using the available methods, Rational Team API clients can examine location specification provided by the end user to determine if they are appropriate for the context in which they are being used. Based on this analysis, a client can fill in parts of the location specification omitted by the end-user if the context defines those missing parts unambiguously.

The `StpProvider.stpLocation()` method facilitates this process by filling in a missing location string scheme from its Namespace parameter and filling in a missing repository from the default repository identified to the Provider by the client. (See the `StpProvider.setDefaultRepository` method in the Javadoc.

For operations requiring a user-specified file-system artifact, a file-system pathname into a workspace file area is sufficient and appropriate. For operations requiring a user-specified object of another type (such as an activity, project, record, or requirement) you use a syntax that specifies the general, fully-specified form of a location string that identifies an object by name:

```
<domain>.<namespace>:<object-name>@<repository-name>
```

For example:

```
"stream:mystream@projects" (where the default <domain> is implied)
```

- <domain > is used to distinguish between different repository types, implementations, or *providers*, of similar objects (for example, between a UCM 1.0 project and a Requisite Pro project).
- <namespace> identifies a namespace in which <object-name> is recognized. In the Rational Team API, several different kinds of objects may appear in the same namespace. For example, in the record namespace can be found database sets, databases, record types, records, fields, and attachments. A given resource may also appear in multiple namespaces. Repositories, for example, appear in each namespace that contains a resource within the repository. Hence a ClearQuest user database appears in the record, action, and query namespaces. The namespace used to name a folder controls the meaning of doReadMemberList since the locations for the members of a folder must be simple extensions of the name of the folder and hence they must be in the same namespace as the folder.

Software that understands location specifications may allow various fields to be omitted. For example:

- <domain> may be omitted if the type can be inferred, or there is a default type.
- <namespace> prefix may be omitted if the namespace of objects is understood by the context, or if there is a default namespace.
- <object-name> is omitted when referring to a repository itself, or to the root of a namespace.
- <repository-name> may be omitted if there is a default repository.

## Filename location specifications

ControllableResources in a file area can be referenced by an absolute or relative file system pathname, using the host operating system naming conventions. For example:

```
/my_views/my_dir/my_file.txt
```

The namespace of a file location specification is Namespace.PNAME. The object name field is the pathname and the remaining fields are null. The pathname is not modified by the Provider or Location class.

The HTTP file scheme prefix, "file://" is also recognized as a valid flag for a file selector. For example:

```
file:///c:/my_views/my_testview/my_testDir/my_file.txt
```

### Stable locations

An alternate form of location specification uses repository and object IDs (typically UUIDs, but not always), instead of names. Stable locations:

- are more efficient to resolve than filename location strings
- provide a more stable reference for an object (i.e. independent of any renaming)
- provide identifiers for objects that do not have names

Note that all named objects are considered to have stable locations, even if the <object-id> and <repository-id> fields of the selector happen to be identical to the <object-name> and <repository-name> fields of the user-friendly location. (That is, if an object can only be identified by name, then its name is the most efficient and stable form of identification, because it is the *only* form.)

The general form of a stable location is,
<domain>.repo.<resource-type>:<object-id>@<repository-id>

For a ClearQuest record, the form is,
cq.repo.cq-record:<record-type>/<record-id>@ <db-set-name>/<database-name>

For example,
"cq.repo/cq-record:Defect/SAMPL00000005@7.0.0/SAMPL"

# Exceptions

All problems are reported via an exception object - a subclass of the exception object defined by WVCM. From such an exception, the client may obtain the following information:

- A reason code (extension of WVCM enumeration), classifying the type of incident being reported (for example, WvcmException.ReasonCode.READ_FAILED).
- A subordinate reason code, providing a finer classification of the incident within the classification of the reason code (for example, StpException.StpReasonCode.CONFLICT)

- A locale-independent message identifier (catalog index) and argument values specific to the incident being reported. This information would be suitable for logging purposes and could also be used to generate a localized message.
- A list of nested exceptions, each of which describes a subordinate incident that contributed to the one being reported by the exception.
- For resource access problems, the primary resource involved in the incident—typically the resource targeted by the operation that failed.
- For property access problems, the property of the primary resource involved in the incident.
- For operations addressing multiple resources, a list of the resources for which the operation was successful.
- For selected incidences, additional information specific to the incident being reported.

Each operation defines a set of preconditions that must be met for the operation to succeed (for example, a resource must exist to read its properties, a resource must not already exist with the same name for one to be created, a resource must be versioned and checked-in to be checked-out). Violations of such preconditions cause the operation to throw an exception.

Operations can often be applied to a collection of resources. If the operation fails on any one of them, an exception is thrown (with the successes reported in the exception object).

Problems encountered by the server while reading or writing the properties or meta-properties of a resource do not cause a Rational Team API operation to throw an exception. Instead, the exception is associated with the property within the returned proxy. Only when the client attempts to extract that specific property value from the proxy will the exception be thrown. The client may also interrogate the proxy prior to extracting the property value to determine if there were problems and obtain the exception without it being thrown. See Additional information on properties and meta-properties for more information.

The principal types of exceptions in the Rational Team API are:

- StpException (extends WvcmException)
- StpPropertyException (extends StpException)
- StpPartialResultsException (extends PropertyException)

## StpException

StpException is an extension of WvcmException and is the root of all checked exceptions thrown by the Rational Team API. All implementations of WVCM-defined methods are documented to throw WvcmException. All public methods of the Rational Team API that are extensions to WVCM are also documented to throw WvcmException. However, the implementations of all these methods consistently throw only StpExceptions – not WvcmExceptions. The conventions that apply are:

- `throws StpException` is never used in any method declaration (public or otherwise). Even when a method throws an StpException it is declared as if it throws a WvcmException.
- a method never throws a new WvcmException. Even though the exception could be expressed as a WvcmException, it is always thrown as a new StpException() instead.

## StpPropertyException

StpPropertyException extends StpExtension and is the base exception class for errors associated with the reading or writing of resource properties.

After a property value is requested from a server, its name is associated with the result and stored in a proxy. That name is associated with the retrieved value if the retrieval attempt was successful or with status information (in the form of an StpPropertyException object) if the retrieval attempt was unsuccessful.

## StpPartialResultsException

StpPartialResultsException extends StpPropertyException and is used for reporting the failure of an operation or property involving multiple resources. It becomes a substitute for the ResourceList that would normally be returned by the operation or property. It contains a ResourceList, which has the proxies for the resources that were successfully processed, and a list of StpExceptions, each corresponding to a resource for which the operation failed. See the Javadoc for more information.

# Use case examples

A tutorial is available for each sub-provider that illustrates some of the common use cases available for each Rational Team product by using the Rational Team API. Common use cases include:

- RequisitePro sub-provider (See the RequisitePro use case tutorial)
    - Open a project and retrieve properties
    - Display the project views
    - View requirement types and their attributes
    - Create, retrieve, and modify requirements, requirement attributes, and traceability
- ClearQuest sub-provider (See the ClearQuest use case tutorial)
    - Create, modify, and delete a record (for example, create a defect change request record type)
    - Select an action and change a state
    - Create a query, select a query and execute it – also, modify an existing query and save with a new name
    - Create, modify, and delete field values in a record
- ClearCase sub-provider (See the ClearCase use case tutorial)
    - Create, populate, and remove Web views (ClearCase Remote Client (CCRC) or ClearCase Web views)
    - Operate on elements within Web views (such as checkout, checkin, and hijack operations)
    - Navigate VOB-object hierarchies and inquire properties of ClearCase objects

# Rational Team API class overviews

The Javadoc that is included with the Rational Team API provides an Overview section that includes descriptions of the available classes. Each class represents a major collection of operations supported by the API.

Descriptions of the classes include:

- The base classes (WVCM and Stp package classes)
- ClearQuest Change Management classes
- RequisitePro Requirements Management classes
- ClearCase Asset Management classes

In addition to class summaries, the Javadoc provides detailed reference information for each package.