IBM VisualAge TeamConnection

Toolbuilder’s Development Kit

Version 2.0

SC34-4553-00

IBM VisualAge TeamConnection

Toolbuilder's Development Kit

Version 2.0

SC34-4553-00

Second Edition (September 1997)

Note

Before using this document, read the general information under “Notices” on page ix.

Contents

Figures

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. 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. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, USA 10594.

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 the Site Counsel, IBM Corporation, P.O. Box 12195, 3039 Cornwallis Road, Research Triangle Park, NC 27709-2195, USA. 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.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

IBM may change this publication, the product described herein, or both. These changes will be incorporated in new editions of the publication.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

BookManager	MVS/XA
Common User Access	Operating System/2
C Set++	OS/2
CUA	SOM Developer's Toolkit
C/370	TeamConnection
IBM	VisualAge C++
MVS/ESA	XGA

The following terms are trademarks of other companies:

ObjectStore
ObjectStore Design, Inc.

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

About this book

This book is part of the documentation library supporting the Toolbuilder's Development Kit (TBDK) feature of the IBM VisualAge TeamConnection licensed programs. It explains how to create and extend tools and the information model for accessing objects in TeamConnection. It contains guidance and reference information.

How this book is organized

This book is divided into the following parts:

"Chapter 1. Toolbuilder's Development Kit concepts and requirements" on page 1 , provides an overview of the Toolbuilder's Development Kit and the tasks you will need to perform to create a tool or extend the information model.

"Chapter 2. Designing a model" on page 3, explains some of the concepts involved in designing a model for a tool.

"Chapter 3. Creating a model for a tool" on page 7, explains how to create a model for a tool using the browser/editor (TBDK Breditor) shipped with the Toolbuilder's Development Kit.

"Chapter 4. Working with CDL and IDL" on page 11, explains how to write class definition language (CDL) and interface definition language (IDL) for generating code for a model.

"Chapter 5. Working with views" on page 21, describes the construction and uses of views.

"Chapter 6. Build processes for the Toolbuilder's Development Kit" on page 25, provides build information related to the model integration process.

"Chapter 7. Object handles" on page 29, explains the significance and use of object handles.

"Chapter 8. Accessing information model objects" on page 33, explains how to code your tool to access information model objects stored in the TeamConnection database.

"Chapter 9. Exception handling" on page 53, explains how your tool can handle TeamConnection exceptions.

"Chapter 10. Performance and scalability issues" on page 63, provides a discussion of performance and scalability issues related to the Toolbuilder's Development Kit.

Conventions

This book uses the following highlighting conventions:

- *Italics* are used to indicate the first occurrence of a word or phrase that is defined in the glossary. They are also used for information that you must replace.
- **Bold** is used to indicate items on the GUI.
- `Monospace` font is used to indicate exactly how you type the information.
- File names follow Intel conventions: **mydir\myfile.txt.** AIX and HP-UX users should render this file name **mydir/myfile.txt.**

Tips or platform specific information is marked in this book as follows:

Shortcut techniques and other tips

OS/2

IBM VisualAge TeamConnection for OS/2

3.1

IBM VisualAge TeamConnection for Windows 3.1

NT

IBM VisualAge TeamConnection for Windows/NT

95

IBM VisualAge TeamConnection for Windows 95

IBM VisualAge TeamConnection for AIX

hp

IBM VisualAge TeamConnection for HP-UX

Tell us what you think

In the back of this book is a comment form. Please take a few moments to tell us what you think about this book. The only way for us to know if you are satisfied with our books or if we can improve their quality is through feedback from customers like you.

Chapter 1. Toolbuilder’s Development Kit concepts and requirements

The Toolbuilder’s Development Kit is a platform for creating tools to enhance or facilitate application development and management in the TeamConnection client-server environment. The Toolbuilder’s Development Kit extends TeamConnection’s storage, versioning, and configuration management capabilities to object-oriented development environments. With the Toolbuilder’s Development Kit the objects you create in a C++ or Smalltalk development environment can be subject to the same rigor that has been the norm in the industry for file-based development for some time.

TeamConnection is a client-server system, with information stored persistently on the server in an object-oriented database. The client retrieves objects from the database and makes them accessible through the TeamConnection cache services (TCCS) interfaces. Tools access these objects through the TeamConnection cache; when appropriate, the objects (if changed) are stored back into the database.

Servers interact with the TeamConnection client cache through the TeamConnection API (TCAPI). The TCAPI is used to retrieve, store, list, and query objects from the persistent store and to navigate to objects currently in the cache. The persistent store is managed by the ObjectStore object-oriented database.

For supplemental information related to the TeamConnection information model , see the *TeamConnection Information Model Reference*.

Development using the Toolbuilder’s Development Kit consists of developing a model for your tool, and then extending or augmenting the existing information model used by TeamConnection to support the tool model. Your model will consist of one or more entity classes, tied together in a network by one or more relationship classes.

Once you define your model, you will add your model to the server’s database schema (also called the storage view), extend the cache to support your classes (the cache view), and construct your tool using the TCCS interfaces.

Tool-building tasks

If you have a background in object-oriented technology, it will be easier for you to understand the Toolbuilder’s Development Kit, because it is an object-oriented environment. In particular, developing a tool in this environment requires you to perform the following tasks:

- Design an entity-relationship model for your tool
- Use the TBDK Breditor to define the Class Definition Language (CDL) or Interface Definition Language (IDL) files for your model, or write your own CDL/IDL. Currently the TBDK Breditor is equipped to perform cache code generation using CDL; server and view definitions use IDL for code generation.
- Build the model for the cache and the server (two separate steps)
- Define views of your model for use by tool code
- Write your tool’s code using the views that you have defined and the interfaces provided by TCCS

If you have previously written or generated IDL or CLS to describe your model, you can load your model into the TBDK Breditor to continue working on it. See “Loading CDL or IDL (via CLS) into the TBDK Breditor” on page 9 for more information about loading an existing model into the TBDK Breditor. See “Chapter 4. Working with CDL and IDL” on page 11 for more information about using CDL or IDL.

Hardware and software requirements

In order to complete the task of building tools, you need the following:

- All software required by or included in the TeamConnection Version 2.0 base product
- VisualAge C++ and/or IBM Smalltalk

Refer to the TeamConnection Version 2.0 Release for Announcement (dated 11/26/96) for detailed version and release level information for software products required by the Toolbuilder’s Development Kit.

OS/2

The SOM Developer’s Toolkit (including the SOM compiler) is required if you want to convert existing IDL to a CLS file. See “Loading CDL or IDL (via CLS) into the TBDK Breditor” on page 9 for an explanation of this process.

Chapter 2. Designing a model

You construct a model for your tool by first identifying the entities (or objects) in the application and the relationships among them. This approach is called entity-relationship modeling. Both entities and relationships can have attributes (such as a *name* or *address*) and relationships (such as *marriedTo* or *worksFor*). Relationships themselves can have relationships with other entities or relationships.

Once you design the network of entities and relationships, you can define views on the information that let you manipulate entire networks, or subsets of a network, easily and concisely.

Entities

Entities are classified in one of two ways: they are either TCParts or dependent objects. (Dependent objects are also referred to as ***plumbing objects***.) TCParts have names that you can use to provide uniqueness or to search the repository. Dependent objects do not have names; they are dependent on a TCPart for their existence. You cannot retrieve or manipulate dependent objects without navigating (that is, traversing a relationship) from a TCPart.

An entity can inherit its properties not only from TCPart but also from another entity (called a ***superclass*** of the entity). Multiple inheritance is not supported.

Handles are the mechanism used by the client cache to access model objects.

See “Chapter 7. Object handles” on page 29 for a detailed explanation of handle allocation and management.

Relationships

The only way to refer to one entity from another is by traversing a relationship. Relationships are a special class of dependent objects; they have two special attributes: the source and the target of the relationship. A relationship cannot exist without both a source and a target; in this way, it is dependent on both the source and the target. If you delete either the source or the target, the relationship is deleted too.

Note: Because pointer values are not persistent, it is not advisable to use pointers to refer to one object from another. The address of the target object could change from one reference to the next.

Relationships can have attributes and can be the source or target of other relationships. Instances of relationships do not have names; they are always dependent objects, controlled by a TCPart.

Relationships also have direction, cardinality, order, and control. Together these properties are referred to as ***relationship semantics***.

Direction

A relationship has a source and a target. Traversing the relationship from the source to the target is traversing the relationship in the primary direction. The source of the relationship owns the relationship, which has ramifications for updating the relationship. These ramifications are related to the TeamConnection merge capability. If two separate development efforts affect a single object, the result is two versions of that object. When it is time to merge the updates made to the two versions into a single version, TeamConnection must examine all of the relationships that are owned by the object. Since an object owns relationships in the primary direction, all relationships in the primary direction are merged when the two versions of the object are merged.

Direction also has an important impact on authorization checking. To change a relationship you must have authority to update the source object. No authorization checks are made for the target object.

Relationships can be traversed in the inverse direction as well. There is no concept of a one-way relationship.

The roles that describe the relationship’s direction are the primary verb (such as employs) and the inverse verb (such as isEmployedBy).

Cardinality

Relationship cardinality controls how many relationships can be constructed between a source and target object. Cardinality is expressed as a range of values: one-to-many (1..m), many-to-one (m..1), one-to-one (1..1), and many-to-many (m..m).

Order

Some tools may require that certain relationships be maintained in a specific order. These relationships can be maintained in an ordered collection and are called ordered relationships. By default, relationships are not ordered. Ordered relationships are kept in the order they are created, but they can be reordered by tools.

Control

Dependent objects must be controlled by a TCPart.

Objects that are subclasses of TCPart cannot be controlled.

For more information on the base classes for models, including TCPart, see “Chapter 3. Creating a model for a tool” on page 7.

Views

Views help you manage and work with extensive networks of objects according to the needs of the tools. You define views of objects and their relationships by writing IDL and compiling it for use by both the client and the server.

You can define views over your object that include or exclude attributes, including attributes used to model relationships. Once you define a view, you can retrieve or store the view as though it were one entity, rather than a (potentially large) set of entities.

Defining a view is a way to define how your tool will access information. It describes a path through the attributes and relationships that constitute the instance data. Views can include relationships that are mutually exclusive, since only one of the relationships can actually be instantiated in the data.

Views can include attributes (including those defined in superclasses), traverse relationships into the same or other objects, or embed another view definition by traversing relationships into the other view.

You can also construct a view by selecting types of relationships in which the object that owns the relationship has specified an abstract superclass as the implementation of the relationship.

For additional information related to constructing views, including a sample view IDL, see “Chapter 5. Working with views” on page 21.

Constraints

The existence of entities, relationships, and attributes is controlled by constraints. Constraints are the conditions under which entities, relationships, and their attributes are created and deleted.

Exceptions

TeamConnection exceptions are provided as a means of evaluating the success of client actions. See “Chapter 9. Exception handling” on page 53 for detailed information on exception handling.

Chapter 3. Creating a model for a tool

This section provides an approach for developing a tool model.

Understanding the cache view hierarchy

After you have designed the model for your tool, you begin constructing it by placing your entity and relationship classes in the cache view hierarchy according to the behavior you want them to have.

Figure 1 shows the base classes (along with related classes) for all models, as represented by the TBDK Breditor.

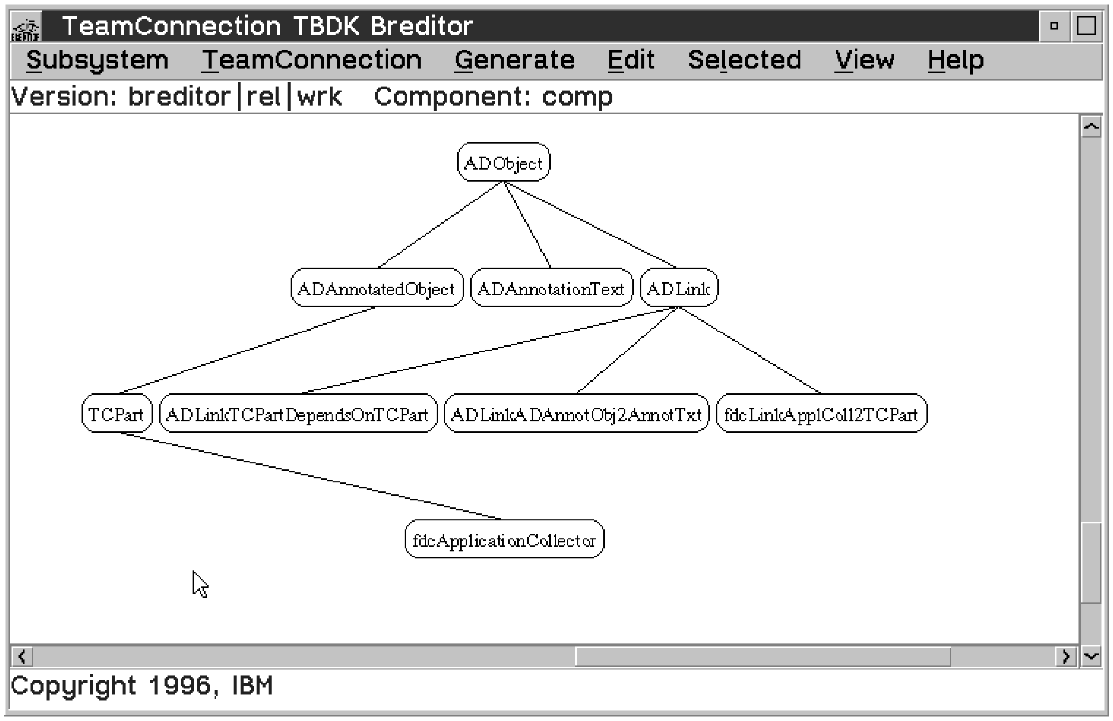

Figure 1. The cache hierarchy

The base classes are defined as follows:

ADObject
: All handles inherit directly or indirectly from this object.

TCPart
: All object handles managed by TeamConnection inherit from this object.

ADLink
: All relationship handles inherit from this object.

_ADObject
: All objects inherit directly or indirectly from this object.

_TCPart
: All objects managed by TeamConnection inherit from this object.

_ADLink
All relationship objects inherit from this object.

_TCPart and TCPart represent the set of entities that are managed by TeamConnection (using check in/out, lock/unlock, and so on), while _ADObject and ADObject represent the class of entities that are controlled by _TCPart and TCPart through relationships or relationship paths, which are represented by ADLink and _ADLink.

You place your classes into the hierarchy by designating a superclass for them.

TCPart names and version context

TCPart names are unique within a class and within a release. The TCPart baseName cannot include the ″:″ and ″|″ characters.

The syntax for the version context is as follows:

```
family|release1 release2 ... releasen|workarea1 workarea2 ... workarean
```

where `releasen` must have a corresponding `workarean`.

The ″workarea″ can be a driver, a release, or a version ID if the tool is doing only reads.

A TCPart can be linked to more than one version context, which might have implications for locking and storing. If you want to lock or store version contexts other than the current set, it is necessary to use the force option discussed in “Chapter 8. Accessing information model objects” on page 33.

Designing a model for a tool

The model you construct determines the capabilities of your tool. When you extend the information model, you can introduce either a totally disjoint model (your tool does not interact with other tools) or a tool that uses other elements of the information model.

Creating a disjoint model

In many cases, it is simpler to develop a model that does not interact with other tools. Your development effort is not subject to changes in the other model; you have complete control. However, you (and your customers) will not derive the benefit of information-sharing between tools.

Using existing models

If you choose to tie your model to other pieces of the information model (refer to the *Information Model Reference, SC34-4554* for details), you need to determine if your extensions can establish the ties through existing relationships, or if your extension will require adding new relationships. Adding new relationships to the information model requires you to add attributes to existing classes, which is known as *schema evolution*.

Advice for new tool builders

If you are just starting to build tools to integrate with TeamConnection, the following pointers might be helpful:

- Start small, with 2 object classes and one relationship. If you can make this work, you have 90% of the integration accomplished.
- Work with the cache implementation first, rather than integrating the storage model immediately. Always set up your test environment to create the objects in the cache, update them in the cache, but stub the store calls that update the persistent store.

  The cache representation can be rebuilt very quickly relative to the server schema, and will allow you to exercise tool semantics to validate your model. Integration with the server should come when your model is reasonably stable, and you require the ability to make your objects persistent.
- Spend a great deal of time architecting the views you use against the model. Views provide mechanisms to enable tool writers to get the right information as one transaction, rather than requiring multiple transactions to achieve a desired state. The cost for multiple views is far outweighed by the cost of multiple transactions.
- Work with the information model constructs, rather than attempting to map the information model objects to an intermediate representation. TeamConnection provides a rich set of operators for handling relationships, objects, and attributes; it largely eliminates the need for tool-defined data structures. Introducing a mapping layer might reduce your dependence on TeamConnection, but will make it more difficult to map your changes back to the information model constructs, and may degrade the performance of your tool.

Using the TBDK Breditor (Browser/editor)

The TeamConnection Toolbuilder’s Development Kit Breditor is a graphical interface for creating, editing, and browsing the class definitions for a tool being integrated with TeamConnection. The capabilities of the TBDK Breditor enable you to do the following:

- Graphically design class definitions and hierarchies
- Graphically view and modify existing classes
- Verify the completeness and accuracy of a defined class subsystem
- Generate CDL, IDL, or C++ files from the defined class definitions

The TBDK Breditor provides integrated help to aid users in all phases of model design.

Loading CDL or IDL (via CLS) into the TBDK Breditor

If you are working on a model that you began developing with CDL or IDL, you can load the existing model definition language files into the TBDK Breditor and continue developing your model there.

To load CDL, simply select **Integrate CDL** from the **Subsystem** menu.

To load IDL into the TBDK Breditor, you must first convert the IDL to a CLS file, which can be integrated into the TBDK Breditor. CLS (and CDL) files are temporary storage files for work in progress. They provide a way for you to save your work for reloading into the TBDK Breditor without generating IDL or C++ code for your model.

The Toolbuilder's Development Kit provides a utility, idl2cls.exe, that you can use to convert existing cache IDL into a CLS file for integration into the TBDK Breditor.

OS/2

The idl2cls.exe utility starts the SOM compiler. Before you attempt this procedure, make sure that you have SOM installed on your system and that the appropriate environment variables are updated with the SOM information (LIBPATH, PATH, and so on).

The IDL-to-CLS converter requires that the class named TCPart be declared by adding an interface statement for it at the beginning of the file. The interface statement should appear as follows:

```
interface TCPart;
```

To convert IDL to a CLS file and integrate it into the TBDK Breditor, follow these steps:

1. Use idl2cls.exe to convert the IDL to a CLS file as follows:

   ```
   idl2cls   sourceFileName
   ```

   Replace *sourceFileName* with the name of your IDL file.
2. Integrate the CLS file into the TBDK Breditor by selecting **Integrate CLS** from the **Subsystem** menu.

Chapter 4. Working with CDL and IDL

You can work with CDL or IDL as generated by the TBDK Breditor, or you can write them yourself. This chapter explains the CDL and IDL generated by the TBDK Breditor. You can use this information to interpret and understand the generated CDL/IDL or to create your own.

CDL is used to define classes that are to be managed within the TeamConnection cache, specifically, in an effort to simplify the process of defining and maintaining tool models.

Currently IDL must be used for server-side modeling and building views. The IDL that the TBDK Breditor generates for a model is a superset of the interface definition language used in the SOM Developer's Toolkit. The Toolbuilder's Development Kit uses this standard IDL to define the classes, attributes, and methods for your server or view model and expands the standard IDL constructs to include entity-relationship concepts specific to the Toolbuilder's Development Kit, such as order, cardinality, direction, and control.

Structure of CDL files

All CDL files consist of the following sections:

- A ***module*** statement, which defines the name of the CDL module
- Zero or more ***class definitions***, which define the flags, attributes, methods, and implementation of the objects in a class.

  Each interface definition consists of a name, zero or more flags, zero or more attributes (including attributes used to hold relationships), and zero or more methods.
- Relationship classes include definitions of target and source attributes.
- Comments in the CDL begin with //.

The format of a statement in a CDL file is: `tag : data`

The following sample cache CDL defines an interface consisting of a subclass of TCPart, a dependent (plumbing) object, and a relationship between them. The CDL is explained in “Description of sample CDL” on page 12. Use the numbers along the left margin of the sample CDL to match the CDL statements to the descriptions in “Description of sample CDL” on page 12.

Sample CDL

1
```
       module DVNT {
```

2
```
           class: Managed
               //
               superClass: TCPart
```

3
```
          flags   : VERSIONED
               attribute   : myValue
                   // Simple integer attribute
                   type: ushort
               attribute   : pSourcePlumbing
```

```
                // rel to the subsystems that are dependent on this subsystem
                targetClass: Plumbing
                targetAttribute: pTargetMgd
                relationshipObject: Plumbing2Mgd
4               cardinality: 1-1
            method   : ManagedInit
                // Initialization function for Managed instances
                type: void
                flags   : INITFUNC
            method   : foo
                // Function foo; behavior for Managed instances
                type: void

5

        class: Plumbing
            //
            superClass: ADObject

6

            flags   : NOEXTENT
            attribute   : adPlumbAttr
                //
                type: string
            attribute   : pTargetMgd
                // rel to the subsystems that this subsystem depends on
                flags   : PRIMARY
                targetClass: Managed
                targetAttribute: pSourcePlumbing
                relationshipObject: Plumbing2Mgd
7               cardinality: 1-1

8

        relClass: Plumbing2Mgd
            //
            superClass: ADLink

9

            flags   : NOEXTENT
            attribute   : source
                //
                flags   : PRIMARY
                targetClass: Plumbing
                targetAttribute: pTargetMgd
                cardinality: 1-1
            attribute   : target
                //
                targetClass: Managed
                targetAttribute: pSourcePlumbing
                cardinality: 1-1
```

Description of sample CDL

1 Module statement for module named DVNT.

3 Managed is a subclass of TCPart.

3 Managed has two attributes: one is a simple integer type, and the second is a relationship type. It provides two methods, one of which is an initialization function. The initialization function is called when the object is constructed; the other method (`foo`) is called by the tool.

4 Cardinality of the relationship (`cardinality`) is indicated in the object classes, in addition to the class used to implement the relationship (`relationshipObject`). The attribute used to maintain the other side of the relationship is provided in the class definition.

5 Plumbing is a subclass of ADObject. The absence of TCPart as a superclass makes it a plumbing object by default.

6 Plumbing has two attributes: one simple string type and one relationship type.

7 Cardinality of the relationship (`cardinality`) is indicated in the object classes, in addition to the class used to implement the relationship (`relationshipObject`). The attribute used to maintain the other side of the relationship is provided in the class definition (`targetAttribute`).

8 Plumbing2Managed is a relationship class, due to its `relationshipClass` identifier (see 10). The fact that Plumbing2Managed is a subclass of ADLink also identifies it as a relationship.

9 Plumbing2Managed has two attributes, source and target, which represent instances of Managed and Plumbing respectively. It has no additional attributes or methods.

The relationship class indicates the cardinality and attribute names in the source and target classes, as well as the semantics associated with the relationship (primary and controlling).

This description is sufficient to generate the TeamConnection interfaces for this model. Some methods not specified in the CDL (such as get and set methods, and a number of cache mechanics methods) are automatically generated.

If you choose to generate cache C++ files, the Breditor will generate a CPP (C++) file, an HPP (C++ header) file, and a CPO file. The CPO code is only applicable to TeamConnection cache services internal machanisms. Modify the cache CPP file to define methods and further refine your tool model. The CDL described earlier generates the following cache CPP file:

```
/***************************************************************************
*
*  IBM TeamConnection Cache Services (TCCS)
*  (C) Copyright IBM Corporation. 1995,1996. All rights reserved.
*
*  IBM Confidential (IBM Confidential-Restricted when Combined
*  with the Aggregated OCO Source Modules for this Program)
*  OCO Source Materials
*
*
***************************************************************************/
#include "DVNT_.hpp"

void _Managed::ManagedInit()
{
}

void _Managed::foo()
{
}
```

Structure of IDL files

All IDL files consist of the following sections:

- A ***module*** statement, which defines the name of the IDL module
- Zero or more ***forward interface declarations***, which declare the names of the objects to be defined by the IDL

- Zero or more ***interface definitions***, which define the attributes, methods, and implementation of the objects

  Each interface definition consists of a name, zero or more attributes (including attributes used to hold relationships), zero or more methods, and an implementation section. The ***implementation section*** defines some additional constructs and semantics not included in the SOM/CORBA specification, such as order, cardinality, direction, and control.
- Comments in the IDL are delimited by /*...*/.

The following sample IDL defines an interface consisting of a TCPart, a dependent (plumbing) object, and a relationship between them. The IDL is explained in “Description of sample IDL” on page 15. Use the numbers along the left margin of the sample IDL to match the IDL statements to the descriptions in “Description of sample IDL” on page 15.

Sample IDL

1

```
        module DVNT {
```

2

```
        interface ADRoleCollection;
        interface Managed;
        interface TCPart;
        interface Plumbing;
        interface ADObject;
        interface Plumbing2Mgd;
        interface ADLink;
```

3

```
        interface Managed : TCPart
```

4

```
        {
          attribute ushort myValue;
          /* Simple integer attribute */


          attribute Plumbing pSourcePlumbing;
          /* rel to the subsystems that are dependent on this subsystem */


          void ManagedInit();
          /* Initialization function for Managed instances */

          void foo();
          /* Function foo; behavior for Managed instances */
```

5

```
          #ifdef __SOMIDL__
            implementation
            {
            adToolInterface = Managed;
            adClassExtent = versioned;
            dllname = fhcDVNT;
            isObjectClass;
            pSourcePlumbing: adInverse, adLink=Plumbing2Mgd, adTarget="Plumbing::pTargetMgd", card=
            ManagedInit: initFunc;
            };
          #endif
        };
```

6

```
interface Plumbing : ADObject
```

7

```
{
  attribute string adPlumbAttr;

  attribute Managed pTargetMgd;
  /* rel to the subsystems that this subsystem depends on */
```

8

```
  #ifdef __SOMIDL__
    implementation
    {
    adToolInterface = Plumbing;
    adClassExtent = none;
    dllname = fhcDVNT;
    isObjectClass;
    pTargetMgd: adPrimary, adLink=Plumbing2Mgd, adTarget="Managed::pSourcePlumbing", car
    };
  #endif
};
```

9

```
interface Plumbing2Mgd : ADLink
```

1011

```
{
  #ifdef __SOMIDL__
    implementation
    {
    adToolInterface = Plumbing2Mgd;
    adClassExtent = none;
    dllname = fhcDVNT;
    adSource="Plumbing::pTargetMgd";
    adTarget="Managed::pSourcePlumbing";
    };
  #endif
};
```

Description of sample IDL

1 Module statement for module named DVNT.

2 Forward declarations of classes ADRoleCollection, Managed, TCPart, Plumbing, ADObject, Plumbing2Managed, and ADLink.

3 Managed is a subclass of TCPart.

4 Managed has two attributes: one is a simple integer type, and the second is a relationship type. It provides two methods, one of which is an initialization function identified in the SOMIDL implementation section (see 5). The initialization function is called when the object is constructed; the other method (`foo`) is called by the tool.

5 Cardinality of the relationship (card=) is indicated in the object classes, in addition to the class used to implement the relationship (relObj=). The attribute used to maintain the other side of the relationship is provided in the class definition.

6 Plumbing is a subclass of ADObject. The absence of TCPart as a superclass in the interface statement makes it a plumbing object by default.

7 Plumbing has two attributes: one simple string type and one relationship type.

8 Cardinality of the relationship (`card=`) is indicated in the object classes, in addition to the class used to implement the relationship (`relObj=`). The attribute used to maintain the other side of the relationship is provided in the class definition (`adTarget=`).

9 Plumbing2Managed is a relationship class because it is identified as a subclass of ADLink.

10 Plumbing2Managed has two attributes, source and target, which represent instances of Managed and Plumbing respectively. It has no additional attributes or methods.

The relationship class indicates the cardinality and attribute names in the source and target classes, as well as the semantics associated with the relationship (primary and controlling).

This description is sufficient to generate the TeamConnection interfaces for this model. Some methods not specified in the IDL (such as get and set methods) are automatically generated.

See "Chapter 8. Accessing information model objects" on page 33 for examples of code generated from IDL similar to the sample included in this section.

Note: When you generate C++ files from IDL using the Breditor, you should not attempt to edit server CPP file, which contains generated methods used by TeamConnection repository services.

Options available in the TBDK Breditor

The following sections describe the options you can specify for attributes, methods, and relationships when using the TBDK Breditor.

See the *Information Model Reference* for listing of all attributes available for modeling when using the Toolbuilder's Development Kit.

Attribute data types

The following list shows supported data types for attributes. It includes the keyword to use for assigning a data type to an attribute. The description of each data type is followed by the cache implementation in parentheses.

<sequence>bindata
Binary data, not interpreted by the tool (DSBulkData)

boolean
True or false (BOOL)

char A single character (char)

char* A NULL-terminated character string (char *)

string<n>
A NULL-terminated character string with a maximum length of n, not including the NULL terminator (char *)

string A NULL-terminated character string (char *)

double
A double-precision floating-point number (double)

float A single-precision floating-point number (float)

long A four-byte signed integer (INT4)

short A two-byte signed integer (INT2)

ulong A four-byte unsigned integer (uINT4)

ushort
A two-byte unsigned integer (uINT2)

Attribute options

In addition to assigning a data type to an attribute, you can use the following options to indicate how attributes and their get and set methods are generated.

noData
The attribute's value is calculated rather than set. No attribute is generated; default get and set methods are generated in the CPP file. The methods must be overridden with actual implementations supplied by a tool.

static A class variable whose value is shared among all instances of the class. The attribute is not inherited by subclasses of the current class.

readonly
The attribute cannot be altered by a method. No set method is generated for read-only attributes. Read-only attributes can be used only within the scope of a method of their class.

index Support for copy key, ordered, and nonduplicated indexing is provided.

Relationship options

You can use the following options to indicate how relationships are generated.

attrib The name of the attribute in the source class that manages the relationship with the target object.

inverse
The name of the attribute in the target class that manages the relationship with the source object.

ordered
If you specify this option, the order of relationships in the collection is maintained. Constructors and move methods are provided to control the order of objects in the relationship collection.

allowsDups
Specify this option to allow duplicates of this relationship. A duplicate relationship is one that can exist more than once between a given pair of source and target objects. Without this option, an attempt to create a relationship between the same source and target causes an exception. Relationships that have attributes or are the source or target of another relationship are likely to allow duplicates. The default is not to allow duplicates.

card Specifies the cardinality of the relationship. Allowed values are 1-1, 1-N, N-1, and M-N.

controlling
Specify this option to give the relationship control over either its source or target. A relationship is controlling if deleted when either of the following are true:

- The number of instances of the relationship for the source or target class is zero.
- The number of instances of the relationship for the source or target class is less than the *minimum cardinality*, which is expressed in the IDL implementation section as follows (for example):

  ```
  myRel: adMin=3, adMax=5;
  ```

To keep the source or target object independent of the relationship, do not select this option. Controlling semantics are ineffective with (ignored for) TCPart and its subclasses; no error is indicated.

primary
Select this option to define this class as the primary direction of the relationship, which makes the object being modeled the source of the relationship.

relObj Specify the class name of the relationship (the subclass of ADLink that defines the relationship class). Construct a relationship class name by combining the source class name, relationship verb, and target class name as follows: SourceClass_verb_TargetClass (Class1_includes_Class2, for example). When you name your classes, use the following scheme:

```
prefix + className
```

For the prefix, use two or three letters that identify your tool.

isRelationship
Indicates this class is a relationship; generates additional constructors supporting relationship semantics.

Method options

You can use the following options to indicate how methods are generated.

termFunc
Selecting this option generates a method in the CPP file that is called from the destructor. The termFunc option corresponds to the Finalize method.

initFunc
Selecting this option generates a function that is called from the constructor. The initFunc option corresponds to the Initialize method.

private
Selecting this option generates the method as a private method. Private methods can be used only by the class that defines them.

protected
Selecting this option generates the method as a protected method. Protected methods can be used by the class that defines them or by its subclasses.

public Selecting this option generates the method as a public method. Public methods can be used by any class. This is the default option for methods.

virtual Selecting this option generates the method as a virtual method. Virtual methods are inherited from a superclass of the current class.

static Selecting this option generates the method as a static method. Static methods are not inherited from a superclass of the current class.

What happens during code generation

After the CDL or IDL for a tool is complete, you will generate code from it. Several features of this generated code are worth noting:

- For each class defined in the CDL or IDL, two classes are generated. The extra class is a handle class, which the cache uses to provide access to information model objects. See “Chapter 7. Object handles” on page 29 for additional information about handle classes.

  For the sample CDL and IDL shown previously, the following classes are generated. The first three are the handle classes and the last three are the object classes.

  Managed

  Plumbing

  Plumbing2Managed

  _Managed

  _Plumbing

  _Plumbing2Managed
- Attributes defined in the CDL or IDL are always generated as private attributes, while the get and set methods are always generated as public methods.
- Relationship attributes have several different access methods:
  – The class defined as the source attribute for a relationship implements three access methods (potentially):
    - _get_***targetAttribute***() [allows the tool to iterate over the collection of objects that is returned]
    - _getFirst_***targetAttribute***() [returns a single object]
    - _getNext_***targetAttribute***() [returns a single object]

  – The class defined as the target attribute for a relationship implements three access methods (potentially):
    - _get_***sourceAttribute***() [allows the tool to iterate over the collection of objects that is returned]
    - _getFirst_***sourceAttribute***() [returns a single object]
    - _getNext_***sourceAttribute***() [returns a single object]

    In these methods, ***targetAttribute*** and ***sourceAttribute*** are the attribute names defined in the attribute option of the source and target definitions the relationship interface.

    For the sample CDL and IDL, Managed (which is defined as the source for the relationship Plumbing2Managed) implements the following:
    - _get_pSourcePlumbing()
    - _getFirst_pSourcePlumbing()
    - _getNext_pSourcePlumbing()

    Plumbing (defined as the target for the relationship) implements the following:
    - _get_pTargetMgd()
    - _getFirst_pTargetMgd()
    - _getNext_pTargetMgd()

Integrating your model

See “Chapter 6. Build processes for the Toolbuilder’s Development Kit” on page 25 for detailed information regarding the process for integrating your new classes into TeamConnection.

Chapter 5. Working with views

Views provide the ability to aggregate objects into larger logical units, which makes them the essential feature of interest for tools attempting to provide complex (and, often, resource intensive) interactions. The unit of tool requests, called a *view type*, is rooted on a TCPart, but may identify a subset of attributes, relationships, and plumbing objects (including their attributes and relationships), along with other TCParts, to be included in a single tool request.

Views are required for all operations that transfer data between the cache and the server. Each tool request processed by the server represents a single transaction to the database, in which all or none of the request is committed to the persistent store.

Determining the appropriate scope of views is crucial to optimizing the performance of these operations. (See "Chapter 10. Performance and scalability issues" on page 63 for more details on this issue.)

View types

A view type is an interface that identifies a connected set of objects and relationships based on a root object. A set of instances identified in this way can be treated as a unit for several operations, such as locking and access control.

This section describes how you would build a view type by identifying the components of the view type. For a more formal definition of a view type, see "Adding a view type to a repository schema" on page 22.

Building a view type

Begin with a TCPart and select the attributes that you would like to be part of your view type. Then, for each class connected to this class through a relationship (links and targets), you repeat this process.

You can select some, none, or all of the connected classes. For each of the connected classes you can select classes that are connected to the connected class.

If the connected class is also the root of a view type, you have the choice of connecting to it as a view type or as a class. Connecting to it as a view type makes the definition recursive, while connecting to it as a class stops the definition at a fixed number of entries.

You can repeat this process to any level, stopping when you decide to stop selecting connected classes. The process is inherently recursive.

Defining a view type

The following is the simplified IDL for some sample classes that might be part of a tool model:

```
interface A : TCPart {
  attribute long a1;
  attribute BRel a2;
}

interface B : TCPart {
  attribute long b1;
  attribute ARel b2;
  attribute CRel b3;
  attribute DRel b4;
}

interface C : TCPart {
  attribute long c1;
  attribute BRel c2;
}

interface D : TCPart {
  attribute long d1;
  attribute BRel d2;
}
```

Figure 2 is a view type diagram for a view derived from the sample classes above.

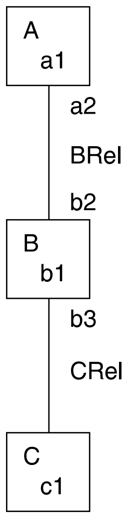

Figure 2. Sample view type diagram

Here is a brief description of how the view type shown in Figure 2 is constructed:

1. Begin with A, and include a1 and a2.
2. By including a2, you can navigate to B, via Brel.
3. B includes b1 and b3. (Note that b2 and b4 are not included in this case.)
4. By including b3, you can navigate to C, via Crel.
5. Finally, C includes c1, but not c2.

Adding a view type to a repository schema

To add a view type to the repository schema, begin by defining an IDL class for it. The view type that is described in the previous section is defined by the following IDL:

```
interface UA : ADView {
  implementation {
    adViewDefn = "(A: a1, a2->(B: b1,\
      b3->(C: c1 as xyz)))"
  }
}
```

The value of the adViewDefn modifier describes the inner structure of the view type, which addresses the classes and relationships that make up the view type. The syntax of this value (an elemList) is described in the example that follows.

```
elemList = "(" qualifier ":" elem {"," elem} ")"

elem = attribExpr | relExpr

attribExpr = attribName ["as" attribViewName]

relExpr = attribName "->"
                (elemList | viewTypeName)
```

The terminal values used in the preceding syntax example are defined as follows:

qualifier
A valid class name (for example, *Managed*). The class can be a subclass of ADLink, TCPart, or ADObject (implicitly). Before a definition view type can be loaded into the repository, all of the qualifiers that are in the definition must already be present in the repository.

The very first qualifier in the adViewDefn must be a managed class.

attribName
An attribute name (for example, empName). This attribute name must be defined in the qualifier class definition or in one of qualifier's parent definitions.

attribViewName
This optional terminal allows the view type to override the name of the attribute for this occurrence of the attribute. This terminal follows the same naming restrictions as any other attribute name. If this terminal is not specified, the attribViewName is assumed to be the same as the attribName.

For example, assume you have an object, Person, with a Name attribute and the relationship PersonMarriedToPerson. Suppose your program needs to display a person's name and that person's spouse's name. The root object's Name attribute could omit this terminal, and the target object's Name attribute could include it with a name of spouseName.

viewTypeName
The name of a view type. A view type can be extended by connecting to either a class or a view type. When connecting to a view type the definition is recursive. When connecting to a class the definition is explicit and fixed.

A collection of view type names is called a *viewTypeList*.

View instances

A view instance consists of the connected object instances that make up a view type. Every view instance begins with a TCPart.

Not every class in the view type is necessarily populated in every view instance of that view type. In addition, some classes can be represented multiple times in a view instance, depending on the cardinality of the relationships involved.

For example, assume that BRel from the earlier example has a 1-M cardinality. The resulting view instance would include many instances of B, each with its own name, if B inherits from a TCPart.

Chapter 6. Build processes for the Toolbuilder's Development Kit

In order to integrate your class objects into TeamConnection, you must have the appropriate software, as described in the TeamConnection Version 2.0 Release for Announcement (dated 11/26/96). It is also necessary to create a database that you are able to communicate with, so that you can run the TBDK Breditor.

Open the Breditor and define the classes that will comprise your subsystem. Refer to "Chapter 3. Creating a model for a tool" on page 7 and the Breditor help for information on using the Breditor to load existing CLS or CDL and defining classes. After you have finished defining your classes, save your subsystem into TeamConnection, so that you will not lose it. Then, follow the steps in "Cache DLL build" or "Server DLL build" on page 26 and "View DLL build" on page 26, depending on whether you want to build cache or server/view DLLs.

After you complete the build processes that follow, you will be able to create tools that use instances of your new classes and to store into and retrieve from TeamConnection. When you link your tool code, be sure to include fhcrscli.lib and fhccmnc.lib in the library specification portion of your link statement.

Cache DLL build

The following steps describe how to build cache DLLs.

1. Select the **Cache C++** option from the Breditor **Generate** menu. Specify the subsystem that you want to generate.

   The Breditor will generate a CPP file, a CPO file, and an HPP file based on your subsystem definition. The names of the generated files will match your subsystem name. These files will be placed in the Breditor's working subdirectory.
2. Copy these files into the subdirectory where you want to perform the build. In addition, copy the generated HPP file into a subdirectory in your INCLUDE path.
3. Copy the sample makefile, cachebld.mak, from the TC_TBDK\TBDK\SAMPLES subdirectory.

   Note: All examples in this file assume that the directory where you installed Toolbuilder's Development Kit is TC_TBDK.

   Modify the cachebld.mak file according to the instructions in the file. If you modify the makefile name, be sure the change the OCTO_MAKEFILE variable in the file. You might find it helpful to rename the makefile to the name of your subsystem with a .MAK extension.
4. Invoke the make process by entering `nmake -f cachebld.mak` from the command line. This will create a DLL file called *ssname*.dll, where *ssname* is the name of your subsystem. Copy this DLL file into a subdirectory in your LIBPATH.
5. The newly-created DLL file and associated LIB file should be placed in the LIBPATH of all clients that will be running any tools that use the new subsystem. Any machine that will be doing development work using these new classes will also need the generated HPP file in its INCLUDE path.

Server DLL build

The following steps describe how to build server DLLs.

1. Select the **Server C++** option from the Breditor **Generate** menu. Specify the subsystem that you want to generate.

   The Breditor will generate a CPP file, an HPP file, and an LCP file based on your subsystem definition. The names of these files correspond to the DLL name of your subsystem. The files will be placed in the Breditor's working subdirectory.

   Note: The DLL name of your subsystem can be found on the **General** page of any of its class definitions.
2. Copy these files into the subdirectory where you want to perform the build.
3. Copy the sample makefile, srvrbld.mak, from the TC_TBDK\TBDK\SAMPLES subdirectory.

   Note: All examples in this file assume that the directory where you installed Toolbuilder's Development Kit is TC_TBDK.

   Modify the srvrbld.mak file according to the instructions in the file. If you modify the makefile name, be sure the change the OCTO_MAKEFILE variable in the file. You might find it helpful to rename the makefile to the name of your subsystem with a .MAK extension.
4. Invoke the make by entering `nmake -f srvrbld.mak` from the command line. This will create two DLL files, *dllname*.dll and fhcschem.dll, where *dllname* is the DLL name of your subsystem, and an ADB file, schema.adb.
5. Copy the schema.adb file to the TEAMC\BIN subdirectory or anywhere else in the PATH, provided that it will be the first schema.adb found. Copy the two DLL files to a subdirectory in your LIBPATH.

View DLL build

After completing the steps outlined in "Server DLL build", remain in the same subdirectory where you did the server build and verify or enact the following conditions:

- The TC_DBPATH environment variable is set to the subdirectory where you want to create the new database (the same path that you copied the schema.adb file to in step 5 under "Server DLL build").
- The TCP/IP libraries are in your LIBPATH.
- The Object Store cache manager and server are started.

Proceed with the following steps to build a view DLL:

1. Copy the sample makefile, viewbld.mak, from the TC_TBDK\TBDK\SAMPLES subdirectory.

   Note: All examples in this file assume that the directory where you installed Toolbuilder's Development Kit is TC_TBDK.

   Do not rename or modify this file.

Copy the bldviews.cmd file from TC_TBDK\TBDK\SAMPLES if you are running on OS/2.

Copy the bldviews.bat file from TC_TBDK\TBDK\SAMPLES if you are running on Windows/NT.

2. Create or copy any view IDL desired for your new subsystem into your build subdirectory. Be sure that no other IDL is in this subdirectory; however, you can have multiple view IDL files for your subsystem.

   All view IDL names should begin with *view*. For example, a view for subsystem X could be named *viewx1.idl*.
3. Enter `emitdllv view*.idl` from the command line. This will produce several intermediate files. Do not modify these files.
4. Enter `bldviews` from the command line. This will produce two DLL files, dllview.dll and dllviews.dll.
5. Copy the dllviews.dll file to a subdirectory in the LIBPATH on the server machine. Copy the dllview.dll file to a subdirectory in the LIBPATH of any client machine that will use the new database created by this process.
6. Create the new database. The database will now be aware of your new classes and allow you to persistently retrieve and store instances of your classes.

Chapter 7. Object handles

Handles are the mechanism that the TeamConnection cache uses to provide access to underlying model objects.

Several requirements are satisfied through the use of handles:

- An object can be referenced and used by several tool processes. Deletion of an object accessed directly (not through a handle) can result either in traps, which happen when another tool process accesses an object that is no longer present, or in the need for complicated code by the tool to prevent such situations.
- Objects that can no longer be reached through navigation can be freed, increasing the amount of resources available for tools.
- Object identity and uniqueness are provided by using handles to search for existing objects within the cache, without requiring each tool to reimplement this function.

Handle classes

Handle classes are generated from the model specification; each model class automatically has a handle class. Handles act as pointers to a model object, and provide access to both attributes and methods that are implemented on the model. Transparent use of the handle class is provided through three operators:

- operator[]
- operator*
- operator->

These operators support pointer notation through the instance, without actually constituting a pointer. Each of these operators is overridden in the handle class to provide a type-safe return type for the kind of model object the handle refers to.

Note: Type-safe return types are provided only for compilers that support this convention.

Handles are very small, making them efficient to embed as needed.

Allocating a handle

The following examples show how of allocate handles in various contexts:

```
// Allocate on the stack; freed when leaving scope
TCPart handle(ADObject::defaultVersion());

// Allocate on the heap; freed using delete
TCPart *pHandle=new TCPart(ADObject::defaultVersion());

// Using createNewHandle(); not sure what kind of model object it is.
ADObject *pHandle=someHandle->createNewHandle();

// Using the copy constructor on the stack
TCPart copyHandle((TCPart &)oldHandle);

// Using the assignment operator
TCPart assignHandle=otherHandle;
```

Handle management

Allocating (or assigning) a handle for a model object has the effect of increasing a use count associated with that object. The use count is used by TeamConnection to determine when an object or network of objects can be removed from the cache. Constructing a new handle to an object is said to *pin* that object in the cache, which implies that the object is in use. All handles that are constructed or allocated must be deleted in order for the related object to be deleted.

Relationships use handles as well, so the use count for an object [calculated by the method _ADObject::_get_useCount()] may seem higher than expected. However, when a network's total use count is represented by the number of handles used to manage the relationship (that is, when there are no external handles to the network), the entire network can be removed from the cache.

The TeamConnection application programming interface can construct handles that eventually must be freed by the caller. Examples include both _TCPart: :list(), and _TCPart::retrieveByName(). These methods return pointers to a DSModelCollection, which contains the handles. To free these handles, either remove them from the collection, or simply delete the collection.

Allocating a handle to a DSModelCollection increases the use count for an object; freeing a handle from the collection decreases the use count. Iterating across the collection [using the first() and next() methods] does not change the use count.

Handle pointers returned by an interface

Some interfaces [such as the navigation interfaces, which return the other side of a relationship, or the DSModelCollection::first() and next() methods], will return a pointer to a handle. These handles must not be freed. Instead, you can reference them while holding a handle to an object in the network, or you can construct a new handle based on the handle returned by the interface. Unless the interface indicates that the handle must be freed by the caller, do not delete handle pointers returned by the interface.

Handles as pointers

There are considerations related to using or copying handle pointers. If you are using handle pointers instead of handles, you have, syntactically, constructed a pointer to a pointer. This must be reflected in the way you use the pointer.

For instance:

```
TCPart handle1(ADObject::defaultVersion());
handle1->_get_adName();               // Handles act like pointers

TCPart *pHandle=new TCPart(ADObject::defaultVersion());
(*pHandle)->_get_adName();            // Pointer to a pointer syntax
```

The latter type of usage can leave you with an invalid pointer if the original handle has been freed. Therefore, it is better to construct a new handle as necessary, rather than trust that the handle supplied to you will continue to exist.

Effect of object deletion on handles

The use of _ADObject::markDeleted() also affects the handles in a network. When markDeleting an object, the object is severed from the network. All relationship handles are removed, which means that navigation to other objects is no longer possible, but all externally-held (tool) handles are preserved.

Access to the object through these handles is still safe, although tools should check to ensure the object’s state makes sense in the context of the tools operation.

Chapter 8. Accessing information model objects

The Toolbuilder's Development Kit provides methods for querying, creating, storing, and deleting objects in a repository schema. This chapter explains how to use these methods to access objects in your repository schema. Both C++ and Smalltalk interfaces are described.

Once a view has been constructed in the cache (or if you wish to continue working with a view you've stored in the repository), the methods described in this section become more pertinent.

These interfaces are defined as methods on TCPart. Many of the methods are virtual methods, allowing tools to override them to augment or replace the actions they represent (for instance, to perform client-side validation before storing an object). Many of the methods—list, retrieveObjects, retrieveByName, and retrieveObjectsByName—are designed to retrieve objects that do not necessarily exist in the cache, and so are provided as static methods.

Each of the following sections describes a method, shows the syntax for the method, and lists the arguments you pass with the method and their meaning.

For methods and messages related to exception handling, see "Chapter 9. Exception handling" on page 53.

Creating objects

Objects are created through the Handle class definition; for our example IDL file (refer to "Sample IDL" on page 14), these interfaces are provided:

```
Managed(const char *vstr, const char* id=defaultObjID(),
                              int bCreate=True);
```

The version string and id uniquely identify the object, but can be assigned values when the object is stored. The third parameter is used to control construction of the underlying model object, and should not be specified by tools.

Using generated constructors

The following examples are provided to illustrate the use of generated constructors.

```
In C++:

// Allocates a handle, but no underlying model object:
Managed handle;

//  Allocates a handle and a model object, using defaults for version and name:
Managed pObj1(DSModelObject::defaultVersion(), DSModelObject::defaultObjID());

//  Allocates a handle and a model object, using defaults for version:
Managed pObj2(DSModelObject::defaultVersion())

//  Allocates a handle and a model object from the heap; the handle
//  should eventually be freed (using delete); the cache will manage
//  the deletion of the model object when there are no further
//  references to it.
Managed *pHeapObj=new Managed(DSModelObject::defaultVersion())
```

```
//  Construct a plumbing object using defaults:
Plumbing pDependentObj(DSModelObject::defaultVersion());

//  Construct a relationship; no need to save the handle if there are no
//  attributes that need to be set. When the relationship handle goes out of
//  scope, the relationship object will continue to exist, held in place by the
//  model objects for pHeapObj and pPlumbing.  If pHeapObj had been
//  allocated from the stack, the entire network would be released upon
//  leaving scope.
Plumbing2Managed( pHeapObj, &pDependentObj);

In Smalltalk:

"Allocates a handle, but no underlying model object:"
anADObjectHandle := Managed new newHandle.

"Allocates a handle and a model object, using defaults for name and version:"
anADObjectHandle := ManagedHandle version:ADObject defaultVersion id:ADUserObjectName
                              defaultObjectId.

"Allocates a handle and a model object, using defaults for version:"
anADObjectHandle := ManagedHandle version:ADObject defaultVersion.

"Construct a plumbing object using defaults:"
aDependentObjectHandle := PlumbingHandle version:ADObject defaultVersion

"Construct a relationship:"
anADLinkHandle := Plumbing2ManagedHandle source:anADObjectHandle
                              target:aDependentObject.
```

Changing an object’s instance data

For every attribute, get and set methods are automatically generated. For the myValue attribute defined in Managed, the following methods are provided:

```
In C++:

uINT2 _get_myValue(void);
void _set_myValue(uINT2 value, omSetProcessing setFunc=SET_CHANGE_FLAG);

In Smalltalk:

myValue
myValue:anInteger
myValue:anInteger processMask:aMask
```

These methods allow you to access the values of the attributes. If the attribute had been specified with the ’readonly’ modifier, or if the attribute had represented a relationship, there would be no generated set method.

Similarly, if the attribute had a ’nodata’ modifier, the interfaces would have been generated, but they would have been generated with a default implementation in the CPP file and Smalltalk application, and would have to be tailored by the programmer.

The setFunc argument to the C++ set methods is used to indicate a change in the value of the attribute. The default set methods set the change flag. If the set methods do not supply this argument, the object does not know that it has changed, much as if you had modified the instance variable directly. At some point before the object is stored, however, the object must be marked as changed in order to be included in the data stream sent to the server.

The possible values of setFunc are as follows:

SET_NONE
: Sets the value, but does not affect the change flag.

SET_CHANGE_FLAG
: Sets the value, and sets the change flag to TRUE.

SET_IGNOREIFCHANGED
: Examines the change flag before setting values. One of the following actions occurs:

  - If the change flag value is TRUE, the function returns immediately and nothing happens.
  - If the change flag value is FALSE, the function behaves as it would for SET_CHANGE_FLAG.

SET_RESET
: Sets the value, and clears the change flag.

In C++, these access methods can be exercised through the handle redirection operators [operator *(), operator[](), and operator->()] as though the handle were a pointer:

```
uINT2 temporaryValue;
Managed pObj(DSModelObject::defaultVersionID());
temporaryValue=pObj->_get_myValue();
pObj->_set_myValue(2);
```

In Smalltalk, the messages can be sent to the handle object, or sent directly to the underlying model object. If sent to the handle object, they are rerouted to the underlying model object using *doesNotUnderstand*.

```
oldValue := anADObjectHandle myValue.
oldValue := anADObjectHandle adObject myValue.  "same result as preceding statement"
newValue := oldValue + 100.
anADObjectHandle myValue:newValue.
anADObjectHandle adObject myValue:newValue.  "same effect as preceding statement"
```

In C++, if you are working with a pointer to a handle (for instance, as returned by new()), the handle must be dereferenced before applying redirection operators:

```
uINT2 temporaryValue;
Managed *pObj=new Managed(DSModelObject::defaultVersionID());
temporaryValue=(*pObj)->_get_myValue();
(*pObj)->_set_myValue(2);
```

Preparing objects for storage (the fixcrlf method)

The fixcrlf method makes adjustments to an object’s bulk data to prepare it for storage on the file system. The default, CRLF_DEFAULT, adds carriage returns, line feeds, Control-Z characters, and converts tab characters to spaces, as needed for bulk data storage.

Return Type
: integer

Parameters

Name	<Direction>Type
flags	<Input > integer

Default
: CRLF_DEFAULT

Valid Values

- CRLF_DEFAULT (CRLF_ADDCR | CRLF_CTRLZ | CRLF_EXPANDTAB for Intel, none of the above for UNIX)
- CRLF_ADDCR (adds carriage returns)
- CRLF_CTRLZ (adds end-of-file characters)
- CRLF_ADDTAB (replaces spaces with tabs)
- CRLF_EXPANDTAB (replaces tabs with spaces)

Method syntax

```
In C++:

int _TCPart::fixcrlf(int flags=CRLF_DEFAULT);
```

Listing objects

There are two methods for listing objects: list and listCache.

For the list method, objects are partially represented in the cache for purposes of the query, although they currently reside as complete objects on the server. The listCache method addresses cache objects only.

list

The list interface provides the ability to bring partial information about a set of objects into the cache. The only information retrieved is the name of the object(s), the key information used by subsequent retrievals, and the time stamp. Although the complete root object is instantiated in the cache for the this method, only these attributes are set; other objects represented by the view type are not instantiated.

Handles to the objects found on the server are returned in the DSModelCollection in C++ (ADModelCollection in Smalltalk). In C++, tools are responsible for freeing the collection when they are done with it.

Method syntax

```
In C++:

static _TCPart::list(char* viewList, char* wildName, char* strVersion);

In Smalltalk:

(TCPart class method) list:aviewListString
                        wildName:aWildNameString version:aVersionString
```

Arguments

char* viewList
: A representation of a collection of view types, used to search for TCParts as several different views. When combined with a wildcard name, use of the correct viewList effectively becomes several searches within one transaction. The form of a viewList is a string of view type names separated by spaces.

char *wildName
: A character representation of an object name, potentially with embedded wildcard characters. TeamConnection supports '_' as matching one character, and '%' as matching zero or more characters.

char *strVersion

Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

listCache

The listCache method returns a collection of handles to cache objects satisfying search criteria defined by a query.

Method syntax

```
In C++:

static _TCPart::listCache(char* viewList, char* wildName,
                          char* strVersion);


In Smalltalk:  not yet implemented
```

Arguments

char* viewList

A representation of a collection of view types, used to search for TCParts as several different views. When combined with a wildcard name, use of the correct viewList effectively becomes several searches within one transaction. The form of a viewList is a string of view type names separated by spaces.

char *wildName

A character representation of an object name, potentially with embedded wildcard characters. TeamConnection supports '_' as matching one character, and '%' as matching zero or more characters.

char *strVersion

Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

Retrieving objects

The repository provides four methods for retrieving objects: retrieve, retrieveByName, retrieveObjects, retrieveObjectsByName.

retrieve

The retrieve interface instantiates the view type in the cache. The retrieve method requires key information (such as that provided by the list interface, or a prior retrieval) in order to function. All attributes and objects included by the view definition view type are instantiated in the cache.

No handle is returned from the interface, as the tool already has a handle to the object being retrieved.

Method syntax

```
In C++:

virtual int retrieve(char* viewType, int lock,
                        int refreshFlag, int forceOption, char* strVersion);

In Smalltalk:

(TCPart instance method) retrieve:aViewTypeString
                        lockOption:aLockOptionInteger
                        refresh:aRefreshOptionInteger
                        aForce:forceIt version:aVersionString
```

Arguments

char* viewType
The name of the view that is used to retrieve the instance data. For a given retrieve request, only a single view type can be specified, although an object can be retrieved as any number of view types, as long as the view types have the same root entity.

int lock
Instructs TeamConnection cache services that objects should be locked upon retrieval:

lock==1
All NamedObjects in the view are locked.

lock==0
No additional locks are acquired.

int refreshFlag
Instructions to TeamConnection cache services used to determine the response if the storage representation of an object differs from the cache representation.

lock==REFRESH_OVERWRITE
Indicates that the storage representation will replace the cache representation.

lock==REFRESH_NONE
Indicates that the retrieval will fail if the cache object has changed since retrieval from the repository.

lock==REFRESH_PRESERVE
: Allows the retrieve to succeed if the time stamps on the cache and server do not match, but will preserve changes made in the cache.

int forceOption
: An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

The force option is important only if LOCK is specified. If you want to retrieve or store a locked part in a particular release or workarea that is linked to another release or workarea, you might want to specify the force option when you are checking in or checking out the part, even if someone else might have the part checked out in another context.

forceOption==1
: Indicates that links should be broken.

forceOption==0
: Indicates that links should be preserved.

Note: If 0 is supplied, and the part is linked with versions not specified in the version string, the action will fail.

char *strVersion
: Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

retrieveByName

The retrieveByName method allows tools to retrieve a view instance in the cache without a preceeding list operation. The name of the object (possibly with wildcards) is used to resolve the object or objects on the server that are represented by the indicated view types in the viewList.

Handles to the view roots matching the qualifications (wildName, viewList) that are found on the server are returned in the DSModelCollection. Tools are responsible for freeing the collection (which, in turn, frees the handles) when they are done with it.

Method syntax

```
In C++:

static DSModelCollection * _TCPart::retrieveByName(
                           char* viewList, char* wildName,
                          int lock, int refreshFlag,
                          int forceOption, char* strVersion);
```

```
In Smalltalk:

(TCPart class method) retrieveByName:aviewListString
                          wildName:aWildNameString lockOption:aLockOptionInteger
                          refresh:aRefreshOptionInteger force:aForceOptionInteger
                          version:aVersionString
```

Arguments

char* viewList
A representation of a collection of view types, used to search for TCParts as several different views. When combined with a wildcard name, use of the correct viewList effectively becomes several searches within one transaction. The form of a viewList is a string of view type names separated by spaces.

char *wildName
A character representation of an object name, potentially with embedded wildcard characters. TeamConnection supports '_' as matching one character, and '%' as matching zero or more characters.

int lock
Instructs TeamConnection cache services that objects should be locked upon retrieval:

lock==1
All NamedObjects in the view are locked.

lock==0
No additional locks are acquired.

int refreshFlag
Instructions to TeamConnection cache services used to determine the response if the storage representation of an object differs from the cache representation.

lock==REFRESH_OVERWRITE
Indicates that the storage representation will replace the cache representation.

lock==REFRESH_NONE
Indicates that the retrieval will fail if the cache object has changed since retrieval from the repository.

lock==REFRESH_PRESERVE
Allows the retrieve to succeed if the time stamps on the cache and server do not match, but will preserve changes made in the cache.

int forceOption
An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

The force option is important only if LOCK is specified. If you want to retrieve or store a locked part in a particular release or workarea that is linked to another release or workarea, you might want to specify the force option when you are checking in or checking out the part, even if someone else might have the part checked out in another context.

forceOption==1
Indicates that links should be broken.

forceOption==0
Indicates that links should be preserved.

Note: If 0 is supplied, and the part is linked with versions not specified in the version string, the action will fail.

char *strVersion
Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

retrieveObjects

The retrieveObjects method allows tools to instantiate the view types specified for multiple objects. This method is an extension of the retrieve method; it allows tools to specify a collection of object-viewtype pairs to be retrieved, whereas retrieve only allows for specification of a single object and view type. All attributes and objects included by the view types are instantiated in the cache.

Handles to the view roots matching the qualifications that are found on the server are returned in the DSModelCollection. Tools are responsible for freeing the collection (which, in turn, frees the handles) when they are done with it.

Method syntax

```
In C++:

static DSModelCollection * _TCPart::retrieveByName(
                          ObjectViewList * list,
                          int lock, int refreshFlag,
                          int forceOption, char* strVersion);

In Smalltalk:  not yet implemented
```

Arguments

ObjectViewList * list
ObjectViewList is a class that contains a collection of handle/view pairs used to specify the objects to be retrieved. The ’list’ portion of the argument is a pointer to an instance of ObjectViewList.

ObjectViewList provides some simple interfaces and a cursor class to aid in its creation and maintenance, as follows:

```
class ObjectViewList {
        ObjectViewList();

        ObjectViewList &add( ADObject *pObj, char *view );
        ObjectViewList &remove( ADObject *pObj, char *view );
        int             isMember( ADObject *pObj, char *view );
        int             Count();
};

class ObjectView {
        ObjectView(ADObject *pObj, char *view);

        char     *View();
        ADObject *Object();
};

class ObjectViewList_Cursor {
        ObjectViewList_Cursor(ObjectViewList *ovl);
        ObjectView *first();
        ObjectView *next();
        ObjectView *prev();
};
```

int lock

Instructs TeamConnection cache services that objects should be locked upon retrieval:

lock==1
: All NamedObjects in the view are locked.

lock==0
: No additional locks are acquired.

int refreshFlag

Instructions to TeamConnection cache services used to determine the response if the storage representation of an object differs from the cache representation.

lock==REFRESH_OVERWRITE
: Indicates that the storage representation will replace the cache representation.

lock==REFRESH_NONE
: Indicates that the retrieval will fail if the cache object has changed since retrieval from the repository.

lock==REFRESH_PRESERVE
: Allows the retrieve to succeed if the time stamps on the cache and server do not match, but will preserve changes made in the cache.

int forceOption

An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

The force option is important only if LOCK is specified. If you want to retrieve or store a locked part in a particular release or workarea that is linked to another release or workarea, you might want to specify the force option when you are checking in or checking out the part, even if someone else might have the part checked out in another context.

forceOption==1
: Indicates that links should be broken.

forceOption==0
: Indicates that links should be preserved.

Note: If 0 is supplied, and the part is linked with versions not specified in the version string, the action will fail.

char *strVersion

Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

retrieveObjectsByName

The retrieveObjectsByName method allows tools to instantiate the view types specified for multiple objects. This method is an extension of the retrieveByName method; it allows tools to specify a collection of name-viewtype pairs to be retrieved, whereas retrieveByName only allows for specification of a single name and view type. All attributes and objects included by the view types are instantiated in the cache.

Handles to the view roots matching the qualifications that are found on the server are returned in the DSModelCollection. Tools are responsible for freeing the collection (which, in turn, frees the handles) when they are done with it.

Method syntax

```
In C++:

static DSModelCollection * _TCPart::retrieveByName(
                         NameViewList * list,
                         int lock, int refreshFlag,
                         int forceOption, char* strVersion);

In Smalltalk:  not yet implemented
```

Arguments

NameViewList * list

NameViewList is a class that contains a collection of handle/view pairs used to specify the objects to be retrieved. The ’list’ portion of the argument is a pointer to an instance of NameViewList.

NameViewList provides some simple interfaces and a cursor class to aid in its creation and maintenance, as follows:

```
class NameViewList {
        NameViewList &add( char *name, char *view );
        NameViewList &remove( char *name, char *view );
        int          isMember( char *name, char *view );
        int          Count();
};

class NameView {
        NameView(char *wName, char *view);

        char     *View();
        char     *WildName();
};

class NameViewList_Cursor {
        NameViewList_Cursor( NameViewList *nvl );
        NameView *first();
        NameView *next();
        NameView *prev();
```

int lock
: Instructs TeamConnection cache services that objects should be locked upon retrieval:

lock==1
: All NamedObjects in the view are locked.

lock==0
: No additional locks are acquired.

int refreshFlag
: Instructions to TeamConnection cache services used to determine the response if the storage representation of an object differs from the cache representation.

lock==REFRESH_OVERWRITE
: Indicates that the storage representation will replace the cache representation.

lock==REFRESH_NONE
: Indicates that the retrieval will fail if the cache object has changed since retrieval from the repository.

lock==REFRESH_PRESERVE
: Allows the retrieve to succeed if the time stamps on the cache and server do not match, but will preserve changes made in the cache.

int forceOption
: An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

The force option is important only if LOCK is specified. If you want to retrieve or store a locked part in a particular release or workarea that is linked to another release or workarea, you might want to specify the force option when you are checking in or checking out the part, even if someone else might have the part checked out in another context.

forceOption==1
: Indicates that links should be broken.

forceOption==0
: Indicates that links should be preserved.

Note: If 0 is supplied, and the part is linked with versions not specified in the version string, the action will fail.

char *strVersion
Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

Storing objects

Tools use the store method to cause changes to the object to be persistently stored in the database. All modified objects in the view definition are stored in a single transaction, and either succeed or fail as a unit.

If constraints have been defined for the object(s) in the view type, the constraints are checked before the transaction is committed. If any one of the constraints fail, the entire transaction is rolled back, and no changes are made persistent.

No handle is returned from the interface, as the tool already has a handle to the object being stored.

store

This method stores cache objects identified by a view type into the repository.

Method syntax

```
In C++:

virtual int store(char* viewType, int bCheckTimeStamp,
                         int forceOption, uINT4 lockOption,
                         char* comments, char* strVersion);

In Smalltalk:

(TCPart instance method) store:aViewTypeString
                         checkTimeStamp:checkTimeStampInteger
                         forceOption:aForceOptionInteger
                         lockOption:aLockOptionInteger
                         comments:comments version:aVersionString
```

Arguments

char* viewType
The name of the view that is used to store the instance data. For a given store request, only a single view type can be specified, although an object can be stored as any number of view types, provided that the view types have the same root entity.

int bCheckTimeStamp
Instructs TeamConnection cache services (TCCS) to evaluate client and server time stamps when objects are stored.

bCheckTimeStamp==1
Causes the server to compare the time stamp of the object being stored with the time stamp on the server version.

Note: This option is recommended. If not specified, it can result in overwriting competing changes from another source.

bCheckTimeStamp==0
Causes TeamConnection cache services to ignore time stamps when objects are stored.

int forceOption
An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

The force option is important only if LOCK is specified. If you want to retrieve or store a locked part in a particular release or workarea that is linked to another release or workarea, you might want to specify the force option when you are checking in or checking out the part, even if someone else might have the part checked out in another context.

forceOption==1
Indicates that links should be broken.

forceOption==0
Indicates that links should be preserved.

Note: If 0 is supplied, and the part is linked with versions not specified in the version string, the action will fail.

uINT4 lockOption
Instructs TeamConnection cache services (TCCS) on how handle the locking of TCParts:

LOCK_OBTAINANDRELEASE
Also known as optimistic locking, TCCS will attempt to check out the part(s) before checking in changes. The part(s) are not locked after the TeamConnection action.

LOCK_OBTAINANDRETAIN
TCCS will attempt to check out the part(s), and will keep the part(s) locked after checking in changes.

LOCK_RELEASE
Indicates that the part(s) is/are already locked, and the lock should be released after applying the changes.

LOCK_RETAIN
Indicates that the part(s) is/are already locked, and the lock should be retained after applying the changes.

char *comments
Free-form text commentary associated with the TeamConnection action.

char *strVersion

Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

storeObjects

This method stores cache objects identified by list of view types into the repository.

Method syntax

```
In C++:

virtual int storeObjects(char* viewList, int bCheckTimeStamp,
                         int forceOption, uINT4 lockOption,
                         char* comments, char* strVersion);

In Smalltalk:  not yet implemented
```

Arguments

ObjectViewList * list

ObjectViewList is a class that contains a collection of handle/view pairs used to specify the objects to be stored. The ’list’ portion of the argument is a pointer to an instance of ObjectViewList.

ObjectViewList provides some simple interfaces and a cursor class to aid in its creation and maintenance, as follows:

```
class ObjectViewList {
        ObjectViewList();

        ObjectViewList &add( ADObject *pObj, char *view );
        ObjectViewList &remove( ADObject *pObj, char *view );
        int            isMember( ADObject *pObj, char *view );
        int            Count();
};

class ObjectView {
        ObjectView(ADObject *pObj, char *view);

        char     *View();
        ADObject *Object();
};

class ObjectViewList_Cursor {
        ObjectViewList_Cursor(ObjectViewList *ovl);
        ObjectView *first();
        ObjectView *next();
        ObjectView *prev();
};
```

int bCheckTimeStamp
Instructs TeamConnection cache services (TCCS) to evaluate client and server time stamps when objects are stored.

bCheckTimeStamp==1
Causes the server to compare the time stamp of the object being stored with the time stamp on the server version.

Note: This option is recommended. If not specified, it can result in overwriting competing changes from another source.

bCheckTimeStamp==0
Causes TeamConnection cache services to ignore time stamps when objects are stored.

int forceOption
An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

The force option is important only if LOCK is specified. If you want to retrieve or store a locked part in a particular release or workarea that is linked to another release or workarea, you might want to specify the force option when you are checking in or checking out the part, even if someone else might have the part checked out in another context.

forceOption==1
Indicates that links should be broken.

forceOption==0
Indicates that links should be preserved.

Note: If 0 is supplied, and the part is linked with versions not specified in the version string, the action will fail.

uINT4 lockOption
Instructs TeamConnection cache services (TCCS) on how handle the locking of TCParts:

LOCK_OBTAINANDRELEASE
Also known as optimistic locking, TCCS will attempt to check out the part(s) before checking in changes. The part(s) are not locked after the TeamConnection action.

LOCK_OBTAINANDRETAIN
TCCS will attempt to check out the part(s), and will keep the part(s) locked after checking in changes.

LOCK_RELEASE
Indicates that the part(s) is/are already locked, and the lock should be released after applying the changes.

LOCK_RETAIN
Indicates that the part(s) is/are already locked, and the lock should be retained after applying the changes.

char *comments
Free-form text commentary associated with the TeamConnection action.

char *strVersion

Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

Locking objects

The lock interface performs a TeamConnection checkout against each TCPart in the view type. Objects that have already been locked have their lock count increased in the cache, as TeamConnection does not support multiple checkouts against a part in the storage view. As such, exiting a tool or unlocking the object through another interface may not preserve the lock count in the cache as expected. For instance, a tool may edit a given object in several windows, locking the object in each. If the user unlocks the object through the TeamConnection GUI, the locks for each window will be lost.

Objects that are added to the view instance through other means (such as those added to the database by other users) will not be locked; the cache representation is used to determine which objects are locked.

It is important to realize that the cache lock function does not necessarily represent the current lock state. It only provides an indication of locking attributes within one instance of the cache, as set by the retrieve interfaces, and maintained by store.

There are limits on the reliability of the cache representation of lock states. For example, there would be no indication in the cache that the following actions have taken place:

- An object is locked when first retrieved, assuming the object was already locked before the retrieve action.
- A user locks an object just retrieved into the cache through another interface.
- The same tool is started in another window, or another tool is started, and one of those tools locks an object just retrieved.
- A tool abends or otherwise quits without unlocking.
- Any TeamConnection command is performed.

Method Syntax

```
In C++:

virtual void lock(char* viewType, int forceOption, char* strVersion);

In Smalltalk:

(TCPart instance method) lock:aViewTypeString forceOption:aForceOptionInteger
                         version:aVersionString
```

Arguments

char* viewType
The name of the view that is used to lock the instance data. For a given lock request, only a single view type can be specified, although an object can be locked as any number of view types, provided that the view types have the same root entity.

int forceOption
An indication that changes should be forced into the repository, possibly breaking links with the part in other version contexts. Its intent is to indicate that, although the version string supplied might not match the current set of versions applicable to the object in the persistent store, the changes in those versions specified in the version string are to be made, breaking the links to those current versions not included in the version string.

forceOption==TRUE
Indicates that links should be broken.

forceOption==FALSE
Indicates that links should be preserved.

Note: If FALSE is supplied, and the part is linked with versions not specified in the version string, the action will fail.

char *strVersion
Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

Unlocking objects

The unlock interface unlocks each TCPart in the view type. Objects that have multiple locks in the cache have their lock count decreased; when the lock count reaches 0, the cache unlocks the object on the server.

Objects that are added to the view instance through other means (such as those added to the database by other users) will not be unlocked; the cache representation is used to determine which objects are unlocked.

Method syntax

```
In C++:

virtual void unlock(char* viewType, char* strVersion);

In Smalltalk:

(TCPart instance method) unlock:aViewTypeString version:aVersionString
```

Arguments

char* viewType
The name of the view that is used to lock the instance data. For a given lock request, only a single view type can be specified, although an object can be locked as any number of view types, provided that the view types have the same root entity.

char *strVersion
Specifies which version contexts are to have the action applied. Single version contexts are specified with the form ″family|release|workarea″ or ″family|release|release″; multiple version contexts can be supplied for some actions, and take the form: ″family|release1 release2...releasen|workarea1 workarea2...workarean″.

Note: A driver or versionID can be used in place of workarea.

If the strVersion is specified as ″family|release|release″, the context is set to refer to the release; only read-only operations are allowed.

Note that an individual object can have only one context; an object with a different version string is considered a separate object.

Deleting objects

Deleting objects from the repository is performed through a combination of methods. First, the object is marked for deletion via the ADObject::markDeleted() method. Secondly, the object is transmitted to the server (and the delete operation made persistent) through the store method.

```
In C++:

ADObject::markDeleted(DSModelCollection *log, BOOL bLog)

In Smalltalk:

(ADObject instance method) markDeleted:anADModelCollection log:aIntegerLogOption
```

The markDeleted method requires a collection that the delete action should be logged against (keeping in mind that the deletion will not be persistent until the server is notified through a store operation). The log can be either one that is provided and managed by the tool, or one that is associated with a named object. In the C++ example, the value of bLog determines whether delete actions are added to the collection (TRUE), or not (FALSE).

All TCParts in a view instance have a delete log associated with them in the cache, as a result of loading the view into the cache. One way to access the delete log is supplied in the following example:

```
(*pObj) ->markDeleted( (*(TCPart*)pPart ->_get_deleteLog() );
```

During the store operation, any TCParts encountered while walking the view are processed, and their contents included in the data stream to be sent to the server.

As such, if you choose to provide your own delete log management, you are responsible for transferring the objects on that log to a TCPart included in the view before the store operation proceeds. If the store is successful, the server will update the cache and the delete logs for the associated named objects are emptied.

Objects can appear on their own delete log. Attempting to markDelete the same object several times has no effect. Finally, objects that have just been created are not logged, since there is no persistent store representation to delete.

Chapter 9. Exception handling

TeamConnection uses exceptions to notify tools that TeamConnection client actions have not succeeded. Exception messages indicate the cause of errors. The class of exceptions relevant to tools is called *TCException*.

Exception handling methods

TCException is a C++ class that has several public methods used by tools to access information about an error. Methods used to access error information include the following:

long getMsgNum();
Returns the exception number.

char *getMessage();
Returns the actual text of the error.

long getSeverity();
Returns the severity of the error:

0	Informational
4	Warning
8	Error
12	Severe
16	Unrecoverable

When the exception is detected, it is the tool's responsibility to free up any memory used by the exception. By including the TC Client include file tcrs.hpp, tools should have access to all prototypes and defines needed. However, if necessary, they can include tcexp.hpp to get to the TCException class prototype directly.

Exception messages

When tools detect an exception, they should compare and map the exception message number returned with the exception message macros included in cmvc_msg.h.

The following is a list of the exception message macros with a description of each:

```
CC_NAMED_OBJECT_EXISTS
An attempt to recreate a %1$s named %2$s was made.  This action will
create a duplicate Named Object either in the cache or in the database.
There cannot be identically named objects in either the cache or
the database.


CC_HLQ_NOT_SET
The MVS Dataset High Level Qualifier (HLQ) needs to be changed
in the user Profile.
Currently, this is done by updating the
appropriate field in the Profile Table found in
EWSPPROF.TXT. In most cases the HLQ should be your MVS Userid.


CC_CONSTRAINT_FAIL
The store operation failed because one or more objects violated
```

```
constraints.
Number of constraint violations: %1$s
Constraint Message(s):
%2$s


CC_CANT_CONNECT_DB
An attempt to connect to a %1$s Database was made, but the
connection cannot be made.  The reason code is
%2$s.  The database alias that cannot be connected is %3$s.


CC_SQL_RC
A SQL command returned with a return code
that was non-zero, indicating that some condition occurred during
execution, other than successful completion.  A return code of
%1$s was returned by the SQL command %2$s.


CC_COMPILER_CODE
The a %1$s compiler returned with a non-zero
return code %2$s or could not be invoked.


CC_REMOTE_COMP_CODE
A call to a remote function returned a completion code
that was non-zero, indicating that some condition occurred during
execution, other than successful completion.  The remote function
invoked was %1$s, the completion code returned was %2$s.


CC_UNIMPLEMENTED_METHOD
An unexpected attempt was made to call a method that
is not implemented in the DSObject subclass against which it was
invoked. The object type against which the method was invoked is
%1$s.  The method that is invoked was %2$s.


CC_ADD_DUPID_TO_OBJMGR
An attempt to recreate a %1$s named
%2$s was made.  This action will create a duplicate
Named Object either in the cache or in the database.  There cannot
be identically named objects in either the cache or the database.


CC_CANT_FIND_OBJ_IN_OBJMGR
Object %1$s was not found in the cache.


CC_DOS_CALL_FAILED
An Operating System call, %1$s, failed.
The return code is:  %2$s.


CC_CANT_ACCESS_REMOTE_SYSTEM
The remote system, %1$s, cannot be
accessed for the desired action (%2$s).
For some reason a connection to the remote system cannot be
established.
The reason code is:  %3$s.


CC_CANT_ACCESS_LIBRARY
The library, %1$s, cannot be accessed for
the desired action (%2$s).
A connection to the host may not be establishable, or the library
may not exist.
```

```
CC_CANT_RETRIEVE_FILE
The remote file, %1$s cannot be retrieved from
the host operating system.  The host id being used for retrieval is
%2$s.  The local file name where the file will be stored
is %3$s.  The return code from the internal call is
<%4$s>.
A connection to the host may not be establishable, the file
may not exist, or the file may be being accessed by another process.


CC_CANT_SEND_FILE
The local file, %1$s cannot be sent to
the host operating system.  The host id being used for transfer is
<%2$s>.  The remote file name where the file should be
stored is <%3$s>.  The return code from the internal
call is <%4$s>.
A connection to the host may not be establishable, the file
may not exist, or the file may be being accessed by another process.


CC_NO_PLATFORM_FILE_INFO
The operating system platform information that is supposed to
participate in this file access, (Platform = %1$s)
cannot be accessed.  Either this information has not yet been defined
or the given platform is not supported.


CC_SEM_ERROR
The attempt to create a semaphore was not successful.  This
can occur when your workstation does not have enough available
resources.


CC_THREAD_ERROR
The attempt to start an OS/2 thread was not successful.
This can occur when your workstation does not have enough available
resources.


CC_NO_REQUEST
There is nothing in the queue to process.


CC_API_NOT_FOUND
The required TeamConnection API %1$s
was not found in the module %2$s.


CC_DLL_LOAD_ERROR
The required module, %1$s, could not be loaded.
If you are trying to access the DB2/2 Catalog,
ensure that you have DB2/2 installed.


CC_SUBSYS_NO_INIT
An attempt to execute a request in subsystem
%1$s failed because the subsystem has not yet
been initialized.


CC_QSERV_NA
The request cannot be submitted because the queuing service
is not activated.
```

```
CC_NOTHING_IN_DEVICE
There is nothing in the the device to search.  Nothing can be
listed.


CC_NO_MATCHES_IN_DEVICE
The search for an object using the following match pattern
yielded no matches:  %1$s.


CC_OBJECT_TYPE_NOTFOUND
A processing error occurred when a reference was made to a
type %1$s that is not in the table of
supported TeamConnection types.


CC_CANT_ACCESS_FILE
The file, %1$s cannot be accessed.
It may not exist, or it may be being accessed by another process.


CC_INVALID_CC
An unrecognized exception has occurred.  <%1 = %2>.
The condition code for the exception code cannot be found in the
TeamConnection message table.  See the TeamConnection logs
for details as to where the exception occurred.


CC_UNEXPECTED_CLASS
A processing error has occurred due to the unexpected use of
a particular class in processing.  The class name specified,
%1$s is not expected at this point in the object
class processing.  See the
TeamConnection logs for details as to where the exception occurred.


CC_UNEXPECTED_ATTRIB
A processing error has occurred due to the unexpected use of
an attribute in processing.  Either the attribute,
%1$s is not a member of the object class %2$s or
is not expected at this point in the object class processing.  See the
TeamConnection logs for details as to where the exception occurred.


CC_OBJ_MEMALLOC
The attempt to create an object was not successful because
there is not sufficient memory.


CC_COLL_ADD
An internal processing error occurred while
attempting to add an object, %1$s, to a
collection.  Incorrect results may occur.


CC_MODCOLL_ADD
An internal processing error occurred while attempting to
add an object, %1$s, to a collection.


CC_MODCOLL_DUP
An internal processing error occurred while attempting
to add an object, %1$s, to a collection.
The object being added is a duplicate of an object that is already
in the collection.  The existing index is %2$s.
```

```
CC_INPUT_SNO
A processing error occurred while attempting to
internally pass an incompatible parameter.  The internal
variable name that caused the exception is %1$s.
The value passed was %2$s.


CC_GET_SNO
A processing error occurred while attempting to
access an attribute or object.  The object type being
accessed was %1$s.  The attribute that could not be gotten was %2$s.


CC_SET_SNO
A processing error occurred while attempting to
set an attribute for an object via a set method.
The object type was %1$s.  The attribute that could not
be set was %2$s.


CC_RSS_OS_ERROR
An ObjectStore processing error occurred.  The message
returned from ObjectStore follows: %1$s.


CC_RSS_INV_VERSION
A processing error occurred while trying to access versioned
data in the repository because of the specification of a
NULL or invalid version string.  The invalid version string
was %1$s.


CC_RSS_NO_CACHE_OBJECT
TeamConnection encountered an error while trying to locate
an object %1$s with key or name
%2$s in the Cache.  The specified object
was expected to be found in the Cache but could not be found
by the key or name indicated.  See the TeamConnection logs
for details as to where the exception occurred.


CC_RSS_INV_CDF_UNIT
An error occurred in TeamConnection because an
invalid CDF unit %1$s was encountered during the parsing of
an input CDF data stream.  A communication failure may have
caused the error.


CC_RSS_VIEW_NOT_INITIALIZED
This error occurs only when the View Type meta-model
methods is used outside of the Repository Services code.


CC_RSS_INV_VIEW_DEF
The View Type is not defined as expected.


CC_RSS_WRONG_METHOD_INVOCATION
According to the input data stream, the method being invoked
was %1$s, but the actual method invoked was %2$s.


CC_RSS_VERS_CONTEXT_ERR
A processing error occurred while trying to set versioned
context data %1$s.
```

```
CC_RSS_CHECKIN_ERROR
Could not check in %1$s with name %2$s.
%3$s


CC_RSS_CHECKOUT_ERROR
Could not check out %1$s with name %2$s.
%3$s


CC_RSS_CREATEMO_ERROR
Could not create %1$s with name %2$s in TeamConnection.
%3$s


CC_RSS_INV_KEY_FORMAT
Persistent store keys must be in the form of .


CC_RSS_CREATE_ERROR
An error occurred while trying to create an object
of type '%s'.


CC_RSS_OBJ_NOT_FOUND
The object could not be found with the specified key(%1$s).
Another user may have deleted it.


CC_RSS_NO_CLASS_EXTENT
No class extent has been defined in the persistent store
model for the indicated class.


CC_RSS_CACHE_CREATE_ERROR
An error occurred while attempting to create object
'%1$s' in the cache.  The object was not created successfully.
Be sure the required models are loaded.


CC_RSS_RENAMEMO_ERROR
An error occurred while trying to rename an object of
type '%s' from '%s' to '%s'.
%s"


CC_RSS_REMOVEMO_ERROR
%1$s could not be removed with name %2$s.
%3$s


CC_EWSX_BAD_DATA
No data was found in the CDF Data Stream passed to
the builder or parser or the data stream passed is not
valid.  A communication failure may have caused the error.


CC_EWSX_BAD_UNIT
A bad unit was encountered in the CDF Data Stream.  A
communication failure may have caused the error.


CC_TCAPI_INITFAILURE
The family may not be started.  The entries in your TCP/IP
hosts and services files may not be correct.
```

```
CC_RSS_WORKAREA_NOT_EXIST
Workarea %1$s is not associated with release %2$s.
The specified workarea may not have been created or the
specified name is not correct.


CC_RSS_CANT_RERETRIEVE_OBJECTS
The specified objects can not be retrieved again
because the objects in the cache are out of date with
the server.


CC_EWSX_CANT_OPEN_FILE
Can't open the file '%1$s'. The attempt
to open '%1$s' was not successful.  This can occur when:
  - The file does not exist
  - Your workstation does not have enough available resources.


CC_EWSX_ERROR_IN_FILE
An error occurred while reading a CDF file.  A communication
failure may have caused the error.


CC_RSS_FAILURE_CREATING_COMPONENT
An error was detected while trying to create a component
during import.


CC_RSS_CONNECT_ERROR
An error occurred trying to perform a connect.


CC_RSS_DISCONNECT_ERROR
An error occurred trying to perform a disconnect.


CC_RSS_CANT_ACCESS
You are not authorized to access the specified object.


CC_RSS_TIMESTAMP_CHECK_FAILED
The object you are trying to store, %1$s,
has been updated since it was extracted from the repository.


CC_RSS_VERS_CONTEXT_RELEASE_WORKAREA_SAME
The name of the workarea was the same as the name of the release for a
store operation.  This is not allowed. Correct the version specification,
and retry the operation.


CC_RSS_VERS_CONTEXT_WORKAREA_NOT_FLUID
The workarea specified is frozen.  Stores are not permitted to
a frozen workarea.  Correct the version specification, and
retry the operation.


CC_RSS_VERS_CONTEXT_WORKAREA_READ_ONLY
The context for the workarea specified is marked as read only.  Stores are
not permitted to a read-only context.  Correct the version specification,
and retry the operation.


CC_RSS_CANNOT_COPY
The specified object cannot be copied because it is currenlty
opened for updates.  Objects cannot be copied in the repository
```

```
if they are in the process of being changed by a repository tool.
Complete all potential updates to the object and store changes in
the repository prior to attempting a copy on this object.


CC_RSS_CANNOT_DELETE
The specified object cannot be deleted because it is currenlty
opened for updates.  Objects cannot be deleted from the repository
if they are in the process of being changed by a repository tool.
Complete all potential updates to the object and store changes in
the repository prior to attempting to delete this object from
the repository.


CC_RSS_DB_DEADLOCK
A deadlock situation has occurred on the server causing the transaction
to be aborted.  Please retry the operation.


CC_SQL_ERROR
An SQL error has occurred.
%s


CC_RSS_NAME_TOO_LONG
An error occurred while trying to set the fullname of an object.
The name specified was too long.


CC_RSS_NOCONTROLLER
A dependent object, '%s', was encountered but no controlling
ManagedObject could be found.


CC_BAD_VIEWTYPE
The specific viewType, '%s', is not valid.  Be sure the required
models are loaded.


CC_RSS_PART_NOT_FOUND
Part %s of type %s was not found.


CC_RSS_RELEASE_EMPTY
The release %s does not contain any parts to export.


CC_RSS_WORKAREA_EMPTY
The workarea %s does not contain any parts to export.


CC_RSS_MISSING_SRC_TGT
An error occurred while trying to create a relationship object
of type '%s'.  Either the source or the target of this relationship
is not present.


CC_RSS_BAD_LOCK_FLAG
An error occurred while trying to perform a store operation.
An invalid 'Lock' flag was used.


CC_RSS_DRIVER_NOT_EXIST
Driver 1$s is not associated with release %2$s.
The specified driver may not have been created or the
specified name is not correct.
```

```
CC_RSS_DRIVER_EMPTY
The driver %s does not contain any parts to export.


CC_RSS_MAX_CARD_ERROR
Cardinality violation:
The attribute '%s' on object '%s' has a maximum cardinality of %s.


CC_RSS_INVALID_TIMESTAMP
The timeStamp specified, '%s', is invalid.


CC_RSS_INVALID_OID
An object of type '%s' has an invalid OID (%s).


CC_RSS_CONTROLLER_SYNC
An object of type '%s' has a calculated controller that is
not its defined controller.  Verify that the model definition
is correct and that all non-managed objects have at least one controlling
rel defined.


CC_RSS_CONTROLLER_NOT_STORED
An attempt was made to store a new object of type '%s' without storing its
controller.


CC_RSS_CONSTRAINT_ERROR
Constraint errors were encountered while attempting to store an object.
%s


CC_RSS_BAD_SET_OBJECT_TYPE
An object of type '%s' was expected, but an object of type '%s' was
passed to the %s::_set_%s method.


CC_RSS_CANT_RETRIEVE_NEW_OBJ
The specified object can not be retrieved from the server since
it is a newly created object and has not been stored yet.


CC_RSS_SQL_CANT_SET_CONTEXT
The attempt to set the version context to
release '%s' and workarea '%s' failed.
Verify that the workarea has not been pruned
and that the release and workarea names
specified are correct.  The query could not be executed.


CC_RSS_USE_SMALLER_CHUNKSIZE
There is not enough memory available to allocate space for the
CDF file being generated.  Please set TC_CDFCHUNKSIZE to a smaller
value.  It's current value is '%s'.


CC_RSS_CANT_CONTROL_MOS
The relationship object '%s' is defined to have controlling semantics,
but managed objects can not be controlled.


CC_RSS_DUP_ANNOT_LABEL
The label '%s' is used in more than one ADAnnotationText object.
```

```
CC_RSS_WRONG_SLASH
An attempt was made to use a backward slash instead of a forward slash
in a name('%s').  Convert all slashes to forward slashes and try again.


CC_RSS_IMPORT_FAILED
An error occurred during import.
%s


CC_RSS_LOST_SERVER_CONNECTION
The connection to the server was broken.  Please try the request again.


CC_RSS_LOST_CLIENT_CONNECTION
The connection to the client was broken.


CC_RSS_INCOMPATIBLE_CLIENT
The client is incompatible with the server.
ToolName: %s
Client Version: %s
Server Version: %s


CC_RSS_MIN_CARD_ERROR
Cardinality violation:
The attribute '%s' on object '%s' has a minimum cardinality of %s.


CC_RSS_DIFFERENT_VERSIONS
The version context specified on the object(%s) does not match the
version context of the object on the other end of the relationship(%s).
Relationship objects require the source and target objects to have
the same version context.


CC_RSS_BAD_STORE_VERSION
The primary version context specified on the store operation doesn't
match one or more of the version contexts of the objects being stored.


CC_RSS_CANT_WRITE_TO_FILE
An error occurred while trying to write data to file '%s'.
This will usually happen when the disk is full.


CC_RSS_ONLY_SU_CAN_SU
Only a superuser can use the alternate user fields.
```

Chapter 10. Performance and scalability issues

For purposes of this section, *performance* is defined in terms of user responsiveness, throughput, and other traditional measures of software capability. *Scalability* involves the extension of performance to an increasingly larger, multiuser server system, and addresses parameters such as the number of active users, the size of data, and the arrival rate of transactions.

For tool builders, the primary focus for improving tool performance involves designing models that access the appropriate scope of repository data for a user requests (through the use of views) and minimizing the number of unnecessary transactions. For many complex requests, both view scope and the number of transactions must be considered and balanced to provide optimum performance.

The TeamConnection server controls how TCParts are clustered in the database. Tool builders affect the degree of object clustering by determining the scope of individual TCParts. An efficient clustering strategy increases the likelihood that, for common access patterns, the next object in a request or the next request will reside in a location already mapped from the persistent store to the database cache.

A *server daemon* is an individually dispatched process capable of processing an entire tool request. The TeamConnection server might consist of many server daemons. If all daemons are busy processing requests, additional requests are queued until a server daemon is free, which is apparent to the tool user as a delay. By minimizing the number of individual transactions initiated by user requests, a tool builder can minimize the amount of queueing at the server.

This section is offered as a guide for improving tool performance during design. Actual performance testing techniques are beyond the scope of this document.

Specific recommendations

During the design process, tool builders have an opportunity to improve overall tool performance (in terms of user responsiveness) by addressing aspects of object modeling, transaction scheme, and view type design. These issues are interrelated, so each should be considered as part of a broader performance improvement strategy. It is also necessary to take into account the related system parameters, such as number of users, database size, and data volume within individual user interactions.

Object modeling recommendations

Design recommendations related to object modeling are based on the following tool builder tasks:

- Walk through the object model periodically to identify object sizes, expected cardinalities, and nesting depths.
- Identify and justify all TCPart and class extents (data structures that allow navigation to each instance of a TCPart subclass).
- Consider any special object clustering requirements.
- Identify indexes that may exist on an attribute of an object class (usually a name or an ID).

  Note: The TBDK Breditor supplies an indexing option.

It is advisable to justify each TCPart, because there is general overhead related to TCParts. The recursive potential of view types (see “Building a view type” on page 21 for an explanation) can increase this overhead exponentially.

The justification process involves determining whether or not the object in question should be independently managed. In many cases, you might find that a prospective TCPart could be relegated to the status of plumbing object or attribute, based on the following (and similar) criteria:

1. Is the object likely to to be the root of a view?
2. Should the object be independently versioned?
3. Would you define the scope of a lock with the object?

If your answers to these and related questions are *yes*, then the object probably should be defined as a subclass of TCPart. If not, you might be able to develop an alternative model that provides the desired behavior, without the additional overhead.

Transaction and view type design recommendations

Design recommendations related to transactions and view type design are based on the following tool builder tasks:

- Identify the primary user interface (UI) interactions with the object model and their frequency.
- For each UI interaction:
  - Identify and justify the corresponding view types and TCAPI server transactions.
  - Estimate data volumes and view type complexity.
- Consider concurrency issues between transactions.

Common and high-impact performance considerations for transaction and view type design include the following:

1. Justify each instance of nested view types, which may cause the tool to retrieve much more data than is necessary for most typical user requests. This is an especially important consideration when view types are nested at two or more levels.
2. Determine the correspondence between the data retrieved in a view and what the user actually sees on the GUI. The rule of thumb is to retrieve from the repository only the data necessary to populate the UI screen, and no more.
3. Check views for impossible links, such as mutually exclusive relationships. Again, only include information necessary for the view type.
4. If you break a view into multiple view types to improve performance, consider the following coding techniques:
   - Use subviews that are loaded at user request (for example, a GUI action).
   - Obtain a small amount of repository data at first, displaying it to the user immediately, and spawn a backgound load to retrieve the remainder of the desired data while the user evaluates the initial UI screen.
   - Create a hierarchy of views, in which the lowest-level parts would be retrieved only if their owning object was actually accessed by the user.
5. Use access patterns that consolidate transactions, when possible. For example, instead of using a listByName method (which returns a handle to an object) followed by a retrieve, use the TeamConnection retrieveByName API, which performs the same work in a single transaction.

Customer support

Your options for IBM VisualAge TeamConnection support, as described in your License Information and Licensed Program Specifications, include electronic forums. You can use the electronic forums to access IBM VisualAge TeamConnection technical information, exchange messages with other TeamConnection users, and receive information regarding the availability of fixes. The following forums are available.

- **IBM Talklink**

  Use the TEAMC CFORUM. For additional information about TalkLink, call

  – United States 1-800-547-1283

  – Canada 1-800-465-7999 ext. 228
- **CompuServe**

  From any ! prompt, type GO SOFSOL, then select TeamConnection. For additional information, call 1-800-848-8199 and ask for representative 239.
- **Internet**

  Go to the IBM homepage, http://www.ibm.com. Use the search function with keyword TeamConnection to go to the TeamConnection area.

If you cannot access these forums, contact your IBM representative.

There are several other support offerings available after purchasing IBM VisualAge TeamConnection. For a list of these offerings, please contact your IBM representative.

Bibliography

IBM VisualAge TeamConnection library

The following is a list of the TeamConnection publications.

- **License Information (GC34-4497):**

  Contains license, service, and warranty information.
- **Installing the TeamConnection Servers (GC34-4551):**

  Lists the hardware and software that are required before you can install and use the IBM VisualAge TeamConnection product, provides detailed instructions for installing and configuring the TeamConnection family and build servers, and provides instructions for administering a TeamConnection family.
- **TeamConnection Getting Started (SC34-4552):**

  Tells first-time users how to install the TeamConnection clients on their workstations, and familiarizes them with the command line and graphical user interfaces.
- **TeamConnection User's Guide (SC34-4499):**

  A comprehensive guide for TeamConnection administrators and client users that helps them install and use TeamConnection.
- **Commands Reference (SC34-4501):**

  Describes the TeamConnection commands, their syntax, and the authority required to issue each command. This book also provides examples of how to use the various commands.
- **Quick Commands Reference (GC34-4500):**

  Lists the TeamConnection commands along with their syntax.
- **Staying on Track with TeamConnection Processes (83H9677):**

  Poster showing how objects flow through the states defined for each TeamConnection process.
- The following publications can be ordered as a set (SBOF-8560):

  Installing the TeamConnection Servers

  TeamConnection Getting Started

  TeamConnection User's Guide

  Commands Reference

  Quick Commands Reference

  Staying on Track with TeamConnection Processes

Tool Builder's Development Kit

The following publications are part of the Tool Builder's Development Kit feature:

- **Tool Builder's Development Guide (SC34-4553):**

  Explains how to create and extend tools for accessing objects in the TeamConnection database. It contains guideance and reference information.
- **Information Model Reference (SC34-4554):**

  Details the TeamConnection information model. This publication is available in softcopy only.

ObjectStore

The following publications are part of the ObjectStore library of documents and are available for order from Object Design, Inc. To order these documents call (617) 674-5000, Monday through Friday from 8:30 AM to 5:30 PM Eastern Time.

- **ObjectStore C++ Installation:**

  Contains step-by-step procedures for installing the latest release of ObjectStore on a specific platform:

  310-100-40 I
  UNIX

  310-310-40 I
  Windows

  310-320-40 I
  OS/2

- **ObjectStore C++ API User Guide (310-000-40 U):**

  Provides information about the application programming interface for application programmers.

- **ObjectStore C++ API Reference (310-000-40 R):**

  Describes the API to the features provided by ObjectStore for application programmers.

- **ObjectStore C++ Building Applications (310-000-40 B):**

  Provides information and instructions for compiling code, generating schemas, and linking files using all supported compilers; and provides instructions for developing ObjectStore client applications for use on multiple platforms.

- **ObjectStore Management (310-000-40 M):**

  Provides information and instructions for perfroming management tasks on ObjectStore server and client systems. It includes server parameters, environment variables, and database utilities.

- **ObjectStore C++ Performance (310-000-40 P):**

  Explains the fundamentals of designing and tuning ObjectStore applications for optimal performance.

IBM Exchange library

The publications listed below can be ordered as a set (SBOF-6098) or separately as indicated below. IBM Exchange will be available at a later date.

- *Licensed Programming Specification (GC34-4525):*
- *Installation Guide (SC34-4509):*
- *Bridge Builder's Guide (SC34-4508):*
- *User's Guide 1 (SC34-4506):*
- *User's Guide 2 (SC34-4507):*

Related publications

- Transmission Control Protocol/Internet Protocol (TCP/IP)
  - *TCP/IP 2.0 for OS/2: Installation and Administration* (SC31-6075)
  - *TCP/IP for MVS Planning and Customization* (SC31-6085)
- MVS

- *MVS/XA JCL User’s Guide* (GC28-1351)
- *MVS/XA JCL Reference* (GC28-1352)
- *MVS/ESA JCL User’s Guide* (GC28-1830)
- *MVS/ESA JCL Reference* (GC28-1829)

Part Number: 33H2571

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

33H2571

SC34-4553-00